

Connecting Apama Applications to External Components

5.3.0

May 2015

This document applies to Apama 5.3.0 and to all subsequent releases.

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

Copyright © 2013-2015 Software AG, Darmstadt, Germany and/or Software AG USA Inc., Reston, VA, USA, and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA Inc. and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at <http://softwareag.com/licenses>.

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software AG Products / Copyright and Trademark Notices of Software AG Products". These documents are part of the product documentation, located at <http://softwareag.com/licenses> and/or in the root installation directory of the licensed product(s).

Document ID: PAM-Connecting_Apama_Applications_to_External_Components-5.3.0-20150518@254863

Table of Contents

Preface.....	12
Documentation roadmap.....	12
Contacting customer support.....	14
Using Standard Adapters.....	15
Chapter 1: Using the Apama Database Connector.....	16
Overview of using ADBC.....	16
Registering your ODBC database DSN on Windows.....	18
Adding an ADBC adapter to an Apama Studio project.....	18
Configuring the Apama database connector.....	20
Configuring an ADBC adapter.....	21
Manually editing a configuration file.....	24
Configuring ADBC localization.....	25
Configuring ADBC Automatic Database Close.....	25
Service monitors.....	25
ADBC blocks.....	26
Codecs.....	26
The ADBCHelper Application Programming Interface.....	27
ADBCHelper API Overview.....	27
Opening databases.....	28
Specifying the adapter instance.....	30
Checking to see if a database is open.....	31
Issuing and stopping SQL queries.....	31
Issuing SQL commands.....	32
Committing database changes.....	32
Performing rollback operations.....	32
Handling query results for row data.....	33
Handling query results for event data.....	33
Handling acknowledgments.....	33
Handling errors.....	34
Reconnection settings.....	35
Closing databases.....	35
Getting schema information.....	35
Setting context.....	35
Logging.....	36
Examples.....	36
The ADBC Event Application Programming Interface.....	37
Discovering data sources.....	37
Discovering databases.....	38
Opening a database.....	40
Closing a database.....	41
Storing event data.....	42
Storing non-event data.....	43
Creating and deleting store events.....	44

Handling data storing errors.....	45
Committing transactions.....	45
Rolling back transactions.....	46
Running commands.....	46
Executing queries.....	46
Executing standard queries.....	47
Stopping queries.....	49
Preserving column name case.....	49
Prepared statements.....	49
Using a prepared statement.....	49
Stored procedures.....	50
Using a stored procedure.....	51
Named queries.....	52
Using named queries.....	52
Creating named queries.....	54
The Visual Event Mapper.....	55
Using the Visual Event Mapper.....	56
Playback.....	60
Sample applications.....	61
Format of events in .sim files.....	61
Chapter 2: Using the Apama Web Services Client Adapter.....	63
Web Services Client adapter overview.....	64
Adding a Web Services Client adapter to an Apama Studio project.....	64
Configuring a Web Services Client adapter.....	65
Specify the Web Service and operation to use.....	66
Specifying Apama events for mapping.....	71
Editing Web Services Client adapter configurations.....	74
Adding multiple instances of the Web Services Client adapter.....	78
Mapping Web Service message parameters.....	78
Simple mapping.....	80
Convention-based XML mapping.....	81
Convention-based Web Service message mapping example.....	82
Template-based mapping.....	87
Combining convention and template mapping.....	88
Mapping complex types.....	90
Difference between doc literal and RPC literal WSDLs.....	90
Using custom EL mapping extensions.....	91
JUEL mapping expressions reference for Web Client Services adapter.....	92
Specifying a correlation ID field.....	94
Specifying transformation types.....	94
Specifying an XSLT transformation type.....	95
Specifying an XPath transformation type.....	95
Customizing mapping rules.....	96
Mapping SOAP body elements.....	96
Mapping SOAP header elements.....	97
Mapping HTTP header elements.....	99
Using EPL to interact with Web Services.....	99
Configuring logging for Web Services Client adapter.....	102

Web Services Client adapter artifacts.....	105
Chapter 3: Using the Apama File Adapter.....	106
File Adapter plug-ins.....	107
File Adapter service monitor files.....	108
Adding the File adapter to an Apama Studio project.....	108
Configuring the File adapter.....	109
Overview of event protocol for communication with the File adapter.....	111
Opening files for reading.....	111
Opening files for reading with parallel processing applications.....	113
Specifying file names in OpenFileForReading events.....	114
Opening comma separated values (CSV) files.....	115
Opening fixed width files.....	116
Sending the read request.....	116
Requesting data from the file.....	117
Receiving data.....	118
Opening files for writing.....	119
Opening files for writing with parallel processing applications.....	120
LineWritten event.....	121
Monitoring the File adapter.....	121
Chapter 4: Using Adapter Plug-ins.....	123
The Null Codec plug-in.....	123
Null codec transport-related properties.....	124
The File Transport plug-in.....	125
The String Codec plug-in.....	127
The Filter Codec plug-in.....	127
Filter codec transport-related properties.....	128
Specifying filters for the filter codec.....	130
Examples of filter specifications.....	130
The XML codec plug-in.....	131
Supported XML features.....	131
Adding XML codec to adapter configuration.....	132
Setting up the classpath.....	132
About the XML parser.....	133
Specifying XML codec properties.....	133
Required XML codec properties.....	134
XML codec transport-related properties.....	134
Message logging properties.....	136
Downstream node order suffix properties.....	136
Additional downstream properties.....	137
Sequence field properties.....	137
Upstream properties.....	137
Performance properties.....	137
Description of event fields that represent normalized XML.....	139
Examples of conversions.....	141
Sequence field example.....	142
XPath examples.....	143
The CSV codec plug-in.....	143
Multiple configurations and the CSV codec.....	144

Decoding CSV data from the sink to send to the correlator.....	145
Encoding CSV data from the correlator for the sink.....	145
The Fixed Width codec plug-in.....	145
Multiple configurations and the Fixed Width codec.....	146
Decoding fixed width data from the sink to send to the correlator.....	147
Encoding fixed width data from the correlator for the sink.....	147
Using Message Services.....	149
Chapter 5: Using Correlator-Integrated Messaging for JMS.....	150
Overview of correlator-integrated messaging for JMS.....	150
Samples for using correlator-integrated messaging for JMS.....	151
Key concepts for correlator-integrated messaging for JMS.....	151
Getting started with simple correlator-integrated messaging for JMS.....	152
Adding and configuring connections.....	153
Adding JMS receivers.....	156
Configuring receiver event mappings.....	157
Using conditional expressions.....	158
Configuring sender event mappings.....	161
Using EPL to send and receive JMS messages.....	162
Getting started with reliable correlator-integrated messaging for JMS.....	163
Mapping Apama events and JMS messages.....	164
Simple mapping for JMS messages.....	165
Using expressions in mapping rules.....	165
Template-based XML generation.....	166
Specifying an XPath transformation for JMS messages.....	166
Specifying an XSLT transformation for JMS messages.....	167
Convention-based XML mapping.....	167
Convention-based JMS message mapping example.....	168
Using convention-based XML mapping when receiving/parsing messages.....	169
Using convention-based XML mapping when sending/generating messages.....	170
Combining convention-based XML mapping with template-based XML generation.....	171
Using EDA events in Apama applications.....	171
About convention-based EDA mapping.....	171
Rules that govern automatic conversion between of EDA events and Apama events.....	171
Rules that govern EPL name generation.....	174
Creating Apama event type definitions for EDA events.....	176
Automatically mapping configurations for EDA events.....	177
Manually mapping configurations for EDA events.....	179
Handling binary data.....	180
Using custom EL mapping extensions.....	181
JUEL mapping expressions reference for JMS.....	181
Implementing a custom Java mapper.....	186
Dynamic senders and receivers.....	188
Durable topics.....	189
Receiver flow control.....	189
Monitoring correlator-integrated messaging for JMS status.....	190
Logging correlator-integrated messaging for JMS status.....	191
JMS configuration reference.....	197

Configuration files.....	197
XML configuration file format.....	198
XML configuration bean reference.....	200
Advanced configuration bean properties.....	206
Designing and implementing applications for correlator-integrated messaging for JMS.....	207
Using correlator persistence with correlator-integrated messaging for JMS.....	207
How reliable JMS sending integrates with correlator persistence.....	208
How reliable JMS receiving integrates with correlator persistence.....	208
Sending and receiving reliably without correlator persistence.....	209
Receiving messages with APP_CONTROLLED acknowledgements.....	209
Sending messages reliably with application flushing notifications.....	211
Using the correlator input log with correlator-integrated messaging for JMS.....	213
Reliability considerations when using JMS.....	213
Duplicate detection when using JMS.....	214
Performance considerations when using JMS.....	217
Performance logging when using JMS.....	218
Receiver performance when using JMS.....	219
Sender performance when using JMS.....	219
Configuring Java options and system properties when using JMS.....	220
Diagnosing problems when using JMS.....	220
JMS failures modes and how to cope with them.....	222
Chapter 6: Using Universal Messaging in Apama Applications.....	226
Overview of using UM in Apama applications.....	226
Comparison of Apama channels and UM channels.....	227
Choosing when to use UM channels and when to use Apama channels.....	228
General steps for using UM in Apama applications.....	229
About events transported by UM.....	229
Using UM channels instead of engine_connect.....	230
Using UM channels instead of configuring adapter connections.....	230
Enabling automatic creation of UM channels.....	230
Considerations for using UM channels.....	232
Setting up UM for use by Apama.....	233
Starting correlators that use UM.....	235
Configuring adapters to use UM.....	235
EPL and UM channels.....	237
Defining UM properties for Apama applications.....	238
Monitoring Apama application use of UM.....	239
Developing Custom Adapters.....	240
Chapter 7: The Integration Adapter Framework.....	241
Overview.....	241
Architecture.....	242
The transport layer.....	244
The codec layer.....	244
The Semantic Mapper layer.....	245
Contents of the IAF.....	246
Chapter 8: Using the IAF.....	247
The IAF runtime.....	247

IAF library paths.....	247
IAF command line options.....	248
IAF log file status messages.....	249
IAF log file rotation.....	249
IAF Management – Managing a running adapter I.....	251
IAF Management options.....	252
IAF Management exit status.....	253
IAF Client – Managing a running adapter II.....	254
IAF Watch – Monitoring running adapter status.....	255
The IAF configuration file.....	255
Including other files.....	256
Transport and codec plug-in configuration.....	257
Event mappings configuration.....	259
Apama event correlator configuration.....	269
Configuring adapters to use UM.....	271
Logging configuration (optional).....	273
Java configuration (optional).....	273
IAF samples.....	274
Chapter 9: C/C++ Transport Plug-in Development.....	277
The C/C++ transport plug-in development specification.....	277
Transport functions to implement.....	278
Defining the transport function table.....	281
The transport constructor, destructor and info functions.....	284
Other transport definitions.....	286
Transport utilities.....	287
Communication with the codec layer.....	288
Transport Example.....	288
Getting started with transport layer plug-in development.....	289
Chapter 10: C/C++ Codec Plug-in Development.....	291
The C/C++ codec Plug-in Development Specification.....	291
Codec functions to implement.....	292
Codec encoder functions.....	293
Codec decoder functions.....	294
Defining the codec function tables.....	295
The codec function table.....	295
The codec encoder function table.....	296
The codec decoder function table.....	298
Registering the codec function tables.....	299
The codec constructor, destructor and info functions.....	301
Other codec definitions.....	302
Codec utilities.....	303
Communication with other layers.....	304
Working with normalized events.....	306
The AP_NormalisedEvent structure.....	307
The AP_NormalisedEventIterator structure.....	308
Transport Example.....	309
Getting Started with codec layer plug-in development.....	309
Chapter 11: C/C++ Plug-in Support APIs.....	311

Logging from plug-ins in C/C++.....	311
Using the latency framework.....	313
C/C++ timestamp.....	313
C/C++ timestamp set.....	313
C/C++ timestamp configuration object.....	313
C/C++ latency framework API.....	314
Chapter 12: Transport Plug-in Development in Java.....	317
The Transport Plug-in Development Specification for Java.....	317
Java transport functions to implement.....	318
Communication with the codec layer.....	321
Sending upstream messages received from a codec plug-in to a sink.....	321
Sending downstream messages received from a source on to a codec plug-in.....	322
Transport exceptions.....	323
Logging.....	324
Example.....	324
Getting started with Java transport layer plug-in development.....	324
Chapter 13: Java Codec Plug-in Development.....	326
The Codec Plug-in Development Specification for Java.....	326
Java codec functions to implement.....	327
Communication with other layers.....	331
Sending upstream messages received from the Semantic Mapper to a transport plug-in.....	331
Sending downstream messages received from a transport plug-in to the Semantic Mapper.....	332
Java codec exceptions.....	332
Semantic Mapper exceptions.....	334
Logging.....	334
Working with normalized events.....	334
The NormalisedEvent class.....	335
The NormalisedEventIterator class.....	336
Java Codec Example.....	337
Getting started with Java codec layer plug-in development.....	337
Chapter 14: Plug-in Support APIs for Java.....	339
Logging from plug-ins in Java.....	339
Using the latency framework.....	340
Java timestamp.....	341
Java timestamp set.....	341
Java timestamp configuration object.....	341
Java latency framework API.....	342
Chapter 15: Monitoring Adapter Status.....	344
IAFStatusManager.....	345
Application interface.....	345
Input events.....	346
Output events.....	347
Returning information from the getStatus method.....	348
Connections and other custom properties.....	351
Asynchronously notifying IAFStatusManager of connection changes.....	352
Mapping AdapterConnectionClosed and AdapterConnectionOpened events.....	354
StatusSupport.....	355
StatusSupport events.....	355

Chapter 16: Out of Band Connection Notifications.....	358
Mapping example.....	358
Ordering of out of band notifications.....	360
Chapter 17: The Event Payload.....	363
Creating a payload field.....	363
Accessing the payload in the correlator.....	364
Developing Custom Clients.....	365
Chapter 18: The Client Software Development Kits.....	366
Basic operations.....	366
The client software development kits.....	367
Chapter 19: The Client Software Development Kits for C++ and Java.....	369
The library classes.....	369
Main classes.....	369
Data classes.....	371
Event correlator interrogation and status.....	371
Additional functionality.....	372
Exceptions.....	374
Logging in C++.....	374
Logging in Java.....	374
Thread-safety.....	375
The complete definitions.....	375
The C++ header file.....	375
Chapter 20: Using the SDKs – C++ and Java Examples.....	395
A simple echo server in C++.....	395
The full example in C++.....	397
The Java example.....	401
Chapter 21: The C Client Software Development Kit.....	404
Using the C SDK.....	404
The complete C example.....	407
Chapter 22: The EngineClient API.....	411
The Java EngineClient API.....	411
The key elements.....	411
Overview of the EngineClientBean.....	411
Functionality of the EngineClientBean.....	412
Recommended usage.....	413
Logging.....	414
Inject operations.....	414
Delete operations.....	415
Inspect operations.....	416
Receive operations.....	416
Send operations.....	417
Watch operations.....	418
GenericComponentManagementBean.....	418
The .NET Engine Client library.....	419
Using the .NET client library.....	420
Java and .NET namespace/class mapping.....	421
Chapter 23: The EventService API.....	425

The key elements.....	425
The IEventService interface.....	425
The IEventServiceChannel interface.....	426
The EventServiceFactory class.....	428
Examples of use.....	428
Chapter 24: The ScenarioService API.....	432
The key elements.....	432
The IScenarioService interface.....	433
The ScenarioDefinition interface.....	434
The IScenarioInstance interface.....	436
The ScenarioServiceFactory class.....	438
The ScenarioServiceConfig class.....	439
Examples of use.....	440

Preface

■ Documentation roadmap	12
■ Contacting customer support	14

You can connect Apama applications to any event data source, database, messaging infrastructure, or application. The general alternatives for doing this are as follows:

- Implement standard Apama Integration Adapter Framework (IAF) adapters.
- Create applications that use correlator-integrated messaging for JMS or Software AG's Universal Messaging
- Develop adapters with Apama APIs for Java and C++.
- Develop client applications with Apama APIs for Java, .NET, and C++.

Connecting Apama Applications to External Components provides information and instructions for

- ["Using Standard Adapters" on page 15](#)
- ["Using Message Services" on page 149](#)
- ["Developing Custom Adapters" on page 240](#)
- ["Developing Custom Clients" on page 365](#)

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 5.3 > 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	Description
<i>Release Notes</i>	Describes new features and changes since the previous release.

Title	Description
<i>Installing Apama</i>	Instructions for installing the Developer, Server, or User Apama installation options.
<i>Introduction to Apama</i>	Introduction to developing Apama applications, discussions of Apama architecture and concepts, and pointers to sources of information outside the documentation set.
<i>Using the Apama Studio Development Environment</i>	Instructions for using Apama Studio to create and test Apama projects, develop EPL programs, define Apama queries, develop JMon programs, and store, retrieve and play back data.
<i>Developing Apama Applications</i>	Describes the different technologies for developing applications: EPL monitors, Apama queries, Event Modeler, and Java. You can use one or several of these technologies to implement a single Apama application. In addition, there are C++, C, and Java APIs for developing components that plug-in to a correlator. You can use these components from EPL.
<i>Connecting Apama Applications to External Components</i>	<p>Describes how to connect Apama applications to any event data source, database, messaging infrastructure, or application. The general alternatives for doing this are as follows:</p> <ul style="list-style-type: none"> • Implement standard Apama Integration Adapter Framework (IAF) adapters. • Create applications that use correlator-integrated messaging for JMS or Software AG's Universal Messaging • Develop adapters with Apama APIs for Java and C++. • Develop client applications with Apama APIs for Java, .NET, and C++.
<i>Building and Using Dashboards</i>	Describes how to build and use an Apama dashboard, which provides the ability to view and interact with scenarios and DataViews. An Apama project typically uses one or more dashboards, which are created in the Dashboard Builder. The Dashboard Viewer provides the ability to use dashboards created in Dashboard Builder. Dashboards can also be deployed as simple Web pages, applets, or WebStart applications. Deployed dashboards connect to one or more correlators by means of a Dashboard Data Server or Display Server.
<i>Deploying and Managing Apama Applications</i>	<p>Describes how to use the Management & Monitoring console to configure, start, stop, and monitor the correlator and adapters across multiple hosts. Also provides information for:</p> <ul style="list-style-type: none"> • Improving Apama application performance by using multiple correlators and saving and reusing a snapshot of a correlator's state.

Title	Description
	<ul style="list-style-type: none"> Managing and monitoring over REST (REpresentational State Transfer). Using correlator utilities.
<i>Using the Dashboard Viewer</i>	In a User installation of Apama, this document describes how to view and interact with dashboards that are receiving run-time data from the correlator. In the Developer and Server installations, this information is included in <i>Building and Using Dashboards</i> .

[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](#)

I Using Standard Adapters

Software adapters provide the interface between Apama applications and middleware buses and other message sources. Apama provides a number of standard adapters.

- With the Apama Database Connector (ADB), Apama applications can store and retrieve data in standard database formats. You can retrieve data using the ADBHelper API or the ADB Event API to execute general database queries or retrieve the data for playback purposes using the Apama Data Player.
- The Apama Web Services Client adapter is a SOAP-based adapter that allows Apama applications to invoke Web services.
- The Apama File adapter uses the Apama Integration Adapter Framework (IAF) to read information from text files and write information to text files by means of Apama events. This lets you read files line-by-line from external applications or write formatted data as required by external applications.
- The File Transport, String Codec and Null Codec plug-ins are useful examples of how event transports and codecs can be written. These plug-ins are provided for your convenience as an aid to testing and the development of custom adapters that make use of them 'as is'. They are available in both C and Java.

Chapter 1: Using the Apama Database Connector

■ Overview of using ADBC	16
■ Registering your ODBC database DSN on Windows	18
■ Adding an ADBC adapter to an Apama Studio project	18
■ Configuring the Apama database connector	20
■ The ADBCHelper Application Programming Interface	27
■ The ADBC Event Application Programming Interface	37
■ The Visual Event Mapper	55
■ Playback	60
■ Sample applications	61
■ Format of events in .sim files	61

The Apama Database Connector (ADBC) is an Apama adapter that uses the Apama Integration Adapter Framework (IAF) and connects to standard ODBC and JDBC data sources as well as to Apama Sim data sources. With the ADBC adapter, Apama applications can store and retrieve data in standard database formats as well as read data from Apama Sim files. Data can be retrieved using the ADBCHelper API or the ADBC Event API to execute general database queries or retrieved for playback purposes using the Apama Data Player.

There are three versions of the ADBC adapter, one each for ODBC, JDBC, and Sim data sources.

For information about playing back data, see "Using the Data Player" in *Using the Apama Studio Development Environment*, which is available if you chose Developer during installation.

Using Standard Adapters

Overview of using ADBC

ADBC is implemented as an Apama adapter that uses the Apama Integration Adapter Framework (IAF) to connect to standard ODBC and JDBC data sources as well as to Apama Sim data sources.

When connected to JDBC or ODBC data sources, ADBC provides access to most open source and commercial SQL databases. With either of these data sources, Apama applications can capture events flowing through the correlator and play them back at a later time. In addition to storing and retrieving event data, Apama applications can store non-event data and execute queries against the data. Dashboards in Apama applications can directly access JDBC and ODBC database data.

An Apama Sim data source is a file with data stored in a comma-delimited format with a .sim file extension. Apama release 4.1 and earlier captured streaming data to files in this format. The Apama ADBC adapter can read .sim files but it does not store data in that format. For information on the format of .sim files, see ["Format of events in .sim files" on page 61](#).

Apama provides ODBC and JDBC database drivers for the following Apama-certified databases (note, the ODBC drivers are available only for Windows platforms):

- DB2
- Microsoft SQL Server
- Oracle

Using the Apama database drivers eliminates the need to install vendor-supplied drivers. In addition, they are pre-configured so when you select an Apama database driver in an Apama Studio project, the adapter instance is automatically configured with appropriate JDBC settings.

The Apama ODBC database drivers are licensed to be used only with the Apama ADBC adapter. The Apama JDBC drivers can be used with any Apama component.

For more information on the supplied database drivers, see the documentation available at `apama_install_dir\doc\db_drivers\jdbc` and `apama_install_dir\doc\db_drivers\odbc`.

Apama provides two Application Programming Interfaces (APIs) for using the ADBC Connector: the ADBCHelper API and the ADBC Event API.

The ADBCHelper API contains the basic features you need for most common use cases, such as opening and closing databases and executing SQL commands and queries. For more information on the ADBCHelper API, see ["The ADBCHelper Application Programming Interface" on page 27](#).

The ADBC Event API contains features for more complex use cases. For example, in addition to opening and closing databases, it contains actions for discovering what data sources and databases are available. For more information on the ADBC Event API, see ["The ADBC Event Application Programming Interface" on page 37](#).

The ADBC Adapter editor in Apama Studio includes an Event Mapping tab that lets you quickly specify the mapping rules for storing events in existing database tables. Apama Studio generates a service monitor that listens for the events of interest and stores them in the database. This monitor provides a quick and straight forward way of writing event data to a database for general analytical purposes; however, it is not meant to be a fail-safe management system.

The ADBC adapter uses separate thread pools for executing queries and commands and will execute each command and query in its own thread. The thread pools are created with a minimum of four threads but for machines with more than four CPU cores the number of threads will match the number of cores. The adapter log will show the number of threads in the thread pools, for example:

```
Query and Command threadpools using 4 threads
```

The maximum number of concurrent queries running will match the number of threads in the thread pool. As an example, on a machine with less than four cores, this would be four concurrent queries and four concurrent commands.

Additional queries and commands submitted will be queued for execution until a thread becomes free. If more than four long running queries are submitted, additional queries will be queued. If a mix of short and known long running queries are being used, the application may want to control the submission of long running queries to ensure the shorter duration queries do not have to wait. If the execution of the short duration queries are required to be run without delay, a second adapter can also be started and used to service just the shorter duration queries.

[Using the Apama Database Connector](#)

Registering your ODBC database DSN on Windows

On Windows it is necessary to register your database and give your database configuration a unique Data Source Name (DSN) before using it from Apama:

1. From the Windows Control Panel, double-click on the ODBC Data Sources icon. If this icon is not listed, double-click on the Administrative Tools icon and then double-click on the Data Sources (ODBC) icon.

This will open the **ODBC Data Source** dialog.

2. On the **User DSN** tab, click Add.
3. In the **Create New Data Source** window, select the driver for which you want to setup a data source.
4. Click Finish to display the **Setup** dialog.
5. Enter a Data Source Name. This is the name you will use in Apama when creating data attachments.
6. Click OK in the **Setup**, and **ODBC Data Source** dialogs.

Note: Standard UNIX systems do not provide an ODBC driver. On UNIX systems, it is currently unsupported to set up an ODBC driver to communicate with your database.

Using the Apama Database Connector

Adding an ADBC adapter to an Apama Studio project

When you add an ADBC adapter to an Apama Studio project, Studio automatically includes all the resources associated with the adapter such as service monitors and configuration files.

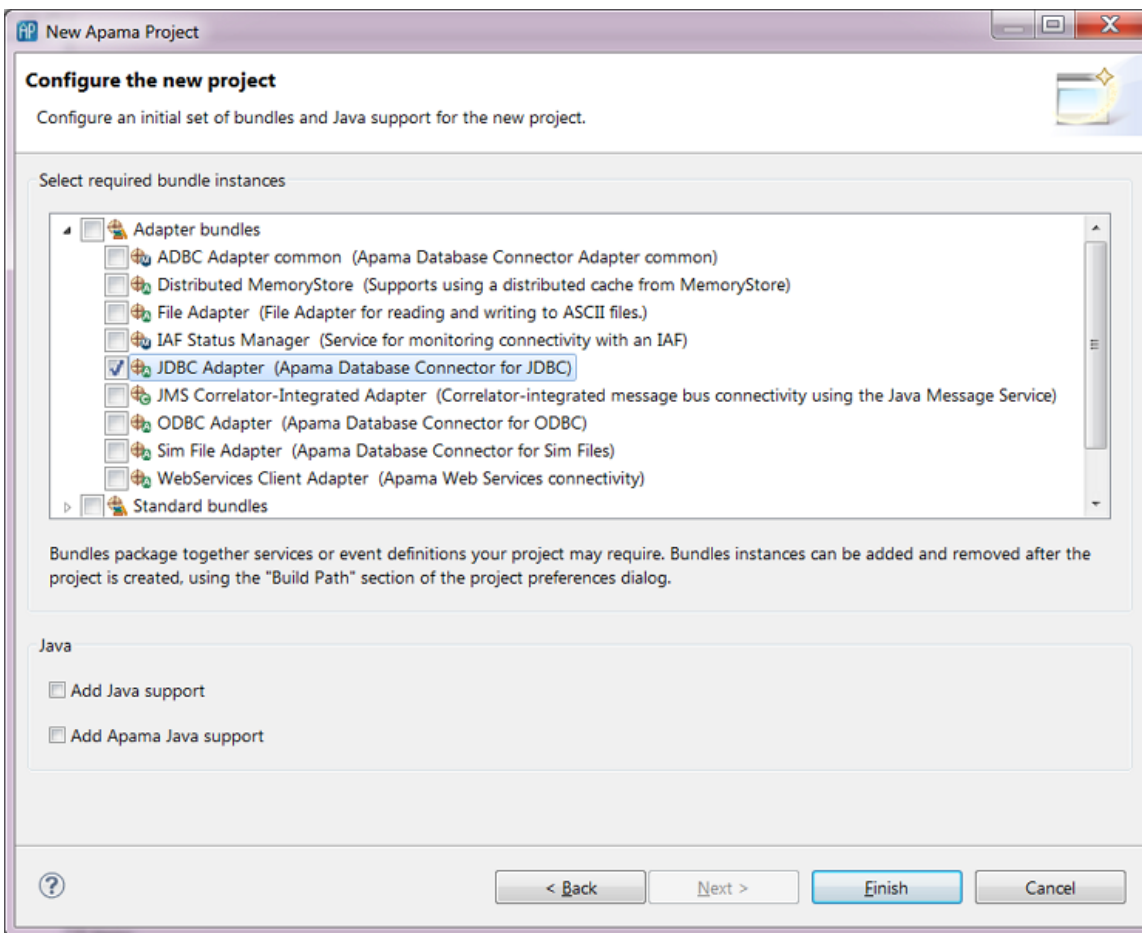
ADBC Adapters are available for three different data sources:

- JDBC Adapter (Apama Database Connector for JDBC)
- ODBC Adapter (Apama Database Connector for ODBC)
- Sim File Adapter for (Apama Database Connector for Sim Files)

To add an adapter to a project:

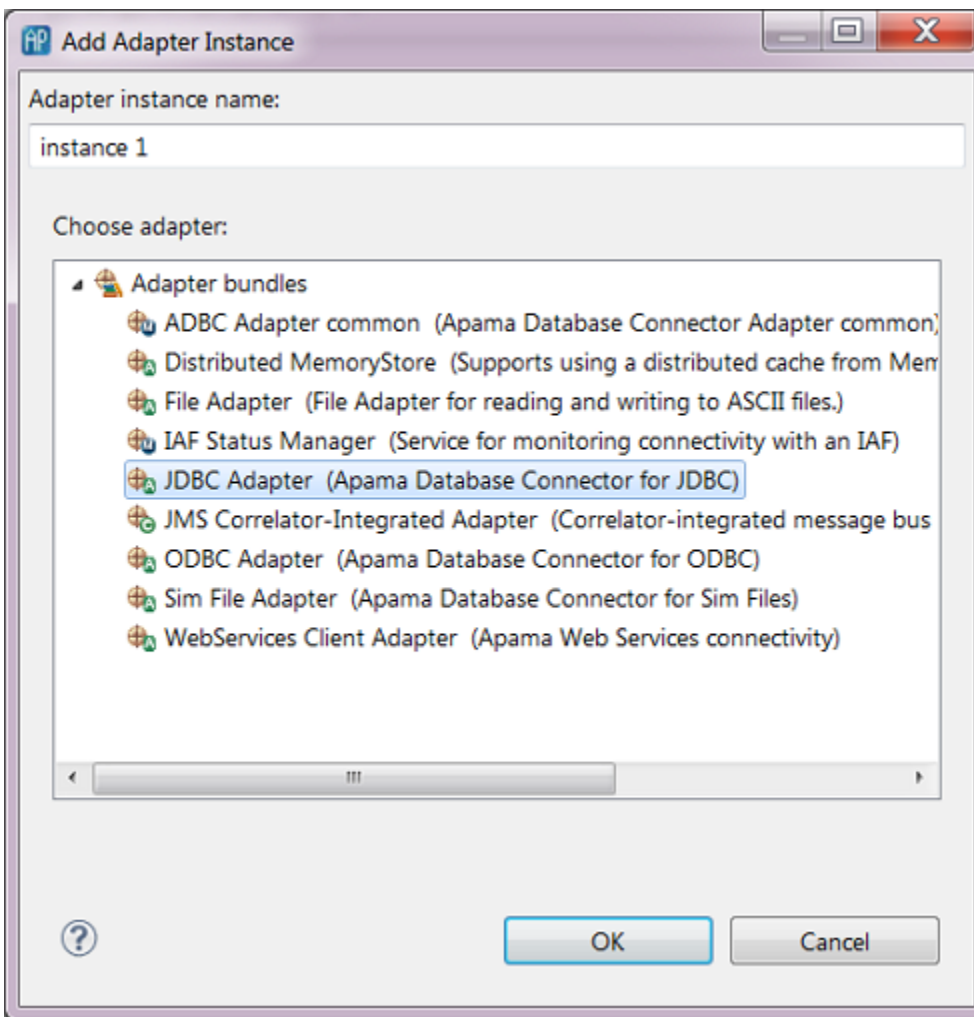
1. There are two ways of adding an ADBC adapter to a project.

If you are creating a new Apama project, select File > New > Apama Project, give it a name, and click Next.



If you are adding an ADBC adapter to an existing project:

- a. In the Project Explorer right-click the project and select Apama > Add Adapter. The **Add Adapter Instance** dialog opens.
- b. If desired, in the **Add Adapter Instance** dialog, create a new name for the adapter instance or accept the default instance name. Apama Studio prevents you from using a name that is already in use.



2. In the **New Apama Project** dialog or the **Add Adapter Instance** dialog, select the ADBC adapter bundle that is appropriate to the kind of data source your application will use. Click Finish or OK.

When you add a data source-specific adapter, the ADBC Adapter common (Apama Database Connector Adapter common) bundle will be added to the project automatically.

[Using the Apama Database Connector](#)

Configuring the Apama database connector

The Apama Database Connector is an adapter that is instantiated with the Apama Integration Adapter Framework (IAF). The IAF enables Apama applications to connect to sources of messages and events and to consumers of messages and events; with ADBC, these sources and consumers can be databases. Before using the ADBC adapter, you need to supply the correct information in the adapter's configuration file.

If you develop your Apama application using Apama Studio, the correct configuration files are included in the application's project file when you add the appropriate ADBC adapter bundle to the project. In order to connect to a database, you need to specify in the adapter's configuration

file the properties such as the type and name of data source and the name of the database that the application will use.

If you are not using Apama Studio, you need to manually create the configuration file from the ADBC adapter template file. For more information on creating the configuration file manually, see ["Manually editing a configuration file" on page 24](#).

Using the Apama Database Connector

Configuring an ADBC adapter

Apama Studio opens an adapter's configuration file in the adapter editor. By default the file is displayed in the editor's graphical view, which is accessed by clicking the Settings tab. The editor's other tabs are:

- **Event Mapping** - Displays the Visual Event Mapper where you can quickly map Apama event fields to columns in a database table.
- **XML Source** - Displays the configuration file's raw XML code.
- **Advanced** - Provides access to other configuration files associated with the adapter instance. These other files specify, for example, the instance's mapping rules, generated monitors and events responsible for storing events in a database, and named queries.

To configure an instance of an ADBC adapter:

1. In the **Project Explorer**, expand the project's **Adapters** node and open the adapter folder (either **Adapter for ODBC**, **Adapter for JDBC**, or **Adapter for Sim**).
2. Double-click the entry for the adapter instance you want to configure. The configuration file opens in the adapter editor. For example, a configuration file for an instance of the ADBC-JDBC adapter looks like this:

Name	Value
JDBC_DRIVER_JARFILE	
ADAPTER_INSTANCE_ID	INSTANCE_1
ADAPTERS_JAR_DIR	\$(apama_home)/adapters/lib
DATABASE_LOCATION	
JDBC_DRIVER_NAME	
PROJECT_DIR	\$(PROJECT_DIR)
ADAPTER_CONFIG_DIR	\$(Adapter for JDBC.configDir)
MAPPING_INSTANCE_FILE	ADBC-mapping_instance_1.xml
CORRELATOR_HOST	\$(Default Correlator.hostname)
CORRELATOR_PORT	\$(Default Correlator.port)

The **Settings** tab of the editor's graphical display presents configuration information in three separate sections:

- **General Properties**
- **Advanced Properties**
- **Variables**

For an instance of the ADBC-ODBC adapter, the display is similar but with fewer items in the **General Properties** and **Variables** section. For an instance of the ADBC-Sim adapter, the display only shows the **Variables** section.

3. In the **General Properties** section, add or edit the following:

- **Database Type** — This drop-down list allows you to select one of the database types from the list of certified vendors.
- **Database URL** — This specifies the complete URL of the database. By default it uses the value of the `DATABASE_LOCATION` variable; for more information on this variable see Step 5.
- **Driver** — For the ADBC-JDBC adapter, this specifies the class name of the vendor's JDBC driver. By default it uses the value of the `JDBC_DRIVER` variable as described in Step 5.
- **Driver Classpath** — For the ADBC-JDBC adapter, this specifies the classpath for the vendor's driver jar file. By default it uses the value of the `JDBC_DRIVER_JARFILE` variable as described in Step 5.
- **AutoCommit** — This controls the use of the ODBC/JDBC driver autocommit flag. The default value is `false`.
- **StoreBatchSize** — This defines the number of events (rows) to persist using the ODBC/JDBC batch insert API. The use of this setting will significantly increase store performance, but it is not supported by all drivers. A value of 100 is appropriate and will provide good performance in most cases.

If store performance is critical, testing is required to find the optimal value for the data and driver being used. The default is 0 which disables the use of batch inserts.

- **StoreCommitInterval** — This defines the interval in seconds before the ADBC adapter will automatically perform a commit for any uncommitted SQL command or store operations. The default value is 0.0 which disables the use of the timed commits.

4. In the **Advanced Properties** section, add or edit information for the following:

- **Transaction Isolation Level** — This specifies what data is visible to statements within a transaction. The `Default` level uses the default level defined by the database server vendor. To change this setting, enter the appropriate value. For JDBC and ODBC the values can be `READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, or `SERIALIZABLE`.
- **Alternate Discovery Query** — In most situations an entry here is not required and the ADBC `Discovery` method lists the database available based on the `DATABASE_LOCATION` variable. In some cases you may need to use a server vendor-specific SQL query statement to list the available databases, such as MySQL's `SHOW DATABASES`.
- **Log Inbound Events** — A boolean that specifies whether or not the application logs inbound ADBC API events with information such as the exact query or command being executed. Logging these events is used for diagnostic purposes and eliminates the need to turn on IAF debug logging. The default is `false`; do not log incoming events.

- `Log Outbound Events` — The same as `Log Inbound Events` except for outbound ADBC API events.
- `LogCommands` — This property specifies whether or not the starts and completions of commands are written to the IAF log file. A value of "true" (the default) logs this information; a value of "false" turns logging off. This is useful in cases where logging the start and completion of a high rate of commands (many hundreds or thousands per second) does not add usable information to the log file.
- `LogQueries` — This property behaves identically to the `LogCommands` property except that it specifies whether or not to log the start and completion of queries.
- `FlowControlHighWater` — This defines a maximum threshold for the number of query responses that have not been acknowledged by the ADBC flow control monitor. If this value is reached, the query will be paused until the number of outstanding acknowledgments decreases to the `FlowControlLowWater` value. This is used by the ADBC query flow control system to ensure the correlator does not get overwhelmed, especially when performing a fast as possible playback. The default value is 15000.
- `FlowControlLowWater` — This defines a threshold for the number of query responses not acknowledged by the ADBC flow control monitor before a query paused by `FlowControlHighWater` is resumed. This is used by the ADBC query flow control system to ensure the correlator does not get overwhelmed, especially when performing a fast as possible playback. The default value is 6000.
- `Query Template Config File` — This specifies the file containing the query templates that are available to the application. By default, this uses a default template file created for the individual Apama Studio project.

You can add or edit values of the following additional advanced properties by clicking the XML tab and modifying the text of the configuration file:

- `NumericSeparatorLocale` — This allows the numeric separator used in the adapter to be changed, if necessary, to match the one used by the correlator. See ["Configuring ADBC localization" on page 25](#).
- `CloseDatabaseIfDisconnected` — This controls automatic closing of databases whose connection is found to be invalid. See ["Configuring ADBC Automatic Database Close" on page 25](#).
- `FixedSizeInsertBuffers` — This is an ODBC-specific property that allows you to change the default buffer size used when the `StoreData` and `StoreEvent` actions perform batch inserts. Apama uses the `FixedSizeInsertBuffers` property along with the `StoreBatchSize` property to determine how large the insert buffers should be. The value specified by `StoreBatchSize` determines how many rows need to be buffered; the value specified by `FixedSizeInsertBuffers` controls the size of the buffers for the columns. The default "true" uses a fixed buffer size of 10K bytes for each column. If the value is changed to "false", the size of the column buffers is determined dynamically by examining the database table into which the data will be inserted. Allowing the buffer size to be set dynamically can significantly reduce memory usage when performing batch inserts to database tables that contain hundreds of columns or when using a very large `StoreBatchSize`.

5. In the **Variables** section, add or edit the appropriate values for the following tokens.

- `ADAPTERS_JARDIR` — For the ADBC-JDBC adapter this specifies the directory where the Apama adapter jar files are located. By default this is the Apama installation's `adapters\lib` directory.

- `DATABASE_LOCATION` — The location of the database for use with the ADBC Discovery API, for example, `jdbc:mysql://localhost/trades`
 - `PROJECT_DIR` — This specifies the location of the Apama project. By default this is automatically set by Apama Studio.
 - `ADAPTER_CONFIG_DIR` — This specifies the location of the adapter's configuration files. By default this is automatically set by Apama Studio.
 - `CORRELATOR_PORT` — This specifies the port used by the correlator. By default this is automatically set by Apama Studio.
 - `JDBC_DRIVER_NAME` — The class name of the driver, such as `com.mysql.jdbc.Driver`
 - `JDBC_DRIVER_JARFILE` — The name of the data source driver file, for example, `C:/Program Files/MySQL/mysql-connector-java-5.1.7/mysql-connector-java-5.1.7-bin.jar`
 - `CORRELATOR_HOST` — This specifies the name of the host machine where the project's default correlator runs. By default this is automatically set by Apama Studio.
6. Specify the event mapping rules of the configuration that are specific to your application using the adapter editor's Visual Event Mapper, available on the Event Mapping tab. For more information on specifying mapping rules, see ["The Visual Event Mapper" on page 55](#).

Configuring the Apama database connector

Manually editing a configuration file

If you are not using Apama Studio to develop your application, you need to manually copy the correct template files to your development environment. The Apama installation provides template files to use as the basis for creating the IAF configuration file to start the ADBC adapter. The templates are located in the `adapters\config` directory of the Apama installation. The following templates are available:

- `ADBC-Sim.xml.dist` — Use this configuration file template for accessing a Sim data source.
- `ADBC-ODBC.xml.dist` — Use this configuration file template for accessing an ODBC data source.
- `ADBC-JDBC.xml.dist` — Use this configuration file template for accessing a JDBC data source.

To create the configuration file for starting the ADBC adapter:

1. Copy the appropriate template to your project.
2. Edit the name attributes of the various transport properties as necessary.

When you start the IAF with the modified configuration file using the syntax `iaf path_to_modified_config_file`, it automatically includes the appropriate common configuration files shown below.

- `ADBC-static.xml` — Common event mapping for the ADI adapter events.
- `ADBC-static-codecs.xml` — The codecs to use (currently null-codec).
- `ADBC-application.xml` — Application specific event mappings.
- `ADBC-namedQuery-Sim.xml` — The named query definitions for a Sim data source.

or

- `ADBC-namedQuery-SQL.xml` — The named query definitions for ODBC and JDBC data sources.

- `ADBC-mapping_instance_name.xml` — Contains the mappings defined by the user using the Visual Event Mapper.

Configuring an ADBC adapter

Configuring ADBC localization

The ADBC adapter internally handles all string data as UTF-8, and provides the same internationalization support as the correlator. The correlator internally uses the C programming language locale for formatting string versions of numeric values, so there can be conditions under which the ODBC and JDBC drivers may use a locale that is not compatible with the English numeric separator format used in ADBC. In locales that do not use English numeric separators, the ODBC and JDBC drivers for some SQL vendors may not correctly handle numeric values passed from the correlator. To address these cases, the ADBC adapter configuration property `NumericSeparatorLocale` allows the numeric separator used in the adapter to be changed to match the one used by the correlator. The property can be set to one of three values:

- `""` (empty string): Default. Don't change/set separator format.
- `c`: Set numeric separator format to English.
- `Native`: Set numeric separator format to system default.

A value of `c` causes the adapter's numeric separator locale to match that used by the correlator, so that the JDBC and ODBC drivers correctly handle the numeric values. The value `Native` causes the adapter to set the locale to the system default. This value is not generally needed and was added for future use and for special cases in which technical support would direct it to be set. If you notice incorrect numeric values when inserting or querying data from the database when running in a locale that doesn't use the English-style numeric separators, then changing the `NumericSeparatorLocale` property to `c` should correct the problem. In Studio, you can access this property by using the XML tab in the ADBC configuration file editor.

Configuring an ADBC adapter

Configuring ADBC Automatic Database Close

The ADBC adapter performs a connectivity check when a JDBC or ODBC error is encountered, and can be configured to automatically perform the database close operation if a connection is found to be invalid. The IAF status manager will detect the database connection has been closed and report the change in connection status. Applications need to monitor the database connection status in order to take advantage of the automatic closing; this functionality is not integrated into the ADBC APIs.

The ADBC adapter configuration property `CloseDatabaseIfDisconnected` is used to enable the closing of databases that are detected as invalid.

- `False`: Default. Don't perform automatic closing.
- `True`: Close databases detected as invalid (that is, disconnected) .

Configuring an ADBC adapter

Service monitors

If your Apama application uses ADBC, you need to inject several required service monitors. In Apama Studio this is done automatically when you add the appropriate data source adapter bundle to the application's project as described in ["Configuring the Apama database connector" on page](#)

20. If you are not using Apama Studio to develop your application, you need to manually inject the following required service monitors in the order they are listed:

- `ADBCAdapterEvents.mon` — Provides definitions for all events sent to or from the ADBC Adapter.
- `ADBCEvents.mon` — Provides the public API for ADBC, implemented as actions on the following events:
 - `Discovery` — This event type defines the actions for discovering ADBC resources. It is used to find the available data sources (ODBC, JDBC, Sim, etc.) and the default databases and query templates configured for those data sources.
 - `Connection` — This event type defines actions for performing all operations on a database except those involving queries
 - `Query` — This event type defines actions for performing queries on a database.
- `ADBCAdapterService.mon` — Provides actions for the following:
 - Forwarding database request events to the adapter.
 - Forwarding database response events to the ADBC Service API layer.
 - Supporting parallel execution of blocks and event actions.
- `IAFStatusManager.mon`
- `StatusSupport.mon`
- `ADBCStatusManager.mon` — Manages status subscriptions for the ADBC adapter and the application.
- `ADBCHelper.mon` — Include this monitor for applications that use the ADBCHelper API.

The ADBC monitors and `IAFStatusManager.mon` are located in the `adapters\monitors` directory of the Apama installation. The `StatusSupport.mon` monitor is located in the Apama installation's `monitors` directory

Configuring the Apama database connector

ADBC blocks

To facilitate the use of ADBC with scenarios, Apama's Event Modeler includes standard blocks for storing and retrieving data. For more information on these blocks, see "ADBC Storage v10" and "ADBC Retrieval v10" in "Developing Apama Applications in Event Modeler", which is in *Developing Apama Applications*.

Configuring the Apama database connector

Codecs

By default, the ADBC adapter uses the standard Apama `NullCodec`. During playback, if your application needs to modify, aggregate or perform analytics on events, you can create and specify IAF codecs to perform these operations instead of using the standard `NullCodec`. For example, capital market applications might convert quote to depth events during playback from a market database. You define the logic for performing this type of conversion in the codec.

For more information on developing codecs, see "[C/C++ Codec Plug-in Development](#)" on page 291 and "[Java Codec Plug-in Development](#)" on page 326.

Configuring the Apama database connector

The ADBCHelper Application Programming Interface

The ADBCHelper Application Programming Interface (API) is a simplified, streamlined API for communicating with databases. In most common use cases, this API is the appropriate way to develop applications. For applications that require more complex ways of accessing databases, see ["The ADBC Event Application Programming Interface" on page 37](#).

Using the Apama Database Connector

ADBCHelper API Overview

The ADBCHelper API is defined in the file `apama_dir\adapters\monitors\ADBCHelper.mon`. The API is implemented with the following events:

- `com.apama.database.DBUtil`
- `com.apama.database.DBAcknowledge`

The `DBUtil` event defines the actions that Apama applications call in order to interact with databases. The `DBAcknowledge` event is used by the ADBCHelper API to specify the success or failure for database actions that request an acknowledgement. Note if you specify the following lines in your code, you do not need to use the fully qualified name for `DBUtil` or `DBAcknowledge`.

```
using com.apama.database.DBUtil;  
using com.apama.database.DBAcknowledge;
```

The basic steps for using the ADBCHelper API are:

1. Create an instance for the `DBUtil` event in your application code, for example:

```
com.apama.database.DBUtil db;
```

2. Call the `DBUtil setAdapterInstanceName` action to identify the adapter instance. This step is only required if the adapter instance name is not the default, `INSTANCE_1`. This action is necessary, for example, if the project uses multiple adapter instances.

For more information, see ["Specifying the adapter instance" on page 30](#).

3. Check whether the database is already open or is in the process of being opened. This step is optional, but it is good programming practice to check for these situations before calling an open event action by calling the `DBUtil isOpen` action. This returns a boolean that specifies if the database is already open or in the process of being opened.

For more information, see ["Checking to see if a database is open" on page 31](#).

4. Call one of the `DBUtil` open actions to open the database.

For more information on open actions, see ["Opening databases" on page 28](#).

5. Call one or more `DBUtil` event actions, depending on the database task you want to implement:
 - Call a SQL query event action to retrieve data from the database, in either a result set or in Apama event format.

For more information on query actions, see ["Issuing and stopping SQL queries" on page 31](#).

- Call a SQL command event action to add, update, or delete data in the database.

For more information on SQL command actions,, see ["Issuing SQL commands" on page 32](#).

- Optionally, if the `autoCommit` setting has been turned off, call a commit event action to commit database changes, or call a rollback event action to rollback uncommitted changes.

For more information on commit actions see ["Committing database changes" on page 32](#).

For more information on rollback actions, see ["Performing rollback operations" on page 32](#).

6. Create actions as required to handle returned result sets. If the query returns events, create listeners for events returned by the query.

For more information on handling query results, see ["Handling query results for row data" on page 33](#).

7. For action calls that request an acknowledgement, your application needs to do the following:

- a. Create an instance of the `com.apama.database.DBAcknowledge` event.

Note, if your code contains the line `using com.apama.database.DBAcknowledge;`, you do not need to use the fully qualified name for `DBAcknowledge`.

- b. Create a listener for the `DBAcknowledge` event that indicates when the `DBUtil` event action call is complete.

For more information on handling acknowledgments, see ["Handling acknowledgments" on page 33](#).

8. Create an action that handles errors that could occur during execution of a `DBUtil` event action call.

For more information on handling errors, see ["Handling errors" on page 34](#).

9. Call the `DBUtil` event's close action to close the database.

For information on closing databases, see ["Closing databases" on page 35](#).

The ADBCHelper Application Programming Interface

Opening databases

The ADBCHelper API provides several actions for opening databases. The "quick" open actions allow you to open JDBC and ODBC databases by passing in a minimal set of parameters, while the "full" open action provides more control by passing in a complete set of parameters. The "shared" open action allows you to use an already open existing matching connection or open a new connection if a matching one does not exist..

In the following quick open actions, you need to pass in values for the following parameters:

- `URL` — database connection string
- `user` — user name
- `password` — user password
- `handleError` — name of a default error handler

See ["Handling errors" on page 34](#) for more information on creating actions to handle errors.

The quick open actions use the default settings for the `autoCommit` (`true`), `batchSize` (`100`), and `timeOut` (`30.0`) properties.

```
action openQuickJDBC(  
    string URL,  
    string user,  
    string password,  
    action < string > handleError )  
action openQuickODBC(  
    string URL,  
    string user,  
    string password,  
    action < string > handleError )
```

The following code snippet shows a use of the `openQuickJDBC` action.

```
com.apama.database.DBUtil db;  
action onload {  
    string dbUrl:= "jdbc:mysql://127.0.0.1:3306/exampledb";  
    string user := "thomas";  
    string password := "thomas-123";  
    db.openQuickJDBC(dbUrl, user, password, handleError );  
    // ...  
}
```

For the following open action you need to pass in all parameters.

```
action open(  
    string type,  
    string serviceId,  
    string URL,  
    string user,  
    string password,  
    string autoCommit,  
    boolean readOnly,  
    integer batchSize,  
    float timeOut,  
    action < string > handleError )
```

Setting the `autoCommit`, `batchSize`, and `timeOut` parameters in the `open` action over-rides the adapter properties specified in the IAF configuration file.

- `type` — The data source type (ODBC, JDBC, Sim, etc.)
- `serviceId` — The service id for the adapter
- `URL` — The database connection string
- `user` — The user name
- `password` — The user password
- `autoCommit` — The auto commit mode to use. If this parameter is not set, the `open` action uses a combination of the `AutoCommit` and `StoreCommitInterval` properties specified in the adapter's configuration file. For information on these properties, see ["Configuring an ADBC adapter" on page 21](#). The value for the `autoCommit` parameter can be one of the following modes:
 - `""` — An empty string specifies that the value set in the configuration file should be used.
 - `true` — Enables the ODBC/JDBC driver's auto commit.
 - `false` — Disable `autoCommit`.
 - `x.x` — Use timed auto commit interval in seconds.
- `readOnly` — Specifies if the connection should be read-only. If the connection is read-only an error will be reported for any API action that requires writes (`Store`, `Commit`, or `Rollback`). Most databases

do not prevent writes from a connection in read-only mode so it is still possible to perform writes using the `Command` actions.

- `batchSize` — The query results batch size to be used for any queries performed.
- `timeOut` — Controls how long the ADBC `open` action will wait for the adapter to become available if it is not running when the `open` action is called.

The following code snippet shows a use of the `open` action. It creates variables for each of the parameters and passes them with the `openaction`.

```
com.apama.database.DBUtil db;
action onload {
    string type := "jdbc";
    string serviceId := "com.apama.adbc.JDBC";
    string dbUrl:= "jdbc:mysql://127.0.0.1:3306/exampledb";
    string user := "thomas";
    string password := "thomas-123";
    string commit := "15.0";
    boolean readMode := false;
    float openTimeout := 30.0;
    Integer queryBatchSize := 100
    db.open(type, serviceId, dbUrl, user, password, commit,
        readMode, openTimeout, queryBatchSize, handleError );
    // ...
}
```

The following `open` action allows you to use a connection that is already open; the action opens a connection if an existing matching connection is not found. The `openShared` action uses the same parameters as the `open` action, above.

```
action openShared(
    string type,
    string serviceId,
    string URL,
    string user,
    string password,
    string autoCommit,
    boolean readOnly,
    integer batchSize,
    float timeOut,
    action < string > errorHandler )
```

The ADBCHelper Application Programming Interface

Specifying the adapter instance

By default ADBCHelper actions use the default name given to the adapter instance, which is `INSTANCE_1`. If the adapter instance name is different from the default, for example if your Apama project has multiple adapter instances, you need to specify the name of the adapter instance you want to use. Use the DBUtil `setAdapterInstanceName` action to do this.

To specify an adapter instance:

Call the `setAdapterInstanceName` action, passing in the adapter instance name variable.

For example:

```
com.apama.database.DBUtil db;
action onload {
    string adapterInstanceName := "EXAMPLE_ADBC_INSTANCE";
    db.setAdapterInstanceName(adapterInstanceName);
    // ...
}
```

The ADBCHelper Application Programming Interface

Checking to see if a database is open

Checking to see whether the database is already open or is in the process of being opened before calling an open event action is optional, but it is good programming practice. An application may also want to check if a database is open before executing a query.

The following example checks these fields to ensure that the application does not try to open an already opened database.

```
com.apama.database.DBUtil db;
//...
if not db.isOpen then {
    db.openQuickODBC(dbUrl,"","","handleError );
}
```

The ADBCHelper Application Programming Interface

Issuing and stopping SQL queries

The following actions execute SQL queries. The actions expect a response and a `handleResult` action needs to be defined to handle each row returned.

```
action doSQLQuery(
    string queryString,
    action< dictionary< string, string > > handleResult )
action doSQLQueryOnError(
    string queryString,
    action< dictionary< string, string > > handleResult)
action doSQLQueryAck(
    string queryString,
    action < dictionary< string, string > > handleResult,
    integer ackNum,
    boolean onError )
```

The following query action allows you to specify a callback action for when the query completes. The parameters are (1) the query string, (2) the handler action for each row returned and (3) the handler for when the query completes. The handler for when the query completes has two parameters, an error string and an integer that specifies the number of rows returned by the query.

```
action doSQLQueryWithCallback(
    string queryString,
    action< dictionary< string, string > > handleResult,
    action < string, integer> handleDone )
```

The following actions are similar to the above query actions except they return Apama events instead of results sets.

```
action doSQLEventQuery(
    string queryString,
    string eventType )
action doSQLEventQueryWithCallback(
    string queryString,
    string eventType,
    action < string, integer> handleDone )
action doSQLEventQueryOnError(
    string queryString,
    string eventType )
action doSQLEventQueryAck(
    string queryString,
    string eventType,
    integer ackNum,
    boolean onError )
```

The following action cancels all outstanding queries in the queue.

```
action stopAll()
```

For more information on creating a `handleResult` action, see ["Handling query results for row data" on page 33](#).

The ADBCHelper Application Programming Interface

Issuing SQL commands

The following actions execute SQL commands and expect no responses.

```
action doSQLCmd( string queryString )
action doSQLCmdAck(
    string queryString,
    integer ackNum,
    boolean onError )
action doSQLCmdOnError( string queryString )
```

The `doSQLCmdOnError` action executes only if a previous non- `*OnError` operation failed. This is useful for doing, for example, a `select * from table` command and then, if an error occurs, execute a `create table ...` command.

The ADBCHelper Application Programming Interface

Committing database changes

The default auto-commit behavior is driver auto commit, assuming the `AutoCommit` and `StoreCommitInterval` properties specified in the adapter's configuration file and the open action are using the default values. If you want more control over when changes are committed to a database, set the open action's `autoCommit` parameter to false and in your EPL code, manually commit data using one of the following `DBUtil` event actions in your EPL code:

```
action doSQLCommit()
action doSQLCommitAck( integer ackNum )
```

The ADBCHelper Application Programming Interface

Performing rollback operations

For rolling back uncommitted changes to database, use the following `DBUtil` actions. If you want to use rollback actions, you need to turn autocommit off.

```
action doSQLRollback()
```

For rolling back uncommitted changes to database in situations where the previous `SQLCmd`, `SQLQuery`, or `SQLCommit` operation failed, use:

```
action doSQLRollbackOnError()
```

When you want to rollback uncommitted changes to the database and receive a `DBAcknowledge` event to indicate success or failure, use:

```
action doSQLRollbackAck( integer ackNum, boolean onError )
```

The `ackNum` parameter is the identifier for the `DBAcknowledge` event; setting it to -1 will disable sending the `DBAcknowledge` event and instead use the default error handler if an error occurs. For the `onError` parameter, setting its value to true will cause the operation to run only if the previous `SQLCmd`, `SQLQuery`, or `SQLCommit` failed.

The ADBCHelper Application Programming Interface

Handling query results for row data

For query actions that return a result set of rows of data, your application needs to define actions to handle result sets. For example:

```
com.apama.database.DBUtil db;
action onload {
    db.openQuickODBC("exampledb","thomas","thomas123",handleError);
    db.doSQLQuery("SELECT * FROM NetworkInformation", handleNetworkInfo);
    // ...
}
action handleNetworkInfo( dictionary< string, string > data ) {
    log "Network: " + data[ "network" ] + " CountryCode: " +
    data[ "countrycode" ] + " NIC: " +
    data[ "networkidentificationcode" ] at INFO;
}
```

The ADBCHelper Application Programming Interface

Handling query results for event data

For query actions that return a result set in the form of events, your application needs to do the following.

1. Define an event type that represents the returned data.
2. Map the returned data to fields in the event type. The easiest way to do this is to use the Apama Studio's Visual Mapper, which automatically saves the mapping information in a project file. For more information on using the Visual Mapper, see ["Using the Visual Event Mapper" on page 56](#).
3. Create a listener for the event type.
4. Execute a query that returns events.

For example, the following EPL code snippet defines an event, executes a query that returns data in the form of the defined event, and defines a listener for the defined event:

```
event NetworkInfo {
    string network;
    integer countrycode;
    integer nid;
}
//...
com.apama.database.DBUtil db;
action onload {
    db.openQuickODBC("exampledb","thomas","thomas123",handleError);
    db.doSQLEventQuery("SELECT * FROM network_info", NetworkInfo);
    //...
}
on all NetworkInfo := netInfo {
    // Code to do something with the returned event...
}
```

The ADBCHelper Application Programming Interface

Handling acknowledgments

Apama applications can call `DBUtil` SQL command and query actions as well as commit and rollback actions that request a `DBAcknowledgement` event. The `DBAcknowledgement` event indicates the success or failure of the action call. This is useful, for example, to know whether or not a query has completed before performing another application operation.

The `DBAcknowledge` event is defined in `apama_install_dir\adapters\monitors\ADBCHelper.mon` as follows:

```
event DBAcknowledge
{
    integer ackNum;
    boolean success;
    string error;
}
```

- `ackNum` — A unique identifier for the action that requested the acknowledgment.
- `success` — A value of true indicates success; false indicates failure.
- `error` — A string describing the specific error.

For action calls that request an acknowledgment, your application needs to do the following:

1. Call an action that requests an acknowledgment, passing in a unique acknowledgment identifier.
2. Create an instance of the `com.apama.database.DBAcknowledge` event.
3. Create a listener for the `DBAcknowledge` event that matches the acknowledgment identifier in the calling action.

For example:

```
integer ackId := integer.getUnique();
db.doSQLQueryAck("SELECT * FROM NetworkInformation", handleNetworkInfo,ackId,false);
com.apama.database.DBAcknowledge ack;
//...
on DBAcknowledge(ackNum = ackId) : ack {
    if ack.success then {
        log "Query complete" at INFO;
    }
    else {
        log "Query failed: " + ack.error at ERROR;
        die;
    }
}
//...
```

The ADBCHelper Application Programming Interface

Handling errors

The `DBUtil` actions require a user-defined `handleError` action that takes a single `string` parameter. The `handleError` action handles errors that could occur during execution of a `DBUtil` event action call.

The following EPL code snippet shows a simple error handler.

```
//...
com.apama.database.DBUtil db;
//...
action onload {
    db.openQuickODBC("exampleddb","thomas","thomas123", handleError);
    //...
}
action handleError( string reason ) {
    log "DB Error: " + reason;
}
```

The ADBCHelper Application Programming Interface

Reconnection settings

Apama applications can automatically reconnect if a disconnection error is encountered. The reconnection capability is optional and the default is to not reconnect when a disconnection error occurs. The following reconnection actions are defined in the `com.apama.database.DBUtil` event.

- `action setReconnectPolicy(string reconnectPolicy)` — This action sets the policy for dealing with adapter connection errors. The `reconnectPolicy` parameter must be one of the constants specified in the `DBReconnectPolicy` event. The policy constants are as follows:
 - `RECONNECT` — Try to reconnect and leave the management of pending requests to the client, which will handle the pending requests in the error handler.
 - `RECONNECT_AND_RETRY_LAST_REQUEST` — Try to reconnect and leave the pending requests unchanged, retry the last request on a successful database reconnection.
 - `RECONNECT_AND_CLEAR_REQUEST_QUEUE` — Try to reconnect and remove all the pending requests.
 - `DO_NOT_RECONNECT` — Do not try to reconnect.

The default reconnect policy is `DO_NOT_RECONNECT`.

- `action setReconnectTimeout(float timeOut)` — This action sets the timeout for the reconnection after a connection error. A value specified by the `setReconnectTimeout` action overrides the default timeout value, which is equal to twice as long as specified by the open action's `timeOut` parameter.

[The ADBCHelper Application Programming Interface](#)

Closing databases

The following action closes the database. If `doStopAll` is set it also cancels all outstanding queries and commands in the queue and prevents new queries and commands from being placed into the queue.

```
action close( boolean doStopAll )
```

[The ADBCHelper Application Programming Interface](#)

Getting schema information

The following actions return information about a table in a database. The actions are only valid in the `handleResult` action specified in a `doSQLQuery`, `doSQLQueryOnError`, or `doSQLQueryAck` operation when dealing with a returned row.

```
action getSchemaFieldOrder() returns sequence< string >
action getSchemaFieldTypes() returns dictionary< string, string >
action getSchemaIndexFields() returns sequence< string >
```

[The ADBCHelper Application Programming Interface](#)

Setting context

By default the ADBCHelper API sends requests to an internal service monitor running in the main context with the EPL `route` command. However, if your application uses parallel processing and spawns to multiple contexts, you have to add code that identifies the main context so the ADBCHelper API can determine whether to send an event with `route` or with `enqueue`.

In applications with multiple contexts, use the following action to specify the main context before spawning.

```
setPrespawnContext( context c )
```

The ADBCHelper Application Programming Interface

Logging

This action specifies whether or not to log all SQL queries, commands, and commit operations to the correlator's log file.

```
action setLogQueries( boolean logQueries )
```

The default is `false`, which disables logging.

The ADBCHelper Application Programming Interface

Examples

The code listings in this section are adapted from the `api-helper-example.mon` application. The actual code can be found in the Apama installation's `samples\adbc\api-helper-example` directory.

Opening and closing a database and executing SQL commands

```
monitor ADBCHelper_Example
{
    com.apama.database.DBUtil db;
    action onload {
        db.openQuickODBC( "MySQL", "fred", "fred-123", handleError );
        db.doSQLCmd( "insert into NetworkInformation values (
            'Vodafone', 'FR', 104 );");
        db.doSQLCmd( "insert into NetworkInformation values (
            'O2', 'FR', 101 );");
        db.doSQLCmd( "insert into NetworkInformation values (
            'Three', 'FR', 102 );");
        db.doSQLCmdAck( "insert into NetworkInformation values (
            'Orange', 'FR', 103 );", 100, false);
        com.apama.database.DBAcknowledge ack;
        on com.apama.database.DBAcknowledge(ackNum = 100) : ack {
            if ack.error.length() = 0 then {
                log "Action complete" at INFO;
                // Other success handling code ...
            }
            else {
                log "Action failed: " + ack.error at ERROR;
                // Other failure handling code ...
            }
        }
        db.close( false);
    }
    action handleError( string reason ) {
        log "DB Error: " + reason at ERROR;
    }
}
```

Executing SQL queries

```
monitor ADBCHelper_Example
{
    com.apama.database.DBUtil db;
    action onload {
        db.openQuickODBC( "DBName", "user", "password", handleError );
        db.doSQLQuery(
            "SELECT * FROM NetworkInformation", handleResult );
        db.close( false);
    }
    action handleResult( dictionary< string, string > data ) {
        log "Network: " + data[ "network" ] +
            " CountryCode: " + data[ "countrycode" ] +

```

```

    " NIC: " + data[ "networkidentificationcode" ] at INFO;
}
action handleError( string reason ) {
    log "DB Error: " + reason at ERROR;
}
}

```

The ADBCHelper Application Programming Interface

The ADBC Event Application Programming Interface

The Apama Database Connector Event Programming Interface (API) provides operations for more complex, lower level interactions with databases than the operations included with the ADBCHelper API. The ADBC Event API is implemented with the following Apama event types and actions associated with those events.

- **Discovery** — This event type provides actions to obtain the names of data sources, databases, and named queries. Discovery actions are not necessary if your application knows the names of data sources, databases, and query templates.
- **Connection** — This event type provides actions for all operations on a database except for those involving queries.
- **Query** — This event type provides actions for creating and executing queries on databases.
- **PreparedQuery** — This event type provides actions for creating prepared query statements that are, in turn, used in queries.

The above events and associated actions are defined in the `ADBCEvents.mon` file.

In addition, some of the actions for `Discovery` events use the following event types, which are defined in the `ADBCAdapterEvents.mon` file.

- `DataSource`
- `Database`
- `QueryTemplate`

Using the Apama Database Connector

Discovering data sources

If your application needs to find available data sources, implement the following steps:

1. Create a new `Discovery` event.
2. Use the `Discovery` event's `findAvailableServers` action.
3. Create a handler action to perform callback actions on the results of the `findAvailableServers` action.
4. In the handler action, declare a variable for a `DataSource` event.

The definitions for the two forms of the `findAvailableServers` action are:

```

action findAvailableServers(
    float timeout,
    action < string, sequence<DataSource> > callback )

```

and

```

action findAvailableServersFull(
    float timeout,
    dictionary<string,string> extraParams,
    action < string, sequence<DataSource> > callback )

```

The definition of the `DataSource` event is:

```

event DataSource
{
    string serviceId;
    string name;
    dictionary<string,string> extraParams;
}

```

- `serviceID` — The `serviceID` to talk to this `DataSource`.
- `name` — The name of the `DataSource` such as ODBC, JDBC, or Sim.
- `extraParams` — Optional parameters.

The relevant code in the `samples\adbc\api-example\ADBC_Example.mon` file is similar to this:

```

com.apama.database.Discovery adbc :=
    new com.apama.database.Discovery;
adbc.findAvailableServers(TIME_TO_WAIT, handleAvailableServers);
action handleAvailableServers(string error,
    sequence<com.apama.database.DataSource> results)
{
    if error.length() != 0 then {
        log "Error occurred getting available data sources: " +
            error at ERROR;
    }
    else {
        if results.size() > 0 then {

            // Save off first service ID found.
            // Assumes first data source has at least one db
            if getDbServiceId() = "" then {
                dbServiceId := results[0].serviceId;
            }
            com.apama.database.DataSource ds;
            log "  DataSources: " at INFO;
            for ds in results {
                log "    " + ds.name + " - " + ds.serviceId at INFO;
            }
            log "Finding Databases ..." at INFO;
            // ... other logic ...
        }
        else {
            log "  No DataSources found" at INFO;
        }
    }
}

```

The ADBC Event Application Programming Interface

Discovering databases

If your application needs to find available databases, implement the following steps:

1. Given a `Datasource` event, call the event's `getDatabases` action.
2. Create a handler action to perform callback actions on the results of the `getDatabases` action.
3. In the handler action, declare a variable for a `Database` event.

The definitions for the two forms of the `getDatabases` action are:

```

action getDatabases(
    string serviceId,
    string user,

```

```

string password,
action< string, sequence<Database> > callback)

```

and

```

action getDatabasesFull(
    string serviceId,
    string locationURL,
    string user,
    string password,
    dictionary<string,string> extraParams,
    action < string, sequence<Database> > callback)

```

Note: JDBC data sources will usually require user and password values.

The definition of the Database event is:

```

event Database
{
    string shortName;
    string dbUrl;
    string description;
    dictionary<string,string> extraParams;
}

```

- shortName — A short display name
- dbUrl — The complete URL of the database, for example, jdbc:sqlserver://localhost/ApamaTest.
- extraParams — Optional parameters.

The relevant code in the samples\adbc\api-example\ADBC_Example.mon file is similar to this:

```

action handleAvailableServers(string error,
    sequence<com.apama.database.DataSource> results)
{
    if error.length() != 0 then {
        log "Error occurred getting available data sources: " +
            error at ERROR;
    }
    else {
        if results.size() > 0 then {

            // Save off first service ID found.
            // Assumes first data source has at least one db
            if getDbServiceId() = "" then {
                dbServiceId := results[0].serviceId;
            }
            com.apama.database.DataSource ds;
            log "  DataSources: " at INFO;
            for ds in results {
                log "    " + ds.name + " - " + ds.serviceId at INFO;
            }
            log "Finding Databases ..." at INFO;
            for ds in results {
                adbc.getDatabases(ds.serviceId, USER, PASSWORD,
                    handleAvailableDatabases);
            }
        }
        else {
            log "  No DataSources found" at INFO;
        }
    }
}

string dbName;
action handleAvailableDatabases(string error,
    sequence<com.apama.database.Database> results)
{
    if error.length() != 0 then {
        log "Error occurred getting available databases: " +

```

```

        error at ERROR;
    }
    else {
        if results.size() > 0 then {
            // Save name of first db found
            if getDbName() = "" then {
                dbName := results[0].shortName;
            }
            com.apama.database.Database db;
            log " Databases: ";
            for db in results {
                log "      " + db.shortName + " - " +
                    db.description + " - " + db.dbUrl at INFO;
            }
            // ... other logic...
        }
        else {
            log " No Databases found" at INFO;
        }
    }
}

```

The ADBC Event Application Programming Interface

Opening a database

In order to open a database, your application should implement the following steps:

1. Create a new `Connection` event.
2. Call the `Connection` event's `openDatabase` action with the database's `serviceID`, database URL, `autocommit` preference, and the name of the callback action.
3. Create the handler action for the `openDatabase` callback action.

The definitions for the different forms of the `openDatabase` actions are:

```

action openDatabase(
    string serviceId,
    string databaseURL,
    string user,
    string password,
    string autoCommit,
    action <Connection, string> callback)

```

and

```

action openDatabaseFull (
    string serviceId,
    string databaseURL,
    string user,
    string password,
    string autoCommit,
    boolean readOnly,
    dictionary<string,string> extraParams,
    action <Connection, string> callback)

```

In addition to these open actions you can also open a database using an already open matching connection if one exists using the `openDatabaseShared` action. If an existing connection is not found, the action opens a new connection.

```

action openDatabaseShared (
    string serviceId,
    string databaseURL,
    string user,
    string password,
    string autoCommit,
    boolean readOnly,
    dictionary<string,string> extraParams,

```

```
action <Connection, string> callback)
```

The value for the `autocommit` parameter is a combination of the `AutoCommit` and `StoreCommitInterval` properties. For information on these properties, see ["Configuring an ADBC adapter" on page 21](#). The value for the `autocommit` parameter can be one of the following modes:

- "" — An empty string specifies that the value set in the configuration file should be used.
- true — Use the data source's value as determined by the ODBC or JDBC driver.
- false — Disable `autocommit`.
- x.x — Use time auto commit interval in seconds.

The `readOnly` parameter specifies if the connection should be read-only. If the connection is read-only an error will be reported for any API action that requires writes (`Store`, `Commit`, or `Rollback`). Most databases do not prevent writes from a connection in read-only mode so it is still possible to perform writes using the `Command` actions.

Specifying parameter values in the open actions overrides the property values set in the configuration file.

The relevant code in the `samples\adbc\api-example\ADBC_Example.mon` file is similar to this:

```
com.apama.database.Connection conn :=
    new com.apama.database.Connection;
action handleAvailableDatabases(string error,
    sequence<com.apama.database.Database> results)
{
    if error.length() != 0 then {
        log "Error occurred getting available databases: " +
            error at ERROR;
    }
    else {
        if results.size() > 0 then {
            // Save name of first db found
            if getDbName() = "" then {
                dbName := results[0].shortName;
            }
            com.apama.database.Database db;
            log " Databases: " at INFO;
            for db in results {
                log " " + db.shortName + " - " +
                    db.description + " - " + db.dbURL at INFO;
            }
            log "Opening Database " + dbName + " ..." at INFO;
            string serviceId := getDbServiceId();
            conn.openDatabase(serviceId, results[0].dbUrl, USER,
                PASSWORD, "", handleOpenDatabase);
        }
        else {
            log " No Databases found" at INFO;
        }
    }
}
```

The ADBC Event Application Programming Interface

Closing a database

In order to close a database your application should implement the following steps:

1. Call the `closeDatabase` action of the `Connection` event (for the open database) with the name of the callback action.

2. Create a handler action for the `closeDatabase` callback action.

The definitions for the two forms of the `closeDatabase` action are:

```
action closeDatabase(  
    action <Connection, string> callback)
```

and

```
action closeDatabaseFull(  
    boolean force,  
    dictionary<string,string> extraParams,  
    action<Connection,string> callback)
```

The relevant code in the `samples\adbc\api-example\ADBC_Example.mon` file is similar to this:

```
com.apama.database.Connection conn :=  
new com.apama.database.Connection;  
// ...  
    conn.openDatabase(serviceId, results[0].dbUrl, "",  
        handleOpenDatabase);  
// ...  
    conn.closeDatabase(handleCloseDatabase);  
action handleCloseDatabase(com.apama.database.Connection conn,  
    string error)  
{  
    if error.length() != 0 then {  
        log "Error closing database " + getDbName() + ": " +  
            error at ERROR;  
    }  
    else {  
        log "Database " + getDbName() + " closed." at INFO;  
    }  
}
```

The ADBC Event Application Programming Interface

Storing event data

In order to store an event in a database, your application needs to use the `Connection` event's `storeEvent` action. The definition of the `storeEvent` action is:

```
action storeEvent(  
    float timestamp,  
    string eventString,  
    string tableName,  
    string statementName,  
    string timeColumn,  
    dictionary<string,string> extraParams) returns integer
```

The `getTime()` call on the event is used to set the `timestamp` value.

Similarly, the `toString()` call on an event sets the `eventString` field.

The `tableName` parameter specifies the name of the database table where you want to store the data.

The `statementName` parameter specifies the name of a `storeStatement` that references a prepared statement or stored procedure. The `storeStatement` is created with the `Connection` event's `createStoreStatement` action. See ["Creating and deleting store events" on page 44](#) for more information on creating a `storeStatement`. If you do not want to specify a prepared statement or stored procedure, the `statementName` parameter should be set to "" (an empty string).

The `timeColumn` parameter specifies the column in the database where you want the event timestamp to be stored.

The `storeEvent` action returns an integer value, which is the identifier for the event being stored. The `setStoreErrorCallback` action is used to specify an action to be used when an error is reported.

To store an event and provide acknowledgement, implement the `storeEventWithAck` action and a callback handler. The definition of the `storeEventWithAck` action is:

```
action storeEventWithAck(  
    float timestamp,  
    string eventString,  
    string tableName,  
    string statementName,  
    string timeColumn,  
    string token,  
    dictionary<string,string> extraParams,  
    action <Connection, string, string> callback)
```

In addition to the parameters used with the `storeEvent` action, the `storeEventWithAck` action includes `token` and `callback` parameters. The `token` parameter specifies a user-defined string to be passed in that will be returned in the callback action. This allows the callback to perform different operations depending on the token value. In this way, a single callback action can perform different operations, eliminating the need to create separate callbacks for each operation. If the `token` parameter is not needed for the callback, it should be set to "" (an empty string).

The `callback` parameter specifies the callback action that handles the success or failure of the `storeEventWithAck` action.

If you want to avoid the overhead of receiving acknowledgements each time event data is added to a database table, use the `storeEvent` action. If your application needs to handle a failure during a call to the `storeEvent` action, it should call the `setStoreErrorCallback` action; for more information, see ["Handling data storing errors" on page 45](#).

The ADBC Event Application Programming Interface

Storing non-event data

In order to store non-event data in a database, your application needs to use the `Connection` event's `storeData` action. The definition of the `storeData` action is:

```
action storeData(  
    string tableName,  
    string statementName,  
    dictionary<string,string> fields,  
    dictionary<string,string> extraParams) returns integer
```

The `tableName` parameter specifies the name of the database table where you want to store the data.

The `statementName` parameter specifies the name of a `StoreStatement` that references a prepared statement or stored procedure. The `storeStatement` is created with the `Connection` event's `createStoreStatement` action. See ["Creating and deleting store events" on page 44](#) for more information on creating a `storeStatement`. If you do not want to specify a prepared statement or stored procedure, the `statementName` parameter should be set to "" (an empty string).

The `fields` parameter specifies the column values to be stored.

To store an event and provide acknowledgement, implement the `storeDataWithAck` action and a callback handler. The definition of the `storeDataWithAck` action is:

```
action storeDataWithAck(  
    string tableName,  
    string statementName,  
    dictionary<string,string> fields,  
    string token,  
    dictionary<string,string> extraParams,
```

```
action <Connection, string, string> callback)
```

In addition to the parameters used with the `storeData` action, the `storeDataWithAck` action includes `token` and `callback` parameters. The `token` parameter specifies a user-defined string to be passed in that will be returned in the callback action. This allows the callback to perform different operations depending on the token value. In this way, a single callback action can perform different operations, eliminating the need to create separate callbacks for each operation. If the `token` parameter is not needed for the callback, it should be set to "" (an empty string).

The `callback` parameter specifies the callback action that handles the success or failure of the `storeDataWithAck` action. The acknowledgement `callback` string contains any errors reported as well as the returned `token`, an empty acknowledgement string indicates success.

If you do not have to take additional action each time a row of data is added to a database table, you can avoid the overhead of receiving acknowledgements by using the `storeData` action. If your application needs to handle a failure during a call to the `storeData` action, it should call the `setStoreErrorCallback` action; for more information, see ["Handling data storing errors" on page 45](#).

The ADBC Event Application Programming Interface

Creating and deleting store events

If your application will use a prepared statement or a stored procedure in a `store` action (such as `storeData` or `storeEvent`) you need to first create a `storeStatement` with `createStoreStatement` action.

The `createStoreStatement` is defined as:

```
action createStoreStatement(  
    string name,  
    string tableName,  
    string statementString,  
    sequence<string> inputTypes,  
    dictionary<integer,string> inputToNameMap,  
    dictionary<string,string> extraParams,  
    action<Connection,string,string> callback)
```

The arguments for this action are:

- `name` - The name of the `storeStatement` instance that will be used in a `store` action. The name must be unique. Specifying a value for `name` is optional and if omitted, one will be created in the form `Statement_1`.
- `tableName` - The name of the database table where the data will be written when the `store` action that uses the `storeStatement` is called.
- `statementString` - The SQL string that will be used as a template when the `store` action that uses the `storeStatement` is called. You can use question mark characters to indicate replaceable parameters in the statement. For example, `"insert into myTable(?,?,?) values(?,?,?)"`.

If you want to use a stored procedure, in the `statementString` enclose the name of the database's stored procedure in curly brace characters `{ }` and use question mark characters `?` to indicate replaceable parameters. For example, `"{call myStoredProcedure(?,?,?)}"`. Stored procedures used in this way can only take input parameters. The stored procedure must exist in the database.

- `inputTypes` - Specifies the types that will be used as replaceable parameters in the `statementString`.
- `inputToNameMap` - Specifies what data item should be used for each input parameter of the store statement. If storing data it would be the name from the dictionary of data to be stored. If storing events it would be the event field name. When you specify the dictionary, the `integer` is the

position and the `string` is the data name. For example, you might specify the `inputToNameMap` parameter as follows:

```
inputToNameMap :=
    {1:"timefield",2:"strfield",3:"intfield",4:"floatfield",5:"boolfield"};
```

- `extraParams` - Not required
- `callback` - The action's callback handler. The definition of the callback action should take the error message as the first string parameter followed by the `storeStatement` name.

The `deleteStoreStatement` is defined as:

```
action deleteStoreStatement(
    string statementName,
    string tableName,
    dictionary<string,string> extraParams,
    action<Connection,string,string> callback)
```

The ADBC Event Application Programming Interface

Handling data storing errors

If your application uses the `storeData` or `storeEvent` actions, you can use the `setStoreErrorCallback` action to handle failures. This is useful for applications that make a large number of store calls where high performance is important and acknowledgement for an individual store operation call is not required. A single `setStoreErrorCallback` action can handle the failure of multiple store calls. The `setStoreErrorCallback` action is defined as follows:

```
action setStoreErrorCallback(
    action<Connection, integer, integer, string> callback)
{
```

Calls to `storeData` and `storeEvent` actions return unique integer identifiers; use these identifiers in the `setStoreErrorCallback` action. The first integer specifies the identifier of the first store action where an error occurred; the second integer specifies the identifier of the last store action error. `callback` specifies the name of the user-defined error handling action.

The ADBC Event Application Programming Interface

Committing transactions

By default, the auto-commit behavior assumes the `AutoCommit` and `StoreCommitInterval` properties specified in the adapter's configuration file and the open action are using the default values. If you want more control over when changes are committed to a database, set the `openDatabase` action's `autoCommit` parameter to false and in your EPL code, manually commit data using the `Connection` event's `commitRequest` action.

1. Create a callback action to handle the results of the `commitRequest` action.
2. Call the `commitRequest` action of the `Connection` event (for the open database) with the name of the callback action.

The definitions for the two forms of the `commitRequest` action are:

```
action commitRequest(
    action<Connection, integer, string, string> callback) returns integer
action commitRequestFull(
    string token,
    dictionary<string, string> extraParams,
    action<Connection, integer, string, string> callback) returns integer
```

The ADBC Event Application Programming Interface

Rolling back transactions

To rollback a database transaction, your application should use the `Connection` event's `rollbackRequest` action. If you want to use rollback actions, you need to turn `autocommit` off.

1. Create a callback action to handle the results of the `rollbackRequest` action.
2. Call the `rollbackRequest` action of the `Connection` event (for the open database) with the name of the callback action.

The definitions for the two forms of the `rollbackRequest` action are:

```
action rollbackRequest(  
    action<Connection, integer, string, string> callback) returns integer  
action rollbackRequestFull(  
    string token,  
    dictionary<string, string> extraParams, string token,  
    action<Connection, integer, string, string> callback) returns integer
```

The ADBC Event Application Programming Interface

Running commands

To execute database commands, such as creating a table or SQL operations such as Delete and Update, use the `Connection` event's `runCommand` action.

1. Call the `runCommand` action of the `Connection` event (for the open database) with the a string containing the SQL command to execute and the name of the callback action.
2. Create a handler action for the `runCommand` callback action.

The definitions for the two forms of the `runCommand` are:

```
action runCommand(  
    string commandString,  
    string token,  
    action <Connection, string, string> callback)
```

and

```
action runCommandFull(  
    string commandString,  
    string token,  
    dictionary<string, string> extraParams,  
    action<Connection, string, string> callback)
```

The ADBC Event Application Programming Interface

Executing queries

An Apama application can execute three types of SQL queries on databases:

- Standard query — An SQL query that you write in your EPL code. This is typically a simple query provided as a string when your EPL code initializes the query. The query string is used when the query is submitted to the database when your EPL code calls the action that starts the query. See ["Executing standard queries" on page 47](#).
- Prepared query — An SQL query that uses a *prepared statement* or *stored procedure*, both of which are stored in the database. Because they are stored in the database, prepared queries are more efficient than standard and named queries as they do not need to be compiled and destroyed each time they are run. Input parameters for prepared queries are not set during initialization. They are set after initialization, but before the query is submitted to the database when the query

start action is called. See ["Prepared statements" on page 49](#) and ["Stored procedures" on page 50](#).

- **Named query** — An SQL query that you write in an XML file as part of the Apama Studio project. Typically, you use a named query if you plan to use the query multiple times (as a template, supplying parameterized values). If the query is relatively complex, it is useful to separate it from your EPL code for readability. Your EPL code specifies the query template name and the template parameter names and values to use when it initializes the query. The template name and parameters are used when the query is submitted to the database when your EPL code calls the action that starts the query. See ["Named queries" on page 52](#).

The ADBC Event Application Programming Interface

Executing standard queries

In order to execute a standard query, your application needs to implement the following steps:

1. Create a new `Query` event.
2. Initialize the query by calling the `Query` event's `initQuery` action passing in the name of the database's `Connection` event and the query string.
3. Call the `Query` event's `setReturnType` action to specify the return type. Apama recommends specifying the return type using one of the following constants:

- `Query.RESULT_EVENT`
- `Query.RESULT_EVENT_HETERO`
- `Query.NATIVE`
- `Query.HISTORICAL`

See ["Return Types" on page 48](#) below for more information on return types.

4. If the return type is `Native`, indicate the event type to be returned by specifying it with the `Query` event's `setEventType` action.

The `setEventType` action is defined as:

```
action setEventType(string eventType)
```

In addition, you need to add mapping rules to the ADBC adapter's configuration file for the event type being returned.

5. In addition, if the return type is `Native`, specify the database table column that stores the event's timestamp with the `Query` event's `setTimeColumn` action.

The `setTimeColumn` action is defined as:

```
action setTimeColumn(string timeColumn)
```

6. If the query will return a large number of results, call the `Query` event's `setBatchSize` action passing in an integer setting the batch size.
7. If you set a batchsize, also use the `Query` event's `setBatchDoneCallback` action passing in values for the `token` and `callback` parameters.

```
action setBatchDoneCallback(  
    string token,  
    action<Query, string, integer, float, string, string> callback)
```


8. If the application needs to know the query's result set schema, call the `Query event`'s `setSchemaCallback` action passing in the name of the handler action.
9. Call the `Query event`'s `start` action passing in the name of the handler action that will be called when the query completes.

Return Types

- **NATIVE** — This return type is most commonly used for playback. When a query is run, each row of the query will be passed through the IAF mapping rules and the matching event will be sent as-is to the correlator. The `Native` return type would not be used for general database queries.

In addition to specifying the `Native` return type, your query needs to specify the event type to be returned and the name of the database table's column that contains the event's time stamp. Specify the event by using the `Query event`'s `setEventType` action; specify the time column by using the `Query event`'s `setTimeColumn` action. You also need to add mapping rules for this event type to the ADBC adapter's configuration file.

- **HISTORICAL** — This return type is also used for playback. When a query is run, each row of the query will be passed through the IAF mapping rules and then the matching event will be “wrapped” in a container event. The container event will have a name based on that of the event name. For example a `Tick` event would be wrapped in a `HistoricalTick` event. Event wrapping allows events to be sent to the correlator without triggering application listeners. A separate user monitor can listen for wrapped events, modify the contained event, and reroute it such that application listeners can match on it. The `Wrapped` return type would not be used for general database queries.
- **RESULT_EVENT** — This return type is used for general database queries. When a query is run, each row in the result set will be mapped to a dictionary in a generic `ResultEvent`. The ADBC adapter will generate a `SchemaEvent` containing the schema (name and type) of the fields in the result set of the query. The `SchemaEvent` will be sent first, before any `ResultEvents`.

The definition for `ResultEvent` is:

```
event ResultEvent {
    integer messageId; // Unique id of query
    string serviceId;
    integer schemaId; // ResultSchema event schemaId to use with this ResultEvent
    dictionary <string, string> row; // Data
}
```

The definition for `ResultSchema` is:

```
event ResultSchema {
    integer messageId; // Unique id of query
    string serviceId;
    integer schemaId;
    sequence <string> fieldOrder;
    dictionary <string, string> fieldTypes;
    sequence <string> indexFields;
    dictionary<string,string> extraParams;
}
```

- **RESULT_EVENT_HETERO** — This return type is intended for advanced database queries. It is not applicable to SQL databases. Some market databases support queries which can, in essence, return multiple tables. For example a market database might allow queries which return streams of both `Tick` and `Quote` data. For such databases multiple `SchemaEvents` would be generated indexed by id.

Executing queries

Stopping queries

The following action cancels all outstanding queries in the queue.

```
action stopAllQueries(  
    action<Connection,string> callback)
```

Executing queries

Preserving column name case

In order to provide compatibility for a wide number of database vendors, the ADBC adapter normally converts column names to lower case. However, if you want to execute complex queries where the `_ADBCType` or `_ADBCTime` are returned as part of the query rather than being specified using the `setEventType` and `setTimeColumn` actions on the query, you need to set the `ColumnNameCase` property in the ADBC adapter's configuration file to `unchanged`.

Setting the `ColumnNameCase` property is done by manually editing the `ColumnNameCase` property to the configuration file.

1. In the **Project Explorer**, in the project's `Adapters` node, expand the ODBC or JDBC adapter, and double-click the adapter instance to open it in the ADBC adapter editor.
2. Display the ADBC adapter editor's **XML source** tab.
3. In the `<transport>` element, edit the `ColumnNameCase` property as follows:

```
<property name="ColumnNameCase" value="unchanged"/>
```

4. Save the ADBC adapter instance's configuration.

When the `ColumnNameCase` property is set to `unchanged`, you can specify a query string in the form

```
string queryString := "SELECT *, 'Trade' AS _ADBCType FROM TradeTable  
                      WHERE symbol = \"ADL\";
```

The other values for the `ColumnNameCase` property can be `lower`, (the default) and `upper`.

Executing queries

Prepared statements

Apama applications can use prepared statements when executing queries. Prepared statements have the following performance advantages over standard queries:

- The query does not need to be re-parsed each time it is used.
- The query allows for replaceable parameters.

The ADBC Event Application Programming Interface

Using a prepared statement

To use a prepared statement, follow the steps below. Note that `PreparedQuery` events support only ODBC/JDBC data types. Vendor-specific data types are not allowed.

1. Create a new `Query` event.
2. Create a new `PreparedQuery` event.

3. Call the new `PreparedQuery` event's `init` action, passing in the database connection, the query string, the input types if using replaceable parameters and the output types if it will be used as a stored procedure.

The definition for the `init` action is:

```
action init (  
    Connection conn,  
    string queryString,  
    sequence<string> inputTypes,  
    sequence<string> outputTypes)
```

The arguments for the `init` action are:

- `conn` — The name of the database's `Connection` event.
- `queryString` — The SQL query string; you can use question mark characters `?` to indicate replaceable parameters.
- `inputTypes` — This is optional, but if you use replaceable parameters in the `queryString`, you need to specify the types that will be used in the query.
- `outputTypes` — This is optional, but if the `PreparedQuery` event is to be used for a stored procedure and it uses output parameters, you need to specify the output types.

For example:

```
sequence<string> inputTypes := ["INTEGER","INTEGER"];  
myPreparedQuery.init (  
    myConnection,  
    "SELECT * FROM mytable WHERE inventory > ? and inventory <?",  
    inputTypes, new sequence<string>);
```

4. Call the new `PreparedQuery` event's `create` action, passing in the name of the callback action.
5. In the callback action's code, call the `Query` event's `initPreparedQuery` action (instead of the `initQuery` action), passing in the name of the `PreparedQuery` event. See ["Executing standard queries" on page 47](#).
6. Call the `Query` event's `setInputParams` action, passing in the values to be used for the replaceable parameters. The definition of the `setInputParams` action is:

```
setInputParams(sequence<string> inputParams)
```

If you want to use `NULL` for the value of a replaceable parameter, use `ADBC_NULL`.
7. If necessary, call any of the other `Query` actions, such as `setBatchSize`, as required.
8. Call the `Query` event's `start` action as you would when executing any other query. See ["Executing standard queries" on page 47](#).

Prepared statements

Stored procedures

Apama applications can use stored procedures when executing queries. Using stored procedures is similar to using prepared statements. The difference is that a stored procedure needs to specify the name of the stored procedure and the output types returned by the query.

The ADBC Event Application Programming Interface

Using a stored procedure

Queries in Apama application use stored procedures by specifying the name of the stored procedure in a prepared statement's query string.

To use a stored procedure:

1. Create a new `Query` event.
2. Create a new `PreparedQuery` event.
3. Call the new `PreparedQuery` event's `init` action, passing in the database connection, the query string, the input types, and the output types.

The definition for the `init` action is:

```
action init (  
    Connection conn,  
    string queryString,  
    sequence<string> inputTypes,  
    sequence<string> outputTypes)
```

The arguments for the `init` action are:

- `conn` — The name of the database's `Connection` event.
- `queryString` — The SQL query string; enclose the name of the database's stored procedure in curly brace characters `{ }` and use question mark characters `?` to indicate replaceable parameters.
- `inputTypes` — Specify the types that will be used for the replaceable parameters in the `queryString`.
- `outputTypes` — Specify the types that will be used for the replaceable parameters in the result.

For example:

```
sequence<string> inputTypes := ["INTEGER", "NULL", "INTEGER"];  
sequence<string> outputTypes := ["NULL", "INTEGER", "INTEGER"];  
myPreparedQuery.init (  
    myConnection,  
    "{call myprocedure(?,?,?)}",  
    inputTypes,  
    outputTypes);
```

- If a parameter is used as both an input and output type, it must be specified in both places.
- If it is only an input type it must be specified as `NULL` in `outputType`.
- If it is only an output type it must be specified as `NULL` in `inputType`.

Therefore, in the example above, the first parameter is just an input type; the second parameter is just an output type; and the third parameter is both an input and output type.

4. Call the new `PreparedQuery` event's `create` action, passing in the name of the callback action.
5. In the callback action's code or once the callback action has been called, call the `Query` event's `initPreparedQuery` action instead of the `initQuery` action, passing in the name of the `PreparedQuery` event. An error will be reported if the `Query` event's `initPreparedQuery` is called before the `PreparedQuery` `create` callback has been called. See ["Executing standard queries" on page 47](#).
6. Call the `Query` event's `setInputParams` action, passing in the values to be used for the replaceable parameters. The definition of the `setInputParams` action is:

```
setInputParams(sequence<string> inputParams)
```

If you want to use NULL for the value of a replaceable parameter, use `ADBC_NULL`.

7. If necessary, call any of the other `Query` actions, such as `setBatchSize`, as required.
8. Call the `Query` event's `start` action as you would when executing any other query. See ["Executing standard queries" on page 47](#).

Stored procedures

Named queries

Apama applications can use named queries. Named queries are templates with parameterized values and are stored in Apama projects. Queries of this type provide advantages for queries that will be used multiple times. They also serve to keep the SQL query strings separate from the application's EPL code.

To use a named query, your EPL code needs to specify the query template name and the template parameter names and values to use when it initializes the query. The template name and parameters are used when the query is submitted to the database.

You define a named query as a query template in the ADBC adapter's `ADBC-queryTemplates-SQL.xml` file. This file contains some pre-built named queries:

- `findEarliest` — Get the row with the earliest time (based on the stored event's timestamp).
- `findLatest` — Get the row with the latest time.
- `getCount` — Get the number of rows in a table.
- `findAll` — Get all the rows from a table.
- `findAllSorted` — Get all the rows from a table ordered by column.

The ADBC Event Application Programming Interface

Using named queries

To use a named query:

1. Create a new `Query` event.
2. Initialize the query by calling the `Query` event's `initNamedQuery` action, passing the name of the database's `Connection` event, the name of the query template, and a `dictionary<string, string>` containing the names and values of the named query's parameters.
3. Call the `Query` event's `setReturnType` action to specify the return type to be `ResultEvent`. When a query is run, each row in the result set will be mapped to a dictionary event field in a `ResultEvent` event.
4. Call the `Query` event's `setReturnEventCallback` action to specify the callback action that will handle the results returned by the query.
5. If the query will return a large number of events (on the order of thousands):
 - a. Call the `Query` event's `setBatchSize` action passing an integer that sets the batch size. The query returns results in batches of the specified size.
 - b. Call the `Query` event's `setBatchDoneCallback` action passing the name of the handler action.
 - c. Define the `setBatchDoneCallback` action to define what to do when a batch is complete. You must call the `Query` event's `getNextBatch` action to continue receiving the query results. The batch size for the

next batch is set by passing an integer parameter for the batch size. You could also call the stop action to stop the query, rather than continuing to receive batches of data.

6. Call the `Query` event's `start` action passing the name of the handler action that will be called when the query completes.
7. Create the callback action that you specified in Step 4, to handle the results returned by the query.
8. Each row of data that matches the query results in a call to the callback action, returning the row results in a parameter of `ResultEvent` type. The `ResultEvent` type contains a dictionary field that contains the row data.
9. Create the action that specifies what to do when the query completes (when all results are returned).

The following example uses the `initNamedQuery` action call to initialize the query, specifying the `findEarliest` named query and `stock_tables` as the value for the named query's `TABLE_NAME` parameter.

```
using com.apama.database.Connection;
using com.apama.database.Query;
using com.apama.database.ResultEvent;

monitor ADBCexample {
    Connection conn;
    Query query;

    string serviceId := "com.apama.adbc.JDBC_INSTANCE_1";
    string dbUrl := "jdbc:mysql://127.0.0.1:3306/exampledb";
    string user := "root";
    string password := "mysql";
    string queryString := "SELECT * FROM sys.tables";
    string tableName := "stock_table";
    dictionary<string,string> paramTable :=
        {"TABLE_NAME":tableName,"TIME_COLUMN_NAME":"tbd"};

    action onload() {
        conn.openDatabase(serviceId, dbUrl, user, password, "",
            handleOpenDatabase);
    }
    action handleOpenDatabase (Connection conn, string error){
        if error.length() != 0 then {
            log "Error opening database : " + error at ERROR;
        }
        else {
            log "Database is open." at INFO;
            runQuery();
        }
    }
    action runQuery {
        query.initNamedQuery(conn, "findEarliest", paramTable);
        query.setReturnType("ResultEvent");
        query.setResultEventCallback(handleResultEvent);
        query.start(handleQueryComplete);
    }
    action handleResultEvent(Query q, ResultEvent result) {
        log result.toString() at INFO;
    }
    action handleQueryComplete(Query query, string error,
        integer eventCount, float lastEventTime) {
        if error.length() != 0 then {
            log "Error running query '" + queryString + "': " +
                error at ERROR;
        }
        else {
            log " Query '" + queryString + "' successfully run." at INFO;
            log " Total events: " + eventCount.toString() at INFO;
            if lastEventTime > 0.0 then {
                log " Last Event Time: " + lastEventTime.toString()
                    at INFO;
            }
        }
    }
}
```

```

    }
  }
  conn.closeDatabase(handleCloseDatabase);
}
action handleCloseDatabase(Connection conn, string error) {
  if error.length() != 0 then {
    log "Error closing database : " + error at ERROR;
  }
  else {
    log "Database closed." at INFO;
  }
}
}
}

```

Named queries

Creating named queries

Each named query in the `ADBC-queryTemplates-SQL.xml` file is defined in an XML `<query>` element. Each `<query>` element has the following attributes:

- `name` — The name of the query.
- `description` — A short description of the query.
- `implementationFunction` — The substitution function that the adapter uses to process the named query. The substitution function allows you to specify tokens that are replaced by parameters with matching names.
- `inputString` — A string that contains the substitution tokens you want to replace with values specified as parameters.

A `<query>` element can also have one or more optional `<parameter>` child elements. Each `<parameter>` element has the following attributes:

- `description` — A short description of the parameter.
- `name` — The name of the parameter.
- `type` — The data type of the parameter.
- `default` — The default value of the parameter.

As an example, the following XML code in the `ADBC-queryTemplates-SQL.xml` file defines the pre-built `findEarliest` named query. The query returns the row with the earliest time.

```

<query
  name="findEarliest"
  description="Get the row with the earliest time."
  implementationFunction="substitution"
  inputString="select * from ${TABLE_NAME} order by ${TIME_COLUMN_NAME}
    asc limit 1">
  <parameter
    description="Name of a table to query"
    name="TABLE_NAME"
    type="String"
    default=""/>
  <parameter
    description="Name of the time column"
    name="TIME_COLUMN_NAME"
    type="String"
    default="time"/>
</query>

```

To create a named query:

1. In the Project Explorer, expand the project's `Adapters` node and open the adapter folder.
2. Double-click the instance configuration file to open it in the adapter editor.
3. In the adapter editor, select the **Advanced** tab.
4. Click the `ADBC-queryTemplates-SQL.xml` file to open it.
5. Select the **Design** tab.
6. On the **Design** tab, right-click the `namedQuery` element and select **Add Child > New Element**.
7. In the **New Element** dialog, type `query`, then click **OK**. A new query row is added to the list.
8. For each of the four attributes (`name`, `description`, `implementationFunction`, `inputString`):
 - a. Right-click the `query` element you added in Step 4, and select **Add Attribute > New Attribute**.
 - b. In the **New Attribute** dialog, provide a **Name** and a **Value** for the attribute.
9. If you want the query to use input parameters, for each parameter:
 - a. Right-click the `query` element and select **Add Child > New Element**.
 - b. In the **New Element** dialog, type `parameter`, then click **OK**.
 - c. Create the following attributes for each parameter:
 - `description`
 - `name`
 - `type`
 - `default`
10. Save the project's version of the query template file.

Named queries

The Visual Event Mapper

The Visual Event Mapper is no longer available for ODBC data sources.

When you add or open an instance of the ADBC Adapter, the adapter editor provides a Visual Event Mapper. The Event Mapper is available by selecting the Event Mapping tab. With the Event Mapper you specify an Apama event type and a table in an existing JDBC database. When you save the adapter configuration file, Apama Studio creates the rules that provide the mapping between the fields in the event and the columns in the database. The mapping rules are stored in the adapter instance's configuration file.

The **Generate Store Monitors** option in the Visual Event Mapper specifies whether or not Apama Studio generates all the necessary EPL code for monitors that listen for events of the specified types as well as for the EPL code that interacts with the database -- opening the database, checking the adapter status, storing event data, etc. This is the default setting. If you turn this option off, you need to write the EPL code for event listeners and for interacting with the database.

The **Auto Start Events** option in the Event Mapper specifies whether or not Apama Studio generates events that cause Apama Studio to automatically start saving event data when the

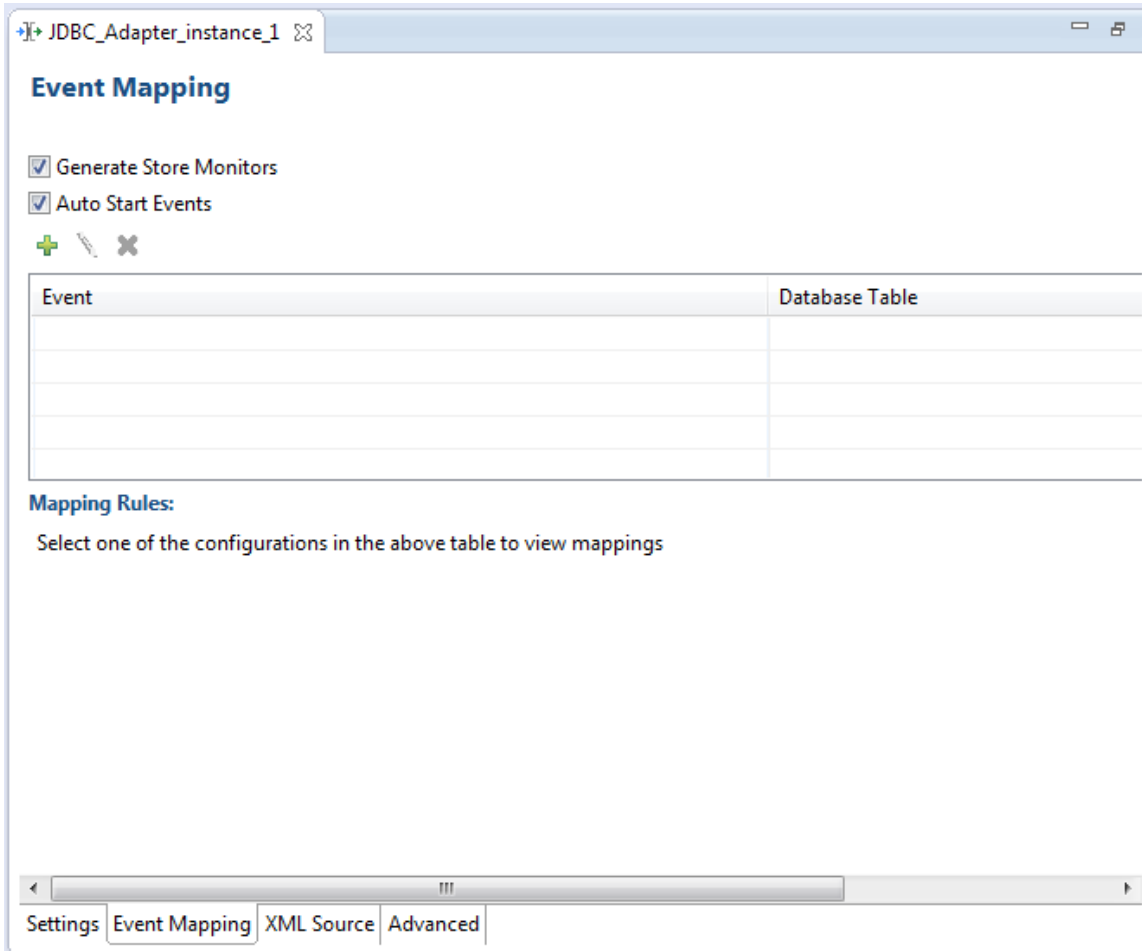
application is launched. If you turn this option off, your application needs to manually send a `StartStoreConfiguration` event in order to start saving data.

Using the Apama Database Connector

Using the Visual Event Mapper

To map an Apama event to a table in a database, follow the steps below. ADBC uses the **SQL** driver to perform the conversion between Apama types and SQL (JDBC) types. Any restrictions are due to the SQL database vendor and the SQL driver being used.

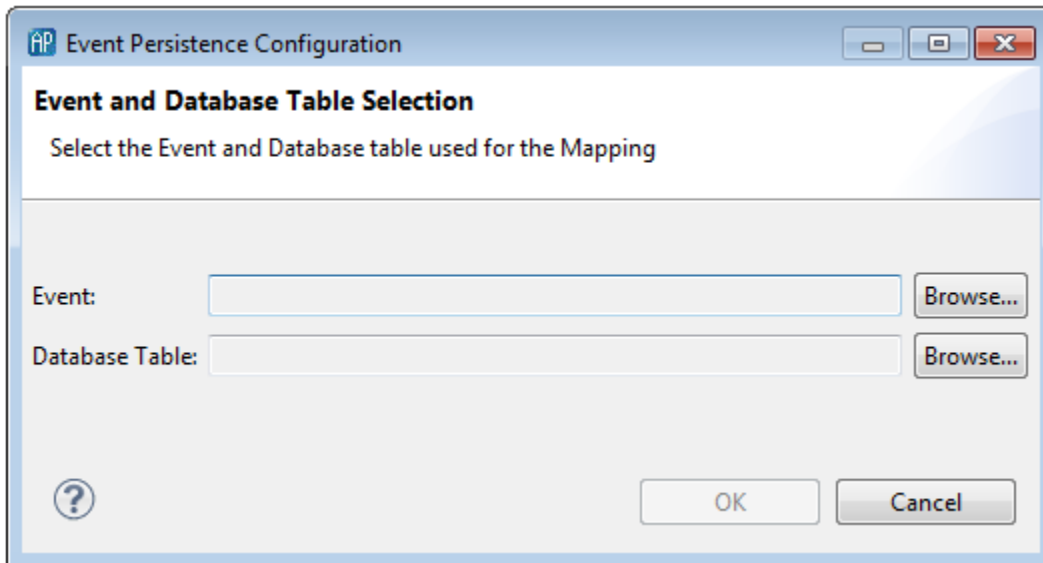
1. Add a new instance of the ADBC Adapter or open an existing instance and select the adapter editor's Event Mapping tab.



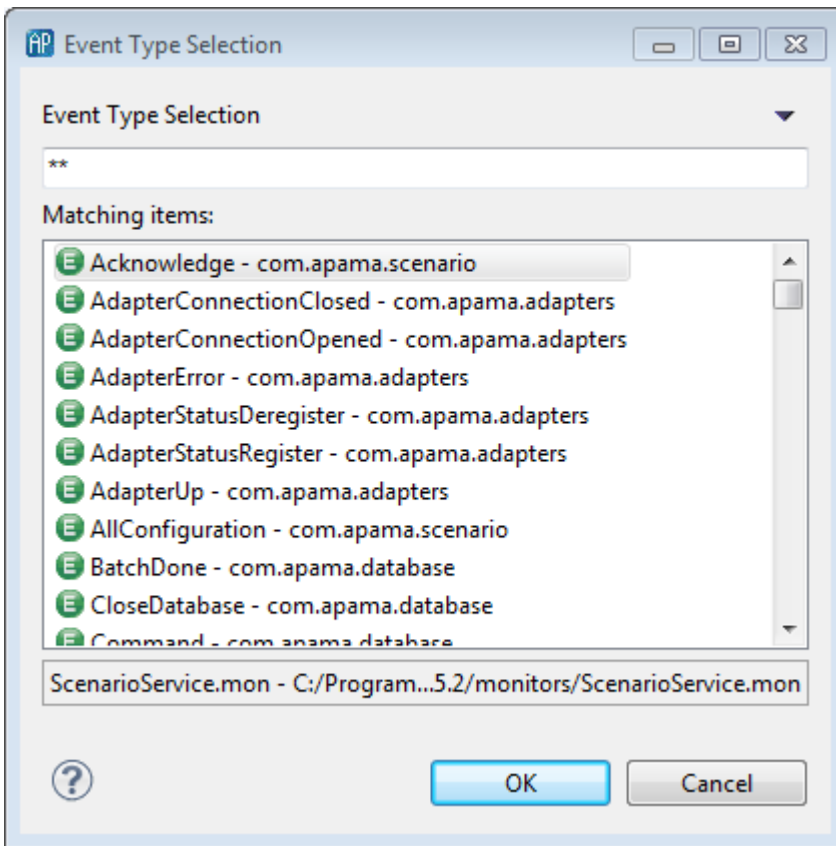
2. If you want Apama Studio to automatically generate an EPL monitor to listen for events of the specified type, make sure the **Generate Store Monitors** option is enabled; this is the default setting. In addition to generating all the necessary EPL code for monitors that listen for events of the specified types, Apama Studio generates all the EPL code that interacts with the database -- opening the database, checking the adapter status, storing event data, etc. This setting is useful if your application does not need to guarantee that each event is persisted. The generated monitor provides a best effort storage implementation suitable for storing data to be analyzed in tools like Analyst Studio. The generated monitor does not perform any filtering so all events of the type specified will be stored.

If your application needs to perform filtering of the events or needs to guarantee that each event will be persisted, you should disable **Generate Store Monitors** option and manually write the required code for the EPL monitors and for interacting with the database.

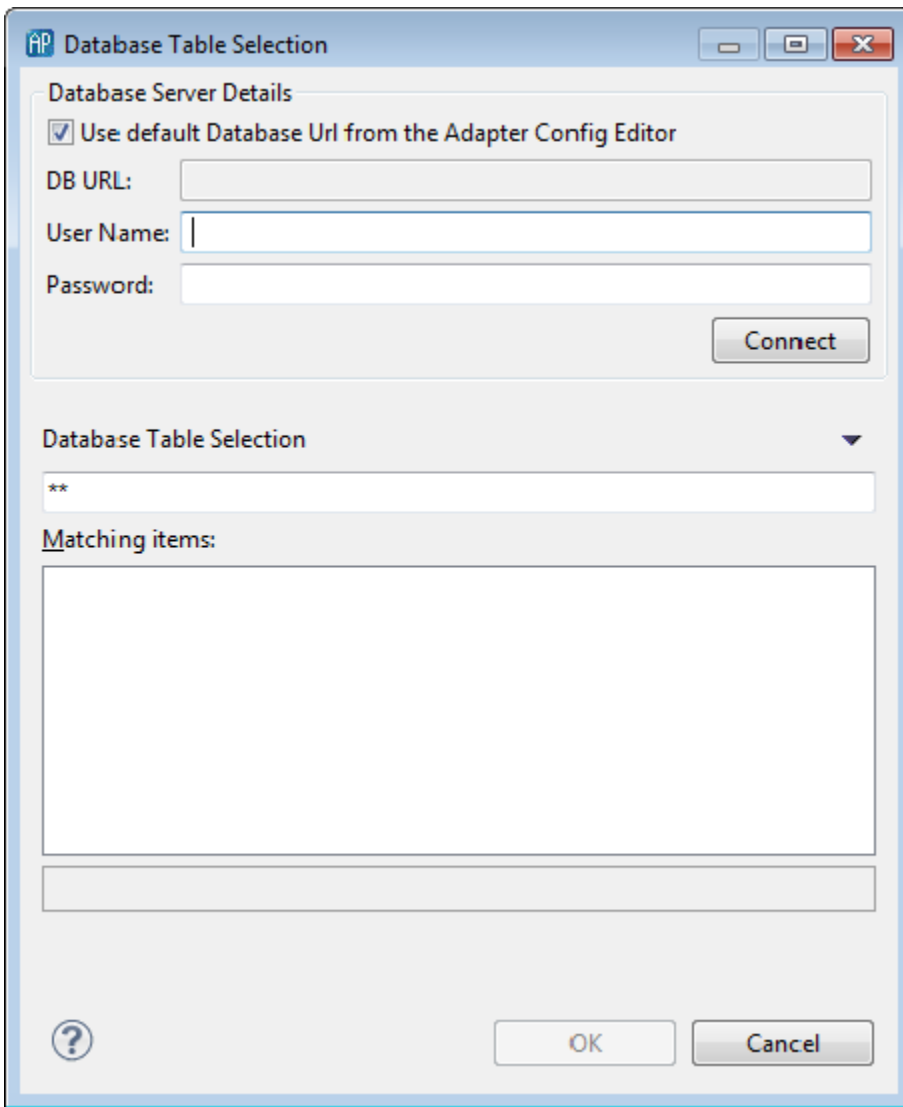
3. Make sure there is a check mark in the **Auto Start** check box (this is the default) if you want to start saving event data immediately when you launch the project. If you clear the check mark in the **Auto Start** check box, your application will need to manually send a `StartStoreConfiguration` event in order to start storing events.
4. In the adapter editor, click the **Add** button. The **Event Persistence Configuration** dialog opens.



5. In the **Event Persistence Configuration** dialog, click the **Browse** button next to the **Event** field. The **Event Type Selection** dialog opens displaying the available event types you can select from. Only events that can be emitted are shown; events that contain fields with contexts or actions are not displayed.

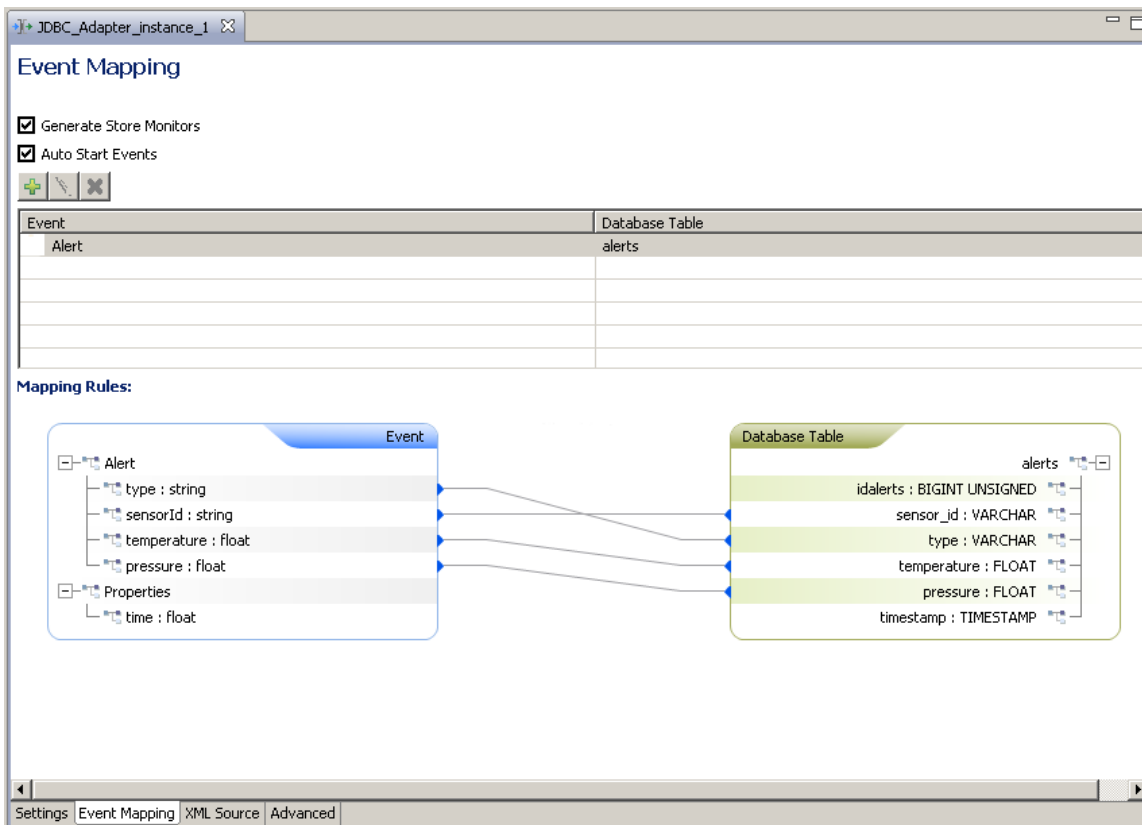


6. In the **Event Type Selection** dialog, select the event type you want to map as follows:
 - a. In the Event Type Selection field, enter the name of the event. As you type, event types that match what you enter are shown in the Matching Items list. When you select an event, the full name is shown on the dialog's status line. You can turn off this display with the dialog's Down Arrow menu icon (▼).
 - b. In the Matching Items list, select the name of the event type you want to map. The name of the EPL file that defines the selected event is displayed in the status area at the bottom of the dialog.
 - c. Click OK.
7. In the **Event Persistence Configuration** dialog, click the Browse button next to the Database table field. The **Database Table Selection** dialog opens.



8. In the **Database Table Selection** dialog, select the database table to which you want to map the event's fields as follows:
 - a. In the Database Server Details section, specify the DB URL, User Name, and Password. By default, the DB URL uses the value used in the adapter configuration settings. You can change the name of the database by un-checking the check box and entering a new name. (Note, you cannot change the type of database.)
 - b. Click Connect to access the database.
 - c. Select the name of the table from the Matching Items list or enter text in the Database Table Selection field. As you type, table names that match what you enter are shown in the Matching Items list. When you select a table, its name is also shown on the dialog's status line. You can turn off this display with the dialog's Down Arrow menu icon (▼).
 - d. In the Matching Items list, select the name of the database table where you want to store the event data.
 - e. Click OK.

9. In the **Event Persistence Configuration** dialog, click OK. The adapter editor display is updated to show the name of the event type and the database table in the Event section. The Mapping Rules section displays lists for Event and Database Table.



10. For each event field you want to store in the Event list click on the field and draw a line to the desired column in the Database Table list.

When you save the adapter instance configuration, Apama Studio generates mapping rules that specify the associations between event fields and database columns. Apama Studio also generates a monitor that listens for events of the specified type. The monitor allows the Apama application to manage when the events are written to the database.

The Visual Event Mapper

Playback

If event data is stored in a database, you can play back the events through the correlator using Apama Studio's Data Player. The Data Player consists of the Query Editor and the Data Player control. In the Query Editor you create and modify queries in order to specify what event data you want to play back. The Data Player control allows you to specify what query to use and how fast to play back the event data.

For full information on the Data Player, see "Using the Data Player" in *Using the Apama Studio Development Environment*.

Command line utilities

When you have stored event data in a database and created queries in Apama Studio, you can also launch a playback session using the Data Player command line utility, `adbc_management`. The `adbc_management` utility is described in .

Using the Apama Database Connector

Sample applications

Several sample applications in the Apama installation illustrate the use of the ADBCHelper and ADBC Event APIs. The samples are located in the `samples\adbc` directory of the Apama installation. The `api-helper-example` uses the ADBCHelper API; the other examples use the ADBC Event API. The samples include:

- `api-helper-example` — An EPL application that shows how to open and close a database and execute SQL commands and queries using the ADBCHelper API.
- `api-example` — An EPL application that uses the ADBC Event API to show how to use all API operations except those for storing data. Included is code for discovering data sources and databases, opening and closing databases, and executing queries.
- `store-data` — An EPL application that shows how to open a database, create a table, and store non-event data using the ADBC Event API.
- `store-events` — An EPL application that shows how to open a database, create a table, and store event data using the ADBC Event API.

Using the Apama Database Connector

Format of events in .sim files

In Apama 4.1 and earlier, Apama captured data streaming through the correlator into proprietary `.sim` files. These files consist of comma-delimited values. You can use the Apama Studio Data Player to play back event data from existing `.sim` files. Note, however, that the ADBC does not write data in `.sim` format.

Apama `.sim` files contain string versions of events and can also contain an optional header that specifies the default timezone for the series. The timezone identifiers can be any supported by Java 1.7. The format of the events contained in a `.sim` file is:

- `timestamp` — a float specifying UTC seconds since 01/01/1970.
- `event origin` — a string specifying whether the event is an internal or external event.
- `event` — a stringified version of the event itself.

Elements of the exported event are separated by commas.

The following is an example of an external event from a `.sim` file (each event is stored on a single line, here they are shown on separate lines for clarity):

```
1161287634.200,
```

```
external,  
com.apama.backtest.RawTick(  
    com.apama.marketdata.Tick("RACK",34.97,11,{}))
```

The following is an example of an internal event from a `.sim` file:

```
1161287629.600,  
internal,  
com.apama.backtest.RawTick(  
    com.apama.marketdata.Tick("RACK",34.96,64,{}))
```

The events in the example are `RawTick` events with embedded `Tick` events.

The following is an example of the optional header containing a specified default timezone:

```
#  
# <Timezone=America/New_York>  
#
```

Comments in `.sim` files

You can add comments when you edit `.sim` files. Introduce lines containing comments with either `#` or `//`.

[Using the Apama Database Connector](#)

Chapter 2: Using the Apama Web Services Client Adapter

■ Web Services Client adapter overview	64
■ Adding a Web Services Client adapter to an Apama Studio project	64
■ Configuring a Web Services Client adapter	65
■ Editing Web Services Client adapter configurations	74
■ Adding multiple instances of the Web Services Client adapter	78
■ Mapping Web Service message parameters	78
■ Specifying a correlation ID field	94
■ Specifying transformation types	94
■ Customizing mapping rules	96
■ Using EPL to interact with Web Services	99
■ Configuring logging for Web Services Client adapter	102
■ Web Services Client adapter artifacts	105

The Apama Web Services Client adapter is a SOAP-based adapter that allows Apama applications to invoke Web services. To use the Apama Web Services Client adapter in an Apama project, you need to do the following:

- Add the pre-packaged bundle of adapter resources for the Web Services Client adapter.
- Specify the location of the Web Service, using the URI of its Web Service Definition Language file (WSDL).
- Specify the Web Service operation or operations to invoke.
- Specify what Apama events will interact with the Web Service operations.
- Create mapping rules that associate the fields in the Apama events with the Web Service operations' parameters.

When you add and configure an instance of the Web Services Client adapter, Apama Studio automatically generates the configuration files, service monitors, and adapter artifacts that are necessary to deploy and run the Apama project's adapter instances.

You can add multiple instances of the Web Services Client adapter to an Apama project. The files generated by Apama Studio are specific to each adapter instance.

Note that Apama applications only invoke Web Service operations in the consume use case; it is not possible to expose actions in Apama applications as Web Services operations.

Using Standard Adapters

Web Services Client adapter overview

The process of adding a Web Services Client adapter to an Apama project involves the following:

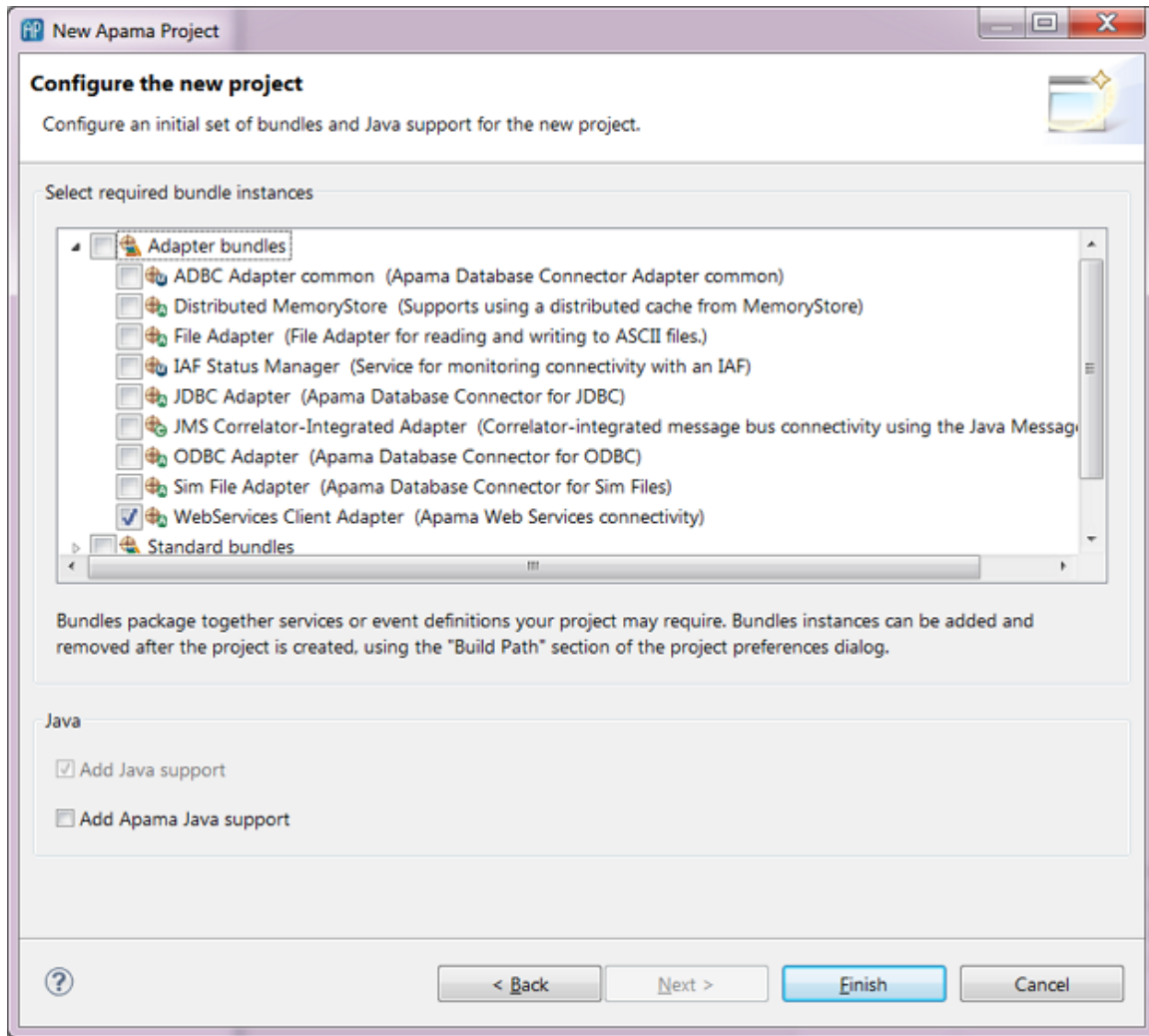
1. Add the Web Services Client adapter resource bundle to an Apama project.
2. Configure the Web Services Client adapter, which involves:
 - a. Specifying the Web Service Definition File (WSDL) that points to the Web Service.
 - b. Specifying a Web Service operation or operations your application will invoke.
 - c. Mapping Web Service Client adapter parameters.

[Using the Apama Web Services Client Adapter](#)

Adding a Web Services Client adapter to an Apama Studio project

To add a Web Services Client adapter to a project:

1. Select **File > New > Apama Project** from the Apama Studio menu. This launches the **New Apama Project** wizard.
2. In the **New Apama Project** wizard, give the project a name, and click **Next**. The second page of the wizard is displayed, listing the available Apama resource bundles.



3. In the **New Apama Project** wizard, in the Select required bundle instances field, select the WebService Client Adapter bundle. When you select this adapter, Apama Studio automatically enables the Add Java Support setting.
4. Click Finish.

Apama Studio generates the basic resources for Web Services Client adapter in the project. From here you need to configure the adapter instance in order to specify the Web Service your application will access. See ["Configuring a Web Services Client adapter" on page 65](#) for information on configuring the adapter instance.

Using the Apama Web Services Client Adapter

Configuring a Web Services Client adapter

After you add the Web Service Client adapter's resource bundle to an Apama project, you need to configure the adapter to interact with the external Web Service. This involves the following:

1. Specify the Web Service Definition File (WSDL) that points to the Web Service.
2. Specify a Web Service operation or operations your application will invoke.

3. Map the parameters of the Web Service operations to Apama events.

When you save the information that you add to a Web Service Client adapter, Apama Studio generates all the run-time support resources necessary for the adapter. For details on these resources, see ["Web Services Client adapter artifacts" on page 105](#).

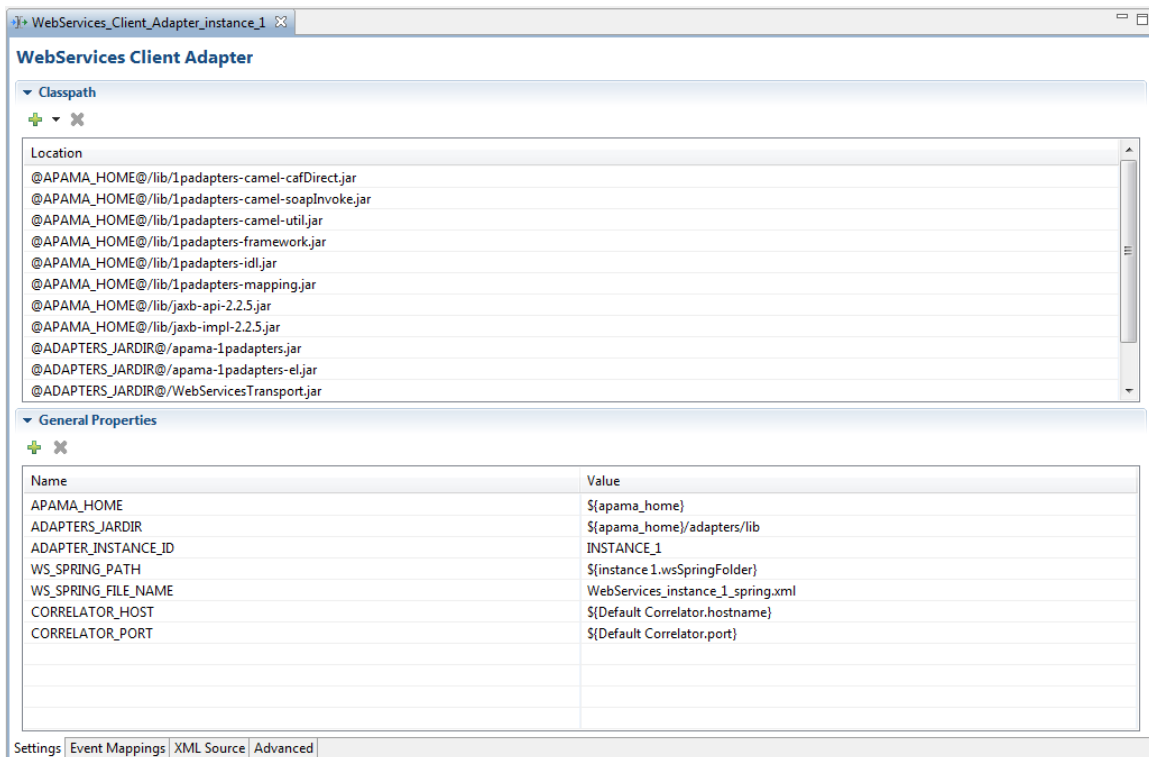
Using the Apama Web Services Client Adapter

Specify the Web Service and operation to use

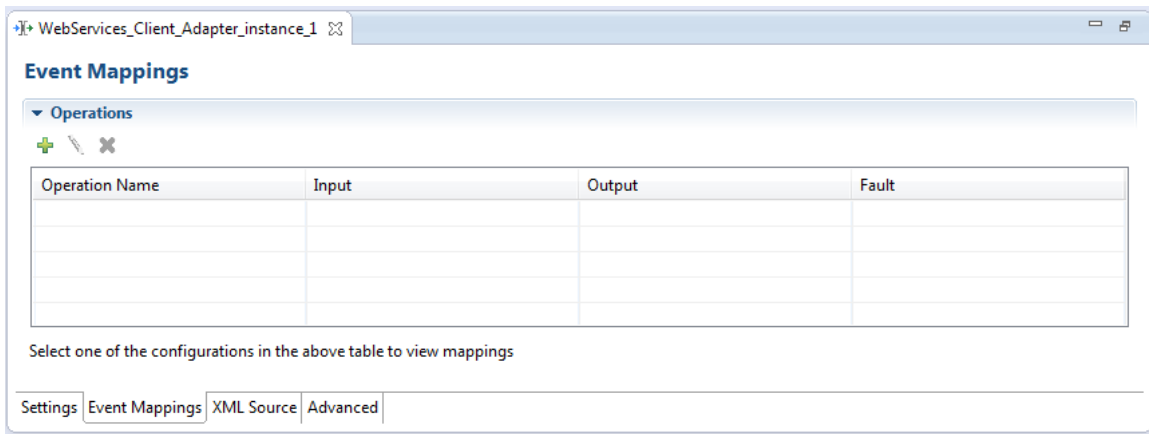
After you add an instance of the Web Services Client adapter to a project you need to specify the Web Service that the application will use and which Web Service operation the application will invoke.


To specify the Web Service to use:

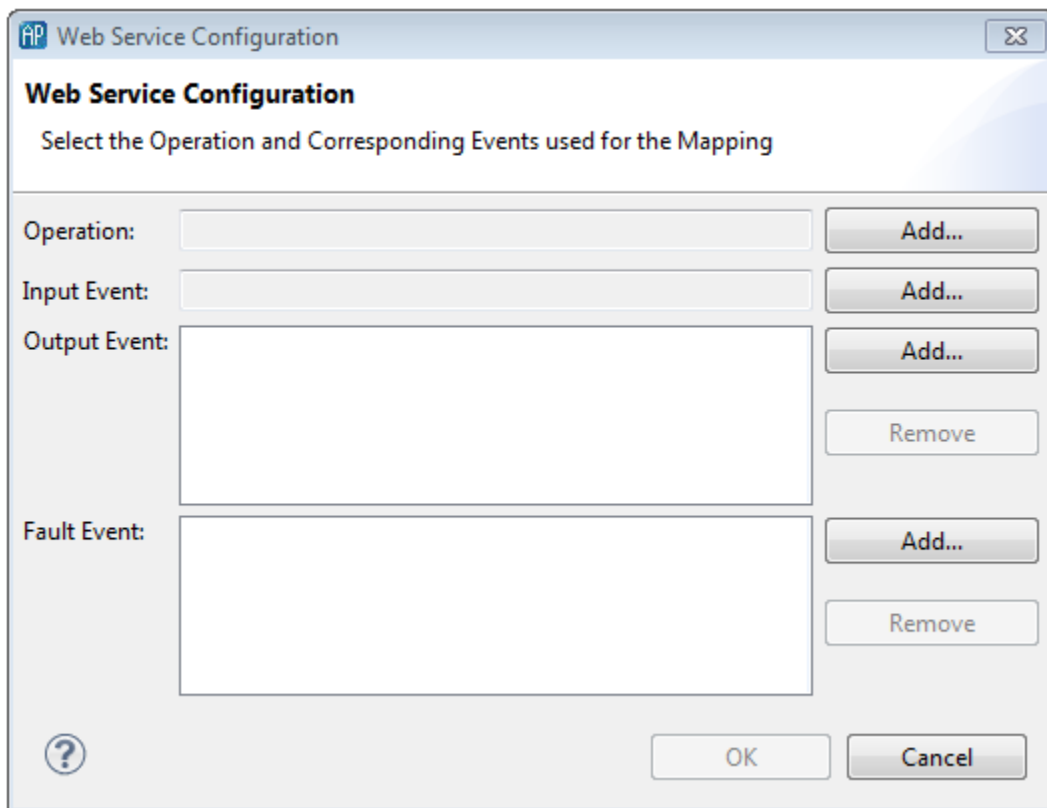
1. In the Project Explorer, expand the project's `Adapters` node and then expand the `WebServices Client Adapter` node.
2. Double-click the entry for the adapter instance you want to configure. This opens the adapter instance configuration in the Web Service adapter editor, showing `CLASSPATH` and other standard properties provided by the adapter's resource bundle.



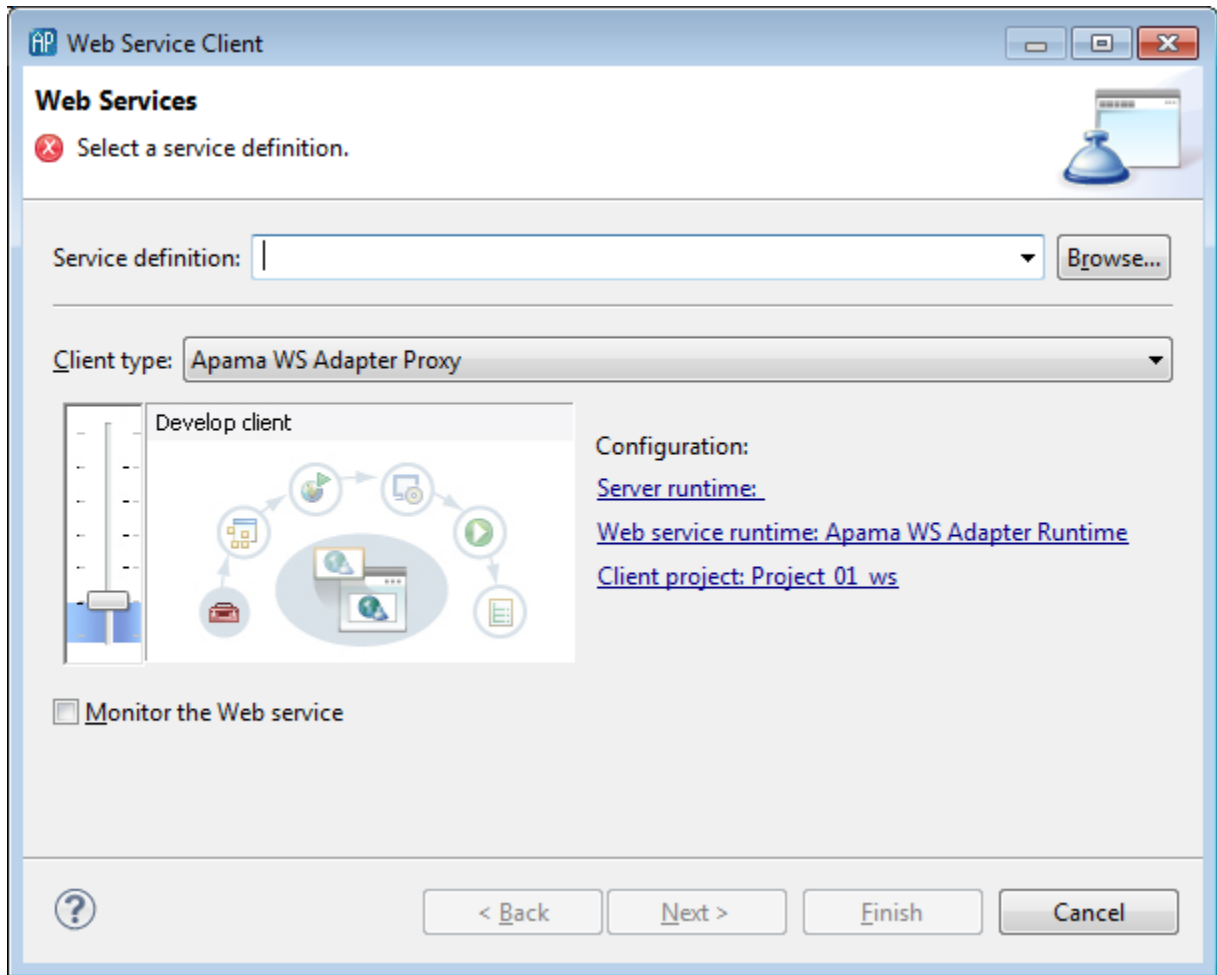
3. In the Web Service adapter editor, select the Event Mappings tab.



- On the adapter editor's Event Mappings tab, click the Add button (). This displays the **Web Service Configuration** dialog.

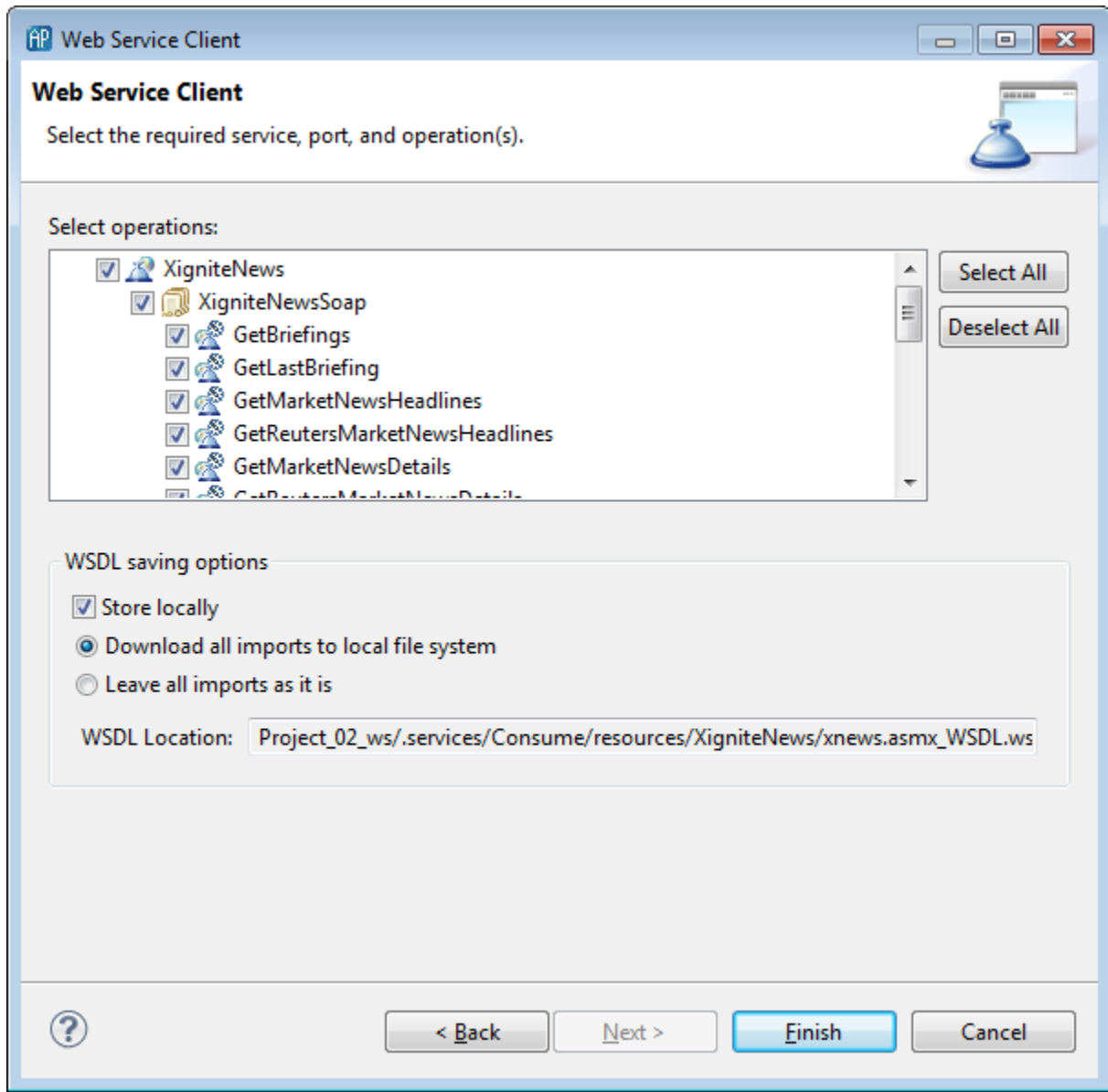


- In the **Web Service Configuration** dialog, to the right of the Operation field, click the Add button and select Create New. This displays the **Web Service Client** wizard.



Note, if you already configured a Web Service in your project, when you click the Add button, you can select Choose from existing instead; for more information, see ["Editing Web Services Client adapter configurations" on page 74](#).

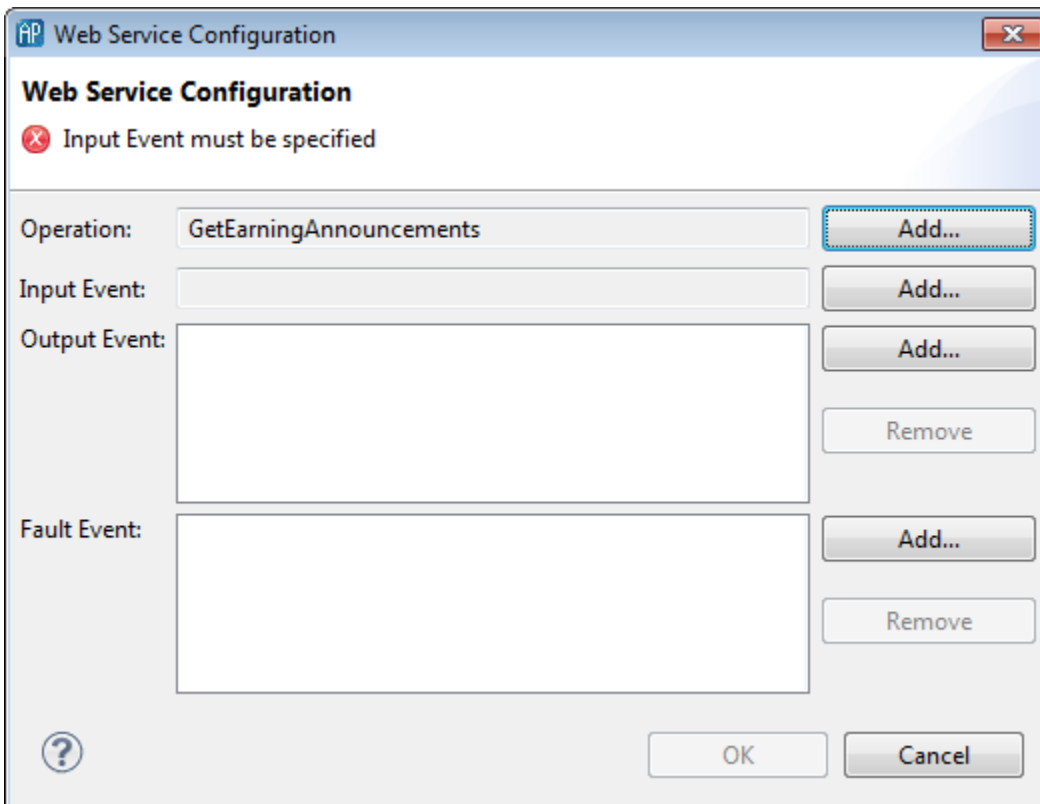
6. In the **Web Service Client** wizard, in the Service definition field, specify a valid URL for the Web Service Definition File (WSDL) that defines the Web Service. This can be a local file or a file at a remote location. Be sure to use the complete syntax when specifying the location for the WSDL file including `file:///` or `http://`. Note, if you click the Browse button, you can select a WSDL file located in your Workspace.
7. In the Client type field, make sure that `Apama Adapters Proxy` is selected from the drop-down list (this is the default).
8. Click **Next**. The second page of the **Web Service Client** wizard opens, showing all the operations that are defined in the WSDL file. The operations are shown with SOAP and SOAP 1.2 bindings depending on how they are defined in the WSDL file.



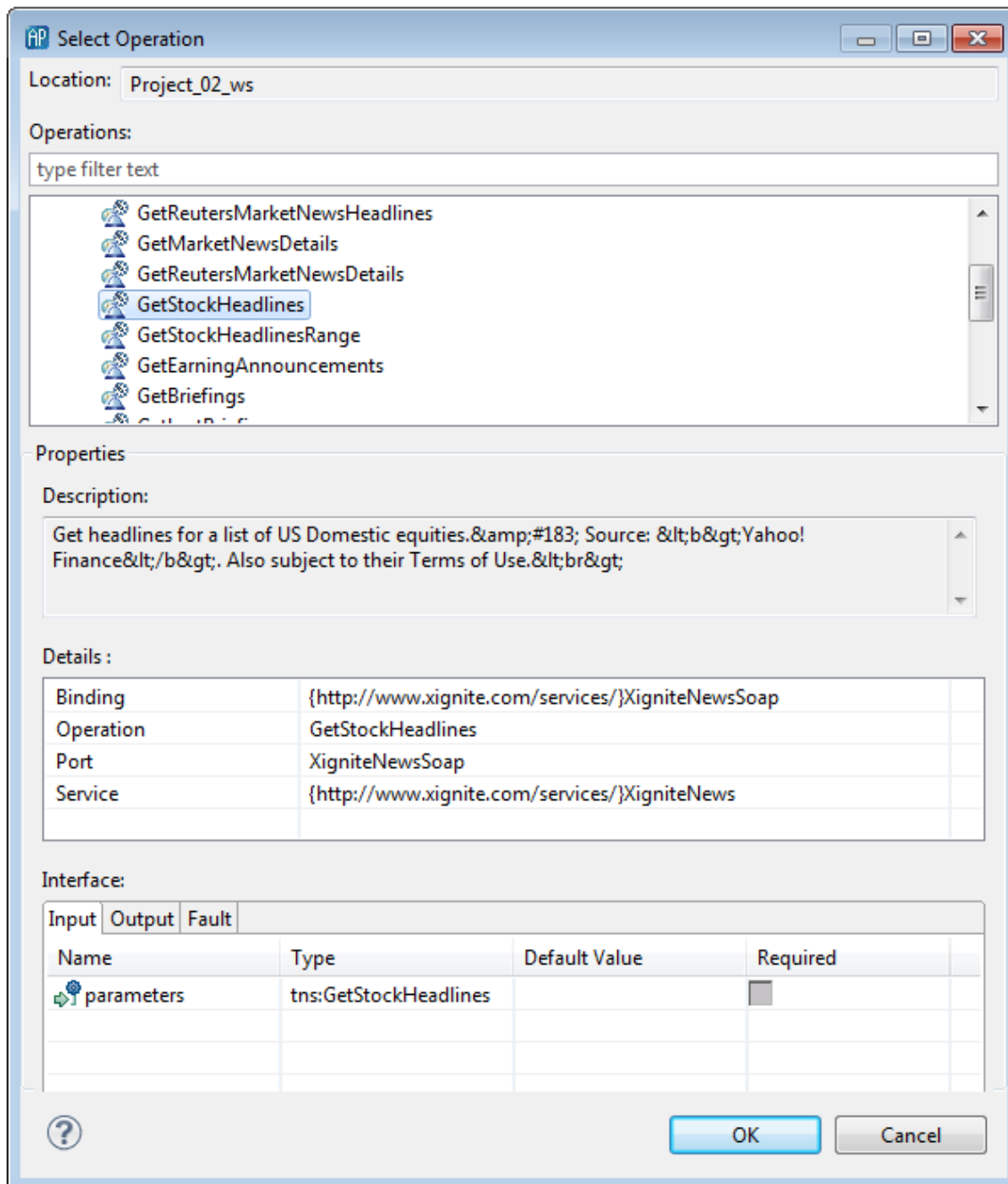
If you want your application to invoke all operations, click Finish instead of Next and skip the next two steps. You are done with this part of the configuration.

9. In the second page of the **Web Service Client** wizard, select the service operation(s) you want your application to invoke.
10. Click Finish.

If you selected one operation the wizard adds it to the Operation field of the **Web Service Configuration** dialog.



If you selected more than one operation the wizard displays the Select Operation dialog, which lets you specify the Apama events that will be mapped to Web Service messages.

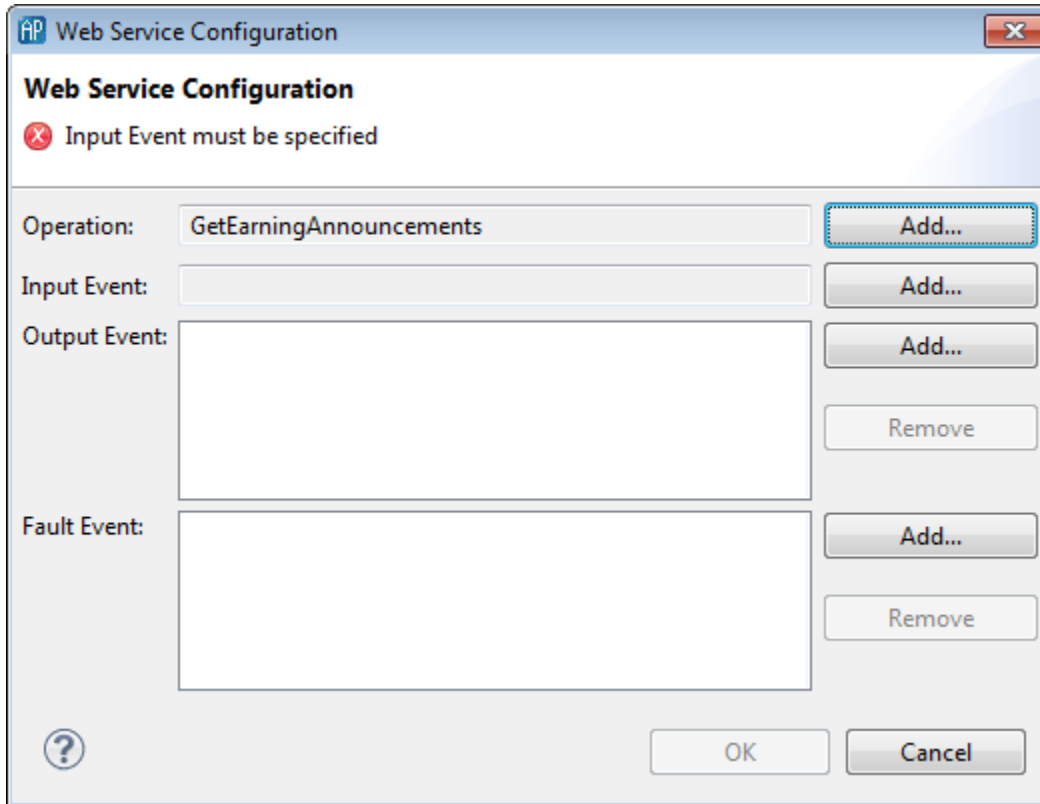


Whether you selected one operation or multiple operations, the next step is to specify the Apama events that will be mapped to Web Service messages; for more information, see ["Specifying Apama events for mapping" on page 71](#).

[Configuring a Web Services Client adapter](#)

Specifying Apama events for mapping

After you specify the Web Service operation your application will invoke, you need to indicate the Apama events with which your application will interact with the messages used by the Web Service operation.

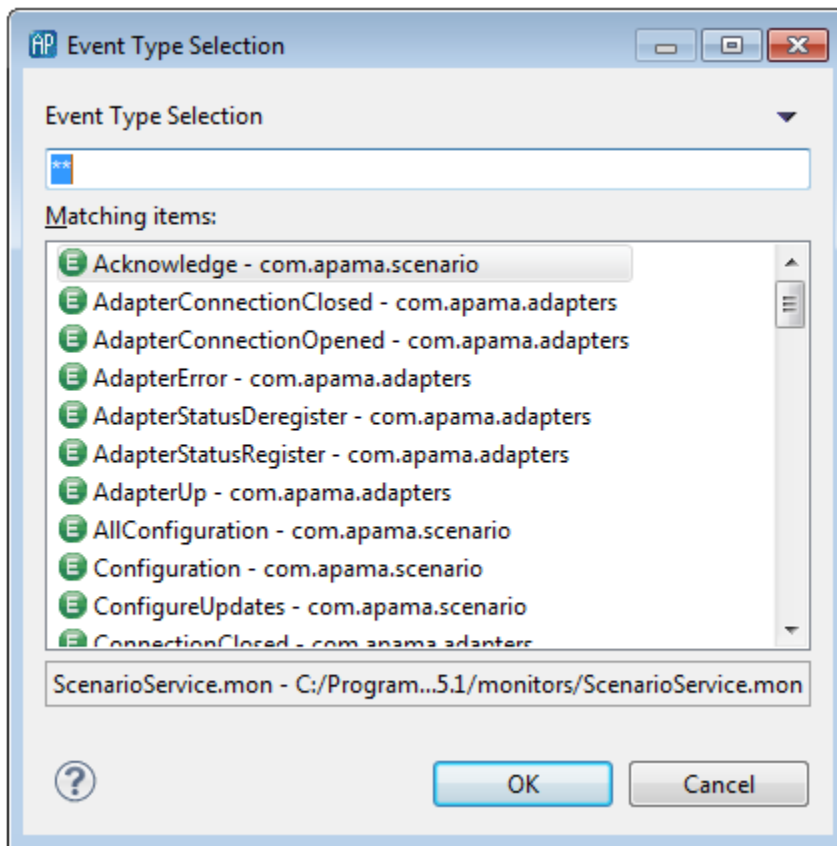


For each operation that you add:

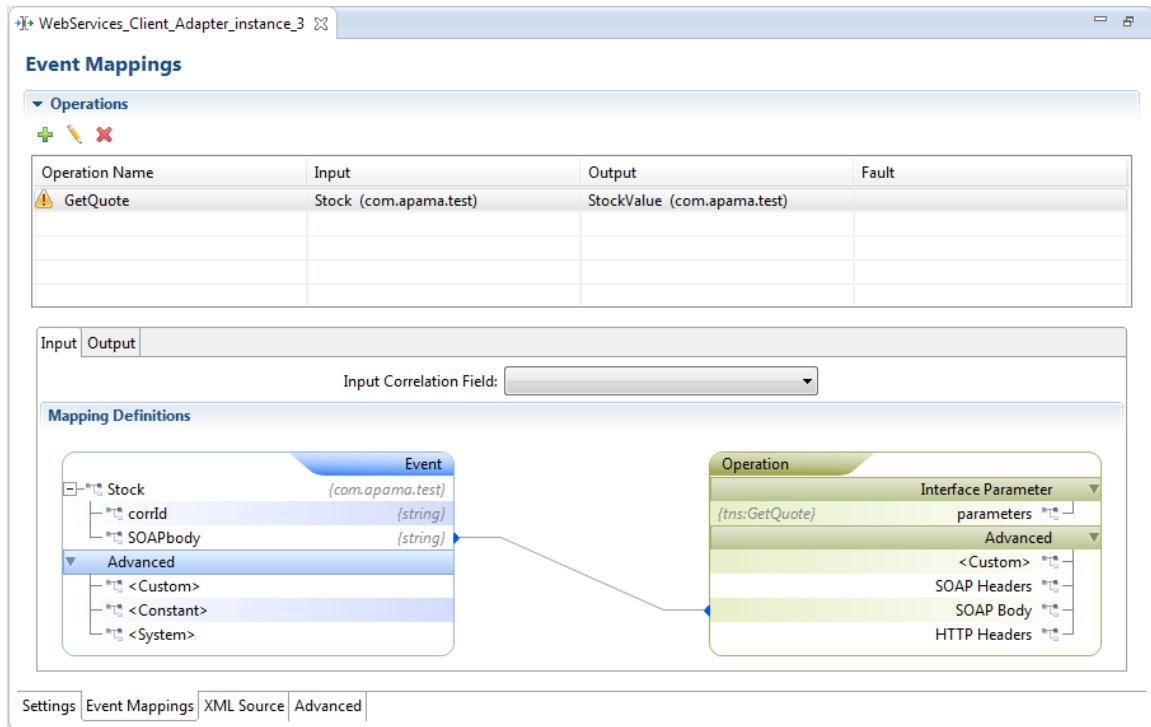
- You must specify an input event. You will map the fields in this input event to the parameters in the input message required to invoke the operation. An Apama application sends an input event to invoke a Web Service operation. You can add the same operation to the adapter more than once. Each time you add the same operation you can specify a different input event or the same input event.
- You can optionally specify output events. If you specify output events, you will map the parameters of the Web Service operation response message to the fields of the events. An Apama application receives an output event when a Web Service sends a response message as the result of an operation request.
- You can optionally specify fault events. If you specify fault events, you will map the parameters of the error message the Web Service might send to the fields of the events. An Apama application receives a fault event if there is an error during invocation of a Web Service operation.

To specify Apama events for mapping:

1. In the **Web Service Configuration** dialog, to the right of the Input Event field, click the Browse button. This displays the **Event Type Selection** dialog.



- a. In the **Event Type Selection** dialog's Event Type Selection field, enter the name of the event. As you type, event types that match what you enter are shown in the Matching Items list.
 - b. In the Matching Items list, select the name of the event type you want to use as the Input Event. The name of the EPL file that defines the selected event is displayed in the status area at the bottom of the dialog.
 - c. Click OK. The **Web Service Configuration** dialog is again displayed, showing the event you selected in the Input Event field
2. If your application will use output messages or fault messages from the Web Service, specify the Apama events that will be associated with those message types in the Output Event and Fault Event fields of the **Web Service Configuration** dialog. For output and fault messages, you can add multiple Apama event types to those fields by selecting the event type and clicking Add.
 3. In the **Web Service Configuration** dialog, click OK. Information about the specified Web Service operations and the associated Apama events is displayed in the Operations section of the adapter editor. In the Mapping Definitions section, the editor displays an Input tab and, if you have specified output events or fault events, Output or Fault tabs.




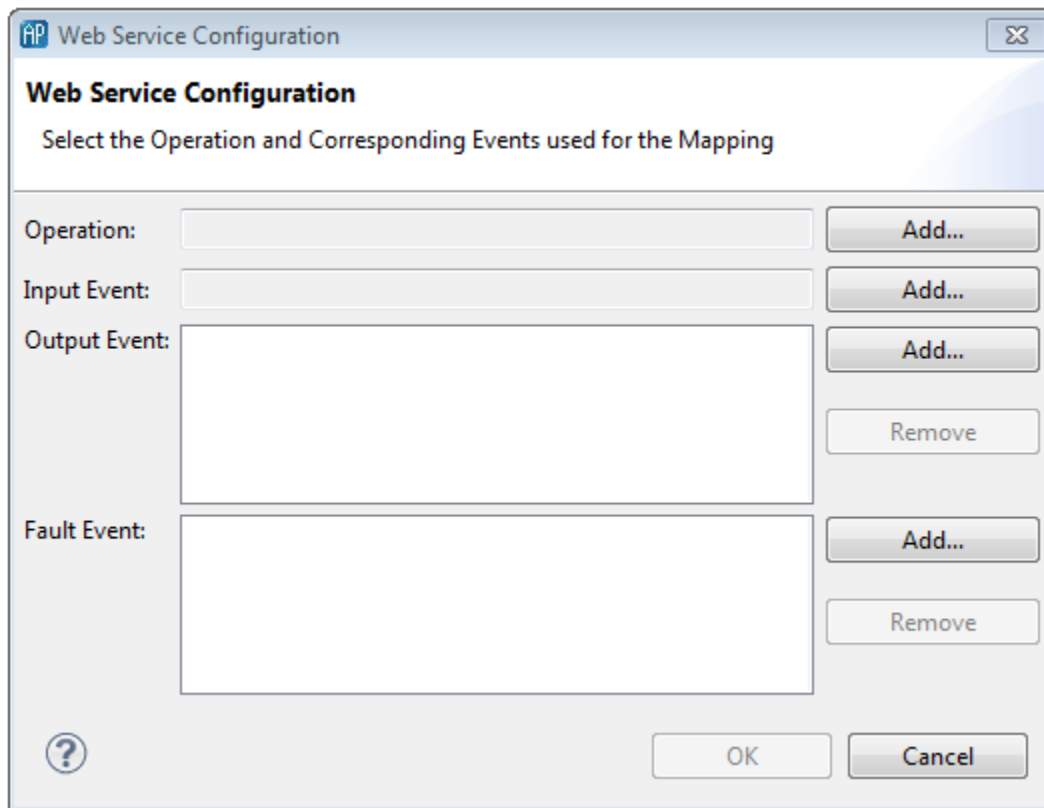
The next step is to map the fields of the Apama events to the parameters in the Web Service operations; for more information, see ["Mapping Web Service message parameters" on page 78](#).

Configuring a Web Services Client adapter

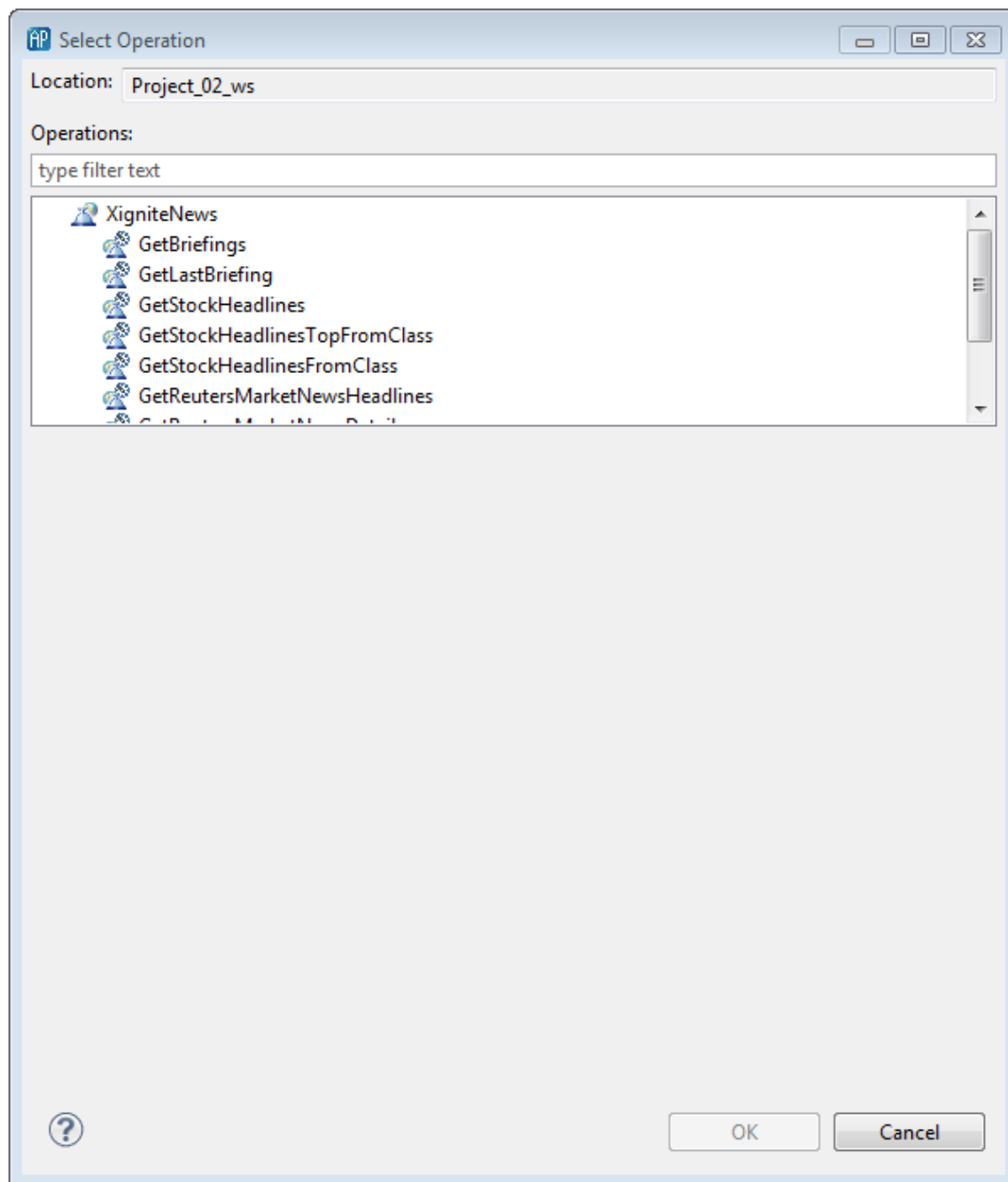
Editing Web Services Client adapter configurations

If you have already configured an instance of the Web Services Client adapter and you want to configure another operation for the application to invoke, add the operation to the adapter configuration as follows:

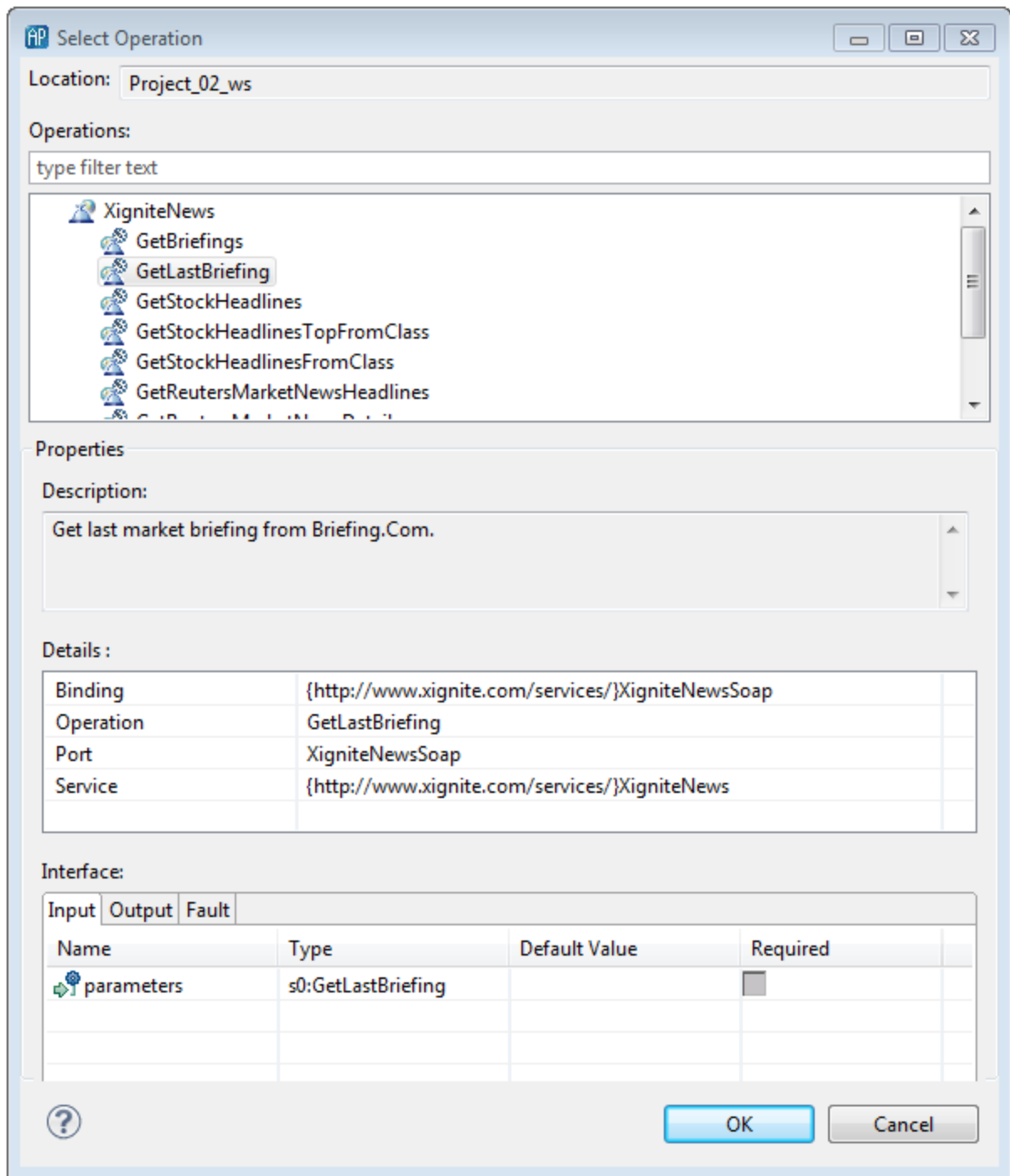
1. In the Project Explorer, expand the project's `Adapters` node and then expand the `WebServices Client Adapter` node.
2. Double-click the entry for the adapter instance you want to configure. This opens the adapter instance configuration in the Web Service adapter editor.
3. In the Web Service adapter editor, select the Event Mappings tab.
4. On the adapter editor's Event Mappings tab, click the Add button (). This displays a blank **Web Service Configuration** dialog.



5. In the **Web Service Configuration** dialog, to the right of the Operation field, click the Add button and select Choose from existing. This displays the **Select Operation** dialog.



6. In the **Select Operation** dialog, in the Operations field, select the operation to invoke. The display in the dialog is updated to show the properties associated with the operation.



- In the **Select Operation** dialog, click OK.

The operation you selected is added to the Operations field of the **Web Service Configuration** dialog.

- Specify the Apama events to use to interact with the operation's Input messages and, if desired, Output and Fault messages as described in ["Specifying Apama events for mapping" on page 71](#).

Using the Apama Web Services Client Adapter

Adding multiple instances of the Web Services Client adapter

You can add multiple Web Services Client adapter instances to an Apama project. Apama Studio generates service monitors and configuration files that are specific to each adapter instance. Different instances can be used, for example, to invoke different Web Service operations and map different Apama events to Web Service input, output, and fault messages. The generated service monitors contain event listeners for the events used in the mappings of the specific adapter instance. If the same event is used in as an input event for multiple adapter instances, multiple events will be emitted to the corresponding channels of each adapter.

To add another adapter instance to a project:

1. In the Project Explorer, expand the project's `Adapters` node.
2. Right-click the `WebServices Client Adapter` node and select **Add Instance** from the pop-up menu. The **Add Instance** dialog opens.
3. In the **Add Instance** dialog, accept the default adapter instance name or give it a new one and click OK. The instance is added to the `WebServices Client Adapter` node.

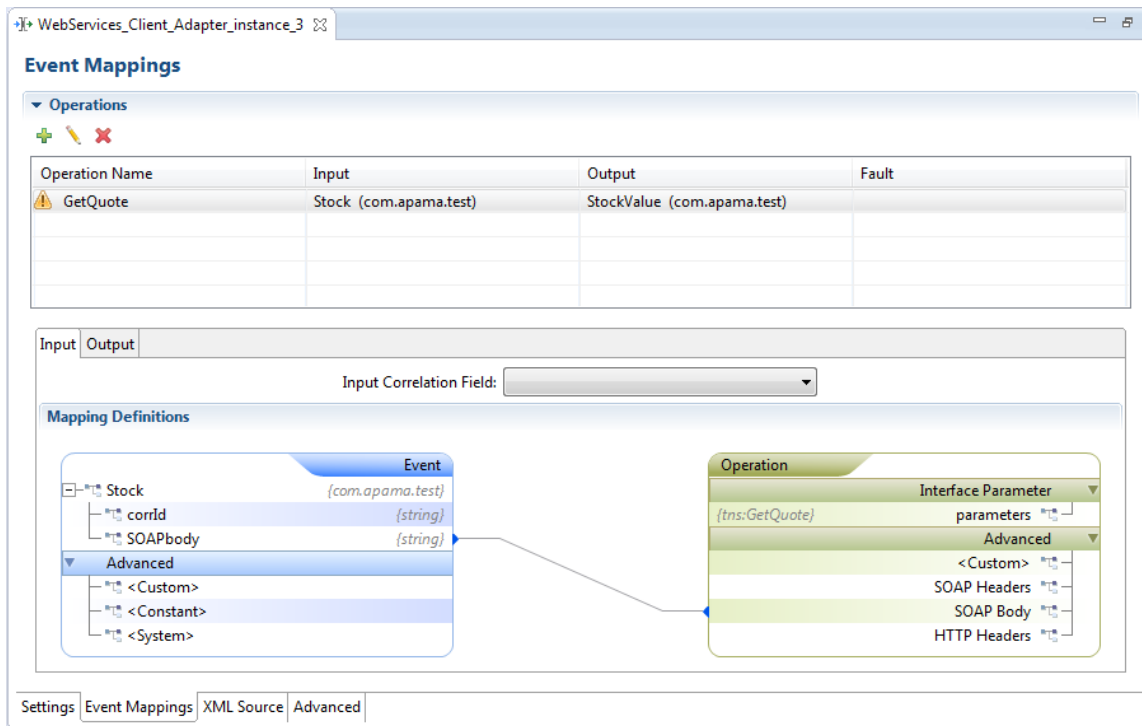
[Using the Apama Web Services Client Adapter](#)

Mapping Web Service message parameters

After you specify which Apama events you want to map to the Web Service messages, you need to create mapping rules that associate Apama event fields with parameters in the Web Service messages. The Apama Studio adapter editor provides a visual mapping tool to create the mapping rules. Web Service messages fall into three categories, each of which can be mapped to Apama events.

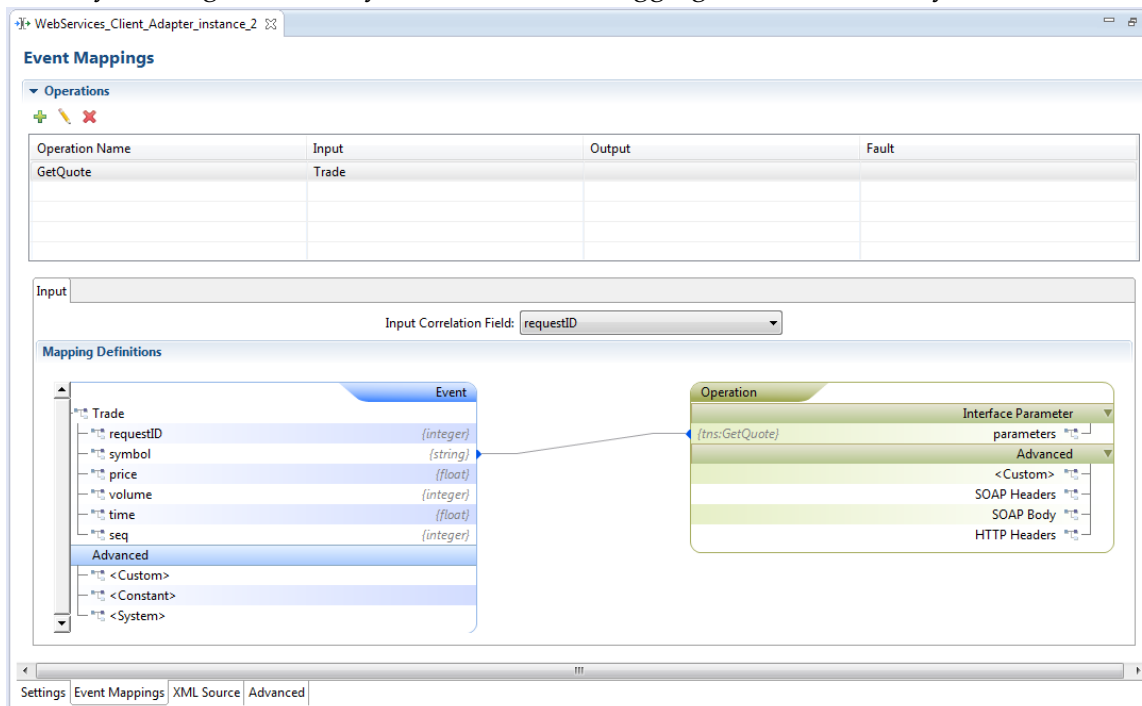
- Input mapping specifies how fields in an Apama event are connected to the parameters in a Web Service input message. These are also known as request messages.
- Output mapping specifies how parameters in a Web Service output message are connected to the fields in an Apama event. These are also known as response messages.
- Fault mapping specifies how Apama event fields and Web service parameters are connected when a fault message is delivered.

When you select an operation in the adapter editor's Event Mappings tab, in the Operations Name column, the Mapping Definitions section of the editor displays hierarchical representations of the Apama event and the Web Service operation. In Apama Studio, the source tree is on the left and the target on the right. For input event messages, the Apama event is the source and the Web Service message is the target; for output and fault messages the Web Service is the source and the Apama event is the target.



There is a tab for each type of Apama event specified: input, output, and fault events. You must specify an input event for each operation that you add. You can add the same operation more than once. If you do, you can specify a different input event or the same input event for each additional instance of the operation. Optionally, you can specify one output event and/or one fault event for each operation instance that you add.

On the adapter editor's Event Mappings tab, in the Mapping Definitions section, you specify the mapping rules by clicking on an entity in one tree and dragging a line to the entity in the other tree.



In the Mapping Definitions section, for input messages, the Apama event fields are displayed to the left and the Web Services operation parameters on the right. To create a mapping rule, click on the event field and drag a line to the desired operation parameter.

There are several approaches for how to map Apama event fields to the parameters in Web Services messages, depending on the complexity of the events and the message parameters.

- **Simple** - Use this approach when a simple Apama event field can be associated with a corresponding type in the Web Service message. A simple Apama event field is of type `integer`, `float`, `boolean`, `decimal`, or `string`. For example, you can use the simple approach to map a `string` event field to a message parameter of type `xsd:string`.
- **Convention-based** - Use this approach when the structure of an Apama event corresponds directly with the XML structure of the Web Service message. The Apama event must have been defined by following the conventions described in ["Convention-based XML mapping" on page 167](#). In this case, Apama (at runtime) automatically converts the event instance to the request XML structure of the Web Service, and also converts the Web Service response XML to an event instance.
- **Template-based** - Use this approach when the XML definition of the Web Service message contains a complex type. You must supply an XML template file that defines the complex type. In the template file, you use variables that will be replaced with values from the Apama event fields. You then map event fields to the variables in the template.
- **Combination** - This approach combines the convention-based and template-based approaches. Use this approach when the adapter can automatically convert at least one event field (`event` or `sequence` type) to XML that models a message parameter and when at least one of the values of the converted fields serves as a value for a variable in a template.

You can also specify if a mapping rule requires a an XPath or XSLT transformation. For more information on how to specify a transformation type, see ["Specifying transformation types" on page 94](#).

The visual mapping tool allows you to add custom entries to the SOAP Headers, HTTP Headers, and SOAP Body if the Web Service requires it. For more information on customizing mapping rules, see ["Customizing mapping rules" on page 96](#).

When you save a Web Service configuration, Apama Studio generates the XML files used to interact with the Web Services and the appropriate service monitors. See ["Web Services Client adapter artifacts" on page 105](#) for more information.

Using the Apama Web Services Client Adapter

Simple mapping

When creating a rule for mapping an Apama event field that contains a simple type (`integer`, `float`, `decimal`, `boolean`, or `string`) to a Web Services parameter that contains a similar type, you can drag a line between the elements as follows:

1. In the Web Services Client adapter editor, display the Event Mapping tab.
2. For each mapping rule, click on the entity you want to map and drag a line to the entity you want to map it to. You must ensure that the types of the two entities match. For example, an Apama `string` type field must map to an XML `xsd:string` field.

Apama Studio represents each rule with a blue line between entities.

Convention-based XML mapping

Convention-based mapping allows XML documents to be created or parsed based on a document structure encoded in the definition of the source or target Apama event type.

The first stage when using convention-based mapping is to examine the structure of the XML document, and create an event definition to represent its root element, with fields for each attribute, text node, sub-element or sequence (of attributes, text nodes or sub-elements). The actual names of the event types are not important, but the event field names and types must follow the following conventions:

- XML attributes can be represented by any EPL simple type such as `string`, `integer`, etc. The name used should be preceded by an underscore, for example `boolean _flag;`.
- XML text nodes are represented by either:
 - A field inside an Apama event representing the parent of the element containing the text, named after the element that encloses the text such as `string myelement;`. This avoids the need to create an event type to represent the element in cases where the element only contains a text node, and no attributes or children. The field type may be any primitive EPL type (for example `string`, `integer`).
 - A field inside an Apama event representing the element that directly contains the text, named `xmlTextNode`. This is necessary in cases where an Apama event type is needed to represent the element so that attributes and/or child elements can also be mapped. The field type may be any primitive EPL type (for example `string`, `integer`).
- XML elements containing attributes or sub-elements of interest are represented by a field of an event type which follows these same conventions. The event type can have any name, but the field must be named after the element, for example, `MyElementEventType myelement.`
- XML attributes, text nodes or elements which may occur more than once in the document are represented by a sequence field of the appropriate primitive or event type, named after the element, for example, `sequence<string> myelement` or `sequence<MyElement> myelement.`

Some special cases to be aware of when naming fields to match element/attribute names are:

- XML nodes which are inside an XML namespace are always referenced by their local name only (the namespace or namespace prefix is ignored).
- XML node names that are Apama EPL keywords (such as `<return>`) must be escaped in the event definition using a hash character, for example, `string #return;`. When generating an XML document, each field in the event will be processed in order and used to build up the output document. When parsing an XML document, each field in the event will be populated with whatever XML content matches the field name and type (based on the conventions above); any XML content that is not referenced in the event definition will be silently ignored.
- XML node names containing any character that is not a valid EPL identifier character (anything other than `a-z`, `A-Z`, `0-9` and `_`) must be represented using a `$hexcode` escape sequence. Of the characters that are not valid EPL identifier characters, only the hyphen and dot are supported. Note that the hexcode based escape sequences are case sensitive. For representing the hyphen or dot use the following:
 - Hyphen '-' is represented as `$002d`.

- Dot '.' is represented as \$002e.

Limitations of convention-based XML mapping

In this release it is not possible to generate documents that contain elements in different XML namespaces (although when parsing this is not a problem).

The following limitations apply to the Apama event definitions that can be used to generate XML:

- Dictionary event field types are not supported
- If an event field is of type `sequence`, the sequence can contain simple types or events. The sequence cannot contain sequences of sequences or sequences of dictionaries

Mapping Web Service message parameters

Convention-based Web Service message mapping example

As an example of using convention-based mapping, consider the following XML documents, which define a request message and a response message:

```
<WSRequest decisionServiceName="dsName"
  xmlns="urn:WSService">
  <WorkDocuments messageType="FLAT">
    <Node id="idAttrValue1">
      <id>id1</id>
      <isLeaf>true</isLeaf>
      <isRoot>false</isRoot>
      <child href="href1"></child>
      <child href="href2"></child>
      <parent href="href1"></parent>
      <parent href="href2"></parent>
      <leaf href="href1"></leaf>
      <leaf href="href2"></leaf>
    </Node>
    <Node id="idAttrValue2">
      <id>id2</id>
      <isLeaf>true</isLeaf>
      <isRoot>false</isRoot>
      <child href="href1"></child>
      <child href="href2"></child>
      <parent href="href1"></parent>
      <parent href="href2"></parent>
      <leaf href="href1"></leaf>
      <leaf href="href2"></leaf>
    </Node>
    <Leaf id="idValue">
      <node href="hrefValue1"></node>
    </Leaf>
  </WorkDocuments>
</WSRequest>

<WSResponse decisionServiceName="dsName"
  xmlns="urn:WSService">
  <WorkDocuments messageType="FLAT">
    <Node id="idAttrValue1">
      <id>id1</id>
      <isLeaf>true</isLeaf>
      <isRoot>false</isRoot>
      <child href="href1"></child>
      <child href="href2"></child>
      <parent href="href1"></parent>
      <parent href="href2"></parent>
      <leaf href="href1"></leaf>
      <leaf href="href2"></leaf>
    </Node>
    <Node id="idAttrValue2">
```

```

    <id>id2</id>
    <isLeaf>true</isLeaf>
    <isRoot>false</isRoot>
    <child href="href1"></child>
    <child href="href2"></child>
    <parent href="href1"></parent>
    <parent href="href2"></parent>
    <leaf href="href1"></leaf>
    <leaf href="href2"></leaf>
  </Node>
  <Leaf id="idValue">
    <node href="hrefValue1"></node>
  </Leaf>
</WorkDocuments>
<Messages version="versionX.Y">
  <Message>
    <severity>Info</severity>
    <text>text1</text>
    <entityReference href="erHref1"></entityReference>
  </Message>
  <Message>
    <severity>Info</severity>
    <text>text2</text>
    <entityReference href="erHref2"></entityReference>
  </Message>
</Messages>
</WSResponse>

```

Following are event definitions that follow the conventions of these XML documents. It is important to understand that the EPL field types directly correspond to the types defined in the XML Schema document used by the request and response documents. Likewise, event field names directly correspond to the element names defined in the XML Schema document.

```

event NodeRef {
    string _href;
}
event NodeType {
    string _id;
    string id;
    boolean isLeaf;
    boolean isRoot;
    sequence<NodeRef> child;
    sequence<NodeRef> parent;
    sequence<NodeRef> leaf;
}
event LeafType {
    string _id;
    NodeRef node;
}
event WorkDocumentsType {
    string _messageType;
    sequence< NodeType > Node;
    sequence< LeafType > Leaf;
}
event WSRequestType {
    string _decisionServiceName;
    WorkDocumentsType WorkDocuments;
}
event MessageType {
    string severity;
    string text;
    NodeRef entityReference;
}
event MessagesType {
    sequence<MessageType> Message;
    string _version;
}
event WSResponseType {
    string _decisionServiceName;
    WorkDocumentsType WorkDocuments;
}

```

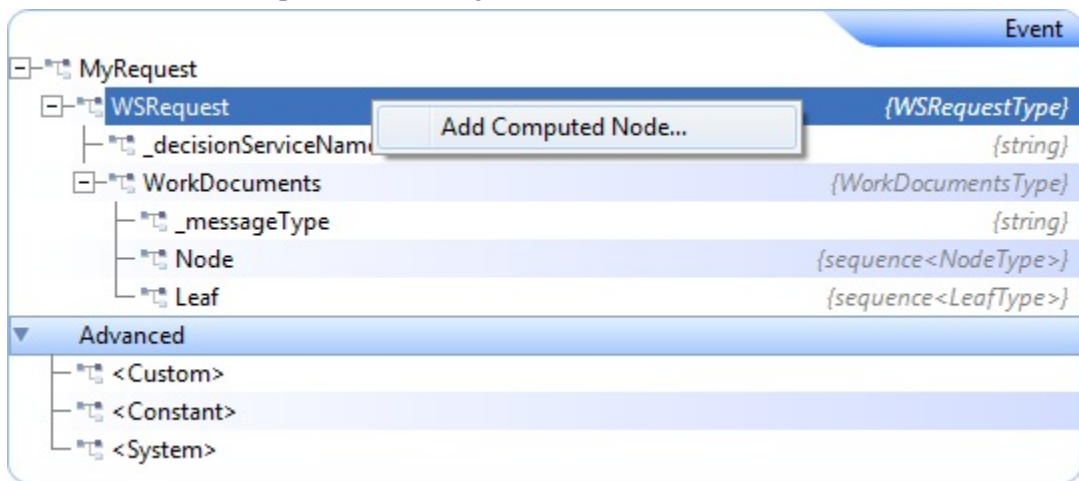
```

    MessageType Messages;
}
event MyRequest {
    WSRequestType WSRequest;
}
event MyResponse {
    WSResponseType WSResponse;
}

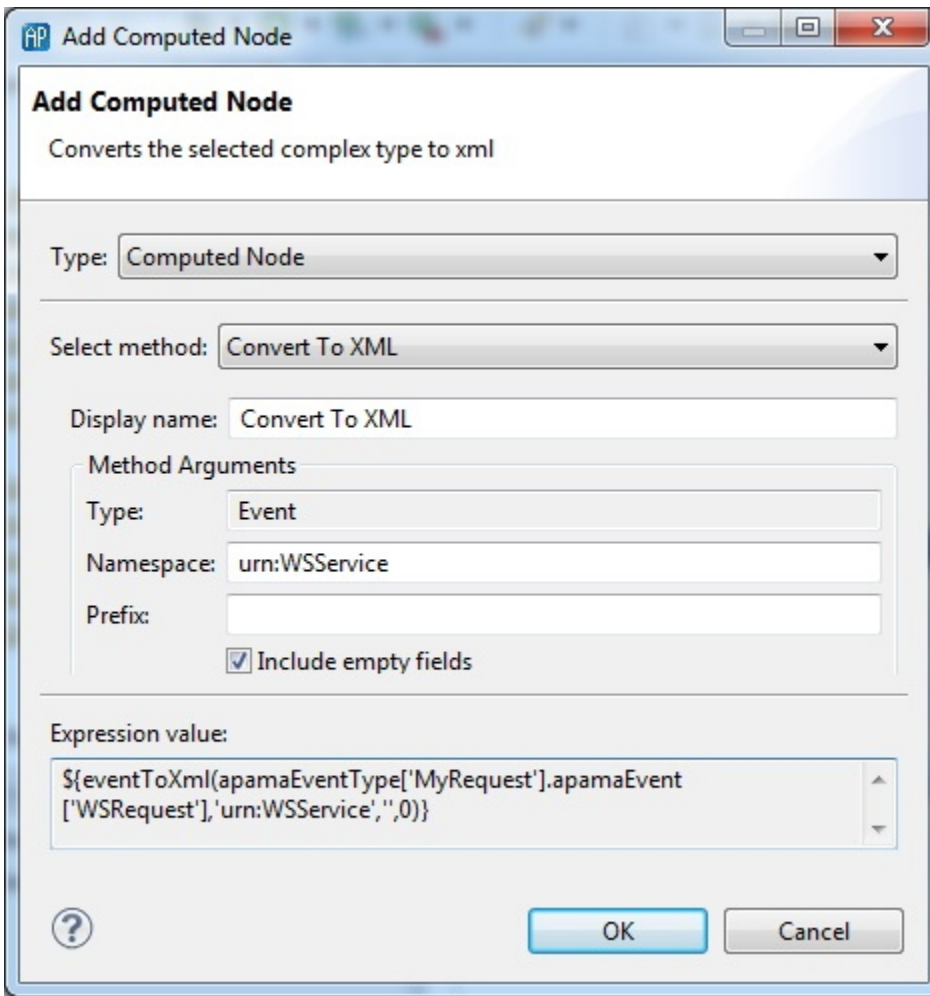
```

To use the convention approach to mapping event fields to message parameters:

1. Create an Apama event with fields that correspond in type and order to the parameters of a Web Service operation.
2. Specify the event as the input, output, or fault event associated with a Web service operation. See ["Specifying Apama events for mapping" on page 71](#).
3. With that operation selected, in the Event Mappings tab, right-click the Apama event and select Add Computed Node. For example, the following tree is in the Input tab:



The **Add Computed Node** dialog appears:



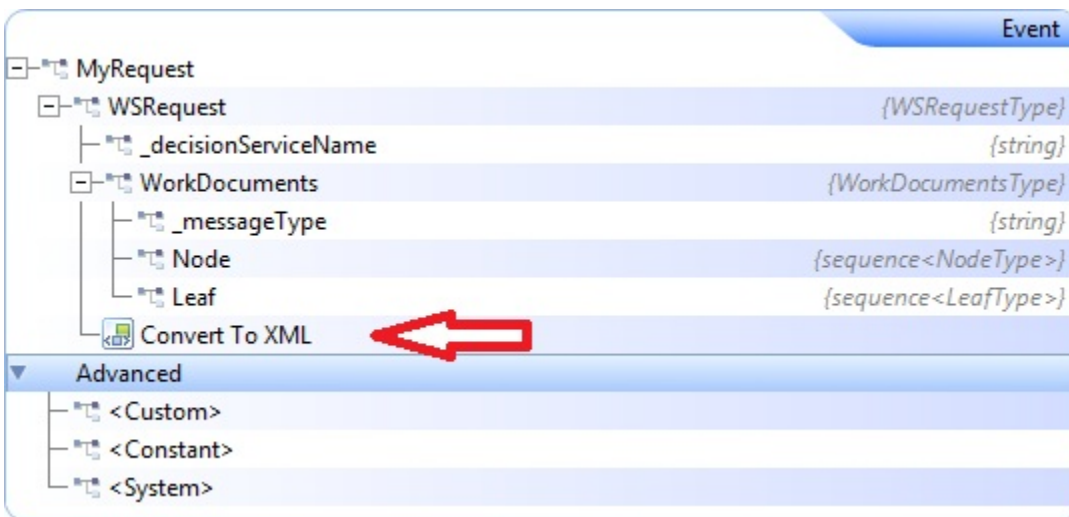
4. In the **Add Computed Node** dialog's Select Method field, select Convert to XML from the drop-down list. The dialog is updated to show more information.

When the adapter generates the request XML structure you can customize the namespace and namespace prefix in the generated XML. If sub-elements in the request XML structure are in different namespaces, you cannot use only the convention mapping approach. You must combine it with the template approach. See ["Combining convention and template mapping" on page 88](#). However, when sub-elements belong to more than one namespace you can use the convention approach without a template to convert the XML structure to an event.

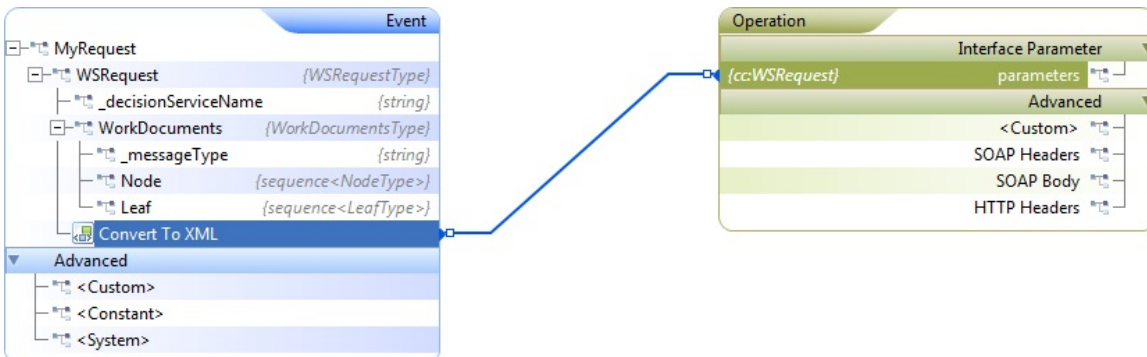
By default the Include empty fields option is enabled. This specifies that empty XML nodes will be generated when empty EPL string fields are encountered within an Apama event. This option does not affect empty strings within a sequence of EPL strings. If you clear the check box to disable the option, empty XML nodes will not be generated.

5. Click OK.

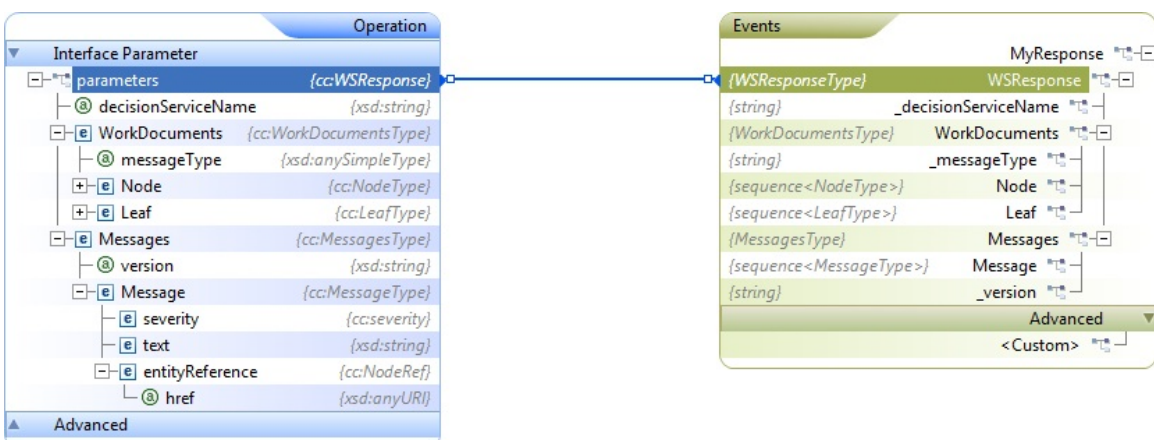
In the mapping tree, Apama Studio adds an entry of type `Convert To XML` to the selected event node. For example:



6. Drag a line from the `Convert To XML` entry to the `parameters` entry. For example:



For the output mapping, map the output Web Service parameter to an event field. If the event field is of an `event` type that models the output XML (per the convention), the adapter automatically creates the event instance at runtime (implicitly) from the XML. Following is an example of mapping an operation's parameter to an output event:



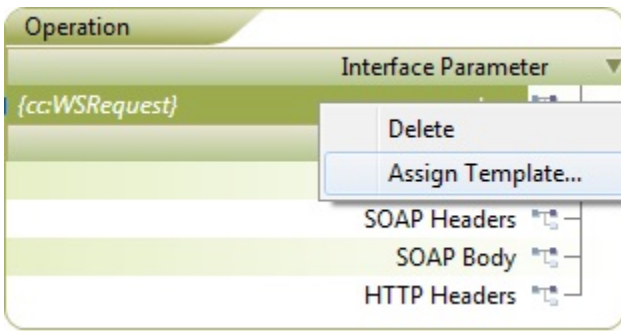
Convention-based XML mapping

Template-based mapping

The template-based approach to mapping lets you map fields in an Apama event to elements and attributes in complex XML structures.

To use the template-based approach:

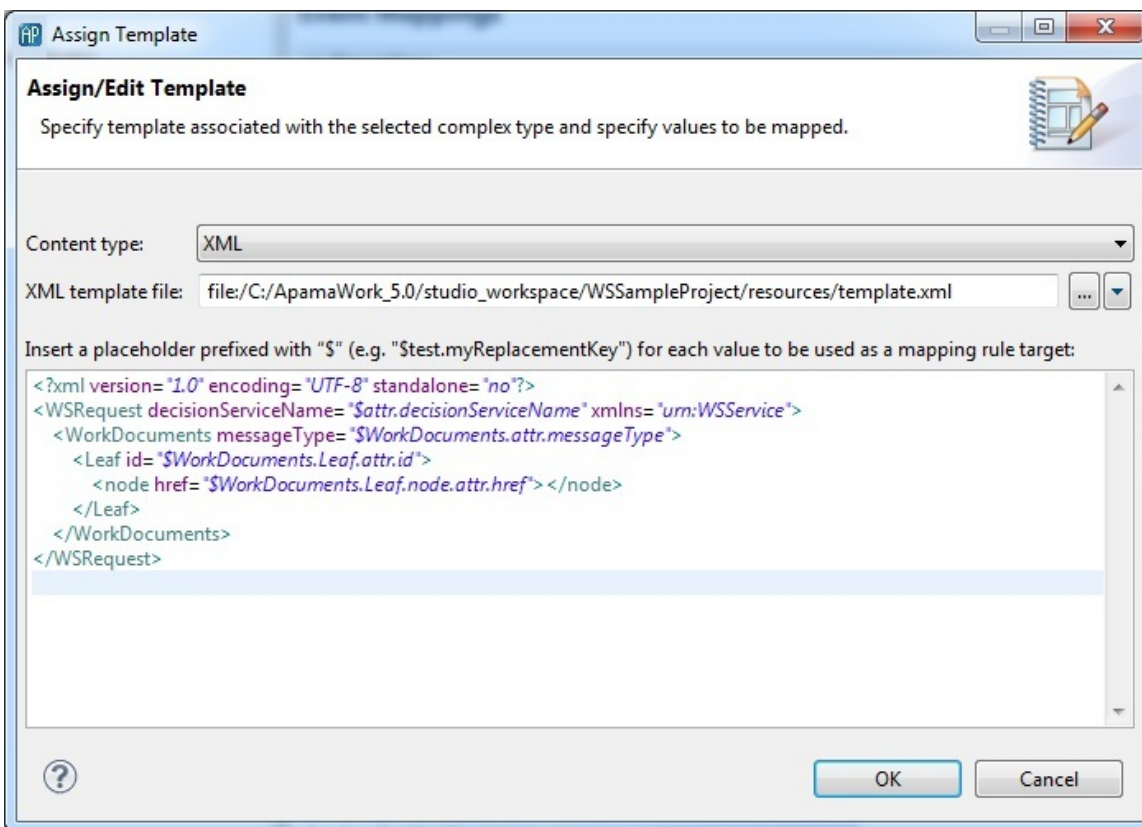
1. In the adapter editor's Event Mappings tab, right-click the operation's parameters entry and select **Assign Template**.



The **Assign Template** dialog appears.

2. In the **Assign Template** dialog's XML Template file field, enter the name of the template file you want to use or use the Browse and Down Arrow buttons to locate the file.

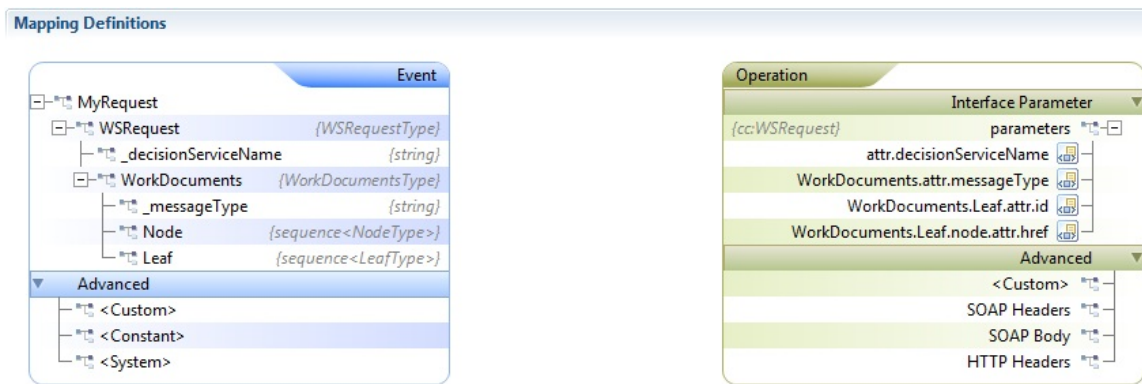
When you specify a template file, the contents of the file are added to the text field in the dialog. For example:



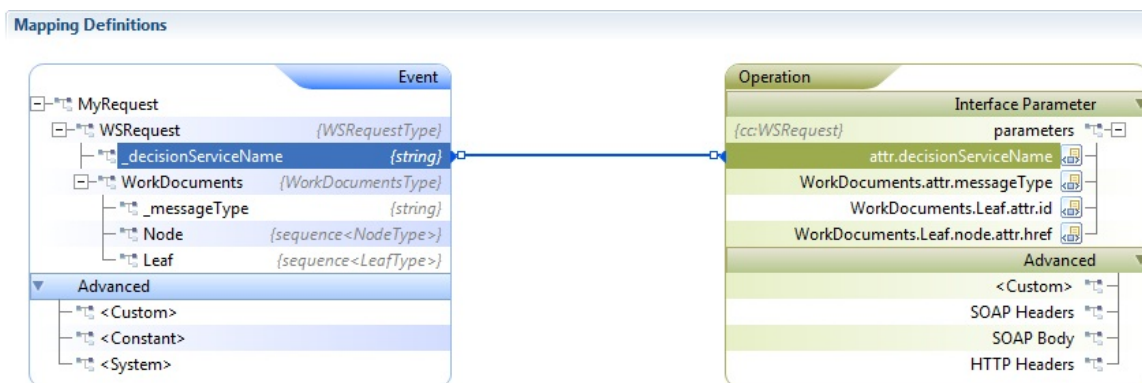
You must have previously written the template file. In the template file, you define variables to represent field values that you want the adapter to obtain from the input event. To define a variable, insert a dollar sign (\$) following by the variable name. After you click OK, the variable name appears as an element in the Operation mapping tree.

3. Edit the template as needed and click OK.

The Operation hierarchical tree is re-displayed showing the various elements and attributes that are defined in the template.



4. In the Event hierarchical tree, click the Apama event field that you want to map to a particular element or attribute in the Operation tree.



Mapping Web Service message parameters

Combining convention and template mapping

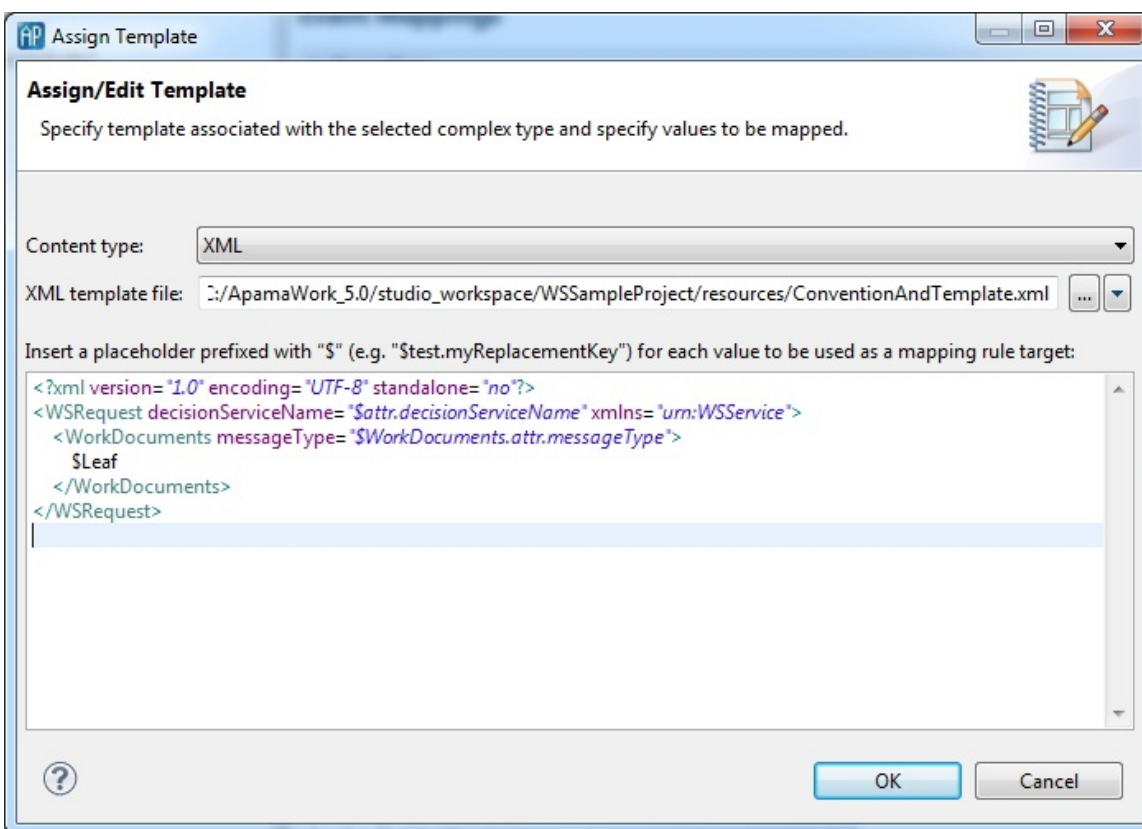
You can combine the convention-based and template-based mapping approaches. In this case, you supply an XML template file and you also use the adapter to automatically convert one or more event fields (of `event` or `sequence` type) to XML. The value of one or more converted fields supplies the value for a variable in the template file.

To use the combined mapping approach:

1. In the adapter editor's Event Mappings tab, right-click the operation's parameters entry and select Assign Template. The **Assign Template** dialog appears.
2. In the **Assign Template** dialog's XML Template file field, enter the name of the template file you want to use or use the Browse and Down Arrow buttons to locate the file.

When you specify a template file, the contents of the file are added to the text field in the dialog.

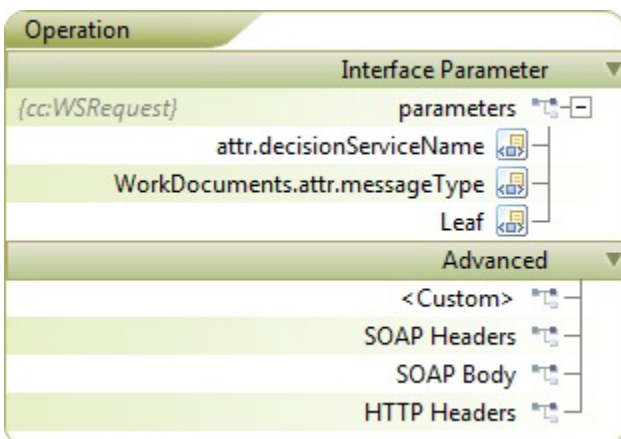
3. Edit the template file to create at least one variable that will get its value from a field in the input event. For example, suppose you are working with the `WSRequest` and `WSResponse` messages described in ["Convention-based Web Service message mapping example" on page 82](#). You might edit the template to create a variable that will get its value from `Leaf` elements.



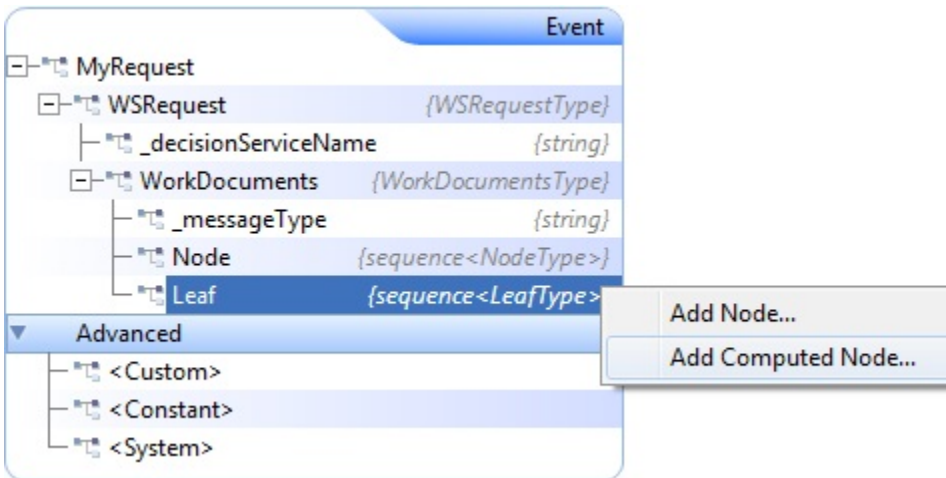
Remember that you must write or have previously written the template file.

4. Edit the template as needed and click OK.

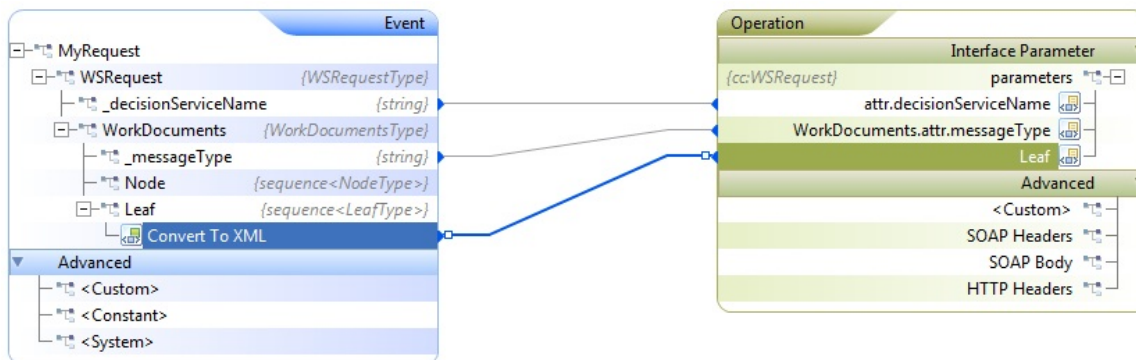
The Operation hierarchical tree is re-displayed showing the various elements and attributes that are defined in the template, including any variables.



5. In the Event hierarchical tree, right-click the Apama event field that contains a field you want to map to a variable and click Convert Into XML. For example:



6. Right-click an event field in the converted field drag a line from that field to the template variable element in the Operation tree. The event field you select provides the value for the target variable. For example, map the converted Leaf sequence to the Leaf variable:



Mapping Web Service message parameters

Mapping complex types

When mapping to a complex Web Service parameter, one option for specifying the input mapping is to provide the entire XML content in an Apama string field, and map that string field to the Web Service parameter. Similarly, in the output mapping, you can map the output Web Service parameter to an Apama string field.

Mapping Web Service message parameters

Difference between doc literal and RPC literal WSDLs

In the **RPC literal** style of WSDL, the XML document that forms the request and response in the SOAP body includes a parent tag of the operation name. All the message parts are provided under that parent tag. On the other hand, the **doc literal** style of WSDL uses only one message part in the SOAP body so the XML document contains just a single message part.

In light of the above, it is important to note that when mapping Apama entities to create the XML request, or mapping the response XML back to Apama, the mapping should always be done to/from

the message parts, regardless of whether it is a doc literal or RPC literal WSDL. The operation name tag for RPC literal WSDLs will be accounted for automatically and should not be supplied in the request by Apama, and it will not be provided in the response to Apama.

Mapping Web Service message parameters

Using custom EL mapping extensions

Apama's Web Services Client adapter and Correlator-Integrated adapter for JMS both use an expression-based mapping layer to map between Apama events and external message payloads. The expressions use Java Unified Expression Language (EL) resolvers and methods, which must be registered to the mapping layer. Apama includes a set of EL resolvers and EL methods that are registered for you and that you can use in mapping expressions. If you want you can register your own EL resolvers and EL methods and then use them as custom mapping extensions.

See the ApamaDoc API reference information for details about the APIs mentioned in the following steps. An example that uses these APIs is in the `samples\correlator_jms\mapping-extensions` folder of your Apama installation directory.

To register and use custom mapping extensions:

1. Define a public class that imports `com.apama.adapters.el.api.ELMappingExtensionProvider` and `com.apama.adapters.el.api.ELMappingExtensionManager`.
2. Implement `ELMappingExtensionProvider`.
3. Override the `ELMappingExtensionProvider.registerExtensions()` method and register each custom EL method and each custom EL resolver with a call to `ELMappingExtensionManager.registerMethod()` or `ELMappingExtensionManager.registerResolver()`, as appropriate. For example:

```
package com.apama.test;

import com.apama.adapters.el.api.ELMappingExtensionManager;
import com.apama.adapters.el.api.ELMappingExtensionProvider;

public class MyStringMethods implements ELMappingExtensionProvider {
    // Register EL methods:
    @Override
    public void registerExtensions(ELMappingExtensionManager manager) {
        throws Exception {
            manager.registerMethod("reverse", getClass().getMethod("reverse", String.class));
            manager.registerMethod("p:prefix", getClass().getMethod("prefix", String.class, String.class));
        }

        public static String reverse(String str) {
            return new StringBuilder(str).reverse().toString();
        }

        public static String prefix(String str, String prefix) {
            if (str != null) {
                return prefix + str;
            } else {
                return prefix;
            }
        }
    }
}
```

4. Register the list of mapping extension providers by adding a `com.apama.adapters.el.config.ELMappingExtensionProviderList` bean to the XML configuration, and setting its `mappingExtensionProviders` property. For example:

```
<bean class="com.apama.adapters.el.config.ELMappingExtensionProviderList">
    <property name="mappingExtensionProviders">
        <list>
```

```

        <bean class="com.apama.test.MyStringMethods"></bean>
        <bean class="com.apama.test.MyIntegerMethods"></bean>
        ...
    </list>
</property>
</bean>

```

The place to set this bean XML snippet is as follows:

- For Correlator-Integrated adapters for JMS, specify the `com.apama.adapters.el.config.ELMappingExtensionProviderList` bean in an existing spring XML file or in a separate file in the same location as other spring files. The recommended location is the `jms-global-spring.xml` file
- For Web Services Client adapters, after Apama Studio generates the `WebServices_instanceName_spring.xml` file, add the `com.apama.adapters.el.config.ELMappingExtensionProviderList` bean. If the `WebServices_instanceName_spring.xml` file must be re-generated, your entry will be overwritten and you will need to re-add it. It is expected that Apama Studio will be enhanced in a future release to avoid the need to re-add this bean.

5. Use mapping extensions in expressions inside the source expressions of mapping rules for both send and receive mappings.

For example, consider a custom static method that takes a string parameter, returns the reverse string, and is registered with the name `my:reverse`. You can use it in a mapping rule as follows:

```

<mapping:rule
    source="${my:reverse(apamaEventType['test.MyMessage'].apamaEvent['body'])}"
    target="${jms.body.textmessage}" type="BINDING_PARAM"/>

```

In this example, `my:reverse` is applied to the expression

`"apamaEventType['test.MyMessage'].apamaEvent['body']"`. This means that the value of the input parameter for the `my:reverse` method will be the value returned by the expression `"apamaEventType['test.TextMessage'].apamaEvent['body']"`, which returns the value of the `"body"` field of the `"test.MyMessage"` event. The result is that the value of the source expression `"my:reverse(apamaEventType['test.MyMessage'].apamaEvent['body'])"` will be the reverse of the string contained in the `"body"` field.

You can use Apama Studio to add custom expressions to event mappings. In the Event Mappings tab of your adapter editor, right-click the `<Custom>` node and select `Add Node....` This displays the `Add Node` dialog, which prompts you to enter a custom expression.

6. Ensure that the `jar` file that contains your mapping extension providers is on the appropriate classpath.

For a Correlator-Integrated adapter for JMS, use a `<jms:classpath>` element to enclose the `ELMappingExtensionProviderList` bean.

For a Web Services Client adapter, the `jar` file must be on the adapter's classpath.

Mapping Web Service message parameters

JUEL mapping expressions reference for Web Client Services adapter

In JUEL mapping expressions, you can use certain string methods in the parts of the mapping expressions that evaluate to `string` types. The table below describes the `string` methods you can use. These methods use the same-named `java.lang.String` methods. The mapping expressions are evaluated first to obtain a result string and then any specified string method is applied. You use these functions in the following way:

```

${some_expression.substring(5)}

```

In the previous format, *some_expression* is an expression that evaluates to a string. In the following examples, *f1* is a field of type `string`:

```
${apamaEvent['f1'].toString().contains('in')}  
${jms.body.textmessage.toString().startsWith('sample')}
```

String method	Description
<code>equalsIgnoreCase('str')</code>	Returns a boolean value that indicates whether the result string is equal to the specified string, ignoring case.
<code>contains('str')</code>	Returns a boolean value that indicates whether the result string contains the specified string.
<code>matches('regex')</code>	Returns a boolean value that indicates whether the result string matches the specified Java regular expression.
<code>startsWith('str')</code>	Returns a boolean value that indicates whether the result string starts with the specified string.
<code>endsWith('str')</code>	Returns a boolean value that indicates whether the result string ends with the specified string.
<code>toLowerCase()</code>	Converts the result string to lowercase and returns it.
<code>toUpperCase()</code>	Converts the result string to uppercase and returns it.
<code>concat('str')</code>	Appends the result string with the specified string and returns this result.
<code>replaceAll('regex', 'regexReplacement')</code>	<p>In the result string, for each substring that matches <code>regex</code>, this method replaces the matching substring with <code>regexReplacement</code>. The string with replacement values is returned.</p> <p>The <code>regexReplacement</code> string may contain backreferences to matched regular expression subsequences using the <code>\</code> and <code>\$</code> characters, as described in the Oracle API documentation for <code>java.util.regex.Matcher.replaceAll()</code>. If a literal <code>\$</code> or <code>\</code> character is required in <code>regexReplacement</code> be sure to escape it with a backslash, for example: <code>"\\$"</code> or <code>"\\"</code>.</p>
<code>substring(startIndex, endIndex)</code>	Returns a new string, which is a substring of the result string. The returned substring includes the character at <code>startIndex</code> and subsequent characters up to but not including the character at <code>endIndex</code> .

String method	Description
<code>substring(startIndex)</code>	Returns a new string, which is a substring of the result string. The returned string includes the character at <code>startIndex</code> and subsequent characters including the last character in the string.
<code>trim()</code>	Returns a copy of the result string with leading and trailing whitespace removed.

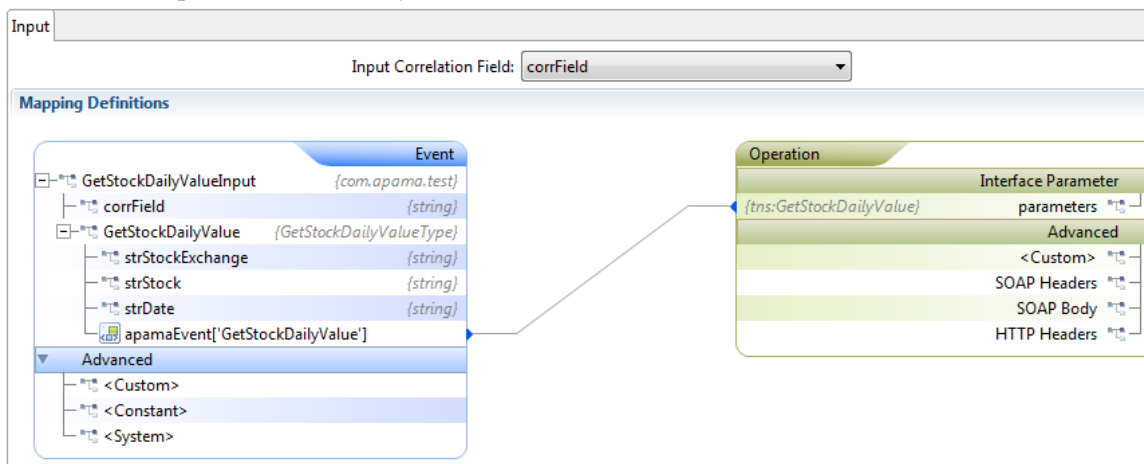
Mapping Web Service message parameters

Specifying a correlation ID field

Because Web Service request (input) and response (output) messages are asynchronous, if an Apama application needs to correlate response messages with a specific request message, you need to specify a field in the Apama event that will contain the correlation ID information.

To use Correlation IDs to associate response messages from Web Services with request messages from Apama applications, specify the name of the event field that will contain the correlation ID by selecting the field in the Input Correlation Field, Output Correlation Field, and Fault Correlation Field drop-down lists. Do this for each event (input, output, and fault) that you associate with the operation. These drop-down lists are located on the InputOutput, and Fault tabs, respectively.

In the following image of an Input Mapping Events tab, you can see the Input Correlation Field at the top of the tab, and the `corrField` field in the input event tree. There is no need to map the correlation ID field. The adapter automatically takes care of this.



Using the Apama Web Services Client Adapter

Specifying transformation types

In the Mapping Element Details section, in the Transformation Type field select the desired type from the drop-down list. You can specify either an XSLT stylesheet or an XPath statement.

Specifying an XSLT transformation type

If the mapping from an Apama event type to a Web Service operation's parameters requires an XSLT transformation, specify the transformation details as follows:

1. If necessary, in the adapter editor's Event Mappings tab, in the Mapping Element Details section, draw the line indicating the mapping from source to target.
2. In the Mapping Element Details section, click on the line that specifies the mapping rule you are interested in.
3. In the Transformation Type field select XSLT Transformation from the drop-down list. This displays the Stylesheet URL field.
4. In the Stylesheet URL field, click Browse [...] to locate the file of the stylesheet to use.

Specifying transformation types

Specifying an XPath transformation type

If the mapping from an Apama event type to a Web Service operation's parameters requires an XPath transformation, specify the transformation details as follows:

1. If necessary, in the adapter editor's Event Mappings tab, in the Mapping Element Details section, draw the line indicating the mapping from source to target.
2. In the Mapping Element Details section, click on the line that specifies the mapping rule you are interested in.
3. In the Mapping Element Details section, in the Transformation Type field select XPath statement.
4. In the XPath Expression field, specify a valid XPath expression. You can either enter the XPath expression directly or you can use the **XPath** builder tool to construct an expression.

To use the **XPath Builder**:

- a. Click the Browse button [...] to the right of the XPath Expression field. The Select input for XPath helper dialog is displayed.
- b. In the Select input for XPath helper dialog, click Browse [...] and select the name of the file that contains a definition of the XML structure (the drop-down arrow allows you to select the scope of the selection process). Click OK. The **XPath Helper** opens, showing the XML structure of the selected file in the left-hand pane.
- c. In the **XPath Helper** build the desired XPath expression by double-clicking on nodes of interest in the left hand pane. The resultant XPath expression is displayed in the XPath tab in the upper right-hand pane.
- d. In the **XPath Helper**, click OK. The **XPath Builder** closes and the XPath Expression field displays the XPath expression you built.

Specifying transformation types

Customizing mapping rules

You can add custom entries to the SOAP Headers, HTTP Headers, and SOAP Body if the Web Service requires it. Create mapping rules for this additional data as described in the following topics.

Using the Apama Web Services Client Adapter

Mapping SOAP body elements

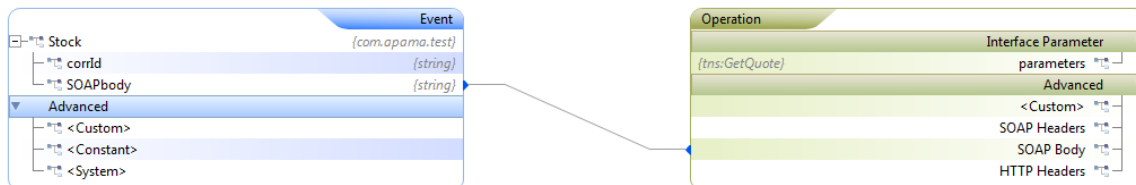
When you invoke an operation you can add custom entries to SOAP body elements if the Web Service requires it. Create mapping rules for SOAP body data as follows.

1. Define Apama events that contain `string` fields to hold the SOAP body data. For example:

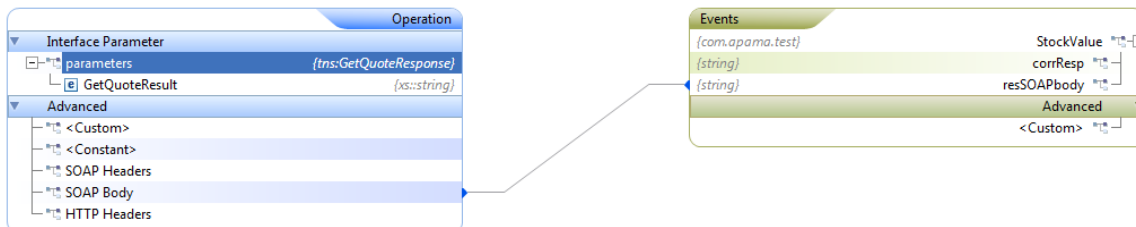
```
event Stock{
    string corrId;
    string SOAPbody;
}

event StockValue{
    string corrResp;
    string resSOAPbody;
}
```

2. In the adapter editor's Mapping Definitions tab, map the event field string that contains the SOAP body to the operation's SOAP body parameter. The following image shows an input mapping for a SOAP body:



The following image shows an output mapping for a SOAP body:



In this example, you would need to create the SOAP body in EPL or in some other fashion. To use the Web Services Client adapter to build the XML that makes up the SOAP body, you can use the convention-based approach shown in the next few steps. If you use convention-based mapping for SOAP body data then the event you use must correlate one-to-one to the expected SOAP body.

3. Define Apama events such as the following. In this example, the `_GetStockDailyValue` event contains the SOAP body data.

```
event _GetStockDailyValue{
    string strStockExchange;
    string strStock;
    string strDate;
}
```



```

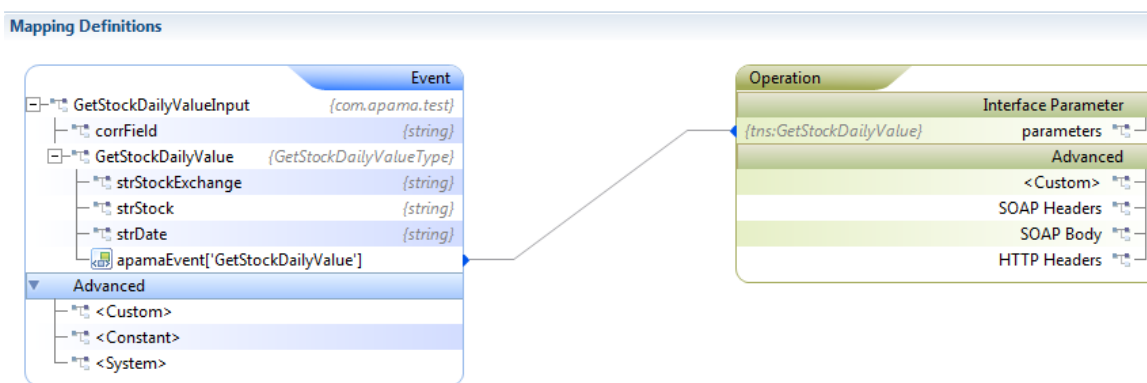
event GetStockDailyValueInput{
    _GetStockDailyValue GetStockDailyValue;
}

event _GetStockDailyValueResponse{
    float GetStockDailyValueResult;
}

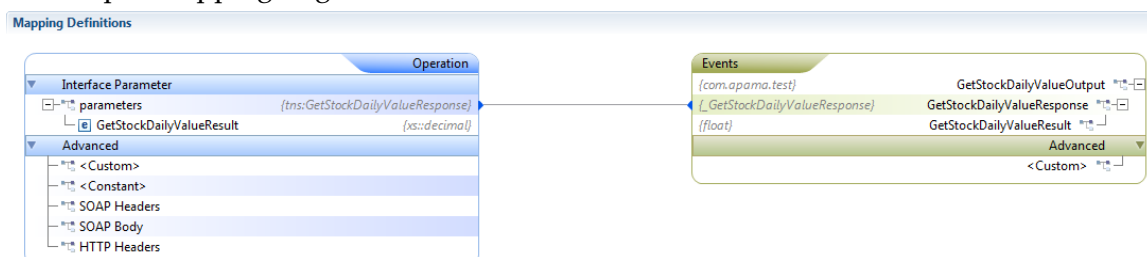
event GetStockDailyValueOutput{
    _GetStockDailyValueResponse GetStockDailyValueResponse;
}

```

4. In the Input tab of the Event Mappings tab, right-click the event that contains the SOAP body data. In this example, this is `_GetStockDailyValue` field. Select **Add Computed Node** and click OK. The **Add Computed Node** dialog appears.
5. In the **Add Computed Node** dialog's Select Method field, select **Convert to XML** from the drop-down list and click OK.
6. Map the `<Custom>apamaEvent` that contains the SOAP body to the operation's parameters field. For example, the input mapping might look like this:



The output mapping might look like this:



Customizing mapping rules

Mapping SOAP header elements

When you invoke an operation you can add custom entries to SOAP header elements if the Web Service requires it. Create mapping rules for SOAP header data as follows:

1. Define Apama events that contain `string` fields to hold the SOAP header data. For example:

```

event Stock{
    string body;
    string header;
}

event StockValue{

```

```

    string resBody;
    string resHeader;
}

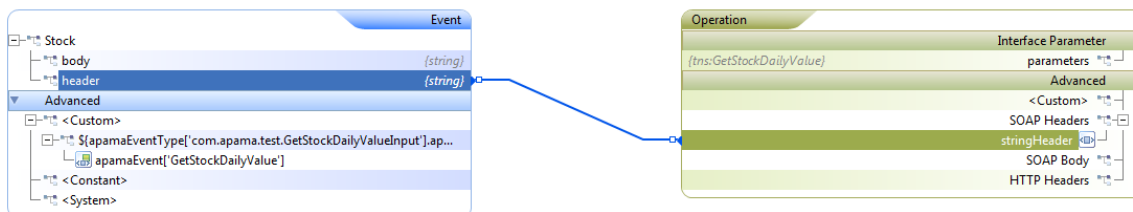
```

2. In the adapter editor's Mapping Definitions tab, right-click the operation entry to which you want to add a SOAP header and select Add. The **Add Node** dialog appears.
3. In the **Add Node** dialog's Type field select SOAP Header as the expression type.
4. In the **Add Node** dialog's Header Name field enter the name for the SOAP header and click OK.

For example:

The new entry is added to the Web Service operation's parameters.

5. Map this entry to the event `string` field that will contain the SOAP header. Following is a sample input mapping:



Customizing mapping rules

Mapping HTTP header elements

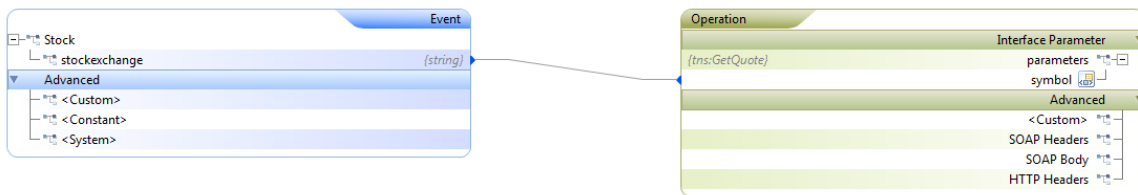
When you invoke an operation you can add custom entries to HTTP header data if the Web Service requires it. The following steps provide an example of obtaining HTTP header data from an operation's response message.

1. Define Apama events that contain `string` fields to hold the HTTP header data. For example:

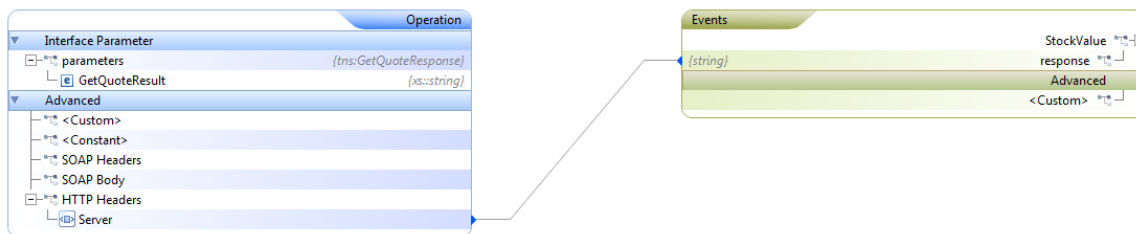
```
event Stock{
    string stockexchange;
}

event StockValue{
    string response;
}
```

2. In the adapter editor's Input mapping tab for the operation you want to invoke, map event fields to operation parameters as needed to invoke the operation. For example:



3. In the output mapping, retrieve the HTTP header information by mapping it to a `string` field event. For example, to retrieve server details, the output mapping would look like the following:



Customizing mapping rules

Using EPL to interact with Web Services

Apama applications that interact with Web Services need to use specific Apama Event Processing Language (EPL) code to do so.

Define the Apama events that are to be used specifically to interact with the Web Services. If the application will use a convention-based approach to mapping, design Apama events such that the event fields correspond to the elements of the associated Web Service messages. Note that fields that correspond to attributes in the Web Service messages should be prefixed with an underscore character (" _ "). For more information on convention-based mapping, see ["Convention-based XML mapping" on page 167](#).

For example:

```
event GetStockDailyValue{
    string strStockExchange;
```

```

    string strStock;
    string strDate;
}

event Stock{
    string stockexchange;
    string stock;
    string date;
}

event StockValue{
    string response;
}

```

The `com.apama.ws` and `com.apama.statusreport` packages define events that are helpful when invoking Web Services. For convenience, specify `using` statements for the events you want to use so that you do not need to specify their full paths. For example:

```

using com.apama.statusreport.Status;
using com.apama.ws.WSError;
using com.apama.ws.SetupContextListeners;
using com.apama.statusreport.SubscribeStatus;

```

Define a listener for an `AdapterUp` event that indicates that the Web Services Client adapter instance is available to pass your application's service requests to the Web Service. In this listener, be sure to specify the name of the adapter instance as it appears in the EPL file that Apama Studio generates. Also, in the body of the listener, enqueue an event to subscribe to the Web Service. For example:

```

on com.apama.adapters.AdapterUp(adapterName = service_id) {
    enqueue SubscribeStatus(service_id,"","");
}

```

Define a listener for an event that indicates that the Web Service is available to respond to requests. When the Web Service is available set up a listener for an event that you specified as the output event for an operation, that is, a listener for response messages from the Web Service. If the application depends on a correlation ID, this listener needs to test the value of the field specified as the correlation field. You can then invoke the Web Service by routing an event you specified as the input event for a particular operation. For example:

```

Status stat;
on Status():stat{
    if (stat.available and stat.serviceID = service_id) then {
        listenResponse();
        route Stock("NASDAQ","MSFT","2011-07-12");
        route Stock("NASDAQ","APMA","2011-07-12");
    }
}
action listenResponse() {
    StockValue stockRes;
    on all StockValue():stockRes {
        log stockRes.toString() at INFO;
    }
    WSError wsError;
    on all WSError():wsError {
        print "At service " + wsError.extraParams["serviceId"];
        // wsError always contains service id
        print "For the requested event " + wsError.requestEvent ;
        print " got the error message as " + wsError.errorMsg ;
        print " Failure message is " + wsError.failureType ;
        // See WSConstants event for types of failure
    }
}

```

The following code provides a complete example of how a monitor can invoke a Web Service in a private context. In this example, the monitor sends requests and listens for responses from the Web Service in a private context.

```

using com.apama.statusreport.Status;

```

```

using com.apama.ws.WSError;
using com.apama.ws.SetupContextListeners;
using com.apama.ws.TerminateContextListeners;
using com.apama.statusreport.SubscribeStatus;

/*
 * Monitor that will spawn to a private context send requests,
 * and listen for Web Service responses in that private context.
 */
monitor UsingContexts {

    context privateContext;
    constant string service_id := "Webservice_INSTANCE_1";
    // ID that the Apama-Studio-generated EPL uses.

    action onload() {

        on com.apama.adapters.AdapterUp(adapterName = service_id) {
            enqueue SubscribeStatus(service_id,"","");
            // Subscribe to Web Service.
        }

        Status stat;
        on Status():stat{
            if (stat.available and stat.serviceID = service_id) then {
                spawnPrivateContext();
            }
        }
    }

    action spawnPrivateContext() {
        privateContext := context("PrivateContext" , false);
        // Create a private context.

        // The application must send a SetupContextListeners event, which
        // contains details such as the service ID (based on generated EPL)
        // and information about the private context in which the monitor
        // listens for service responses.
        route SetupContextListeners(service_id,privateContext);

        spawn sendAndListen() to privateContext;
    }

    action sendAndListen() {
        print "Sending events to " + context.current().toString();
        enqueue Stock("NASDAQ","APMA","2011-07-12") to context.current();
        StockValue stockRes;

        on all StockValue():stockRes {
            log stockRes.toString() at INFO;
            print "Terminating Private context service monitor";
            terminatePrivateContext(); // Terminates service monitor in this context.
        }

        WSError wsError;
        on all WSError():wsError {
            print "At service " + wsError.extraParams["serviceId"];
            // WSError always contains service ID.
            print "For the requested event " + wsError.requestEvent ;
            print " got the error message as " + wsError.errorMsg ;
            print " Failure message is " + wsError.failureType ;
            // See WSConstants event for types of failures.
        }
    }

    action terminatePrivateContext() {
        // The application must send a TerminateContextListeners event,
        // which contains details such as the service ID and the private
        // context that contains the service monitor that contains
        // the service response listeners. When the response listeners
    }
}

```

```

// terminate, the service monitor that contained them also terminates.
// The private context is still available to be used.
route TerminateContextListeners(service_id,privateContext);

//If a request is sent there should not be a response.
print "Sending request...expecting no response" ;
enqueue Stock("NASDAQ","APMA","2011-07-12") to privateContext;
}
}

```

Using the Apama Web Services Client Adapter

Configuring logging for Web Services Client adapter

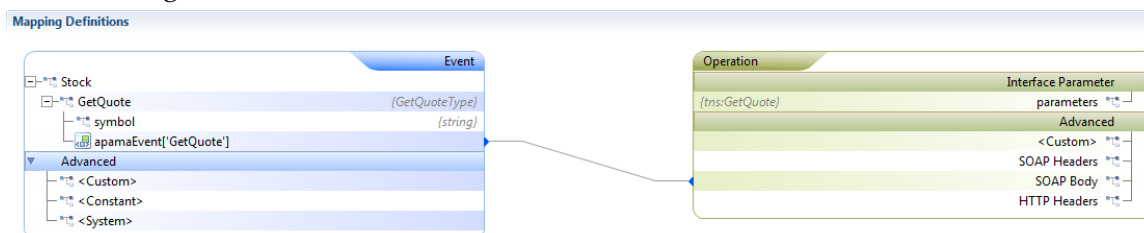
To configure logging for a Web Services Client adapter, modify the `APAMA_HOME/adapters/config/adapters-log4j.properties` file. The default logging level for the properties listed in this file is `INFO`. The `com.progress.el` package contains the mapping framework. Possible logging levels are `INFO`, `WARN`, `ERROR`, `DEBUG`, and `TRACE`.


If you want you can place this file in another location or use a different name for this file. If you do then be sure to update the value of the `log4j.configuration` property in the Web Services Client adapter's IAF configuration file. This is the line you need to change:

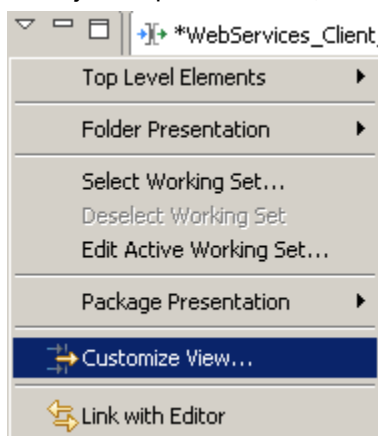
```
<property name="log4j.configuration" value="file:///@APAMA_HOME@/adapters/config/adapters-log4j.properties"/>
```

In addition to general adapter logging, you can configure payload logging. Payload logging can help you diagnose problems by indicating what the constructed request and responses looks like, which can point to whether a problem is in the tooling part of your application or in the runtime execution.

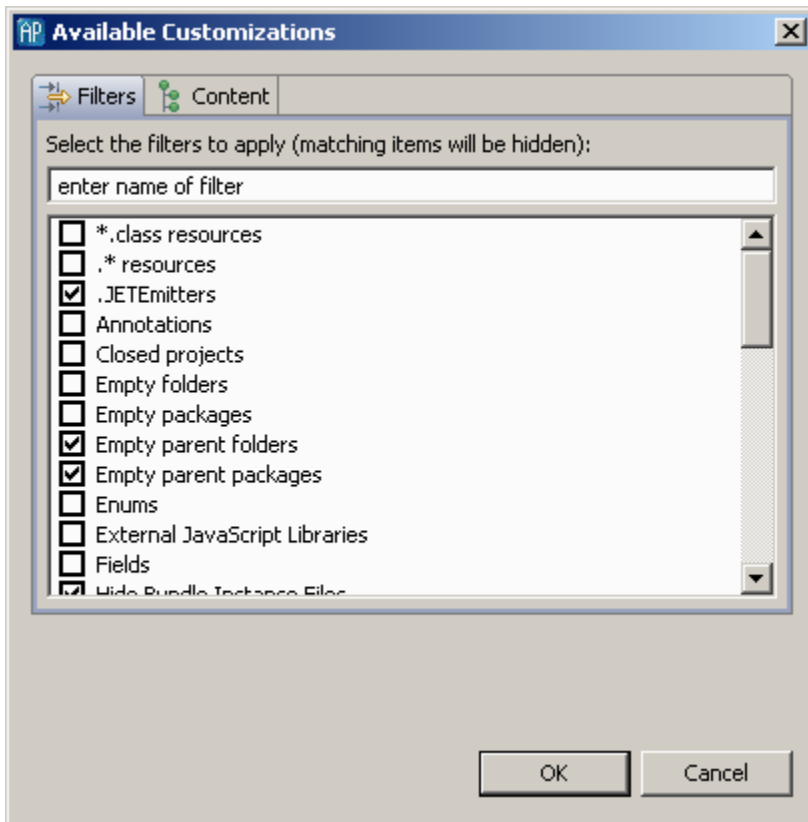
The steps below show how to configure payload logging for a convention-based mapping such as the following:



1. In the Project Explorer toolbar, click View Menu () and select Customize View.



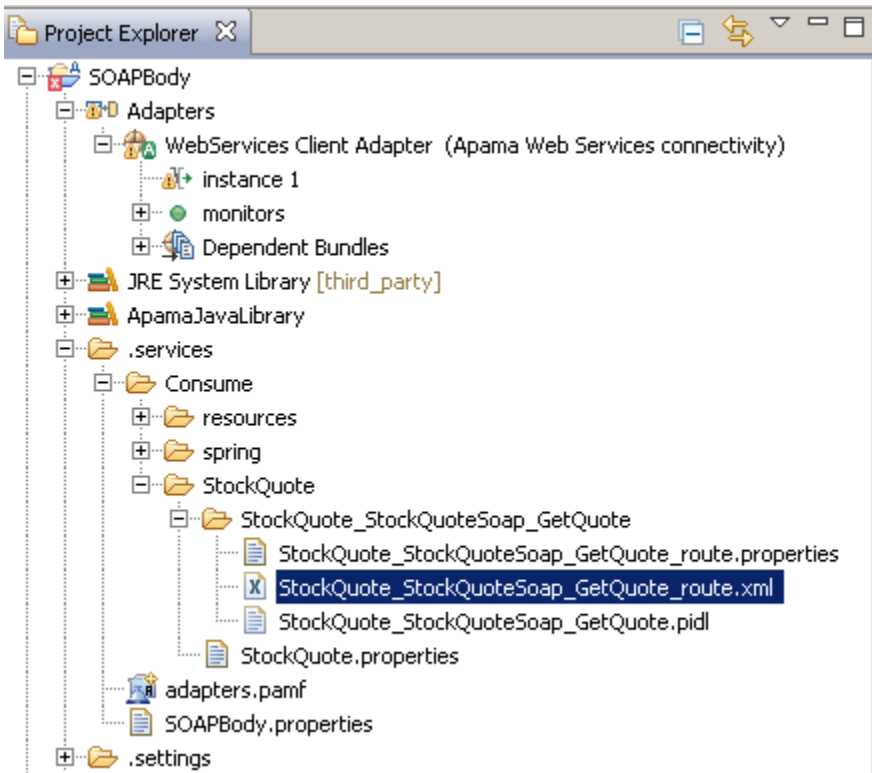
This displays the Available Customizations dialog:



2. In the Filters tab, ensure that the `*.resources` checkbox is not selected. Click OK.

The `.services` folder now appears in the project's hierarchy in the Project Explorer.

3. For the operation for which you want to enable payload logging, in the project's hierarchy in Project Explorer, open the XML file for the operation's `route` statement. For the mapping shown in the image at the beginning of these instructions, you would select the `StockQuote_StockQuoteSoap_GetQuote_route.xml` file:



4. In the operation's `_route.xml` file, add the following lines just after the `<cxf.properties>` section:

```
<cxf:features>
  <bean class="org.apache.cxf.feature.LoggingFeature"/>
</cxf:features>
```

For example:

```
<cxf:cxfEndpoint
. . .>
  <cxf:properties>
    <entry key="mtom-enabled" value="\${CXF_mtom_enabled_StockQuote_StockQuoteSoap_GetQuote_route}"/>
    <entry key="dataFormat" value="\${CXF_dataFormat_StockQuote_StockQuoteSoap_GetQuote_route}"/>
  </cxf:properties>
  <cxf:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </cxf:features>
</cxf:cxfEndpoint>
```

5. In the `APAMA_HOME/adapters/config/adapters-log4j.properties` file, change the default `INFO` setting for `log4j.logger.org.apache.cxf` to the logging level you want.

After you invoke the operation you specified payload logging for, the `iaf.log` file for the Web Services Client adapter displays something like the following:

```
-----
ID: 1
Address: http://www.webserviceX.net/stockquote.asmx
Encoding: UTF-8
Content-Type: text/xml
Headers: {SOAPAction=["http://www.webserviceX.NET/GetQuote"], Accept=[/*/*]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body><GetQuote xmlns="http://www.webserviceX.NET/">
    <symbol>APMA</symbol></GetQuote></soap:Body></soap:Envelope>
-----
```

Using the Apama Web Services Client Adapter

Web Services Client adapter artifacts

Apama Studio automatically generates the various artifacts that enable the application to connect to the Web Service and invoke its operations. The following types of files are generated:

- Web Services files - These include files that define, for example, the interface, beans, and routes used by the Web Services itself.
- Apama files - These are files that define Apama service monitors and mapping rules. These files listen for Apama events that are then mapped to Web Services request messages. The mapping rules also determine how Web Services response messages are associated with fields in Apama events.

[Using the Apama Web Services Client Adapter](#)

Chapter 3: Using the Apama File Adapter

■ File Adapter plug-ins	107
■ File Adapter service monitor files	108
■ Adding the File adapter to an Apama Studio project	108
■ Configuring the File adapter	109
■ Overview of event protocol for communication with the File adapter	111
■ Opening files for reading	111
■ Specifying file names in OpenFileForReading events	114
■ Opening comma separated values (CSV) files	115
■ Opening fixed width files	116
■ Sending the read request	116
■ Requesting data from the file	117
■ Receiving data	118
■ Opening files for writing	119
■ LineWritten event	121
■ Monitoring the File adapter	121

This chapter provides a description of the Apama File adapter that is included when you install the Apama software. Each Apama standard adapter includes the transport and codec plug-ins it requires, along with any required EPL service monitor files. The C++ plug-ins are located in the Apama installation's `adapters\bin` directory (Windows) or `adapters/lib` directory (UNIX); the Java plug-ins are located in `adapters\lib`. The EPL files are located in the `adapters\monitors` directory.

If you develop an application in Apama Studio, when you add a standard adapter to the project, Apama Studio automatically creates a configuration file for it. In addition, the standard Apama adapters include bundle files that automatically add the adapter's plug-ins and associated service monitor files to the Apama Studio project.

If you are not using Apama Studio, you need to create a configuration file that will be used by the IAF to run the adapter. Each adapter includes a template file that can be used as the basis for the configuration file. The template files are located in the installation's `adapters\config` directory and have the forms `adapter_name.xml.dist` and `adapter_name.static.xml`. These template files are not meant to be used as the adapters' actual configuration files — you should always make copies of the template files before making any changes to them.

The Apama File adapter uses the Apama Integration Adapter Framework (IAF) to read information from text files and write information to text files by means of Apama events. This lets you read files line-by-line from external applications or write formatted data as required by external applications.

With some caveats, which are mentioned later in this section, the File adapter supports reading and writing to multiple files at the same time. Information about using the File adapter can be found in the following topics:

- ["File Adapter plug-ins" on page 107](#)
- ["File Adapter service monitor files" on page 108](#)
- ["Adding the File adapter to an Apama Studio project" on page 108](#)
- ["Configuring the File adapter" on page 109](#)
- ["Overview of event protocol for communication with the File adapter" on page 111](#)
- ["Opening files for reading" on page 111](#)
- ["Specifying file names in OpenFileForReading events" on page 114](#)
- ["Opening comma separated values \(CSV\) files" on page 115](#)
- ["Opening fixed width files" on page 116](#)
- ["Sending the read request" on page 116](#)
- ["Requesting data from the file" on page 117](#)
- ["Receiving data" on page 118](#)
- ["Opening files for writing" on page 119](#)
- ["LineWritten event" on page 121](#)
- ["Monitoring the File adapter" on page 121](#)

Using Standard Adapters

File Adapter plug-ins

The Apama File adapter uses the following plug-ins:

- `JMultiFileTransport.jar` — The `JMultiFileTransport` plug-in manages the connections to the files opened for reading and writing.
- `JFixedWidthCodec.jar` or `JCSVCodec.jar` — These plug-ins parse lines of data in fixed-width format or comma separated value format (CSV) into fields>

For details about using these codec plug-ins, see ["The CSV codec plug-in" on page 143](#) and ["The Fixed Width codec plug-in" on page 145](#).

- `JNullCodec.jar`

These plug-ins need to be specified in the IAF configuration file used to start the adapter. If you add this adapter to an Apama Studio project, Apama Studio automatically adds these plug-ins to the configuration file. If you are not using Apama Studio, you can use the `File.xml.dist` template file as the basis for the configuration file. See ["Configuring the File adapter" on page 109](#) for more information about adding the necessary settings to the adapter's configuration file.

Using the Apama File Adapter

File Adapter service monitor files

The File adapter requires the event definitions in the following monitors which are in your Apama installation directory. If you are using Apama Studio, the project's default run configuration automatically injects them. If you are not using Apama Studio, you need to make sure they are injected to the correlator in the order shown before running the IAF.

1. `monitors\StatusSupport.mon`
2. `adapters\monitors\IAFStatusManager.mon`
3. `adapters\monitors\FileEvents.mon`
4. `adapters\monitors\FileStatusManager.mon`

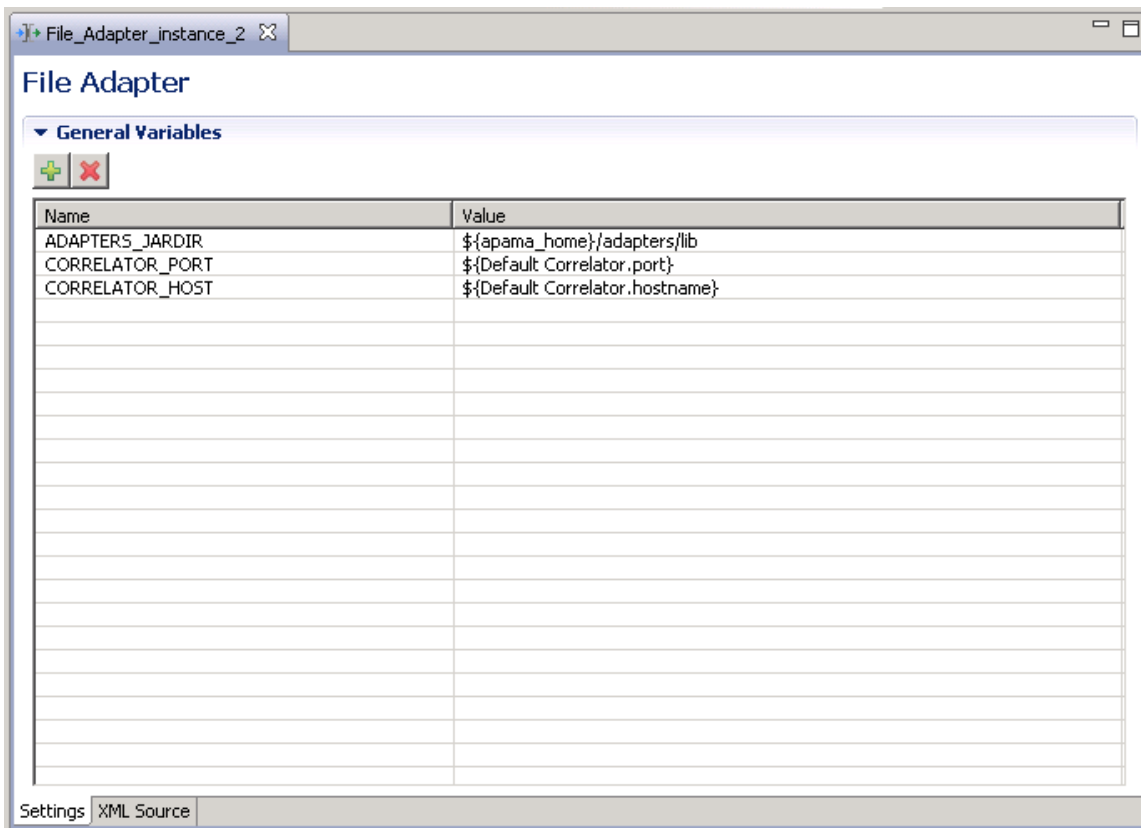
Using the Apama File Adapter

Adding the File adapter to an Apama Studio project

If you are developing an application in Apama Studio, add the File adapter as follows:

1. In the Project Explorer, right-click name of the project and select **Apama > Add Adapter**. The **Add Adapter Instance** dialog is displayed.
2. Select **File Adapter (File adapter for reading and writing to ASCII files)**. Apama Studio adds a default name to the Adapter instance name field that ensures this instance of this adapter will be uniquely identified. You can change the default name, for example, to indicate what type of external system the adapter will connect to. Apama Studio prevents you from using a name already in use.
3. Click **OK**.

Apama Studio adds a File adapter entry that contains the new instance to the project's `Adapters` node and opens the instance's configuration file in the Apama Studio adapter editor as shown in the following illustration.



For the File adapter, the adapter editor's Settings tab displays a listing of General Variables. When first created it lists variables that are used in the Apama Studio project's default launch configuration. You can add variables by clicking the Add button and filling in the variable's name and value.

For editing other configuration properties for the File adapter, display the adapter editor's XML Source tab and add the appropriate information.

Using the Apama File Adapter

Configuring the File adapter

Before using the File adapter, you need to add information to the IAF configuration file used to start the adapter. When you add an adapter to an Apama Studio project, a configuration file for each instance of the adapter is automatically created. In Apama Studio, double-click the name of the adapter instance to open the configuration file in the adapter editor.

If you are using Apama Studio's adapter editor, you can edit or add variables to the General Variables section as displayed on the Settings tab. For other properties, you need to edit the XML code directly; to do this, select the adapter editor's XML Source tab.

If you are not using Apama Studio, the configuration file can be derived from the template `adapters\config\File.xml.dist` configuration file shipped with the Apama installation. *Caution:* before changing any values, be sure to make a copy of the `File.xml.dist` file and give it a unique name, typically with an `.xml` extension instead of `.xml.dist`.

The template configuration file references the `adapters\config\File-static.xml` file using the XML `XInclude` extension. The `File-static.xml` file specifies the adapter's codec and its mapping rules. Normally you do not need to change any information in this file. See ["The IAF configuration file" on page 255](#) for more information on the contents on an adapter's configuration file.

In Apama Studio, adapters are configured using the adapter editor. To open an adapter instance, in the Project Explorer, right-click on `<project name>\Adapters\File Adapter\<instance name>` and select **Open Instance** from the pop-up menu.

You can set the variables used by the File adapter in the main Settings tab. Values of the form `${...}`, such as `${DefaultCorrelator:port}` are set to the correct value automatically by the Apama Studio project's default launch configuration and do not need to be modified. To configure other properties used by the adapter, edit the XML code directly by selecting the XML Source tab.

If you are not using Apama Studio, all adapter properties are configured by editing the adapter `.xml` file in an XML or text editor.

Customize the following properties:

- `<logging level="INFO" file="logs/FileAdapter.log"/>`

If you start the IAF from the Management and Monitoring console, specify an absolute path for the log filename.

```
<classpath path="@ADAPTERS_JARDIR@/JNullCodec.jar" />
<classpath path="@ADAPTERS_JARDIR@/JFixedWidthCodec.jar" />
<classpath path="@ADAPTERS_JARDIR@/JCSVCodec.jar" />
<classpath path="@ADAPTERS_JARDIR@/JMultiFileTransport.jar" />
```

Replace `@ADAPTERS_JARDIR@` with the actual path to the `.jar` files. Typically, this is the `apama_install_dir\adapters\lib` directory.

If you are using Apama Studio, these jar files are automatically added to the classpath in the configuration file and you do not need to replace the `@ADAPTERS_JARDIR@` token.

- In the `<sink>` and `<source>` elements, replace `@CORRELATOR_HOST@` and `@CORRELATOR_PORT@` with valid attribute values:

```
<apama>
  <sinks>
    <sink host="@CORRELATOR_HOST@" port="@CORRELATOR_PORT@" />
  </sinks>
  <sources>
    <source host="@CORRELATOR_HOST@" port="@CORRELATOR_PORT@"
      channels="FILE" />
  </sources>
</apama>
```

If you are using the adapter in an Apama Studio project, the default launch configuration uses the default correlator host and port settings and you do not need to replace the `@CORRELATOR_HOST@` and `@CORRELATOR_PORT@` tokens.

- `<property name="simpleMode" value="false" />`

Indicate whether or not to start the File adapter in simple mode. In simple mode, the File adapter reads lines from a single file or writes lines to a single file. In non-simple mode, you can use the fixed width or CSV codecs to decode/encode field data. Also, the File adapter can read/write to multiple files and additional controls are available for communication between the adapter and the correlator. Non-simple mode is recommended for most situations. Details about simple mode and non-simple mode are in the `File.xml.dist` file. If you are using Apama Studio, switch to the adapter editor's XML Source tab if you want to view these details or to edit the settings.

Overview of event protocol for communication with the File adapter

The `adapters\monitors\FileEvents.mon` file defines the event types for communication with the File adapter. The following event types in the `com.apama.file` package are defined in the `FileEvents.mon` file. These events enable I/O operations on files. See `FileEvents.mon` for details about the events that are not described in the subsequent topics.

- `OpenFileForReading`
- `OpenFileForWriting`
- `FileHandle`
- `FileLine`
- `ReadLines`
- `NewFileOpened`
- `EndOfFile`
- `CloseFile`
- `FileClosed`
- `FileError`
- `LineWritten`

Using the Apama File Adapter

Opening files for reading

The File adapter can read from multiple files at the same time. Send an `OpenFileForReading` event for each file you want the File adapter to read. This involves emitting an event to the channel specified in the adapter's configuration file, typically `FILE`, for example:

```
emit OpenFileForReading(...) to "FILE"
```

The `OpenFileForReading` event definition is as follows:

```
event OpenFileForReading
{
    string    transportName;
    integer   requestId;
    string    codec;
    string    filename;
    integer   linesInHeader;
    string    acceptedLinePattern;
    dictionary<string, string> payload;
}
```

Parameter	Description
-----------	-------------

transportName	Name of the transport being used within the File adapter. This must match the transport name specified in the IAF configuration file so that the transport can recognize events intended for it.
requestId	Request identifier for this open file event. The response, which is either a <code>FileHandle</code> event or a <code>FileError</code> event, contains this identifier.
codec	Name of the codec to use with the file. This must match one of the codecs specified in the <i>adapter-static.xml</i> IAF configuration file. When you want the File adapter to read and write entire lines of data just as they are, specify the null codec (<code>JNullCodec</code>). When you want the File adapter to interpret file lines in some way, you can specify either the CSV codec (<code>JCSVCodec</code>) or the Fixed Width codec (<code>JFixedWidthCodec</code>) according to how the data in the file is formatted. To open fixed width or CSV files, you must add some information to the <code>payload</code> field of the <code>OpenFileForReading</code> event. The codecs needs this information to correctly interpret the data. For details about adding to the <code>payload</code> field, see "Opening comma separated values (CSV) files" on page 115 or "Opening fixed width files" on page 116 .
filename	Absolute path, or file pattern (for example, <code>*.txt</code> , <code>*.csv</code>) within absolute directory path if intending to read all files matching a pattern in order of last time modified. While a relative path might work, an absolute path is recommended. A relative path must be relative to where the IAF has been started, which can be unpredictable. For example: <pre>c:\logfiles*.log /user/local/jcasablancas/logfiles/*.log</pre> For more information, see "Specifying file names in OpenFileForReading events" on page 114 .
linesInHeader	The number of lines in the header, or 0 if there is no header. Text files sometimes contain a number of lines at the beginning of the file that explain the format. As these are usually of some specific format, the Apama File adapter cannot interpret them. By skipping these lines, the File adapter can process just the data contained in the file.
acceptedLinePattern	Regular expression pattern (in the same format supported by Java) to use to match lines to read. The File adapter reads only those lines that match this pattern. To read all lines, specify an empty string.
payload	String dictionary for storing extra fields for use with codecs. For fixed width files the following fields make up the <code>payload</code> ; for other types of files, they will be ignored. <ul style="list-style-type: none"> • <code>sequence<integer> fieldLengths</code>

	<p>The length (number of characters) in each field, in order, where the number of fields is given by <code>fieldLengths.size()</code></p> <ul style="list-style-type: none"> • <code>boolean isLeftAligned</code> <p>Whether the data in the field is aligned to the left or not (that is, right aligned)</p> <ul style="list-style-type: none"> • <code>string padCharacter</code> <p>The pad character used when the data is less than the width of the field</p> <p>For CSV files, the following field makes up the <code>payload</code>; for other types of files it will be ignored.</p> <ul style="list-style-type: none"> • <code>string separator</code> <p>The separator character</p>
--	--

Using the Apama File Adapter

Opening files for reading with parallel processing applications

If your Apama application implements parallel processing, you may want to increase parallelism by processing the incoming events from the File adapter in a separate, private, context, rather than doing everything in the correlator's main context. To request that events from the File adapter are sent to the private context your monitor is running in, the monitor should open the file using the `com.apama.file.OpenFileForReadingToContext` event instead of `OpenFileForReading`. The `OpenFileForReadingToContext` event has a field that contains a standard `OpenFileForReading` event (see ["Opening files for reading" on page 111](#)), in addition to a field specifying the context that file adapter events should go to for processing, (which is usually the context the monitor itself is running in, `context.current()`), and the name of the channel the File adapter is using. When using the `OpenFileForReadingToContext` event, the `OpenFileForReadingToContext` event and all other file adapter events must not be emitted directly to the adapter, but rather enqueued to the correlator's main context, where the adapter service monitor runs. The File adapter's service monitor is responsible for emitting the events that are enqueued from other contexts to the File adapter, and for enqueueing the events received from the File adapter to whichever context should process them (as specified in the `OpenFileForReadingToContext` event).

The `OpenFileForReadingToContext` event is defined as follows:

```
event OpenFileForReadingToContext
{
    context instanceContext;
    string fileChannel;
    OpenFileForReading fileEvent;
}
```

Here is an example of how the `OpenFileForReadingToContext` event is used:

```
com.apama.file.OpenFileForReading openFileForReading :=
    new com.apama.file.OpenFileForReading;
... // populate the fields of the openFileForReading event as needed
// Instead of emitting openFileForReading to "FILE", wrap it in
// the OpenFileForReadingToContext event and enqueue it to the service
// monitor in the main context.
enqueue com.apama.file.OpenFileForReadingToContext(context.current(),
    "FILE", openFileForReading) to mainContext;
com.apama.file.FileHandle readHandle;
on com.apama.file.FileHandle{
```

```

transportName=openFileForReading.transportName,
requestId=openFileForReading.requestId):readHandle
{
    // instead of emitting to the "FILE" channel, enqueue it to the main
    // context
    enqueue com.apama.file.ReadLines(openFileForReading.transportName, -1,
        readHandle.sessionId, 20) to mainContext;
    ...
}

```

Opening files for reading

Specifying file names in OpenFileForReading events

In an `OpenFileForReading` event, the value of the `filename` field can be a specific file name or a wildcard pattern. However, the filename cannot have multiple wildcards.

Specific filename

When you specify a specific filename in an `OpenFileForReading` event, when the adapter receives requests to read lines from the file, the adapter reads till the end of the file and waits until more data is available. An external process, or the adapter itself, might write more data to the file if it is open for write at the same time that it is being read. If more data becomes available, the File adapter sends it. If the File adapter receives a `CloseFile` event, the File adapter closes the file against further reading.

Each time the File adapter reaches the end of the file it is reading, the File adapter sends an `EndOfFile` event to the correlator. If, during this process, more data was appended to the file, the file operations will continue as normal — that is, the File adapter will send more lines if they were requested. Thus, when reading specific files, file appends are acceptable and have a well defined behavior. However, any other modifications, such as changing the lines that have already been read, may have undefined results. An application can ignore or react to an `EndOfFile` event. The definition of an `EndOfFile` event is as follows:

```

event EndOfFile
{
    /* The name of the transport being used within the file adapter */
    string transportName;

    /* The session ID this File is associated with */
    integer sessionId;

    /* The name of the file*/
    string filename;
}

```

Wildcard filenames

Now suppose that in an `OpenFileForReading` event, the value of the `filename` field is a wildcard pattern. In this case, the adapter does the following:

1. Opens a new file that matches the pattern
2. Reads that file in its entirety
3. Sends back an `EndOfFile` event
4. Opens the next file that matches the pattern if one is available

For the application's information, the File adapter sends back an event when it opens each new file. The `NewFileOpened` event contains the name of the file that was opened:

```
event NewFileOpened
{
    /* The name of the transport being used within the file adapter */
    string transportName;

    /* The session Id this File is associated with */
    integer sessionId;

    /* The filename opened */
    string filename;
```

The order of opening the files that match the wildcard pattern is not specified. Currently, the files are ordered by the modification date and then alphabetically by filename.

If a file that has been previously read is externally modified (while in the meantime, the File adapter is reading from other files that match the wildcard pattern), the file is read again in its entirety. That is, any files that are modified after reading from them will be read again (until the `CloseFile` is sent). Note that this includes file appends.

Using the Apama File Adapter

Opening comma separated values (CSV) files

An example of defining an `OpenFileForReading` event that opens a CSV file so that each field is automatically parsed appears below. The additional data required by the CSV codec is stored in the payload dictionary.

```
com.apama.file.OpenFileForReading openCSVFileRead :=
    new com.apama.file.OpenFileForReading;

    //matches transport in IAF config
    openCSVFileRead.transportName := JMultiFileTransport;

    //the request id to use
    openCSVFileRead.requestId := integer.getUnique();

    //read using JCSVCodec
    openCSVFileRead.codec := "JCSVCodec";

    //file to read
    openCSVFileRead.filename := "/usr/home/formby/stocktick.csv";

    //separator char is a ","
    openCSVFileRead.payload["separator"] := ",";

    //emit event to channel in config.
    emit openCSVFileRead to "FILE";
```

Subsequently, when the File adapter receives `FileLine` events, the adapter stores each field in the data sequence in order. You can access the ones you are interested in.

For details about using the CSV codec, see ["The CSV codec plug-in" on page 143](#).

Using the Apama File Adapter

Opening fixed width files

An example of defining an `OpenFileForReading` event that opens a fixed width file so that each field is automatically parsed appears below. The additional data required by the Fixed Width codec is stored in the payload dictionary.

```
com.apama.file.OpenFileForReading openFixedFileRead :=
    new com.apama.file.OpenFileForReading;

//matches transport in IAF instance
openFixedFileRead.transportName := JMultiFileTransport;

//the request id to use
openFixedFileRead.requestId := integer.getUnique();

//read using CSV Codec
openFixedFileRead.codec := "JFixedWidthCodec";

//file to read
openFixedFileRead.filename := "/usr/home/formby/stocktick.txt";
//additional data required to interpret fixed width data

//sequence of field lengths
openFixedFileRead.payload["fieldLengths"] := "[6,4,9,9,9]";

//it is left aligned
openFixedFileRead.payload["isLeftAligned"] := "true";

//the pad character
openFixedFileRead.payload["padCharacter"] := "_";

//emit event to channel in config.
emit openFixedFileRead to "FILE";
```

Subsequently, when the File adapter receives `FileLine` events, the adapter stores each field in the data sequence in order. You can access the ones you are interested in.

For details about using the Fixed Width codec, see ["The Fixed Width codec plug-in" on page 145](#).

Using the Apama File Adapter

Sending the read request

After you construct an `OpenFileForReading` event, emit it to the "FILE" channel. For example:

```
com.apama.file.OpenFileForReading openFileWeWantToRead :=
    new com.apama.file.OpenFileForReading;

//populate the openFileWeWantToRead event
//..
//..
emit openFileWeWantToRead to "FILE";
```

Emitting an `OpenFileForReading` event from EPL code signals the File adapter to open the file. If the open operation is successful, the File adapter returns a `FileHandle` event, whose definition is as follows:

```
event FileHandle
{
    /* The name of the transport being used within the file adapter */
```

```

string transportName;

/* Request ID this file handle is in response to. */
integer requestId;

/* Session ID to use for further communication with the
   File adapter */
integer sessionId;
}

```

The `sessionId` is the most important field; all communication related to this file references this value.

If the open operation is unsuccessful, the File adapter returns a `FileError` event, whose definition is as follows:

```

event FileError
{
    /* The name of the transport being used within the file adapter */
    string transportName;

    /* Request ID this file error is in response to. */
    integer requestId;

    /* This should contain relevant information as to why the
       error occurred */
    string message;
}

```

Using the Apama File Adapter

Requesting data from the file

After your application receives a `FileHandle` event, it can emit a `ReadLines` event, which signals the adapter to start reading lines from the file. The definition of the `ReadLines` event is as follows:

```

event ReadLines
{
    /* The name of the transport being used within the file adapter */
    string transportName;

    /* request Id. Corresponding FileLine events will have the
       same requestId */
    integer requestId;

    /* The session Id this read line event is coming from */
    integer sessionId;

    /* The number of lines to read. As each line is read, a FileLine
       event will be sent from the Adapter in response to this request.
       When opening a single file, the adapter will continue to probe
       the file until lines are available.

       If a wildcard pattern was specified and the end of file is
       reached before the specified number of lines have been read,
       the file will be closed, an EndOfFile event will be sent, and
       if a new file is available, it will send a NewFileOpened event
       and read the remaining number of lines from this. If a new file
       is not available, the adapter will wait until one is available.
    */
    integer numberOfLines;
}

```

The `sessionId` in the `ReadLines` event must be the same as the `sessionId` stored in the `FileHandle` event that the application received when the file was opened.

The adapter tries to read as many lines as specified in the `ReadLines` event. If the file does not contain that many lines, what the adapter does depends on whether the original `OpenFileToRead` event specified a specific file or a wildcard pattern. According to that setting, the adapter either waits until the file contains more data, or tries to open a new file to deliver the balance from.

Using the Apama File Adapter

Receiving data

As the File adapter reads the file, it returns `FileLine` events to your application. Each line is associated with a specific `sessionId`, and the data is stored within a sequence of strings. The definition of `FileLine` events is as follows:

```
event FileLine
{
    /* The name of the transport being used within the file adapter */
    string transportName;

    /* When receiving these events in response to a request to read
    lines, this field will contain the same value as provided in
    the corresponding ReadLines event*/
    integer requestId;

    /* The session Id this FileLine event is for or from */
    integer sessionId;

    /* The data as a sequence of strings. */
    sequence<string> data;
}
```

Notice that the data field is a sequence of strings, rather than a string. However, when you use the null codec for reading, the sequence contains only one element, which contains the entire line read:

```
//the whole line is stored in the first element, we used null codec
string line := fileLine.data[0];
```

For specialized codecs, each field is in a discrete element in the sequence:

```
//The app knows which field contains the data we are interested in:
string symbol := fileLine.data[0];
string exchange := fileLine.data[1];
string currentprice := fileLine.data[2];
//and so on
```

After the File adapter opens a file for reading, the file remains open as long as the adapter is running. If you want to close a file, you must send a `CloseFile` event that specifies the `sessionId` of the file you want to close. For example, if you want to replace the contents of a file, you must close the file before you send an `OpenFileForWriting` event. The definition of the `CloseFile` event is as follows:

```
event CloseFile
{
    /* The name of the transport being used within the file adapter */
    string transportName;
    /* Request ID for this CloseFile event. If an error occurs,
    the response will contain this ID */
    integer requestId;

    /* The session ID the file to close is associated with */
    integer sessionId;
}
```

If there is an error, the File adapter sends a `FileError` event. Otherwise, the File adapter closes the file and sends a `FileClosed` event, and then it is available to be opened again for writing or for reading.

Using the Apama File Adapter

Opening files for writing

To open a file for writing, emit an `OpenFileForWriting` event. The definition of the `OpenFileForWriting` event is similar to the definition of the `OpenFileForReading` event:

```
event OpenFileForWriting
{
    /* The name of the transport being used within the file adapter.
       This should match the transport name used in the IAF config
       file so the transport can recognize events intended for it */
    string transportName;

    /* Request ID for this open file event. The response, either a
       FileHandle or a FileError event, will contain this ID */
    integer requestId;

    /* The name of the codec to use with the file. This should match
       one of the codecs specified in the (static) config file. Use
       the null codec (by default this is called JNullCodec) to write
       entire lines */
    string codec;

    /* Full filename to write to */
    string filename;

    /* Boolean representing whether the file is to be overwritten or
       appended */
    boolean appendData;

    /* This field is used to specify extra parameters to the codecs,
       such as the CSVCodec and the FixedWidthCodec */
    dictionary<string, string> payload;
}
```

The procedure for opening a file for writing CSV or fixed width files is effectively the same as for reading. Specify the relevant fields in the payload to describe the format of the file you want to write. When subsequently sending `FileLine` events, populate the data sequence field with the data for each field.

Again, once constructed, emit the `OpenFileForWriting` event to the "FILE" channel, for example:

```
emit new com.apama.file.OpenFileForWriting("FileTransport",
    integer.getUnique(), "JNullCodec", "/home/writeFile.txt", false);
```

For fixed width files, you can construct a more complex `OpenFileForWriting` event in a similar way to that described in ["Opening fixed width files" on page 116](#).

Again, as with reading a file, the File adapter sends a `FileHandle` or `FileError` event (see ["Sending the read request" on page 116](#)), which your application should listen for, filtering on the `requestId` for the `FileHandle` event you are interested in.

Once a `FileHandle` event has been received, the file has successfully opened and the application can begin to send `FileLine` events to be written:

```
event FileLine
{
    /* The name of the transport being used within the file adapter */
    string transportName;
```

```

/* When sending these upstream (i.e. writing):
 1) If this value is negative, the adapter will send no
    acknowledgement.
    A FileError may be sent back in response if this request
    generates an error. If the user is always using the same
    requestId e.g. -1, then they will not be able to work out
    which line generated the error. The application writer is
    free to ignore listening for these errors should they wish.

 2) If this value is 0 or greater then the adapter will send an
    acknowledgement LineWritten event back to the application.
*/
integer requestId;

/* The session Id this FileLine event is for or from */
integer sessionId;

/* The data as a sequence of strings. */
sequence<string> data;
}

```

Notice that the data field is a sequence of strings, rather than a string. This allows you to have the fields you want to write as separate entries in the sequence, and it lets the File adapter format the sequence for writing according to the chosen codec. For the fixed width codec, the number of elements in the sequence should match the number of fields originally specified when opening the file. For the null codec, if the sequence contains more than one element, each field will be written out using a separator defined in the IAF configuration file. This separator can be blank, in which case each element will be written out immediately after the previous one, with a newline after the last element.

The `FileLine` event is exactly the same as the one received when reading; however, the `requestId` takes on a more important role. If you specify a positive `requestId`, your application receives an acknowledgement

When a file is already open for reading, you can write to that file only by appending new data. Of course, you must send an `OpenFileForWriting` event, and then the File adapter can process `FileLine` events for writing to that file. You receive a `FileError` event if the file is open for reading and for writing and you try to write data into the file but not by appending the new data.

Using the Apama File Adapter

Opening files for writing with parallel processing applications

If your Apama application implements parallel processing, you may want to increase parallelism by processing the incoming events from the File adapter in a separate, private, context, rather than doing everything in the correlator's main context. To request that events from the File adapter are sent to the private context your monitor is running in, the monitor should open the file using the `com.apama.file.OpenFileForWritingToContext` event instead of `OpenFileForWriting`. The `OpenFileForWritingToContext` event has a field that contains a standard `OpenFileForWriting` event (see ["Opening files for writing" on page 119](#)), in addition to a field specifying the context that file adapter events should go to for processing, (which is usually the context the monitor itself is running in, `context.current()`), and the name of the channel the File adapter is using. When using the `OpenFileForWritingToContext` event, the `OpenFileForWritingToContext` event and all other File adapter events must not be emitted directly to the adapter, but rather enqueued to the correlator's main context, where the adapter service monitor runs. The File adapter's service monitor is responsible for emitting the events that are enqueued from other contexts to the File adapter, and for enqueueing the events received from the File adapter to whichever context should process them (as specified in the `OpenFileForWritingToContext` event).

The `OpenFileForWritingToContext` event is defined as follows

```
event OpenFileForWritingToContext
{
    context instanceContext;
    string fileChannel;
    OpenFileForWriting fileEvent;
}
```

Using the `OpenFileForWritingToContext` event is similar to using the `OpenFileForReadingToContext` event. See ["Opening files for reading with parallel processing applications" on page 113](#) for an example use of the `OpenFileForReadingToContext` event.

Opening files for writing

LineWritten event

After the File adapter writes a line to a file, the adapter sends a `LineWritten` event. The event definition is as follows:

```
event LineWritten
{
    /* The name of the transport being used within the File adapter */
    string transportName;

    /* Request ID this event is in response to. */
    integer requestId;

    /* The sessionId the FileLine was sent for */
    integer sessionId;
}
```

This is useful when you want your application to send `FileLine` events in a batch to control flow. If you need to do flow control, you would typically set all the `requestIds` to *positive* values and emit the next `FileLine` events only after receiving the `LineWritten` notification for the previous `FileLine` event you sent. If you do *not* need to do flow control, you could set `requestId=-1` for all but the last `FileLine` event, but set it to a positive value for the very last `FileLine` event so you get a single `LineWritten` notification when everything has been written.

The file remains open for the lifetime of the adapter unless you emit a `CloseFile` event. See ["Opening files for reading" on page 111](#) for the `CloseFile` event definition.

Using the Apama File Adapter

Monitoring the File adapter

You can use the File adapter status manager (`FileStatusManager.mon` in the `adapters\monitors` directory) to monitor the state of the File adapter.

The File adapter sends status events to the correlator, some of which are asynchronous (not requested) status messages. This occurs as a result of connection status changes, which happen in response to a file being closed or opened.

For single files, the File adapter sends an `AdapterConnectionOpenedEvent` when it opens a new file for reading or writing, and an `AdapterConnectionClosedEvent` when it closes a file. When the File adapter uses a wildcard pattern to open a series of files, in addition to those events, the File

adapter sends an `AdapterConnectionClosedEvent` event after it has read everything in a file, and an `AdapterConnectionOpenedEvent` event when it opens the next file. This is an analogous pattern to the `EndOfFile` and `NewFileOpened` events sent by the adapter itself.

Using the Apama File Adapter

Chapter 4: Using Adapter Plug-ins

■ The Null Codec plug-in	123
■ The File Transport plug-in	125
■ The String Codec plug-in	127
■ The Filter Codec plug-in	127
■ The XML codec plug-in	131
■ The CSV codec plug-in	143
■ The Fixed Width codec plug-in	145

Apama provides several standard codec and transport plug-ins for your convenience, which can be used for testing or in combination with custom plug-ins.

The compiled binaries for all the standard plug-ins are copied to the `\bin` and `\lib` directories (for the C and Java versions respectively) automatically if you chose Developer during the Apama installation.

Information on where to find the source code and how to build those plug-ins for which source code is available can be found in ["IAF samples" on page 274](#).

Using Standard Adapters

The Null Codec plug-in

The `NullCodec`/`JNullCodec` codec layer plug-ins are very useful in situations where it does not make sense to decouple the transport and codec layers. The transport layer plug-in might be best placed to perform all the necessary encoding and/or decoding of events, and to supply and receive Apama normalized events, rather than custom transport-specific messages.

The Null Codec plug-in is provided to make it easy to develop such transport plug-ins. This is a trivial codec layer plug-in that passes downstream normalized events from the transport layer to the Semantic Mapper, and upstream normalized events from the Semantic Mapper to the transport layer with no modification.

In order to load this plug-in, the `<codec>` element in the adapter's configuration file needs to load the `NullCodec` or `JNullCodec` library (this represents the filename of the library that implements the plug-in). Note that for the Java version, the full path to the plug-in's `.jar` file needs to be specified.

A configuration file for C/C++ uses this:

```
<codec name="NullCodec" library="NullCodec">
```

In a configuration file for Java:

```
<codec name="JNullCodec"
      jarName="Apama_install_dir\lib\JNullCodec.jar"
      className="com.apama.iaf.codec.nullcodec.JNull
```

Note: The `NullCodec` and `JNullCodec` plug-ins can only be used with transport plug-ins that understand `NormalisedEvent` objects. The Null Codec plug-ins expect downstream `NormalisedEvent`

objects from the transport and pass upstream `NormalisedEvent` objects it receives directly to the transport plug-in. Using the Null Codec plug-ins with a transport that expects any other kind of object does not work and can possibly crash the adapter.

Null codec transport-related properties

This codec plug-in supports standard Apama properties that are used to specify the name of the transport that will send upstream messages.

Transport-related properties

- **transportName** - This property specifies the transport that the codec should send upstream events to. The property can be used multiple times. The codec maintains a list of all transport names specified in the IAF configuration file. A `transportName` property with an empty value is ignored by the codec.

If no transports are provided in the configuration file then the codec saves the last added `EventTransport` as the default transport. An upstream event is sent to the default transport if no transport information is provided in the normalized event or in the IAF configuration file.

- **transportFieldName** - This property specifies the name of the normalized event field whose value gives the name of the transport that the codec should send the upstream event to. You can also provide a transport name by specifying a value in the `__transport` field. Empty values of these fields are ignored and treated as if not present.
- **removeTransportField** - The value of this property specifies whether the transport related fields should be removed from the upstream event before sending it to transport. The default value is `true`. If the property is set then the field specified by the `transportFieldName` property and the field named `__transport` are removed from the upstream event if they are present. Values `'yes'`, `'y'`, `'true'`, `'t'`, `'1'` ignore cases and are treated as `true` for this property; any other value is treated as `false`.

Upstream behavior

The plug-in's behavior when an upstream event is received proceeds in this order:

1. The codec gets the name of the field that contains the transport name from the value of `transportFieldName` property. From the specified field, the codec then gets the transport name and sends the event to that transport. If the `transportFieldName` property is not specified, if the value of the property is empty, if the field is not present in the event, or if the transport name is empty then the codec tries [2].

For example, the following configuration specifies two transports and the filter codec specifies a transport field named `TRANSPORT`:

```
<transports>
  <transport name="MARKET_DATA" library="transport-lib">
    <property name="Host" value="datahost.com" />
    <property name="Port" value="444" />
  </transport>
  <transport name="ORDER_MANAGEMENT" library="transport-lib">
    <property name="Host" value="orderhost.com" />
    <property name="Port" value="1234" />
  </transport>
</transports>
<codecs>
  <codec name="NullCodec" library="NullCodec">
    <property name="transportFieldName" value="TRANSPORT"/>
    ...
  </codec>
</codecs>
```

The IAF can now route any upstream event that defines a `TRANSPORT` field to one of these two transports. The value of the `TRANSPORT` field, either `MARKET_DATA` or `ORDER_MANAGEMENT`, determines the transport. Note: If the `removeTransportField` property is set `true` or not defined, then the `TRANSPORT` field and `__transport` will be removed (if present) from the upstream event before sending it to transport.

2. The codec gets the transport name from the `__transport` field of the normalized event and sends the event to specified transport. If the `__transport` field is not present or if the transport name specified is empty, the codec then tries [3].

For example, in the above configuration, consider an upstream event that does not have a `TRANSPORT` field or the value of the field is empty. If this event has a value in the `__transport` field of either `MARKET_DATA` or `ORDER_MANAGEMENT`, then that value determines the transport.

3. The codec loops through all transports specified in the `transportName` property and sends the event to the transport. If no transport is specified then the codec tries [4]. Note that the codec ignores all transport names that are empty.

If an exception occurs while sending the event to any transport, then the codec logs the exception and continues sending events to the remaining transports. If the codec was able to send the event to at least one transport, then it does not throw an exception; otherwise, it throws the last captured exception.

For example, the following configuration specifies two transports:

```
<transports>
  <transport name="MARKET_DATA" library="transport-lib">
    <property name="Host" value="datahost.com" />
    <property name="Port" value="444" />
  </transport>
  <transport name="ORDER_MANAGEMENT" library="transport-lib">
    <property name="Host" value="orderhost.com" />
    <property name="Port" value="1234" />
  </transport>
</transports>
<codecs>
  <codec name="NullCodec" library="NullCodec">
    <property name="transportName" value="ORDER_MANAGEMENT"/>
    ...
  </codec>
</codecs>
```

In this example, the codec has not defined the `transportFieldName` property. The IAF will route any upstream event that does not contain a `__transport` field or has empty value in that field to the `ORDER_MANAGEMENT` transport.

4. If a default transport name is present, then the codec sends the event to that transport. The default transport is the last-added transport. If a default transport is also not found then, it throws an exception.

The File Transport plug-in

The `FileTransport/JFileTransport` transport layer plug-ins can read and write messages both from and to a text file. This makes it very convenient for testing string encoding and decoding, semantic mappings, and EPL code, because a text file with some sample messages can be put together quickly and then run through the IAF. Similarly in the upstream direction it allows messages to be written to a file instead of an external message sink such as a middleware message bus.

Messages (or events) are read from and written to named files. Each line of the input file is taken to be a single input event. Each output event is written to a new line of the output file.

In order to load this plug-in, the `<transport>` element in the adapter's configuration file must load the `FileTransport` library (this represents the filename of the library that implements the plug-in). Note that for the Java version, the full path to the plug-in's `.jar` file must be specified.

A configuration file for C/C++ would use this:

```
<transport name="FileTransport" library="FileTransport">
```

In a configuration file for Java:

```
<transport name="JFileTransport"
  jarName="Apama_install_dir\lib\JFileAdapter.jar"
  className="com.apama.iaf.transport.file.JFileTransport">
```

The File Transport plug-in takes the following properties:

- `input` — Specifies the name of the input file.
- `output` — Specifies the name of the output file.
- `cycle` — Specifies the number of times that the plug-in should cycle through the input file. Any value less than zero causes the plug-in to cycle endlessly, until the adapter is either shut down or re-configured. A zero value (the default if the property is missing) means 'no cycling' and results in the same behavior as if the value of this property was '1'.

For more information on specifying plug-ins in an adapter's configuration file, see ["Transport and codec plug-in configuration" on page 257](#).

The plug-in automatically stops after reading the entire input file the requested number of times. If the adapter is subsequently asked to reload its configuration, the plug-in starts running again, using the current property values in the configuration file. If the adapter configuration is reloaded while the plug-in is running, the new configuration will not take effect until the plug-in reaches the end of the current input file. In this case, a second reload request is required before the plug-in will actually start reading the new file.

By default, the File Transport plug-in always communicates with the event codec using Java `String` objects. Therefore, the String Codec plug-in is a suitable companion as it provides a mechanism for converting between `String` objects and normalized events.

There are some minor differences between the C and Java implementations:

- In the **C** version, if no input filename is specified, the standard input stream is used; similarly if no output filename is specified the standard output stream is used.
- In the **Java** version, there is an extra property called `upstreamNormalised`. If this is specified and set to `true`, the File Transport communicates with its codec using `NormalisedEvent` objects rather than `String` objects. In this configuration it should be used with the `JNullCodec`, which does not perform any encoding or decoding but simply passes the unchanged `NormalisedEvent` objects between the codec and transport layers. If `upstreamNormalised` is set to `true`, the File Transport uses the functionality of the `JStringCodec` class to perform encoding/decoding, and all the properties available for use with the `JStringCodec` plug-in class can be specified as properties to the `JFileTransport`.

This is one of the sample plug-ins for which source code is available – see ["IAF samples" on page 274](#) for more information.

The String Codec plug-in

The `StringCodec`/`JStringCodec` codec plug-ins read transport events as simple text strings and breaks them into fields, names and values, using delimiter strings supplied by configuration properties.

Events are assumed to have the following general format:

```
<name1><sepA><value1><sepB><name2><sepA><value2><sepB>...<namen><sepA><valuen><terminator>
```

where `<name>` corresponds to the field name, followed by a delimiter character or string `<sepA>`, followed by the field's value, `<value>`. The complete `<name>` and `<value>` pair is then separated from another such sequence by a `<sepB>` delimiter. This pattern is assumed to repeat itself.

Fields with empty values are permitted. Because the terminator is optional, the codec will consume names and values up to the end of the input string if no terminator is found.

In order to load this plug-in, the `<codecs>` element in the adapter's configuration file must load the `StringCodec` library (this represents the filename of the library that implements the plug-in). Note that for the Java version, the full path to the plug-in's `.jar` file must be specified.

A configuration file for C/C++ would use this:

```
<codec name="StringCodec" library="StringCodec">
```

In a configuration file for Java:

```
<codec name="JStringCodec"
  jarName="Apama_install_dir\lib\JFileAdapter.jar"
  className="com.apama.iaf.codec.string.JStringCodec">
```

The String Codec plug-in takes the following properties:

- `NameValueSeparator` — The string used to separate names and values (`<sepA>` above).
- `FieldSeparator` — The string used to separate fields (`<sepB>` above).
- `Terminator` — The string used to mark the end of the event string.

All properties must be specified in the adapter configuration file.

For more information on specifying plug-ins in an adapter's configuration file, see ["Transport and codec plug-in configuration" on page 257](#).

This is one of the sample plug-ins for which source code is available – see the ["IAF samples" on page 274](#) for more information.

The Filter Codec plug-in

The Apama filter codec plug-ins filter normalized event fields. You can use the filter codec to:

- Route upstream events to particular transports
- Remove particular fields from upstream and/or downstream events

To use the filter codec, the `FilterCodec` or `JFilterCodec` library must be available to the IAF at runtime. These are the filenames of the C++ and Java libraries that implements the plug-in.

In order to load this plug-in, the `<codec>` element in the adapter's configuration file needs to load either the `FilterCodec` or `JFilterCodec` library. Note that for the Java version, the full path to the plug-in's `.jar` file needs to be specified.

A configuration file for C/C++ uses this:

```
<codec name="FilterCodec" library="FilterCodec">
```

In a configuration file for Java:

```
<codec name="JFilterCodec"
  jarName="Apama_install_dir\lib\JFilterCodec.jar"
  className="com.apama.iaf.codec.filtercodec.JFilterCodec">
```

To configure the filter codec, add the following to the `<codecs>` section of the IAF configuration file:

```
<codec name="FilterCodec" library="FilterCodec">
  <property name="transportFieldName" value="transport_field_name"/>
  <property name="filter_spec_1" value="filter_condition_1"/>
  <property name="filter_spec_2" value="filter_condition_2"/>
  ...
  <property name="filter_spec_n" value="filter_condition_n"/>
</codec>
```

Details for replacing the variables in the above `codec` section are in the following topics:

- ["Filter codec transport-related properties" on page 128](#)
- ["Specifying filters for the filter codec" on page 130](#)
- ["Examples of filter specifications" on page 130.](#)

Filter codec transport-related properties

This codec plug-in supports standard Apama properties that are used to specify the name of the transport that will send upstream messages.

Transport-related properties

- **transportName** - This property specifies the transport that the codec should send upstream events to. The property can be used multiple times. The codec maintains a list of all transport names specified in the IAF configuration file. A `transportName` property with an empty value is ignored by the codec.

If no transports are provided in the configuration file then the codec saves the last added `EventTransport` as the default transport. An upstream event is sent to the default transport if no transport information is provided in the normalized event or in the IAF configuration file.

- **transportFieldName** - This property specifies the name of the normalized event field whose value gives the name of the transport that the codec should send the upstream event to. You can also provide a transport name by specifying a value in the `__transport` field. Empty values of these fields are ignored and treated as if not present.
- **removeTransportField** - The value of this property specifies whether the transport related fields should be removed from the upstream event before sending it to transport. The default value is `true`. If the property is set then the field specified by the `transportFieldName` property and the field named `__transport` are removed from the upstream event if they are present. Values `'yes'`, `'y'`, `'true'`, `'t'`, `'1'` ignore cases and are treated as `true` for this property; any other value is treated as `false`.

Upstream behavior

The plug-in's behavior when an upstream event is received proceeds in this order:

1. The codec gets the name of the field that contains the transport name from the value of `transportFieldName` property. From the specified field, the codec then gets the transport name and sends the event to that transport. If the `transportFieldName` property is not specified, if the value of the property is empty, if the field is not present in the event, or if the transport name is empty then the codec tries [2].

For example, the following configuration specifies two transports and the filter codec specifies a transport field named `TRANSPORT`:

```
<transports>
  <transport name="MARKET_DATA" library="transport-lib">
    <property name="Host" value="datahost.com" />
    <property name="Port" value="444" />
  </transport>
  <transport name="ORDER_MANAGEMENT" library="transport-lib">
    <property name="Host" value="orderhost.com" />
    <property name="Port" value="1234" />
  </transport>
</transports>
<codecs>
  <codec name="FilterCodec" library="FilterCodec">
    <property name="transportFieldName" value="TRANSPORT"/>
    ...
  </codec>
</codecs>
```

The IAF can now route any upstream event that defines a `TRANSPORT` field to one of these two transports. The value of the `TRANSPORT` field, either `MARKET_DATA` or `ORDER_MANAGEMENT`, determines the transport. Note: If the `removeTransportField` property is set `true` or not defined, then the `TRANSPORT` field and `__transport` will be removed (if present) from the upstream event before sending it to transport.

2. The codec gets the transport name from the `__transport` field of the normalized event and sends the event to specified transport. If the `__transport` field is not present or if the transport name specified is empty, the codec then tries [3].

For example, in the above configuration, consider an upstream event that does not have a `TRANSPORT` field or the value of the field is empty. If this event has a value in the `__transport` field of either `MARKET_DATA` or `ORDER_MANAGEMENT`, then that value determines the transport.

3. The codec loops through all transports specified in the `transportName` property and sends the event to the transport. If no transport is specified then the codec tries [4]. Note that the codec ignores all transport names that are empty.

If an exception occurs while sending the event to any transport, then the codec logs the exception and continues sending events to the remaining transports. If the codec was able to send the event to at least one transport, then it does not throw an exception; otherwise, it throws the last captured exception.

For example, the following configuration specifies two transports:

```
<transports>
  <transport name="MARKET_DATA" library="transport-lib">
    <property name="Host" value="datahost.com" />
    <property name="Port" value="444" />
  </transport>
  <transport name="ORDER_MANAGEMENT" library="transport-lib">
    <property name="Host" value="orderhost.com" />
  </transport>
</transports>
```

```

        <property name="Port" value="1234" />
      </transport>
    </transports>
  </codecs>
  <codec name="FilterCodec" library="FilterCodec">
    <property name="transportName" value="ORDER_MANAGEMENT"/>
    ...
  </codec>
</codecs>

```

In this example, the codec has not defined the `transportFieldName` property. The IAF will route any upstream event that does not contain a `__transport` field or has empty value in that field to the `ORDER_MANAGEMENT` transport.

4. If a default transport name is present, then the codec sends the event to that transport. The default transport is the last-added transport. If a default transport is also not found then, it throws an exception.

Specifying filters for the filter codec

You specify each filter as a codec property. The filter codec plug-in applies each filter you specify to incoming and outgoing events as they pass through the codec. The property name identifies the field(s) that the filter applies to and the property value specifies the condition that must be true for the filter to operate.

The general syntax of a filter specification is:

```
<property name="filter[.direction][.field_name]" value="condition" />
```

<i>direction</i>	Indicates the direction of the events that the filter applies to. Specify downstream, upstream, or both. The default is both.
<i>field_name</i>	Identifies the field that the filter applies to. The default is that the filter applies to all fields in the event.
<i>condition</i>	Specifies the value that the field must have that causes it to be removed from the event.

Examples of filter specifications

The following filter removes the `price` field from upstream events when the value of the `price` field is 0.0:

```
<property name="filter.upstream.price" value="0.0"/>
```

The following filter removes the `name` field from upstream and downstream events when the value of the `name` field is `NULL`:

```
<property name="filter.both.name" value="NULL"/>
```

In upstream events, the following filter removes each field in which the value is 55:

```
<property name="filter.upstream" value="55"/>
```

In upstream and downstream events, the following filter removes each field in which the value is

```
<remove>:
```

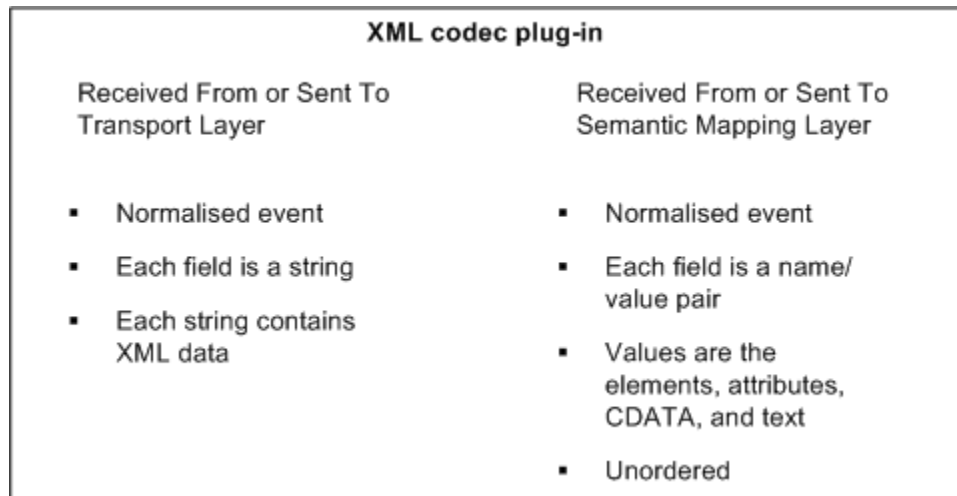
```
<property name="filter" value="<remove>"/>
```

The XML codec plug-in

The Apama XML codec converts messages between the following two formats:

- IAF normalized event whose field values are strings that contain XML data.
- Normalized event in which each field is a name/value pair. These unordered fields contain elements, attributes, CDATA, and text.

Figure 1. XML codec plug-in



To use the XML codec, you must add some information to the IAF configuration file and then set up the classpath. After you do this, you can launch the adapter by running the IAF executable.

For an example configuration file, see `adapters\config\XMLCodec-example.xml.dist` in the Apama installation directory. This file can be changed as required for the purposes of your data and the content added to the adapter configuration file in which the codec is to be used.

Use the information in the following topics to help you configure the XML codec:

- ["Supported XML features" on page 131](#)
- ["Adding XML codec to adapter configuration" on page 132](#)
- ["Setting up the classpath" on page 132](#)
- ["About the XML parser" on page 133](#)
- ["Specifying XML codec properties" on page 133](#)
- ["Description of event fields that represent normalized XML" on page 139](#)
- ["Examples of conversions" on page 141](#)

Supported XML features

The XML codec can convert messages that contain the following:

- Elements

- Attributes
- Text nodes
- CDATA nodes, including CDATA nodes that contain an XML document to be parsed

CDATA nodes are supported only in the downstream direction.

- Namespace prefixes and definitions (only basic support)
- XPath expressions, including functions

Result types of XPath expressions must be simple. For example,

```
string contains();
```

The XML codec cannot convert XML data that contains the following XML features:

- Document type specifiers
- Processing instructions
- Notations and entities
- XML with more than one top-level (root) element
- Node or nodeset XPath expressions

For Node or nodeset XPath expressions, only the first match is returned.

Adding XML codec to adapter configuration

To include the XML codec in the adapter configuration, add the following to the `<codecs>` section of the IAF configuration file:

```
<codec name="XMLCodec"
  className="com.apama.iaf.codec.xml.XMLCodec"
  jarName="@ADAPTERS_JARDIR@\XMLCodec.jar"
>
  <!-- Properties go here -->
</codec>
```

Typically, `@ADAPTERS_JARDIR@` is the `APAMA_HOME\adapters\lib` directory.

For details about the properties that you can specify, see ["Specifying XML codec properties" on page 133](#).

Setting up the classpath

To use the XML codec, ensure the following JAR files in the `APAMA_HOME\lib` directory are in the adapter classpath when you run the IAF.

```
jplugin_public@LIBRARY_VERSION@.jar
util@LIBRARY_VERSION@.jar
jdom.1.0.jar
```

If the XML codec JAR file is in the `APAMA_HOME\adapters\lib` directory, you are all set. The IAF finds these dependencies automatically. Otherwise, set the classpath either as an environment variable or in the `<java>` section of the IAF configuration file.

About the XML parser

On startup, the XML codec logs the names of the classes it is using for XML parsing and XML generation. For example:

```
INFO [11808] - XMLCodec: Encoder initialized: using XML Document builder
               'org.apache.xerces.jaxp.DocumentBuilderImpl'
INFO [11808] - XMLCodec: Decoder initialized: using Streaming API for XML (StAX)
               'com.ctc.wstx.stax.WstxInputFactory'
```

Apama uses Xerces for encoding (creating XML docs) and Woodstox StAX for decoding (parsing).

XML namespace support

If your application relies on the standard XML parsing/generation behavior (that is, not XPath) there is no concept of "declaring namespaces" in the XML codec nor is it required as long as the XML document is valid (that is, it declares any namespace prefixes it uses) then you can just use `namespaceprefix:elementName` when referring to elements in your mapping rules. If there is any doubt, you can run your sample message through the XMLCodec property `logFlattenedXML=true` and it will show you what to specify in your mapping rules, for example, consider the following sample message:

```
<h:table xmlns:h="http://www.myco.com/apama/test/testnamespace_h/"
         xmlns="http://www.myco.com/apama/test/testnamespace_default">
  <h:tr>
    <h:td>Apples</h:td>
    <td>Bananas</td>
  </h:tr>
</h:table>
```

With the above sample message you could use mapping rules such as:

```
<map type="string" default="" apama="default_namespace"
     transport="Body.h:table/@xmlns"/>
<map type="string" default="" apama="prefix_namespace"
     transport="Body.h:table/@xmlns:h"/>
<map type="string" default="" apama="prefixed_element_text"
     transport="Body.h:table/h:tr/h:td/text()"/>
<map type="string" default="" apama="non_prefixed_element_text"
     transport="Body.h:table/h:tr/td/text()"/>
```

If you use XPath in your application, XPath itself contains operators to access the local (non-namespace) name and namespace URI of any XML content. However it is often convenient to define some global prefixes to make it easier to refer to namespaced elements. Apama supports this by allowing any number of `XPathNamespace:myprefix` codec properties, whose value is the URN that the specified prefix should point to. For example,

```
<property name="XPathNamespace:b" value="urn:xmlns:mynamespace"/>
```

would allow XPath expressions to use "b" to refer to elements in the "mynamespace" namespace:

```
<property name="XPath:Test.root/b:elementname/text()"/>
```

Specifying XML codec properties

In the XML codec section of the IAF configuration file, you can set a number of XML properties. For details about setting properties in the IAF configuration file, see ["Plug-in <property> elements" on page 258](#).

When you reload the IAF, any changes to these configuration properties take effect in the codec. In addition to specifying these properties, you must also set up event mappings for XML messages. See ["Event mappings configuration" on page 259](#).

Properties are described in the following topics:

- ["Required XML codec properties" on page 134](#)
- ["Message logging properties" on page 136](#)
- ["Downstream node order suffix properties" on page 136](#)
- ["Additional downstream properties" on page 137](#)
- ["Sequence field properties" on page 137](#)
- ["Upstream properties" on page 137](#)
- ["Performance properties" on page 137](#)

Required XML codec properties

The XML codec requires you to set the `XMLField` and `transportName` properties. All other properties are optional.

`XMLField` — This property identifies the field name that XML will be read from when decoding, and will be written to when encoding. The flattened XML representation is stored in fields with names prefixed with the value you specify for the `XMLField` property.

When you are familiar with how the XML codec behaves, you can specify the `XMLField` property multiple times to parse/generate multiple XML documents per event. Parsing follows the order in which `XMLField` properties appear, and generating XML follows the reverse order.

It is possible to use this mechanism to parse an XML string embedded as CDATA in another XML string. To do this, specify the flattened field name of the CDATA node as an `XMLField`. However, note that sequence fields across separate CDATA nodes are not supported.

`transportName` — The XML codec sends upstream events to the transport that this property identifies. This transport must be defined in the same IAF configuration file.

XML codec transport-related properties

This codec plug-in supports standard Apama properties that are used to specify the name of the transport that will send upstream messages.

Transport-related properties

- `transportName` - This property specifies the transport that the codec should send upstream events to. The property can be used multiple times. The codec maintains a list of all transport names specified in the IAF configuration file. A `transportName` property with an empty value is ignored by the codec.

If no transports are provided in the configuration file then the codec saves the last added `EventTransport` as the default transport. An upstream event is sent to the default transport if no transport information is provided in the normalized event or in the IAF configuration file.

- `transportFieldName` - This property specifies the name of the normalized event field whose value gives the name of the transport that the codec should send the upstream event to. You can also provide a transport name by specifying a value in the `__transport` field. Empty values of these fields are ignored and treated as if not present.
- `removeTransportField` - The value of this property specifies whether the transport related fields should be removed from the upstream event before sending it to transport. The default value is

true. If the property is set then the field specified by the `transportFieldName` property and the field named `__transport` are removed from the upstream event if they are present. Values 'yes', 'y', 'true', 't', '1' ignore cases and are treated as true for this property; any other value is treated as false.

Upstream behavior

The plug-in's behavior when an upstream event is received proceeds in this order:

1. The codec gets the name of the field that contains the transport name from the value of `transportFieldName` property. From the specified field, the codec then gets the transport name and sends the event to that transport. If the `transportFieldName` property is not specified, if the value of the property is empty, if the field is not present in the event, or if the transport name is empty then the codec tries [2].

For example, the following configuration specifies two transports and the filter codec specifies a transport field named `TRANSPORT`:

```
<transports>
  <transport name="MARKET_DATA" library="transport-lib">
    <property name="Host" value="datahost.com" />
    <property name="Port" value="444" />
  </transport>
  <transport name="ORDER_MANAGEMENT" library="transport-lib">
    <property name="Host" value="orderhost.com" />
    <property name="Port" value="1234" />
  </transport>
</transports>
<codecs>
  <codec name="XMLCodec"
    className="com.apama.iaf.codec.xml.XMLCode"
    jarName="@ADAPTERS_JAR\XMLCodec.jar">
    <property name="transportFieldName" value="TRANSPORT"/>
    ...
  </codec>
</codecs>
```

The IAF can now route any upstream event that defines a `TRANSPORT` field to one of these two transports. The value of the `TRANSPORT` field, either `MARKET_DATA` or `ORDER_MANAGEMENT`, determines the transport. Note: If the `removeTransportField` property is set true or not defined, then the `TRANSPORT` field and `__transport` will be removed (if present) from the upstream event before sending it to transport.

2. The codec gets the transport name from the `__transport` field of the normalized event and sends the event to specified transport. If the `__transport` field is not present or if the transport name specified is empty, the codec then tries [3].

For example, in the above configuration, consider an upstream event that does not have a `TRANSPORT` field or the value of the field is empty. If this event has a value in the `__transport` field of either `MARKET_DATA` or `ORDER_MANAGEMENT`, then that value determines the transport.

3. The codec loops through all transports specified in the `transportName` property and sends the event to the transport. If no transport is specified then the codec tries [4]. Note that the codec ignores all transport names that are empty.

If an exception occurs while sending the event to any transport, then the codec logs the exception and continues sending events to the remaining transports. If the codec was able to send the event to at least one transport, then it does not throw an exception; otherwise, it throws the last captured exception.

For example, the following configuration specifies two transports:

```
<transports>
```



```

<transport name="MARKET_DATA" library="transport-lib">
  <property name="Host" value="datahost.com" />
  <property name="Port" value="444" />
</transport>
<transport name="ORDER_MANAGEMENT" library="transport-lib">
  <property name="Host" value="orderhost.com" />
  <property name="Port" value="1234" />
</transport>
</transports>
<codecs>
  <codec name="XMLCodec"
    className="com.apama.iaf.codec.xml.XMLCodec"
    jarName="@ADAPTERS_JAR\XMLCodec\jar">
    <property name="transportName" value="ORDER_MANAGEMENT"/>
    ...
  </codec>
</codecs>

```

In this example, the codec has not defined the `transportFieldName` property. The IAF will route any upstream event that does not contain a `__transport` field or has empty value in that field to the `ORDER_MANAGEMENT` transport.

4. If a default transport name is present, then the codec sends the event to that transport. The default transport is the last-added transport. If a default transport is also not found then, it throws an exception.

Message logging properties

`logFlattenedXML` — If true, the IAF log contains a list of the name/value pairs generated by the XML codec when flattening XML received from the transport, at `CRIT` level. Each field is on a different line, which makes it easy to see what fields are being generated and what the mapping's transport field names should be set to. Turning this on in production impacts performance. The default is false.

`logAllMessages` — If true, the IAF log contains the full contents of every message sent upstream or downstream, before and after encoding, and before and after decoding, all at `CRIT` level. Turning this on in production impacts performance. The default is false.

Downstream node order suffix properties

`generateTwinOrderSuffix` — If true, all field names for text, CDATA and element nodes are appended with `"", "[2]", "[3]"`, and so on. The number specifies the position of this node relative to 'twins', that is, nodes of the same type and name. These order suffixes provide a partial order for the XML nodes. Note that the first child node with a given name is defined to have no suffix (rather than an explicit `"[1]"`), to improve readability. The default is false.

Use this property when you need to map fields without sensitivity to the precise order in which differently named nodes appear in the XML. This is probably a more useful option than setting the `generateSiblingOrderSuffix` property for most users of the XML codec.

`generateSiblingOrderSuffix` — If true, all field names for text, CDATA and element nodes (except the root element) are appended with `"#1", "#2"`, and so on. The number specifies the position of this node relative to all its siblings (of any type, such as element or CDATA.). These order suffixes provide a total order for the XML nodes. The default is true.

Use this property when you need to map fields using the precise order in which differently named nodes appear in the XML, or for total control over node ordering when generating XML upstream.

Examples of both suffixes are in ["Description of event fields that represent normalized XML" on page 139](#) and ["Examples of conversions" on page 141](#).

You can set both node order properties to true. For sample output when both are set to true, see ["Examples of conversions" on page 141](#). The default values of these two properties may change in a future release, so the recommendation is to explicitly specify both properties according to the behavior required.

Additional downstream properties

`XPath: XMLField -> ResultField` — The value of this property specifies an XPath expression that should be evaluated for the specified `XMLField`, with the result put into the `ResultField` in the normalized event. Only simple data types (boolean/float/string) can be returned at present, so XPath expressions that match multiple nodes only return the first matching node. See ["XPath examples" on page 143](#).

`trimXMLText` — If true, the XML codec removes any leading or trailing whitespace characters from XML text data in downstream messages before adding the text to the normalized event. The default is true.

Sequence field properties

`sequenceField` — The value of this property is a field that is treated as a sequence. This means that all XML nodes that match this name are translated to a single entry in the normalized event, in the form of an EPL `sequence` of type `string`. The element name should be a plain name, without a node order suffix. In other words, the value of this property and the field in the outgoing event should be in the form: `elementA/elementB/@attrib`. You can specify this property multiple times.

`ensurePresent` — This property specifies an attribute, text string or CDATA node of an element that will be added to the output event as a blank string even if it is not present in the XML. This is mostly useful for fields identified with the `sequenceField` property, as empty strings get added to the sequence for optional attributes. You can specify this property multiple times.

`separator: elementName` — Whenever the specified element occurs in the XML message, the value of this property is prepended to any sequences in nodes below the specified element. See ["Sequence field example" on page 142](#).

Upstream properties

`indentGeneratedXML` — If true, the generated XML is indented to make it easier to read. The default is false.

`omitGeneratedXMLDeclaration` — If true, the `<?xml ... ?>` declaration at the start of the generated XML is not included. The default is false.

Performance properties

`skipNullFields` — A boolean that indicates whether you want the XML codec to omit nodes with null values from downstream, flattened, normalized events. Specify true to omit nodes with null values. The default is false.

The `skipNullFields` property applies to the name/value pairs for XML elements themselves. These have no associated data, so generating normalized event fields for them is not necessary unless they are required for ID rules. The `skipNullFields` property does not apply to a node whose value is an empty string.

Setting `skipNullFields` to true has no effect on the ordering suffixes that the codec adds to nodes. For example, consider an XML element that is deep within an XML hierarchy such as the following:

```
<root>
  <a>
```

```

    <b>
      <c>
        I want this string
      </c>
    </b>
  </a>
</root>

```

In the downstream direction, the XML codec creates a normalized event that contains a dictionary of name/value pairs that includes an entry for each element. If you specify sibling suffixes and `Test` as the XML field name, the dictionary contains the following:

```

{ "Test.root/":null ,
  "Test.root/a#1/":null ,
  "Test.root/a#1/b#1/":null ,
  "Test.root/a#1/b#1/c#1/":null ,
  "Test.root/a#1/b#1/c#1/text()#1:"I want this string" }

```

Unless you require one of the null value fields for an ID rule, you do not need the null value fields. If you set `skipNullFields` to true, the XML codec drops the null value fields from the normalized event. In this example, the result is a dictionary with one entry:

```

{ "Test.root/a#1/b#1/c#1/text()#1:"I want this string" }

```

As you can see, this is much more lightweight. Turning this feature on can sometimes improve throughput by up to 1.5 times.

`parseNode` — Specify this property one or more times to identify only those nodes that you want parsed, flattened, and added to the normalized event.

By default, the XML codec parses, flattens, and adds all nodes to the normalized event. If you specify one or more `parseNode` property entries, the XML codec processes only the node or nodes specified by a `parseNode` property.

The value of a `parseNode` property can be any node path. The codec ignores order suffixes (`#n` or `[n]`) if you specify them in node paths. In other words, the codec parses all elements of the type specified in the `parseNode` property.

For example, suppose the value of the XML field property is `Test` and you have the following XML:

```

<root>
  <a>ignore me</a>
  <b>look at me</b>
  <c>look at me</c>
  <b>look at me again</b>
</root>

```

You can specify the following `parseNode` properties:

```

<property name="parseNode" value="Test.root/b/text()" />
<property name="parseNode" value="Test.root/c[9999999999]/text()" />

```

The XML codec produces the following dictionary entries:

```

"Test.root/b#1/text()#1" = "look at me"
"Test.root/c#2/text()#1" = "look at me"
"Test.root/b#3/text()#1" = "look at me again"

```

As you can see, the XML codec ignores the `[9999999999]` suffix.

Typically, you would specify the following `parseNode` properties:

- For each mapping rule, specify a `parseNode` property whose value is the transport field for that rule.
- For each ID rule in the adapter configuration file, specify a `parseNode` property whose value is the field name.

It is not necessary to specify `parseNode` properties for nodes identified by `sequenceField` or `separator:elementName` properties.

Setting the `parseNode` property prevents some nodes from being parsed. Consequently, the order of subsequent nodes might change, and therefore they would have different node order suffixes. For this reason, you probably want to set the `logFlattenedXML` property to true to see in what order suffixes are being generated before you add `parseNode` properties. Then add the `parseNode` properties and update the node paths used in mapping and ID rules as needed.

Specifying `parseNode` properties instead of parsing the entire document can result in very substantial throughput improvements. This is especially true for documents in which only a small proportion of the XML is actually going to be mapped.

Description of event fields that represent normalized XML

As mentioned before, a single XML field on the transport side is represented on the correlator side as a series of name/value fields, all prefixed by the value you specified for the `XMLField` property. This section describes how the XML codec names fields, based on the XML data.

Note, any field not specified as an `XMLField` for the `XMLCodec` will pass through the system as normal. These fields are not dropped/ignored.

If there is any uncertainty about the correct transport field names to use in the IAF mapping rules, try setting the `logFlattenedXML` codec property to true.

To preserve XML node ordering information, the codec adds ordering information to node names by appending a suffix according to the suffix generation mode enabled — either "", "#2", "#3", and so on or "[1]", "[2]", "[3]", and so on.

The "#n" sibling format provides a total ordering across all child nodes under a given parent, specifying each node's position relative to all of its sibling nodes. This suffix mode is the default. To turn it off, set the `generateSiblingOrderSuffix` codec property to false. Note that the root node never has a sibling order suffix because only one root exists. Sample field names:

```
Field1.message/element#1/  
Field1.message/other_element#2/  
Field1.message/other_element#3/
```

The twin "[n]" format is insensitive to the order in which nodes appear as long as they have different names, and it specifies a node's position relative to its twin nodes. (Twins are siblings with the same node name.) This suffix mode is disabled by default (for backwards compatibility). To turn it on, set the `generateTwinOrderSuffix` codec property to true. To improve readability the first sibling node with a given name has no suffix. That is, the [1] suffix is implicit. Sample field names:

```
Field1.message/element/  
Field1.message/element[2]/  
Field1.message/other_element/  
Field1.message/other_element[2]/  
Field1.message/other_element[3]/  
Field1.message/yet_another_element/  
Field1.message/yet_another_element[2]/
```

Note that for a message to be correctly translated in the upstream direction (from the correlator), there do not have to be enough suffixes in the event to form a total order, but any suffixes that are provided will be used. In the absence of sibling order suffixes to determine ordering of different node types, the XML codec generates the XML nodes in the following order:

1. Text data

2. CDATA

3. Elements

The XML codec maps XML elements, attributes, CDATA and text data as described in the following sections. In the following topics, assume that the value of the `XMLField` property is `Test`.

Elements

An XML element maps to a field with the following characteristics:

- The name is separated and terminated with the `'/'` character.
- The value is an empty string (`""`).

For example, an element `B` nested inside an element `A` is represented in the normalized event as follows:

```
"Test.A/B#1/" = ""
```

When the XML codec generates XML for upstream events, it is not a requirement to have an associated field for every element. The XML codec automatically creates ancestor XML elements when they do not have associated fields. For example, consider the following field:

```
"Test.A/B#1/@att" = ""
```

If necessary, the codec creates the `A` and `B` element nodes.

Element attributes

XML element attributes map to fields with names equal to the parent element's field name, followed by `'@att'` where `att` is the name of the attribute, and the field's value is the attribute value. For example, an attribute `B` of an element `A` with the value `Hello` is represented as follows:

```
"Test.A/@B" = "Hello"
```

CDATA

XML CDATA in an element maps to a field with a name equal to the parent element's field name followed by `CDATA()` and a value that contains the text data. For example, an element `A` with CDATA `" Hello "` followed by sub-element `B` followed by CDATA `" World "` is represented as follows:

```
"Test.A/CDATA()#1" = " Hello "  
"Test.A/B#2/" = ""  
"Test.A/CDATA()#3" = " World "
```

Text data

Text data in an XML element maps to a field with a name equal to the parent element's field name followed by `text()`. The value of the field is the text data. Unless the `trimXMLText` is false (the default is that it is true), the codec strips leading and trailing whitespace from text data. For example, an element `A` that contains the text `" Hello World "` followed by sub-element `B` followed by text `" ! "` is represented as follows:

```
"Test.A/text()#1" = "Hello World"  
"Test.A/B#2/" = ""  
"Test.A/text()#3" = "!"
```

In the event of errors during XML parsing, the parser

- Logs the errors in the IAF log file

- Tries to send to the semantic mapper a flattened, normalized event that contains the remaining fields

Examples of conversions

Suppose that the value of the `XMLField` property is `Test`, and the value of the `trimXMLText` property is `true`. Consider the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<Message>
  <ElementA>
    Hello there
  </ElementB/>
  !
  <ElementC/>
  <![CDATA[Sample CDATA (with < and > comparison operators). ]]>
  <ElementB att1="X" att2="Y">
    <![CDATA[More CDATA in the same element.]]>
  </ElementB>
</ElementA>
</Message>
```

With sibling order suffixing, this XML maps to the following normalized event fields:

```
"Test.Message/" =
"Test.Message/ElementA#1/" =
"Test.Message/ElementA#1/text()#1" = "Hello there"
"Test.Message/ElementA#1/ElementB#2/" =
"Test.Message/ElementA#1/text()#3" = "!"
"Test.Message/ElementA#1/ElementC#4/" =
"Test.Message/ElementA#1/CDATA()#5" =
  "Sample CDATA (with < and > comparison operators). "
"Test.Message/ElementA#1/ElementB#6/" =
"Test.Message/ElementA#1/ElementB#6/@att1" = "X"
"Test.Message/ElementA#1/ElementB#6/@att2" = "Y"
"Test.Message/ElementA#1/ElementB#6/CDATA()#1" =
  "More CDATA in the same element."
```

With twin order suffixing, the same XML maps to the following normalized event fields:

```
"Test.Message/" =
"Test.Message/ElementA/" =
"Test.Message/ElementA/text()" = "Hello there"
"Test.Message/ElementA/ElementB/" =
"Test.Message/ElementA/text()[2]" = "!"
"Test.Message/ElementA/ElementC/" =
"Test.Message/ElementA/CDATA()" =
  "Sample CDATA (with < and > comparison operators). "
"Test.Message/ElementA/ElementB[2]/" =
"Test.Message/ElementA/ElementB[2]/@att1" = "X"
"Test.Message/ElementA/ElementB[2]/@att2" = "Y"
"Test.Message/ElementA/ElementB[2]/CDATA()" = "More CDATA in the same element."
```

To construct the XML above (assuming element ordering matters, but allowing for `text()` concatenation), the following name/value pairs are all that is required:

```
"Test.Message/ElementA#1/text()#1" = "Hello there"
"Test.Message/ElementA#1/ElementB#2/" =
"Test.Message/ElementA#1/text()#3" = "!"
"Test.Message/ElementA#1/ElementC#4/" =
"Test.Message/ElementA#1/CDATA()#5" =
  "Sample CDATA (with < and > comparison operators). "
"Test.Message/ElementA#1/ElementB#6/@att1" = "X"
"Test.Message/ElementA#1/ElementB#6/@att2" = "Y"
"Test.Message/ElementA#1/ElementB#6/CDATA()#1" = "More CDATA in the same element."
```

With both sibling order suffixing and twin order suffixing set to true, the XML codec generates two field/value pairs for each node. For example, the same XML used in the previous two examples maps to the following:

```

"Test.Message/" =
"Test.Message/ElementA/" =
"Test.Message/ElementA#1/" =
"Test.Message/ElementA/text()" = "Hello there"
"Test.Message/ElementA#1/text()#1" = "Hello there"
"Test.Message/ElementA/ElementB/" =
"Test.Message/ElementA#1/ElementB#2/" =
"Test.Message/ElementA/text()[2]" = "!"
"Test.Message/ElementA#1/text()#3" = "!"
"Test.Message/ElementA/ElementC/" =
"Test.Message/ElementA#1/ElementC#4/" =
"Test.Message/ElementA/CDATA()" =
    "Sample CDATA (with < and > comparison operators)."
"Test.Message/ElementA#1/CDATA()#5" =
    "Sample CDATA (with < and > comparison operators)."
"Test.Message/ElementA/ElementB[2]/" =
"Test.Message/ElementA#1/ElementB#6/" =
"Test.Message/ElementA/ElementB[2]/@att1" = "X"
"Test.Message/ElementA#1/ElementB#6/@att1" = "X"
"Test.Message/ElementA/ElementB[2]/@att2" = "Y"
"Test.Message/ElementA#1/ElementB#6/@att2" = "Y"
"Test.Message/ElementA/ElementB[2]/CDATA()" = "More CDATA in the same element."
"Test.Message/ElementA#1/ElementB#6/CDATA()#1" = "More CDATA in the same element."

```

Since the suffix properties are orthogonal, you can set both to true, and the XML codec generates normalized fields with each kind of suffix. This allows you to use the same instance of the XML codec for XML elements that need sibling suffixing and XML elements that need twin suffixing. While this impacts memory usage according to the amount of XML data being normalized, you can specify mapping rules to filter for the fields of interest.

Sequence field example

Consider the following XML fragment:

```

<root>
  <prices instr="MSFT">
    <info>1.04</info>
    <info type="SELL">1.03</info>
  </prices>
  <prices instr="IBM">
    <info type="BUY"></info>
    <info type="SELL">1.06</info>
  </prices>
</root>

```

Suppose that the following properties are set in the XML codec section of the IAF configuration file:

```

<property name="XMLField" value="Test"/>
<property name="sequenceField" value="Test.root/prices/@instr"/>
<property name="sequenceField" value="Test.root/prices/info/@type"/>
<property name="sequenceField" value="Test.root/prices/info/text()"/>
<property name="ensurePresent" value="Test.root/prices/info/@type"/>
<property name="ensurePresent" value="Test.root/prices/info/text()"/>
<property name="separator:Test.root/prices" value="(prices)"/>

```

With these property values, the XML fragment maps to the following normalized event fields:

```

"Test.root/" =
"Test.root/prices#1/" =
"Test.root/prices#1/info#1/" =
"Test.root/prices#1/info#2/" =
"Test.root/prices#2/" =
"Test.root/prices#2/info#1/" =

```

```

"Test.root/prices#2/info#2/" =
"Test.root/prices/@instr"   = ["(prices)", "MSFT", "(prices)", "IBM"]
"Test.root/prices/info/@type" = ["(prices)", "", "SELL", "(prices)", "BUY", "SELL"]
"Test.root/prices/info/text()" = ["(prices)", "1.04", "1.03", "(prices)", "", "1.06"]

```

If you define the following mapping rules in the IAF configuration file, you can map these normalized event fields to and from string fields in a sequence field of an Apama event.

```

<mapping-rules>
  <map transport="Test.root/prices/@instr"
    apama="instruments" type="reference"
    referencetype="sequence <string>" default="[]"/>
  <map transport="Test.root/prices/info/@type"
    apama="types" type="reference"
    referencetype="sequence <string>" default="[]"/>
  <map transport="Test.root/prices/info/text()"
    apama="prices" type="reference"
    referencetype="sequence <string>" default="[]"/>
</mapping-rules>

```

XPath examples

Consider the following XML:

```

<root>
  text1
  <a att="100.1">A text 1</a>
  <a>A text 2</a>
  <b att="300.0">
    <a att="400.4"/>
  </b>
  This is an interesting text string
</root>

```

Suppose that the following properties are set in the XML codec section of the IAF configuration file:

```

<property name="XMLField" value="Test"/>
<property name="XPath:Test->MyXPathResult.last-a" value="*/a[last()]/>
<property name="XPath:Test->MyXPathResult.first-att" value="//@att"/>
<property name="XPath:Test->MyXPathResult.first-a-text" value="/root/a[1]/text()"/>
<property name="XPath:Test->MyXPathResult.att>200" value="//@att>200"/>
<property name="XPath:Test->MyXPathResult.att-count" value="count(//@att)"/>
<property name="XPath:Test->MyXPathResult.text-contains"
  value="contains(/cdata-root/text()[last()], &quot;interesting&quot;)/>

```

With these property values, the XML fragment maps to the following normalized event fields:

```

"MyXPathResult.last-a"      = "A text 2"
"MyXPathResult.first-att"   = "100.1"
"MyXPathResult.first-a-text" = "A text 1"
"MyXPathResult.att>200"     = "true"
"MyXPathResult.att-count"   = "3"
"MyXPathResult.text-contains" = "true"

```

The CSV codec plug-in

The CSV codec plug-in (`JCSVCodec`) translates between comma separated value (CSV) data and a sequence of string values. This codec (or the Fixed Width codec plug-in) can be used with the standard Apama File adapter to read data from files and to write data to files. (For more information on the Fixed Width codec, see ["The Fixed Width codec plug-in" on page 145](#); for more information on the standard Apama File adapter, see .)

CSV format is a simple way to store data on a value by value basis. Consider an example CSV file that contains stock tick data. The lines in the file are ordered by Symbol, Exchange, Current Price, Day High, and Day Low, as follows:

```
TSCO, L, 395.50, 401.5, 386.25
MKS, L, 225.25, 240.75, 210.25
```

In this example, each field is separated from the next by a comma. You can use other characters as separators as long as you identify the separator character for the CSV codec.

To specify a separator character other than a comma, do one of the following:

- Send a configuration event from the transport that is communicating with the CSV codec using the method described in the topic, ["Multiple configurations and the CSV codec" on page 144](#).
- Set the `separator` property in the IAF configuration file that you use to start the File adapter. For example:

```
<property name="separator" value=" " />
```

If you set the `separator` property, the codec uses the separator you specify by default. If you do not specify the `separator` property, and the codec does not receive any configuration events before receiving messages to encode or decode, the codec refuses to process messages. The codec throws an exception back to the module that called it, which is either the transport or the semantic mapper depending on whether the data is flowing downstream or upstream.

For an example configuration file, see `adapters\config\JCSVCodec-example.xml.dist` in the Apama installation directory. The `JCSVCodec-example.xml.dist` file itself should not be modified, but you can copy relevant sections of the XML code, modify the code as required for the purposes of your data, and then add the modified content to the adapter configuration file in which the codec is to be used.

Multiple configurations and the CSV codec

The CSV codec supports multiple configurations for interpreting separated data from different sources. A transport that is using the CSV codec can use the `com.apama.iaf.plugin.ConfigurableCodec` interface to set up different configurations for interpreting data from multiple sources that use different formats.

The transport can set a configuration by calling the following method on the codec:

```
public void addConfiguration(int sessionId,
                             NormalisedEvent configuration)
    throws java.io.IOException
```

The `sessionId` represents the ID value for this configuration.

The normalized event should contain the following key/value pair stored as strings that will be parsed in the codec:

Key	Value
<code>separator</code>	A string that contains the character to be used as the separator value, for example, <code>" , "</code> or <code>" ; "</code> .

The transport can remove a configuration by calling the following method:

```
void removeConfiguration(int sessionId) throws java.io.IOException
```

The `sessionId` represents the ID value initially used to add the configuration with the `addConfiguration()` method.

Decoding CSV data from the sink to send to the correlator

To decode an event into a sequence of fields, the transport can then call:

```
public void sendTransportEvent(Object event, TimestampSet timestamps)
    throws CodecException, SemanticMapperException
```

The event object is assumed to be a `NormalisedEvent` instance. It must contain a key of `data`, which has a value of `string` type that contains the data to decode. That is, the `string` contains the line containing the separated data. The codec then decodes the data, and stores the value from each field in a string sequence. This value from each field replaces the value for the data key.

If the event object also contains a `sessionId` key with an integer value associated with it, the value of the key identifies the configuration the codec uses to interpret the data. If the event does not contain a `sessionId`, the codec uses the default configuration as specified in the adapter configuration file.

Encoding CSV data from the correlator for the sink

Encoding CSV data works in the exact opposite way as decoding. The semantic mapper calls:

```
public void sendNormalisedEvent(NormalisedEvent event,
                                TimestampSet timestamps)
    throws CodecException, TransportException
```

The `sendNormalisedEvent()` method retrieves the data associated with the `data` key. The retrieved data is a sequence of strings, each of which contains the value of a field. The method then encodes the sequence into a single line to send to the transport so the transport can write the data to the sink. The CSV codec stores the result of the encoding in the `data` field. If the event contains a `sessionId` value, this is the configuration that the codec uses to encode the data. If the event does not contain a `sessionId`, the codec uses the default adapter configuration as specified in the adapter's configuration file initially used to start the adapter.

For a given event mapping in the IAF configuration file, it is not possible to dynamically specify the event decoder property, which identifies the codec that sends this event to the transport. Consequently, an adapter that is using several different codecs is unable to receive the same type of event from each codec. If it is necessary for your adapter to receive the same type of event from multiple codecs, set the event decoder property to the Null codec. This lets the transport receive the event and subsequently reroute the event back to the CSV codec by calling the following method:

```
sendNormalisedEvent(NormalisedEvent event, TimestampSet timestamps)
```

The CSV codec then returns the encoded data to the transport.

The Fixed Width codec plug-in

The Fixed Width codec plug-in (`JFixedWidthCodec`) translates between fixed width data and a sequence of string values. This codec (or the CSV codec plug-in) can be used with the standard Apama File adapter to read data from files and write data to files. (For more information on the CSV codec, see ["The CSV codec plug-in" on page 143](#); for more information on the Apama File adapter, see [.](#))

Fixed width data is a method of storing data fields in a packet or a line that is a fixed number of characters in size. Data stored in a fixed width format can be expressed by the following three parameters:

- The field widths used (that is, the number of characters used for storing each field)

- The padding character used if the data for a given field can be stored in less than the number of characters allocated for it
- Whether or not the data is left or right aligned within the field.

For example, consider the following, which describes a tick with ordered properties:

symbol	6 characters
exchange	4 characters
current price	9 characters
day high	9 characters
day low	9 characters

If the pad character is '-', an example of a left-aligned line is as follows:

```
TSCO--L---392.25---400.25---382.25---
```

The following is an example of a right-aligned line:

```
--TSCO---L---392.25---400.25---382.25
```

To specify fixed width data properties, do one of the following:

- Send a configuration event from the transport that is communicating with the Fixed Width codec using the method described in the topic, ["Multiple configurations and the Fixed Width codec" on page 146](#).
- Set the fixed width properties in the IAF configuration file you use to start the adapter. For example, to obtain the left-aligned fixed width data above:

```
<property name="fieldLengths" value="[6,4,9,9,9]"/>
<property name="padCharacter" value="-"/>
<property name="isLeftAligned" value="true"/>
```

If you set all these properties, the codec uses them by default when decoding or encoding events.

If you do not set any of these properties, the codec expects to receive configuration events (as described in ["Multiple configurations and the Fixed Width codec" on page 146](#)), prior to receiving messages to encode or decode. Otherwise, the codec refuses to process these messages. The codec throws an exception back to the module that called it, which is either the transport or the semantic mapper depending on whether the data is flowing downstream or upstream.

If you require a default configuration, be sure to set all of these properties in the configuration file. If you set some of the properties, but not all of them, the codec cannot start.

For an example configuration file, see `adapters\config\JFixedWidthCodec-example.xml.dist` in the Apama installation directory. The `JFixedWidthCodec-example.xml.dist` file itself should not be modified, but you can copy relevant sections of the XML code, modify the code as required for the purposes of your data, and then add the modified content to the adapter configuration file in which the codec is to be used.

Multiple configurations and the Fixed Width codec

The Fixed Width codec supports multiple configurations for interpreting fixed width data from different sources. A transport that is using the Fixed Width codec can use the

`com.apama.iaf.plugin.ConfigurableCodec` interface to set the configuration that you want the adapter to use.

The transport can set a configuration by calling the following method on the codec:

```
public void addConfiguration(int sessionId,
                             NormalisedEvent configuration)
    throws java.io.IOException
```

The `sessionId` represents the ID value for this configuration.

The normalized event should contain key/value pairs that are stored as strings the Fixed Width codec can parse.

Key	Value
<code>fieldLengths</code>	A string sequence that contains the number of characters each field value is stored in. For example, "[5,6,5,9]" where the first value is stored in the first 5 characters, the second value is stored in the next 6 characters, and so on.
<code>isLeftAligned</code>	"true" or "false", depending on whether data is left or right aligned in a field.
<code>padCharacter</code>	"_" where '_' is the pad character used when the data requires padding to fill the field.

The transport can remove a configuration by calling the following method:

```
void removeConfiguration(int sessionId) throws java.io.IOException
```

The `sessionId` represents the ID value initially used to add the configuration using the `addConfiguration()` method.

Decoding fixed width data from the sink to send to the correlator

To decode an event into a sequence of fields, the transport calls the `sendTransportEvent()` method as follows:

```
public void sendTransportEvent(Object event, TimestampSet timestamps)
    throws CodecException, SemanticMapperException
```

The event object is assumed to be a `NormalisedEvent`. It must contain the key 'data', which has a value of string type containing the data to decode. That is, the line that contains the fixed width data. The Fixed Width codec then decodes the data and stores the value from each field in a string sequence. This value from each field replaces the value for the data key.

If the event also contains a `sessionId` key with an integer value associated with it, this is the configuration that the codec uses to interpret the data. If the event does not contain a `sessionId` the codec uses the default configuration as specified in the configuration file.

Encoding fixed width data from the correlator for the sink

Encoding fixed width data works in the exact opposite way to decoding. The semantic mapper calls:

```
public void sendNormalisedEvent(NormalisedEvent event,
    TimestampSet timestamps)
    throws CodecException, TransportException
```

This method retrieves the data associated with the `data` key. The data is in a string sequence where each member contains the value of a field. The method encodes the sequence members into a single line to send to the transport so the transport can write the data to the sink. Finally, the method stores the result of the encoding in the `data` field again.

If the event contains a `sessionId` value, this is the configuration that the codec uses to encode the data. If the event does not contain a `sessionId`, the codec uses the default File adapter configuration as specified in the File adapter configuration file initially used to start the file adapter.

For a given event mapping in the IAF configuration file, it is not possible to dynamically specify the event decoder property, which identifies the codec that sends the event to the transport. Consequently, an adapter that is using several different codecs is unable to receive the same type of event from each codec. If it is necessary for your adapter to receive the same type of event from multiple codecs, set the event decoder property to the Null codec. This lets the transport receive the event and subsequently reroute the event back to the Fixed Width codec by calling the following method:

```
sendNormalisedEvent(NormalisedEvent event, TimestampSet timestamps)
```

The Fixed Width codec then returns the encoded data to the transport.

II Using Message Services

Apama support for Java Message Service (JMS) messaging is integrated into the Apama correlator. This provides an efficient method for Apama applications to support JMS messages for communication with external systems.

Universal Messaging (UM) is Software AG's middleware service that delivers data across different networks. It provides messaging functionality without the use of a web server or modifications to firewall policy. In Apama applications, you can configure and use the connectivity provided by UM.

For messaging between Apama components, the use of UM described here is typically a simpler and more deeply integrated alternative to connecting to a UM realm using correlator-integrated messaging for JMS. However, you should use JMS when Apama is using UM to send messages to and receive messages from non-Apama systems. See ["Using Correlator-Integrated Messaging for JMS" on page 150](#), which supports configurable mapping between Apama event strings and whatever formats the non-Apama components are using for their JMS messages.

Chapter 5: Using Correlator-Integrated Messaging for JMS

■ Overview of correlator-integrated messaging for JMS	150
■ Getting started with simple correlator-integrated messaging for JMS	152
■ Getting started with reliable correlator-integrated messaging for JMS	163
■ Mapping Apama events and JMS messages	164
■ Dynamic senders and receivers	188
■ Durable topics	189
■ Receiver flow control	189
■ Monitoring correlator-integrated messaging for JMS status	190
■ Logging correlator-integrated messaging for JMS status	191
■ JMS configuration reference	197
■ Designing and implementing applications for correlator-integrated messaging for JMS	207
■ Diagnosing problems when using JMS	220
■ JMS failures modes and how to cope with them	222

Apama support for Java Message Service (JMS) messaging is integrated into the Apama correlator. This provides an efficient method for Apama applications to support JMS messages for communication with external systems.

Using Message Services

Overview of correlator-integrated messaging for JMS

The Java Message Service (JMS) provides a common programming model for asynchronously sending events and data across enterprise messaging systems. JMS supports two models, *publish-and-subscribe* for one-to-many message delivery and *point-to-point* for one-to-one message delivery. Apama's correlator-integrated messaging for JMS supports both these models.

When configured to use correlator-integrated messaging for JMS, Apama applications map incoming JMS messages to Apama events and map outgoing Apama events to JMS messages.

Apama's correlator-integrated messaging for JMS supports the following levels of reliability, built upon the reliability mechanisms provided by JMS:

- `BEST_EFFORT`
- `AT_LEAST_ONCE`
- `EXACTLY_ONCE`
- `APP_CONTROLLED` (can be set for only receivers, not for senders)

When the reliability level is set to `EXACTLY_ONCE` or `AT_LEAST_ONCE` or `APP_CONTROLLED` then delivery is guaranteed because messages are robustly retained by the broker until they are received and acknowledged by the Apama client. The `APP_CONTROLLED` reliability mode lets the application control when messages are acknowledged to the broker.

When the reliability level is set to `BEST_EFFORT`, message delivery is not guaranteed. For applications that do not require guaranteed message delivery, the `BEST_EFFORT` mode provides greater performance.

You can specify configuration for JMS in Apama Studio, either in the correlator-integrated adapter for JMS editor or by editing sections of the XML and `.properties` configuration files directly. Note, however that the mapping configuration should always be edited by using the Apama Studio adapter editor.

Note: For users of Software AG's Universal Messaging (UM): When using UM to send messages between Apama and non-Apama components the recommendation is to use correlator-integrated messaging for JMS to communicate with UM. However, when using UM to send messages only between Apama components (correlators and IAF adapters) the recommendation is to use UM as described in ["Using Universal Messaging in Apama Applications" on page 226](#). Using UM as described in that section requires less configuration because there is no need to provide mapping configuration for each event type.

Using Correlator-Integrated Messaging for JMS

Samples for using correlator-integrated messaging for JMS

Apama Studio provides the following example applications that illustrate the use of correlator-integrated messaging for JMS. The examples are located in the `APAMA_HOME\samples\correlator_jms` directory.

- `simple-send-receive` - This application demonstrates simple sending and receiving. It sends a sample event to a JMS queue or topic as a JMS `TextMessage` using the automatically configured default sender and receives the message using a statically-configured receiver.
- `dynamic-event-api` - This application demonstrates how to use the event API to dynamically add and remove JMS senders and receivers. In addition, it shows how to monitor senders and receivers for errors and availability.
- `flow-control` - This application demonstrates how to use the event API to avoid sending events faster than JMS can handle and a separate demonstration of how to avoid receiving messages from JMS faster than the EPL application can handle.

Overview of correlator-integrated messaging for JMS

Key concepts for correlator-integrated messaging for JMS

The key JMS concepts when implementing an Apama application with correlator-integrated messaging are *connections*, *receivers*, and *senders*.

JMS connections

To use JMS you must configure one or more named connections to the JMS broker. If you need to connect to multiple separate JMS broker instances (which may be using the same JMS provider/vendor or different ones) you need a connection for each; it's also possible to add multiple connections for the same broker (for example, for rare cases where it improves performance

scalability). In Apama Studio you can select from a variety of JMS providers that come with default connection configurations.

JMS receivers

A receiver is a single-threaded context for receiving messages from a single JMS queue or topic (with a single JMS `Session` and `MessageConsumer` object). A connection to a JMS broker can be configured with any number of receivers. Many, but not all, JMS providers support creating multiple receivers for a single queue (or in some cases, topic) either to scale throughput performance, or when using JMS 'message selectors' to partition the messages on a destination.

JMS senders

A sender is a single-threaded context for sending messages (with a single JMS `Session` and `MessageProducer` object). A connection to a JMS broker can be configured with any number of senders. You can add any number of senders, but by default if no senders are explicitly configured, a single sender called "default" will be created implicitly. Each sender can send messages to any JMS destination (a queue or topic); the destination is specified on a per-message basis in the mapping rule set (either hardcoded by specifying a constant value per message type in the mapping rules or mapped from a destination field in the apama event). Messages sent by a single sender with the same JMS headers ("priority" for example) will usually be delivered in order by the provider (although this may not be the case if there is a failure), but the ordering of sends across senders is undefined. Multiple senders can be created for a single connection to scale throughput performance, or for sending messages with different `senderReliability` modes. Each sender is represented by its own correlator output channel.

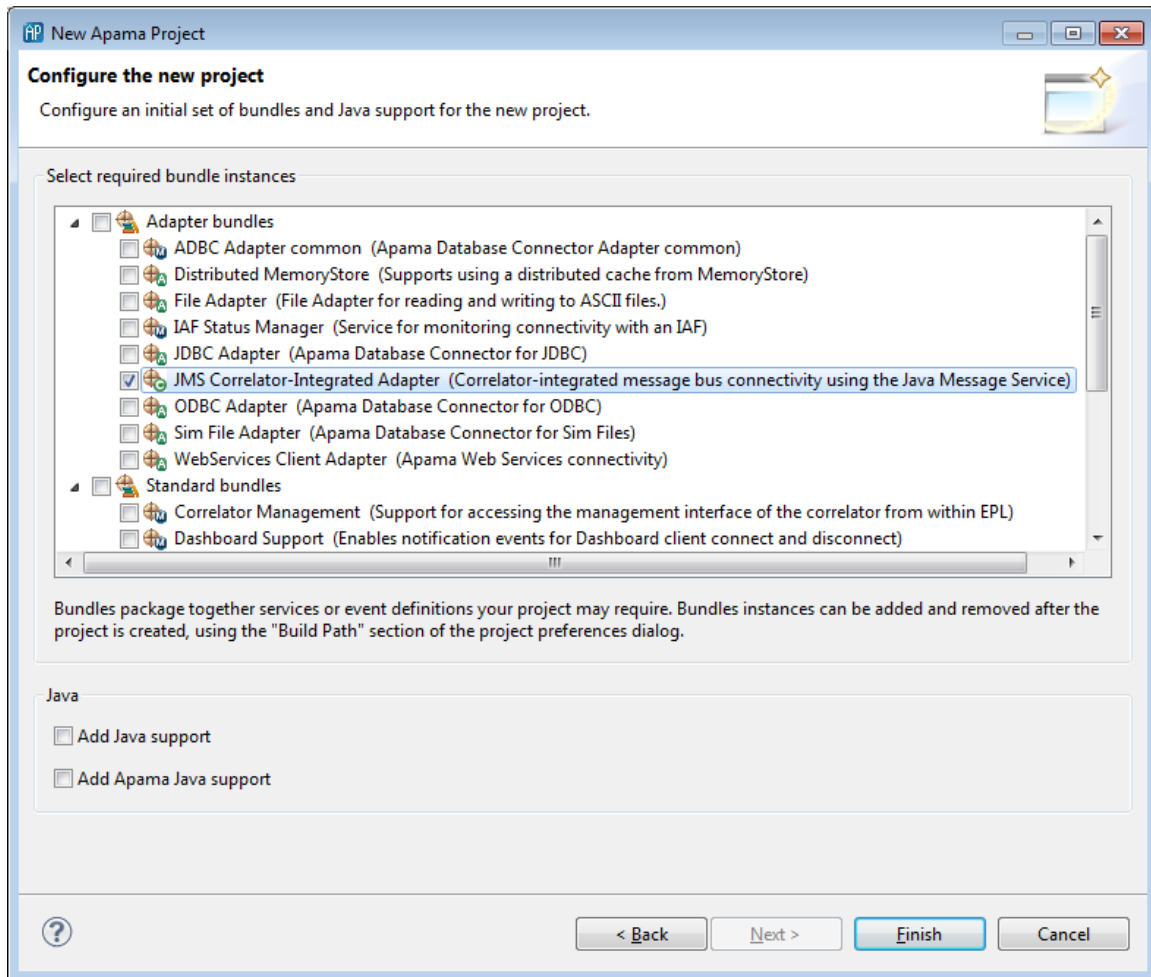
[Overview of correlator-integrated messaging for JMS](#)

Getting started with simple correlator-integrated messaging for JMS

This section describes the steps for creating an Apama application that uses correlator-integrated messaging for JMS where guaranteed delivery is not required. Apama Studio provides an example application that illustrates a simple use of correlator-integrated messaging for JMS in the `APAMA_HOME\samples\correlator_jms\simple-send-receive` directory.

To make correlator-integrated messaging for JMS available to an Apama project:

1. Select **File > New > Apama Project** from the Apama Studio menu. This launches the **New Apama Project** wizard.
2. In the **New Apama Project** wizard, give the project a name, and click **Next**. The second page of the wizard appears, listing the available Apama resource bundles.



3. Apama's correlator-integrated messaging for JMS makes use of the Apama correlator-integrated adapter for JMS. In the New Apama Project wizard, from the Select required bundle instances field, select the JMS Correlator-Integrated Adapter bundle.
4. Click Finish.

Apama Studio adds the correlator-integrated adapter for JMS to the project's `Adapters` node. In addition, Apama Studio generates all the necessary resources to support correlator-integrated messaging for JMS. Note, you can only add a single instance of the correlator-integrated messaging adapter for JMS to an Apama project.

After you add the correlator-integrated adapter for JMS, you need to configure connections to a JMS broker and configure senders and receivers.


Using Correlator-Integrated Messaging for JMS

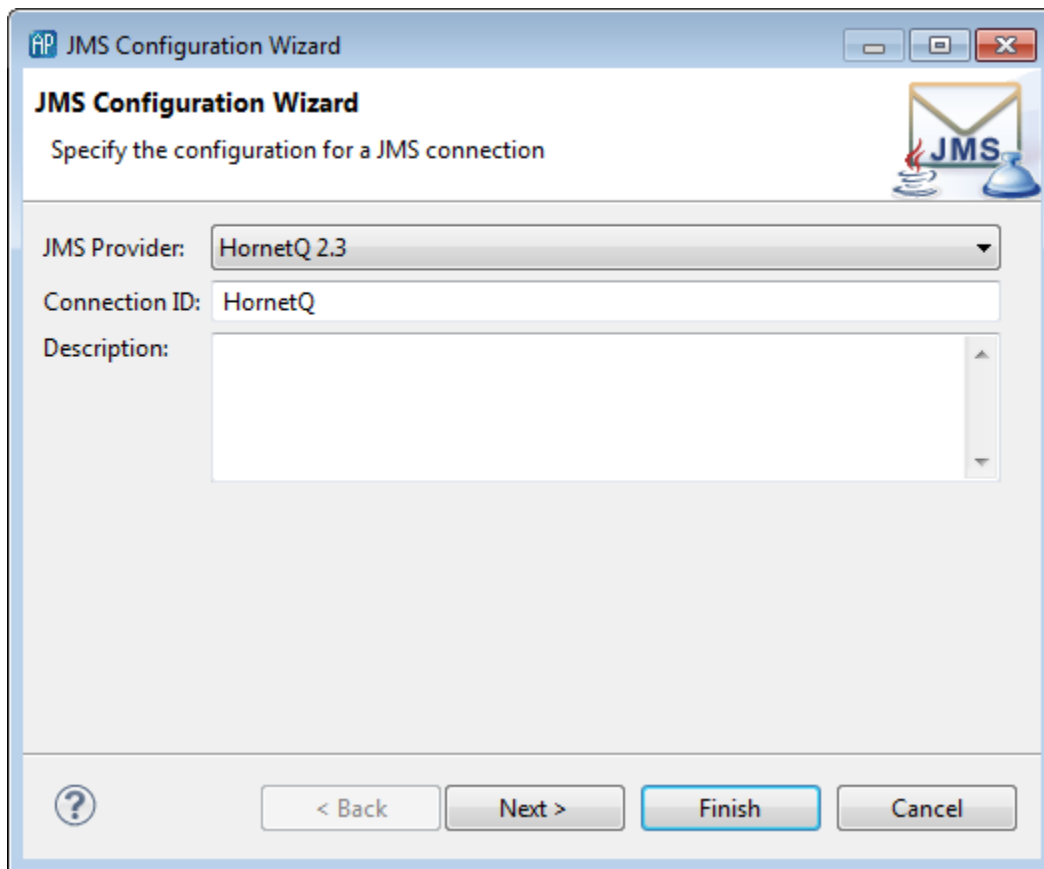
Adding and configuring connections

When you first add the correlator-integrated messaging for JMS bundle to an Apama Studio project, the list of connections is initially empty. You can add one or more connection to JMS providers.

To establish a connection to a JMS broker:

1. In the Project Explorer, expand the project's `Adapters` node and then expand the `JMS Correlator-Integrated Adapter` node.

2. Double-click the adapter instance. This opens the instance's configuration in the editor for the correlator-integrated adapter for JMS.
3. In the adapter editor's Settings tab, click the Add Connection button () to display the JMS Configuration wizard.



4. On the first page of the JMS Configuration wizard specify the following.
 - a. JMS Provider, select from the drop-down list.
 - b. Connection ID, this needs to be unique, and will be used throughout the configuration files and Apama application to identify this broker connection. The value for the connection ID should not contain any spaces. The connection ID is used when sending JMS messages from the Apama application. This Apama connection identifier is not exposed to the JMS provider in any way.
 - c. Description, this is optional and currently unused.
5. Click Next.

This displays the second page of the JMS Configuration wizard.
6. The second page of the JMS Configuration wizard displays the default `CLASSPATH` details for the JMS Provider you selected in the previous step. If necessary, add or modify the values as appropriate for your environment.
7. Click Next.

This displays the third page of the JMS Configuration wizard.

Connection properties
Specify the connection properties to connect to the broker

Connection User Name :

Connection Password :

☒ Use JNDI
☐ Show advanced properties

ConnectionFactory JNDI Lookup Name:

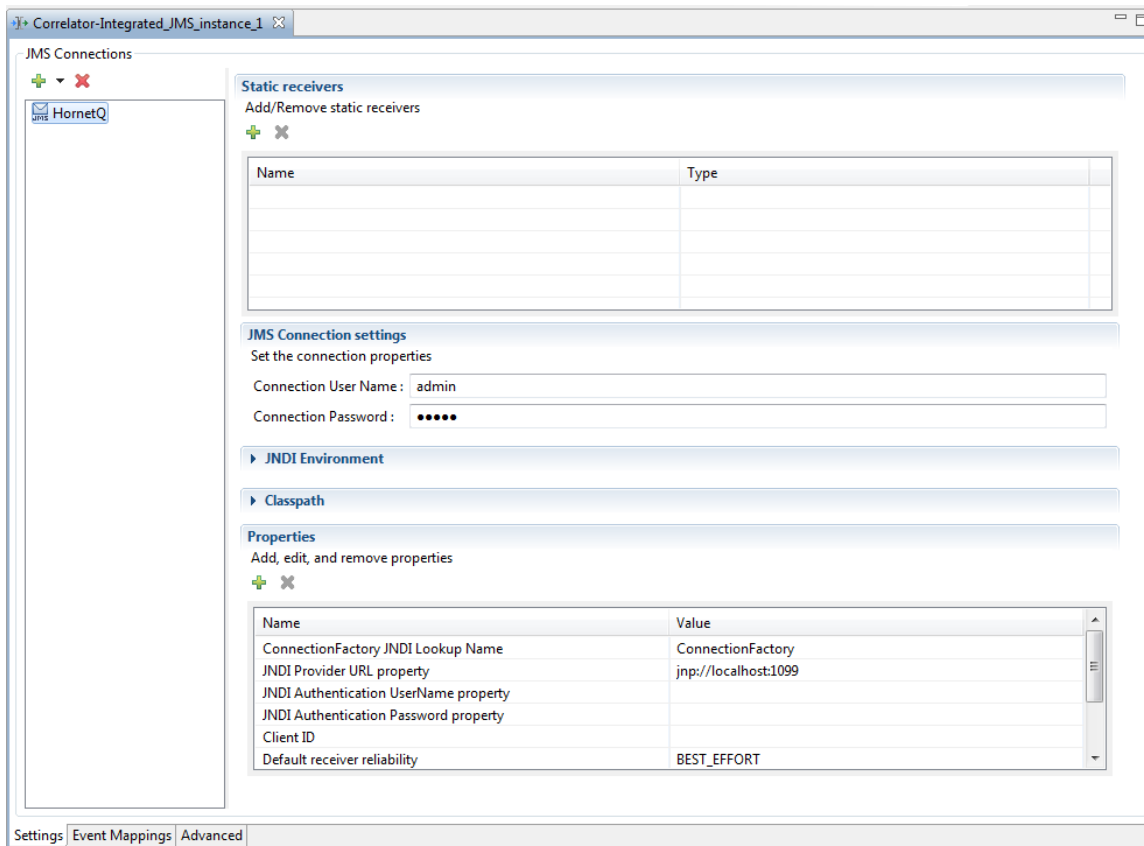
+ ×

Property	Value
JNDI Environment	
Initial context factory	org.jnp.interfaces.NamingContextFactory
Provider URL	jnp://localhost:1099
Security Principal	
Security Credentials	

? < Back Next > Finish Cancel

8. The third page of the JMS Configuration wizard displays the default connection properties for the JMS provider you selected. If necessary, add or modify the values as appropriate.
 - a. By default, the JMS Connection uses JNDI, which is the recommended option. Unselect **Use JNDI** if your application will not use JNDI and your JMS provider supports non-JNDI instantiation of the `ConnectionFactory` class (note that many providers do not).
 - b. By default, the JMS Configuration wizard lists a subset of standard connection properties. If **Use JNDI** is enabled, the Connection details field shows `JNDI Environment` properties. If **Use JNDI** is not enabled, the connection details field shows `ConnectionFactory` properties. To show the complete list of properties add a check to the **Show advanced properties** check box.
 - c. You can add and remove properties and you can modify the properties' values. To modify a value, click in the **Value** column and enter the required information.
9. Click **Finish**.

Apama Studio updates the adapter editor to display the new connection in the JMS Connections section.




After you establish a connection to a JMS broker, you need to add JMS receivers and specify mapping configurations for receivers and senders.

Getting started with simple correlator-integrated messaging for JMS

Adding JMS receivers

JMS receivers are added to JMS Connections. To add a JMS receiver to a project:

1. In the Project Explorer, double-click the project's correlator-integrated adapter for JMS instance. This opens the instance configuration in the Apama Studio adapter editor.
2. Select the desired JMS Connection.

3. In the Static Receivers section click the Add destination button ()

This adds a receiver with a default name to the Name column and a default type (queue) to the Type column.

4. If desired, you can edit the value in the Name column. You can edit the value in the Type column by clicking the value and selecting a new type from the drop-down list at the right.

After you have configured the JMS receivers for each queue or topic of interest, you need to configure how the received JMS messages will be mapped to Apama events.

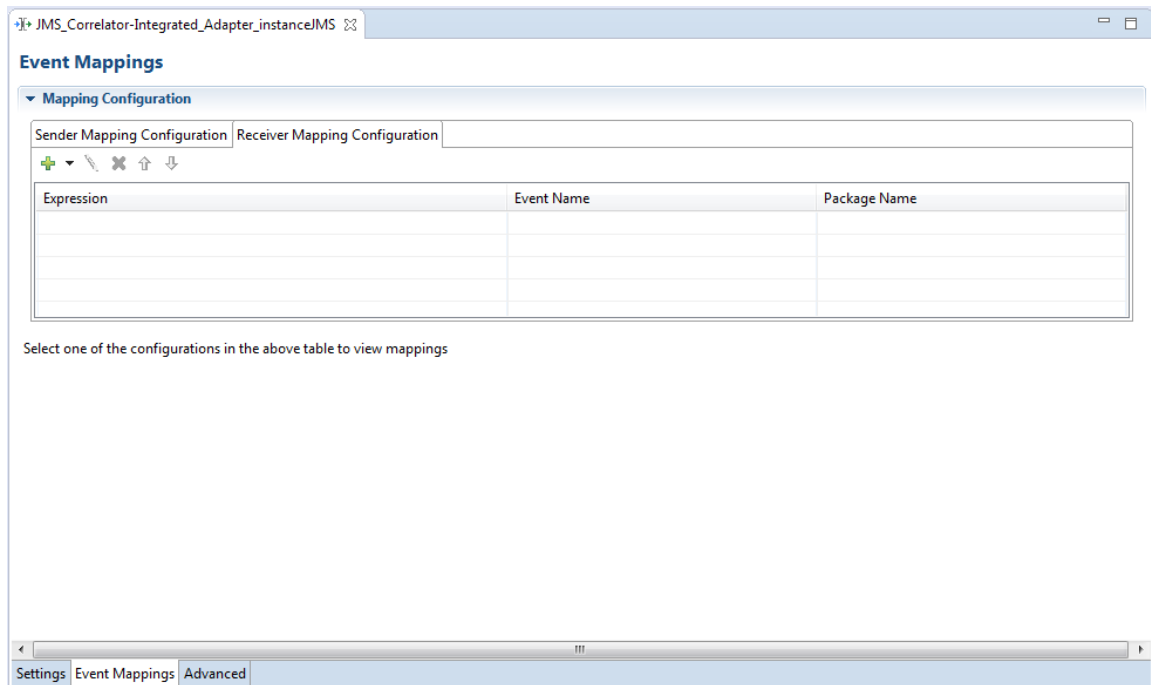
Getting started with simple correlator-integrated messaging for JMS


Configuring receiver event mappings

Each event mapping for a received JMS message is configured by specifying the target Apama event type, a conditional expression to determine which source JMS messages should be mapped to this event type, and a set of mapping rules that populate the fields of the target Apama event based on the contents of the source JMS message.

To configure an event mapping:

1. Ensure that the Apama event types you wish to use for mapping have been defined in an EPL file in your project.
2. In the Project Explorer, double-click the project's correlator-integrated adapter for JMS instance. This opens the instance configuration in the Apama Studio adapter editor.
3. In the adapter editor, select the Event Mappings tab.



4. On the adapter editor's Event Mappings tab in the Mapping Configuration section, select the Receiver Mapping Configuration tab.
5. Click the down triangle next to the Add Event button () and select Add Event to display the Event Type Selection dialog.
6. In the Event Type Selection dialog's Event Type Selection field, enter the name of the event. As you type, event types that match what you enter are shown in the Matching Items list.
7. In the Matching Items list, select the name of the event type you want to associate with the JMS message. The name of the EPL file that defines the selected event is displayed in the status area at the bottom of the dialog.
8. Click OK.

This updates the display in the adapter editor's Event Mappings tab to show a hierarchical view of the JMS message on the left (the mapping source) and a hierarchical view of the Apama event on the right (the mapping target). In addition, the Expression column displays a default JUEL

conditional expression that determines which JMS messages will use the specified mapping rules. If you need to use a different conditional expression, you can edit the default. For more information see ["Using conditional expressions" on page 158](#).

9. Map the JMS message to the Apama event by clicking on the entity in the **Message** tree and dragging a line to the entity in the **Event** tree. For example, the simplest mapping for a standard JMS `TextMessage` would be a single mapping rule from `JMS Body` in the JMS message to a single `string` field in the Apama event. More complex mapping involves mapping the value of one or more JMS headers or properties or parsing XML content out of the text message. For more information see ["Mapping Apama events and JMS messages" on page 164](#).

If a receiver mapping configuration lists multiple events, the mapper evaluates the expressions from top to bottom, stopping on the first mapping whose conditional expression evaluates to true. You can use the up and down arrows to change the order in which the evaluations are performed.

Adding JMS receivers

Using conditional expressions

When you configure event mappings for received JMS messages, you specify Apama event types to which JMS messages will be mapped along with the mapping rules. The correlator-integrated mapper for JMS uses JUEL expressions to indicate which mapping rules to use. JUEL (Java Unified Expression Language) expressions are a standard way to access data. When you specify an event type for a receiver, Apama Studio creates a default conditional expression that evaluates a JMS property named `MESSAGE_TYPE`, testing to see if its value is the name of the specified Apama event type. You can modify the default expression if you need to test for a different condition, depending on the format of the JMS messages that Apama will be receiving.

Depending on your application's needs, you can create a conditional expression for the following cases:

- Match a JMS Header
- Match a JMS Property
- If the XML document root element exists
- Match an XPath into the JMS message body

To specify a custom conditional expression:

1. On the **Receiver Mapping Configuration** tab, click the expression in the **Expression** column.
2. Click the **Browse** button next to the expression. This displays the **Conditional Expression** dialog, where you can edit the default expression.
3. In the **Condition** field, select the type of conditional expression you want from the drop-down list. Depending on your selection, the remaining available fields will vary.
4. Fill in the remaining fields as required. For some fields you select from drop-down lists, for others you enter values directly. If you select the **Custom** type of conditional expression, you can edit the expression directly. If a string literal in the expression contains a single or double quotation mark, it needs to be escaped with the backslash character (`\'` or `\"`).
5. Click **OK**. The new expression is displayed in the **Expression** column of the **Receiver Mapping Configuration** tab.

Conditional operators in custom expressions

The following operators are available:

- `==` equal to
- `!=` not equal
- `lt` less than
- `gt` greater than
- `le` less than or equal
- `ge` greater than or equal
- `and`
- `or`
- `empty` null or empty
- `not`

A number of methods are available for common string operations such as the ones listed below.

- `contains()`
- `endsWith()`
- `equals()`
- `equalsIgnoreCase()`
- `matches()`
- `startsWith()`

For a complete list of the available methods as well as details for using these methods, see ["JUEL mapping expressions reference for JMS" on page 181](#).

Custom conditional expression examples

In most cases the decision about which Apama event type to map to for a given JMS message is based on a JMS message property value or sometimes a header, such as `JMSType`. In other cases, when there is no alternative, the decision is made by parsing XML content in the document body and evaluating an XPath expression over it. Here are some examples of typical conditional expressions.

- JUEL boolean expression based on a JMS string property value:

```
${jms.property['MY_MESSAGE_TYPE'] == 'MyMessage1'}
```
- JUEL boolean expression based on a JMS header value:

```
${jms.header['JMSType'] == 'MyMessage1'}
```
- JUEL boolean expression based on the existence of the XML root element 'message1' in the body of a `TextMessage`:

```
${xpath(jms.body.textmessage, 'boolean(/message1)')}
```
- JUEL boolean expression based on testing the value of an XML attribute in the body of a `TextMessage`:

```
${xpath(jms.body.textmessage, '/message/info/@messageType') == 'MyMessage'}
```
- JUEL boolean expression for matching based on message type (`TypeMessage`, `MapMessage`, `BytesMessage`, `ObjectMessage`, or `Message`):

```
${jms.body.type == 'TextMessage'}
```

The following boolean JUEL expressions show advanced cases demonstrating what is possible using JUEL and illustrating how the syntax works with example XML documents

- JUEL expression that matches all messages:

```
${true}
```

- 'greater than' numeric operator:

```
${jms.property['MY_LONG_PROPERTY'] gt 120}
```

- Using backslash to escape quotes inside a JUEL expression:

```
${jms.body.textmessage == 'Contains \'quoted\' string'}
```

- Operators 'not', 'and', 'or', and 'empty':

```
${not (jms.property['MY_MESSAGE_TYPE'] == 'MyMessage1' or  
      jms.property['MY_MESSAGE_TYPE'] == 'MyMessage2') and  
      not empty jms.property['MY_MESSAGE_TYPE']}
```

- Testing the value of an entry in the body of a MapMessage:

```
${jms.body.mapmessage['myMessageTypeKey'] == 'MapMessage1'}
```

- An advanced XPath query (and use of JUEL double-quoted string literal and XPath single-quoted string literal in the same expression)

```
${xpath(jms.body.textmessage, " (count(/message3/e) > 2) and  
    /message3/e[2] = 'there' and  
    (/message3/e[1] = /message3/e[3]) ")}
```

For an XML document such as

```
<message3><e>Hello</e><e>there</e><e>Hello</e></message3>
```

- XPath namespace support:

```
${xpath(jms.body.textmessage, " /message4/*[local-name()='element1' and  
    namespace-uri()='http://www.myco.com/testns']/text() ") ==  
    'Hello world'}
```

For an XML document such as

```
<message4 xmlns:myprefix="http://www.myco.com/testns">  
    <element1>No namespace</element1>  
    <myprefix:element1>Hello world</myprefix:element1></message4>
```

- Recursively parsing XML content nested in the CDATA section of another XML document:

```
${xpath(xpath(jms.body.textmessage, '/messageA/text()'),  
    '/messageB/text()') == 'MyNestedMessageType'}
```

For an XML document such as

```
<messageA><![CDATA[  
    <messageB>MyNestedMessageType</messageB> ]]>  
</messageA>
```

- Check if a JMS string property value contains the specified value:

```
${jms.property['MY_MESSAGE_TYPE'].contains('Apama')}
```

- Check if a JMS TextMessage body matches the specified regular expression:

```
${jms.body.textmessage.matches('.*inb*[ou]*r')}
```


For a table of expressions for getting and setting values in JMS messages and recommended mappings to Apama event types, see ["JUEL mapping expressions reference for JMS" on page 181](#).

Adding JMS receivers

Configuring sender event mappings

Each event mapping for a JMS message to be sent is configured by specifying the source Apama event type, and a set of mapping rules that populate the target JMS message from the fields of the source Apama event.

To configure an event mapping:

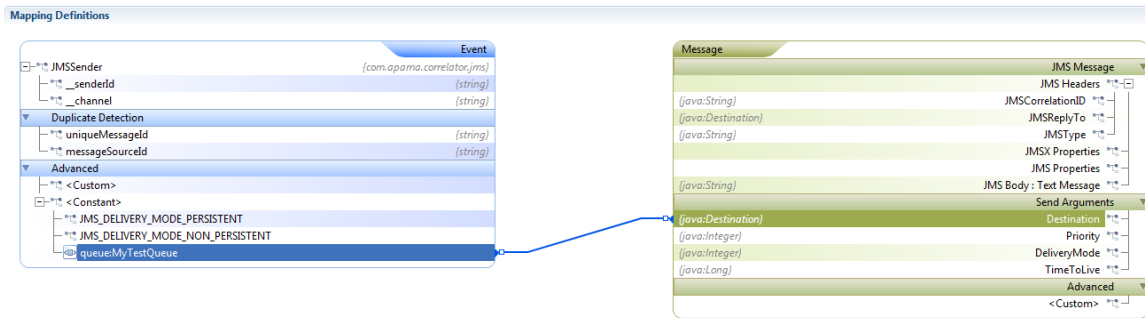
1. Ensure that the Apama event types you wish to use for mapping have been defined in an EPL file in your project.
2. If necessary, in the Project Explorer, double-click the project's correlator-integrated adapter for JMS instance. This opens the instance configuration in the Apama Studio adapter editor.
3. Select the JMS connection.
4. In the correlator-integrated adapter for JMS editor, select the Event Mappings tab.
5. On the adapter editor's Event Mappings tab, select the Sender Mapping Configuration tab.
6. On the Sender Mapping Configuration tab click the down triangle next to the Add Event button () and select Add Event to display the Event Type Selection dialog.
7. In the Event Type Selection dialog's Event Type Selection field, enter the name of the event. As you type, event types that match what you enter are shown in the Matching Items list.
8. In the Matching Items list, select the name of the event type you want to associate with the JMS message. The name of the EPL file that defines the selected event is displayed in the status area at the bottom of the dialog.
9. Click OK.

This updates the display in the adapter editor's Event Mappings tab to show a hierarchical view of the Apama event on the left (the mapping source) and a hierarchical view of the JMS message on the right (the mapping target).

10. Create a mapping rule as follows:
 - a. If necessary, click on the event to be mapped in the Event Name column.
 - b. Click on the entity in the event tree and drag a line to the entity in the message tree.

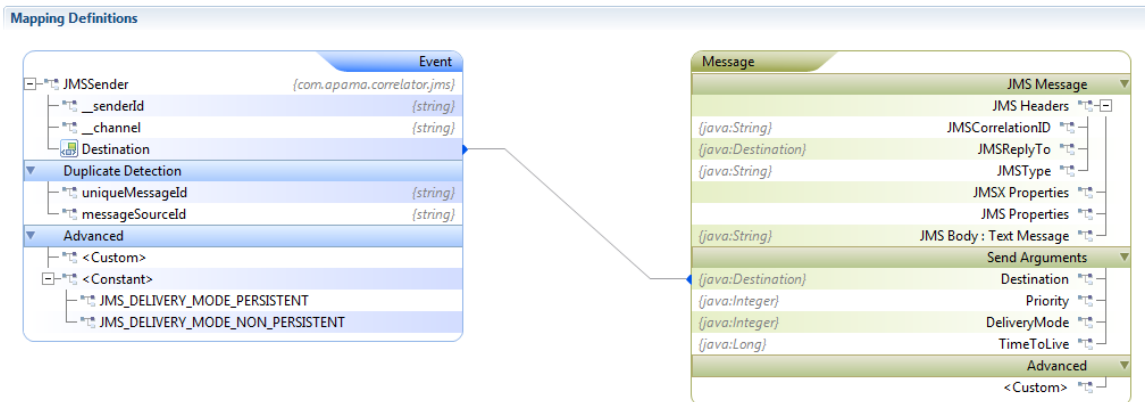
For example, a simple mapping would be from a single `string` field in an Apama event to `JMS Body` in the JMS message. More complex mappings might involve mapping an event field to a specific JMS property. For more information see ["Mapping Apama events and JMS messages" on page 164](#).

11. Specify the message's JMS destination in either of two ways:
 - Specify a constant value in the event type's mapping:



For more information on specifying a constant value, see ["Using expressions in mapping rules"](#) on page 165.

- Specify a destination in an event field and map that field to the message:



Note *destination* is always specified as `topic:name`, `queue:name`, or `jndi:name`

Getting started with simple correlator-integrated messaging for JMS

Using EPL to send and receive JMS messages

The EPL code necessary for using correlator-integrated messaging for JMS is minimal.

- Initialization - Your application needs to notify the correlator that the application has been injected and is ready to process events from the JMS broker.
 1. Apama recommends that after *all* an application's EPL has been injected, the application should send an application-defined 'start' event using a .evt file. Using an event is clearer and more reliable than enabling JMS message receiving using monitor `onload()` actions because it is easier to guarantee that all EPL has definitely been injected and is in a good state before the event is sent and JMS message receiving commences.
 2. Any monitors that need to use the correlator-integrated messaging for JMS event API will have a variable (typically a monitor-global field) holding a JMS event object.
 3. The monitor that handles the application-defined start event (from step 1), should use this JMS event object to notify correlator-integrated messaging for JMS that the application is initialized and ready to receive messages, for example:

```
on com.mycompany.myapp.Start() {
    jms.onApplicationInitialized();
    // Any other post-injection startup logic goes here too.
}
```

- Receiving events - After configuring a JMS receiver, add EPL listeners for the events specified in the mapping configuration.
- Sending events - Send the Apama event associated with the JMS message in the Sender Mapping Configuration by using the following syntax.

```
send event_name to "jms:senderId";
```

Note, *senderId* is typically "*connectionId-default-sender*" unless explicitly configured with a different name. For example to send an event to the default sender on a connection called "MyConnection", use the following:

```
send MyEvent to "jms:MyConnection-default-sender";
```

For more information on specifying the message's JMS destination, see ["Configuring sender event mappings" on page 161](#).

[Getting started with simple correlator-integrated messaging for JMS](#)

Getting started with reliable correlator-integrated messaging for JMS

This section describes the steps for creating an Apama application that uses reliable correlator-integrated messaging for JMS in an environment where guaranteed delivery is required. In order to enable reliable JMS messaging, you set specific JMS connection properties. In addition, reliable JMS messaging makes use of Apama's correlator persistence feature, which specifies that the correlator periodically writes its state to stable storage.

The focus here is on the most widely used reliability modes, which transparently tie JMS message sending and receiving to the correlator's persistence feature. When correlator persistence is enabled the correlator periodically writes its state to stable storage. For more complex applications, there are features to prevent message loss even when not using persistent monitors. See ["Sending and receiving reliably without correlator persistence" on page 209](#).

The steps described in this section build on the example created in ["Getting started - using simple correlator-integrated messaging for JMS" on page 152](#).

To enable reliable correlator-integrated messaging for JMS for an Apama project:

1. If necessary, create an Apama project that uses correlator-integrated messaging for JMS as described in ["Getting started with simple correlator-integrated messaging for JMS" on page 152](#).
2. If necessary, in the Project Explorer expand the project's `Adapters` node, expand the correlator-integrated messaging for JMS adapter node, and double-click the adapter instance. This opens the instance's configuration in the adapter editor.
3. In the adapter editor, display the `Settings` tab and in the `JMS Connection` section, select the JMS connection to use.
4. Click the `Properties` section to expand it.
5. In the `Properties` section, select `EXACTLY_ONCE` or `AT_LEAST_ONCE` for `Default receiver reliability`. Select `EXACTLY_ONCE` or `AT_LEAST_ONCE` for `Default sender reliability`. Each of these reliability modes prevents message loss. `AT_LEAST_ONCE` is simpler and offers greater performance. `EXACTLY_ONCE` adds detection and elimination of duplicate messages (if configured correctly), which may be required for some applications.

6. If receiving with `EXACTLY_ONCE` reliability, it is necessary to configure additional mapping rules to specify an application-level unique identifier for each received message that will function as the key for detecting functionally duplicate messages. To add these mapping rules, display the Event Mappings tab and in the source event tree, map the `uniqueMessageId` and (optionally, but recommended) `messageSourceId` entities to appropriate values in the JMS message. For example, they could be mapped to JMS message properties called `UNIQUE_MESSAGE_ID` and `MESSAGE_SOURCE_ID` (or to nodes within an XML document in the message body). When sending JMS messages, the mapping rules provide a way to expose the `uniqueMessageId` and `messageSourceId` that Apama automatically generates for sending messages to whatever JMS client will be receiving them, so that it can perform duplicate detection.
7. In your application's EPL code, add the `persistent` keyword before the monitor declarations for monitors listening for Apama events associated with JMS messages.
8. In the project's Run Configuration, enable correlator persistence as follows.
 - a. In the Run Configuration dialog, select the Components tab.
 - b. Select Default correlator and click Edit. The Correlator Configuration dialog appears.
 - c. In the Correlator Configuration dialog, select the Persistence Options tab, select Enable correlator persistence, and click OK.

Running a correlator in this way causes the it to periodically write its state to stable storage.

For more information on correlator persistence, see "Using Correlator Persistence" in *Developing Apama Applications*.

Using Correlator-Integrated Messaging for JMS

Mapping Apama events and JMS messages

After you specify which Apama events you want to associate with JMS messages, you need to create mapping rules that associate Apama event fields with parts of the JMS messages. The Apama Studio adapter editor provides a visual mapping tool to create the mapping rules. There are several approaches for how to map Apama events to the JMS messages.

- ["Simple mapping for JMS messages" on page 165](#) — Use this approach when a simple Apama event field can be associated with a corresponding value in the JMS message.
- ["Using expressions in mapping rules" on page 165](#) — Use this when sending or receiving JMS messages and you need to write a customized JUEL expression for a mapping rule.
- ["Template-based XML generation" on page 166](#) — Use this when sending JMS messages that contain XML. You assign a template that will be used to generate an XML document. The template contains placeholders for each of the source event fields whose values will replace the placeholders.
- ["Specifying an XPath transformation for JMS messages" on page 166](#) — Use this when receiving JMS messages containing XML to specify values from the XML document that are to be used to populate the fields in the target Apama event.
- ["Specifying an XSLT transformation for JMS messages" on page 167](#) — Use this when receiving JMS messages containing XML, to change or simplify the structure of the XML document.

- ["Convention-based XML mapping" on page 167](#) — Use this to parse or generate XML documents by using event definitions that follow specific conventions to implicitly encode the structure of the XML document. This approach allows mapping of sequences to elements of the same type. It avoids the need for XPath, but does impose some limitations on the XML naming and structure.
- ["Using EDA events in Apama applications" on page 171](#) — Use correlator-integrated messaging for JMS to consume and publish EDA events.
- ["Implementing a custom Java mapper" on page 186](#) — If the mapping tools provided with Apama do not meet your needs then you can implement your own Java mapper class and use that instead.

Using Correlator-Integrated Messaging for JMS

Simple mapping for JMS messages

When creating a simple 1:1 mapping rule for an Apama event field to part of a JMS message that contains a similar type, you can drag a line between the elements as follows:

1. In the editor for the correlator-integrated adapter for JMS, display the Event Mapping tab.
2. For each mapping rule, click on the entity you want to map and drag a line to the entity you want to map it to.

Apama Studio represents each rule with a blue line between entities. If the types of the source and target do not match, type coercion will be performed automatically at runtime.

Mapping Apama events and JMS messages

Using expressions in mapping rules

In many cases, a mapping rule requires customization. For example, if you map an event field to a `JMS Property` field then you need to specify which JMS property to use. In other cases you may want to use a constant value in a mapping rule or to create a JUEL expression, for example to execute an XPath query on nested XML documents.

To add an expression to a mapping rule:

1. Drag a mapping line from the entry in the source tree to the target. If one side of the mapping rule requires a more specific expression, the Connection Participants dialog is displayed.
2. In the Connection Participants dialog's Type field, select an entry from the drop-down list.
3. In the next field enter the `JMS Body` type, the `JMS Property` name, a constant value, or a custom JUEL expression. As you enter this information, the expression that will be used in the mapping rule is displayed in the Expression Value field.
4. Click OK.

For a table of expressions for getting and setting values in JMS messages and recommended mappings to Apama event types, see ["JUEL mapping expressions reference for JMS" on page 181](#).

Mapping Apama events and JMS messages

Template-based XML generation

With the template-based approach to mapping, you can map fields in an Apama event to elements and attributes in complex XML structures. The template consists of a sample XML document with placeholders that will be replaced with values from the Apama event fields. When you assign a template, these variables are displayed in the JMS message tree. You then map event fields to the variables.

To assign a template for mapping:

1. In the adapter editor's Event Mappings tab, right-click the `JMS Body` entry and select **Assign Template**. The Assign Template dialog appears.
2. In the XML Template file field, enter the name of the template file you want to use or click the browse or down arrow button to locate the file.

When you specify a template file, the contents of the file are added to the text field in the dialog.

It is usually best to create the template file from a sample XML document before opening this dialog, but it is also possible to perform this task from the dialog itself, for small XML documents. To create the XML template, you define placeholders to represent field values that you want the adapter to obtain from the input event. To define a placeholder, insert a dollar sign (\$) following by the placeholder name. After you click OK, the placeholder appears as a new child of the target's JMS body node.

3. In the source event, click the Apama event field and drag a line to the desired element or attribute in the target JMS message.

Mapping Apama events and JMS messages

Specifying an XPath transformation for JMS messages

If a mapping rule from an Apama event to a JMS message requires an XPath transformation, specify the transformation details as follows:

1. If necessary, in the adapter editor's Event Mappings tab, in the Mapping Definitions section, draw the line indicating the mapping from source to target.
2. In the Mapping Definitions section, click on the line that specifies the mapping rule you are interested in.
3. In the Mapping Definitions section, in the Transformation Type field select XPath.
4. In the XPath Expression field, specify a valid XPath expression. You can either enter the XPath expression directly or you can use the XPath builder tool to construct an expression.

To use the XPath Builder:

- a. Click the Browse button [...] to the right of the XPath Expression field. The Select input for XPath helper dialog is displayed.
- b. In the Select input for XPath helper dialog, click Browse [...] and select the name of the file that contains a definition of the XML structure (the drop-down arrow allows you to select the scope of the selection process). Click OK. The XPath Helper opens, showing the XML structure of the selected file in the left pane.
- c. In the XPath Helper, build the desired XPath expression by double-clicking on nodes of interest in the left pane. The resultant XPath expression appears in the XPath tab in the upper right pane. If the XML

document makes use of namespaces, change the namespace option from `Prefix` to `Namespace` or `Local name`.

- d. In the XPath Helper, click OK. The XPath Builder closes and the XPath Expression field displays the XPath expression you built.

Mapping Apama events and JMS messages

Specifying an XSLT transformation for JMS messages

If a mapping rule from an Apama event to a JMS message requires an XSLT transformation, specify the transformation details as follows:

1. If necessary, in the adapter editor's Event Mappings tab, in the Mapping Element Details section, draw the line indicating the mapping from source to target.
2. In the Mapping Element Details section, click on the line that specifies the mapping rule you are interested in.
3. In the Transformation Type field select XSLT Transformation from the drop-down list. This displays the Stylesheet URL field.
4. In the Stylesheet URL field, click Browse [...] to locate the file of the stylesheet to use.

Mapping Apama events and JMS messages

Convention-based XML mapping

Convention-based mapping allows XML documents to be created or parsed based on a document structure encoded in the definition of the source or target Apama event type.

The first stage when using convention-based mapping is to examine the structure of the XML document, and create an event definition to represent its root element, with fields for each attribute, text node, sub-element or sequence (of attributes, text nodes or sub-elements). The actual names of the event types are not important, but the event field names and types must follow the following conventions:

- XML attributes can be represented by any EPL simple type such as `string`, `integer`, etc. The name used should be preceded by an underscore, for example `boolean _flag;`.
- XML text nodes are represented by either:
 - A field inside an Apama event representing the parent of the element containing the text, named after the element that encloses the text such as `string myelement;`. This avoids the need to create an event type to represent the element in cases where the element only contains a text node, and no attributes or children. The field type may be any primitive EPL type (for example `string`, `integer`).
 - A field inside an Apama event representing the element that directly contains the text, named `xmlTextNode`. This is necessary in cases where an Apama event type is needed to represent the element so that attributes and/or child elements can also be mapped. The field type may be any primitive EPL type (for example `string`, `integer`).
- XML elements containing attributes or sub-elements of interest are represented by a field of an event type which follows these same conventions. The event type can have any name, but the field must be named after the element, for example, `MyElementEventType myelement`.

- XML attributes, text nodes or elements which may occur more than once in the document are represented by a sequence field of the appropriate primitive or event type, named after the element, for example, `sequence<string> myelement` OR `sequence<MyElement> myelement`.

Some special cases to be aware of when naming fields to match element/attribute names are:

- XML nodes which are inside an XML namespace are always referenced by their local name only (the namespace or namespace prefix is ignored).
- XML node names that are Apama EPL keywords (such as `<return>`) must be escaped in the event definition using a hash character, for example, `string #return;`. When generating an XML document, each field in the event will be processed in order and used to build up the output document. When parsing an XML document, each field in the event will be populated with whatever XML content matches the field name and type (based on the conventions above); any XML content that is not referenced in the event definition will be silently ignored.
- XML node names containing any character that is not a valid EPL identifier character (anything other than `a-z`, `A-Z`, `0-9` and `_`) must be represented using a \$hexcode escape sequence. Of the characters that are not valid EPL identifier characters, only the hyphen and dot are supported. Note that the hexcode based escape sequences are case sensitive. For representing the hyphen or dot use the following:
 - Hyphen '-' is represented as `$002d`.
 - Dot '.' is represented as `$002e`.

Limitations of convention-based XML mapping

In this release it is not possible to generate documents that contain elements in different XML namespaces (although when parsing this is not a problem).

The following limitations apply to the Apama event definitions that can be used to generate XML:

- Dictionary event field types are not supported
- If an event field is of type `sequence`, the sequence can contain simple types or events. The sequence cannot contain sequences of sequences or sequences of dictionaries

Mapping Apama events and JMS messages

Convention-based JMS message mapping example

The following example shows how to parse a JMS message whose body contains an XML document and map it to an Apama event called `MyEvent`.

Consider a JMS message whose body contains the following XML document:

```
<?xml version='1.0' encoding='UTF-8'?>
<myroot xmlns:p='http://www.myco.com/dummy-namespace'>
  <myelement1>An element value</myelement1>
  <myelement2 myattribute='123' myboolattribute='true'>456</myelement2>
  <ignoredElement>XML content that is not included in the event definition
    is ignored</ignoredElement>
  <el>Hello</el>
  <el>there</el>
  <e-2 e2att='value1'><subElement>e2-sub-value1</subElement></e-2>
  <e-2 e2att='value2'><subElement>e2-sub-value2</subElement></e-2>
  <el>world</el>
  <namespacedElement xmlns='urn:xmlns:foobar'>My namespaced
    text</namespacedElement>
  <p:namespacedElement>My namespaced text 2</p:namespacedElement>
  <namespacedElement>My non-namespaced text 3</namespacedElement>
</myroot>
```



```
<return>Element whose name is an EPL keyword</return>
</myroot>
```

Define the Apama event `MyEvent` as follows:

```
event MyElement2
{
    string _myattribute;
    boolean _myboolattribute;
    string xmlTextNode;
}
event E2
{
    string _e2att;
    string subElement;
}
event MyRoot
{
    string myelement1;
    MyElement2 myelement2;
    sequence<string> e1;
    sequence<string> namespacedElement;
    string #return;
    sequence<E2> e$002d2;
}
event MyEvent
{
    string destination;
    MyRoot myroot;
}
```

Note that the field names and types matter but the event type names do not.

The document above would be parsed to the following Apama event string:

```
MyEvent("queue:MyQueue",
  MyRoot("An element value",
    MyElement2("123",true,"456"),
    ["Hello","there","world"],
    ["My namespaced text","My namespaced text 2","My non-namespaced text 3"],
    "Element whose name is an EPL keyword",
    [E2("value1","e2-sub-value1"),E2("value2","e2-sub-value2")]
  ))
```

The exact same event definitions could be used in the other direction for creating an XML document, although the node order will be slightly different from that of the document shown above (based on the field order) and everything would be in the same XML namespace.

Convention-based XML mapping

Using convention-based XML mapping when receiving/parsing messages


To map a received JMS message to an Apama event using the convention-based approach:

1. Create an Apama event type with fields that correspond in type and order to the structure of the XML document. Ensure that the target event type you are mapping to has a field of this type and that the field's name is the same as the name of the root element in the expected XML document.
2. Drag a mapping line from the JMS message node containing the XML document (for example, the `JMS Body`) to the target Apama event field that has the same name as the root element in the XML document. Assuming the JMS message contains an XML document (a string beginning with an open angle bracket character "<"), the document will be parsed and the results will be used to populate the fields of the target event.

Convention-based XML mapping

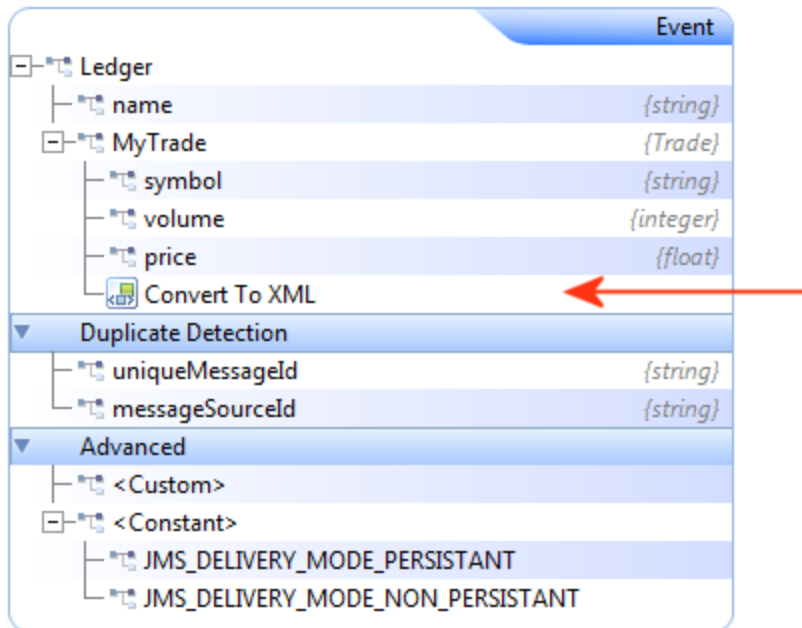
Using convention-based XML mapping when sending/generating messages

To map an Apama event to a JMS message using the convention-based approach:

1. Create an Apama event type with fields that correspond in type and order to the structure of the XML document. In order to use convention-based XML mapping, the event type representing the XML document must be nested inside a parent event type, so ensure that such a parent event type has been created. Typically, the parent event type might have two fields, a `string` field representing the JMS destination, and an `event` field representing the root of the XML document.
2. In the adapter editor's Event Mappings tab, click the Add Event button () to add a mapping for the desired parent event type (that is, the event type that contains the event field that represents the XML root element).
3. In the adapter editor's Event Mappings tab, right-click the Apama event that represents the root node of the XML document and select Add Computed Node to display the Add Computed Node.
4. In the Add Computed Node dialog's Select Method field, select Convert to XML from the drop-down list. The dialog is updated to show more information.

You can specify a namespace and namespace prefix for the generated XML document if desired, or else leave them blank. By default the Include empty fields option is enabled. This specifies that empty XML nodes will be generated when empty EPL `string` fields are encountered within an Apama event. This option does not affect empty strings within a sequence of EPL strings. If you clear the check box to disable the option, empty XML nodes will not be generated.

5. Click OK.
6. In the mapping tree, Apama Studio adds an entry of type `Convert To XML` to the selected event node.



7. Drag a mapping line from the `Convert To XML` entry to the desired node in the XML message, for example, to the `JMS Body`.

Convention-based XML mapping

Combining convention-based XML mapping with template-based XML generation

It is also possible to combine the convention-based approach with template-based XML generation. An XML template can be used to generate the top-level XML document, while one or more placeholders can be added and mapped to XML sub-document strings generated by convention-based XML mapping.

To combine these approaches:

1. Right-click a source event (representing an XML root element) or sequence (representing a list of XML elements) and click Add Computed Node.
2. Select Convert to XML from the drop-down list and click OK. This will result in a `Convert To Xml` node representing a generated XML string.
3. Drag a mapping line from that node to the target `$placeholder` node specifying where the XML snippet should be inserted into the top-level document.

Convention-based XML mapping

Using EDA events in Apama applications

Software AG's Event-Driven Architecture (EDA) allows information to be published so that one or more interested consumers can subscribe to what was published. Published data is in the form of EDA events. Each EDA event adheres to an XML schema (`.xsd` file) that defines an EDA event type.

In an Apama application, you can use correlator-integrated messaging for JMS to consume and publish EDA events. The following topics provide information and instructions to help you do this:

- ["About convention-based EDA mapping" on page 171](#)
- ["Creating Apama event type definitions for EDA events" on page 176](#)
- ["Automatically mapping configurations for EDA events" on page 177](#)
- ["Manually mapping configurations for EDA events" on page 179](#)

Mapping Apama events and JMS messages

About convention-based EDA mapping

The following topics describe the conventions that Apama Studio uses to map EPL events and EDA events.

- ["Rules that govern automatic conversion between of EDA events and Apama events" on page 171](#)
- ["Rules that govern EPL name generation" on page 174](#)

Using EDA events in Apama applications

Rules that govern automatic conversion between of EDA events and Apama events

The following table lists the conventions and rules that Apama Studio follows when it

- Uses EDA event schemas to generate Apama event definitions
- Converts Apama events to EDA events or converts EDA events to Apama events

EDA schema construct	Conventions/rules
Simple XML type	Represented by a simple type field such as <code>string</code> or <code>integer</code> .
Complex XML type	Represented by Apama events having the same name and structure.
XML attribute	Represented by a simple type field. The name of the field is obtained by prefixing an underscore to the attribute name.
Simple XML element	Represented by a simple type field with same name.
Complex XML element	Represented by a field of <code>event</code> type. The <code>event</code> type corresponds directly to the complex type of the element. The name of the field is equal to the element name.
Anonymous complex type element	Represented by a field of <code>event</code> type. The <code>event</code> type corresponds directly to the anonymous complex type. The name of the <code>event</code> type is derived by combining the parent elements/types names and the current element name. The name of the field is equal to the element name.
Namespace	The package of generated Apama events corresponds directly to the namespace of the corresponding XML Schema types. The package is obtained by stripping the fixed part of the namespace name and by replacing "/" with ".". The package name is translated into the namespace name when generating EDA XML from Apama events. For details, see "Rules that govern EPL name generation" on page 174 .
Element cardinality	An element with a cardinality combination other than <code>minOccurs=1</code> and <code>maxOccurs=1</code> is represented as a sequence of corresponding types, that is, a <code>sequence</code> of simple types for simple type elements or a <code>sequence</code> of <code>event</code> types for complex type elements. You must ensure that the size of the <code>sequence</code> is according to the cardinality.
Optional attribute	Represented by a sequence of the corresponding simple type. An empty sequence represents the absence of attributes. The sequence can contain zero or one element.
Sequence	Corresponding fields for all the elements are added to a parent event type that corresponds to the complex type.

EDA schema construct	Conventions/rules
All	Corresponding fields for all the elements are added to a parent event type that corresponds to the complex type.
Choice	Corresponding <code>sequence</code> type fields for all the elements are added to the parent event type. Only the sequence of the chosen element should be filled; the others must be empty.
Restriction	A simple type element with one or more restrictions is represented by a simple type field of the corresponding type. You must ensure that the value specified is within restrictions.
Extension of simple type	Represented by an <code>event</code> type. The <code>event</code> type contains a simple type field named <code>xmlTextNode</code> to hold the text content of the element. Additionally, the <code>event</code> type contains fields for attributes.
Extension of complex type	Represented by two events, one corresponds to the base type and the other corresponds to the extension type. The event corresponding to the extension type contains all fields added by the extension type. It also contains a field named <code>xmlExtends</code> that is of the base event type.
Element reference	A separate event is generated for a referenced element. The name of the event is the referenced element name appended with <code>Element</code> . The event contains a field with the same name as the referenced element name. The event using the reference element contains a field whose name is obtained by prefixing <code>xml</code> to the referenced element name.
<code>anyType</code>	An element of type <code>xsd:anyType</code> is represented by a field of <code>string</code> type or <code>sequence<string></code> type depending on the cardinality. The name of the field is obtained by prefixing <code>xml</code> to field name. The content of the field is the whole element node serialized as a string.
<code>any</code>	The <code>xsd:any</code> element is represented by a field named <code>xmlFragments</code> . The type of the field is <code>string</code> or <code>sequence<string></code> depending on the cardinality. The content of the field is the whole XML node corresponding to <code>xsd:any</code> serialized as a string.
<code>anyAttributes</code>	The <code>xsd:anyAttributes</code> element is represented by a field named <code>xmlAttributes</code> of type <code>dictionary<string,string></code> .

EDA schema construct	Conventions/rules
	The content of the dictionary is name/value pairs for attributes that correspond to <code>xsd:anyAttributes</code> .
Special characters	If an XML name contains a hyphen or a period, or begins with a number, special treatment in EPL is required. For details, see "Rules that govern EPL name generation" on page 174 .
<code>anySimpleType</code> or <code>anyURI</code>	An element of type <code>xsd:anySimpleType</code> or <code>xsd:anyURI</code> is represented by a field of <code>string</code> type. The name of the field is same as the name of element. The content of the field is the same as the content of the element.

About convention-based EDA mapping

Rules that govern EPL name generation

When Apama Studio uses an EDA event type schema to generate EPL event type definitions, it derives the package name from the XSD namespace of the corresponding schema types. This ensures the ability to

- Distinguish between elements that have the same name but different namespaces
- Generate the namespace from the package name when creating an XML payload from Apama events

There is a one to one relationship between a namespace in an EDA event schema and an Apama package. For example, if there are two namespaces in the `.xsd` file then Apama Studio generates EPL event type definitions in two Apama EPL packages.

The following special characters are allowed in namespace and element names that are to be mapped to EPL package names, EPL event type names or EPL event type field names:

- Colon (:))
- Hyphen (-)
- Period (.))
- Underscore (_)

When a name contains one of these characters Apama Studio replaces it with an underscore and prefixes `x` to the name. The `x` in `x` is an ordered list of letters that identifies the characters that were in the name. The following table describes these escape characters:

Escape Character	Description
<code>c</code>	Indicates that a colon (:) was replaced.
<code>f</code>	Indicates that the package name was derived from a non-EDA namespace. If it is the first character in the escape sequence then the package name is the full namespace. For an EDA namespace, the EPL package name does not include the fixed part of the EDA namespace, which is <code>http://namespaces.softwareag.com/EDA/</code> .

Escape Character	Description
h	Indicates that a hyphen (-) was replaced.
n	<p>Indicates that nothing was replaced. However, since an EPL name can contain underscores but cannot have a number as the first character, an <code>n</code> in an escape sequence means one of the following:</p> <ul style="list-style-type: none"> • There was an underscore in the EDA name. • There was a number as the first character in the EDA name or at the beginning of a part of a namespace. <p>Having the <code>n</code> in the sequence provides the information needed to map the EPL name back to an EDA name.</p> <p>Exception: If an EDA name contains one or more underscores, but not any other special characters then Apama Studio does not add an escape prefix. This is because underscores are allowed in EPL names. When an underscore is in an EDA name, the EPL escape prefix is needed only if the name also contains a colon, hyphen, or period, or a number is the first character.</p>
p	Indicates that a period (.) was replaced.
u	Indicates that an underscore (_) was replaced.

Apama Studio maps a namespace to an EPL package name as follows:

- The package name contains a period (.) in place of each slash (/).
- For an EDA namespace, the package name contains only the non-fixed part of the namespace. The package name does not contain `http://namespaces.softwareag.com/EDA/`.
- For a non-EDA namespace, the package name contains the full namespace and `£` is the first letter in the escape sequence prefixed to the package name.

The following table provides examples of how EDA namespace names become EPL package names:

Namespace	EPL package name
<code>http://namespaces.softwareag.com/EDA/Hello/Big/Data</code>	<code>Hello.Big.Data</code>
<code>http://namespaces.softwareag.com/EDA/Hello-World</code>	<code>\$h\$Hello_World</code>
<code>http://namespaces.softwareag.com/EDA/Hello.World</code>	<code>\$p\$Hello_World</code>
<code>http://namespaces.softwareag.com/EDA/Hello_World</code>	<code>Hello_World</code>
<code>http://namespaces.softwareag.com/EDA/Hello_World/101</code>	<code>\$un\$Hello_World._101</code>
<code>http://namespaces.softwareag.com/EDA/Pulse/1.2</code>	<code>\$np\$Pulse._1_2</code>

Namespace	EPL package name
<code>http://namespaces.softwareag.com/EDA/a_small.mixed-bag</code>	<code>\$uph\$a_small_mixed_bag</code>
<code>http://www.example.com/sample</code>	<code>\$fcnpp\$http_._.www_example_com.sample</code>
<code>urn:uddi-org:api_v3</code>	<code>\$fchcu\$urn_uddi_org_api_v3</code>

The following table provides examples of how other EDA names become EPL names:

EDA element name	EPL name
<code><xsd:element name="MAC-Address"/></code>	<code>\$h\$MAC_Address</code>
<code><xsd:element name="Model.Type"/></code>	<code>\$p\$Model_Type</code>
<code><xsd:element name="Model_Type_Special"/></code>	<code>Model_Type_Special</code>

About convention-based EDA mapping

Creating Apama event type definitions for EDA events

To create an Apama event type definition for an EDA event:

1. Ensure that you can access the XML Schema (`.xsd`) file that defines the EDA event type or types for which you want to create Apama (EPL) event type definitions.
2. In Apama Studio's Project Explorer, right-click the eventdefinitions folder in the project in which you want to use EDA events and select **New > Event Definition**.
3. In the New Event Definition dialog, select **EDA Event Type** and click **Next**.
4. Accept the default containing folder or click **Browse** to specify the location of the EPL event type definition(s) to be generated. Click **Next**.

5. Click the **Browse** button to the right of the Schema Element/Type field to display the Type Chooser dialog, which allows selection of an XML Schema element that has the substitution group as `eda (Namespace) :Payload`.

The drop-down arrow lets you change scope among Recent files, Local file system, Workspace, Remote URL, and XML Schema.

6. Select an element, click **OK** and then **Finish**.

Apama Studio generates one EPL (`.mon`) file for each EDA namespace. Each `.mon` file contains event type definitions for all EDA event types defined in that namespace. The files are generated with file names in the following format:

```
payload_element_name_EDA.mon
```

If more than one `.mon` file is generated, file names end with `*_n.mon`. For example, `EDA_test_1.mon`, `EDA_sample_2.mon` and so on.

Generated EPL files include a root wrapper Apama event whose name is the name of the payload element appended by `_EDA`, for example, `CableboxHealth_EDA`. This root event is used for sending and receiving EDA events.

The root event contains a field for the payload element and a dictionary field for EDA headers. The name of the payload field corresponding to the payload element is the same as the name of the payload element. The type of the payload field is an `event` type that corresponds to the type of the payload element. The structure and name of the root event is governed by the conventions described in ["Rules that govern automatic conversion between of EDA events and Apama events" on page 171](#).

The root event also contains the `configure()` action to populate the header dictionary with filterable properties and some standard EDA headers. For example:

- `headers["$Event$Kind"] := "Event"; //Fixed value`
- `headers ["$Event$FormatVersion"] := "9.0"; //Fixed value`
- `headers ["$Event$Type"]` is set to the type of the EDA event from which these Apama event definitions are generated, for example: `"{http://namespaces.softwareag.com/EDA/WebM/Communication/Sms}SmsSent"`.


In the EPL context from which you sent the root event to a JMS topic for consumption by EDA, you must call the `configure()` action before you send the event.

Once you have EPL event type definitions for the EDA event types your application needs to use, you can set up the receiver and sender mapping configurations in a correlator-integrated adapter for JMS.

Using EDA events in Apama applications

Automatically mapping configurations for EDA events

With EPL event type definitions for the EDA events you want to use in your application, you can use Apama Studio to automatically create sender and receiver mapping configurations in a correlator-integrated adapter for JMS. To do this:

1. If you have not already done so, add a correlator-integrated messaging adapter for JMS to your project.
2. In the Project Explorer pane, in the project that uses EDA events, expand the **Adapters** folder and double-click the correlator-integrated adapter for JMS instance to open it in the JMS Connections editor.
3. In the editor, click the **Event Mappings** tab.
4. In the **Sender Mapping Configuration** tab, click the down-pointing carat to the right of the plus sign  and select **Add EDA Event**.

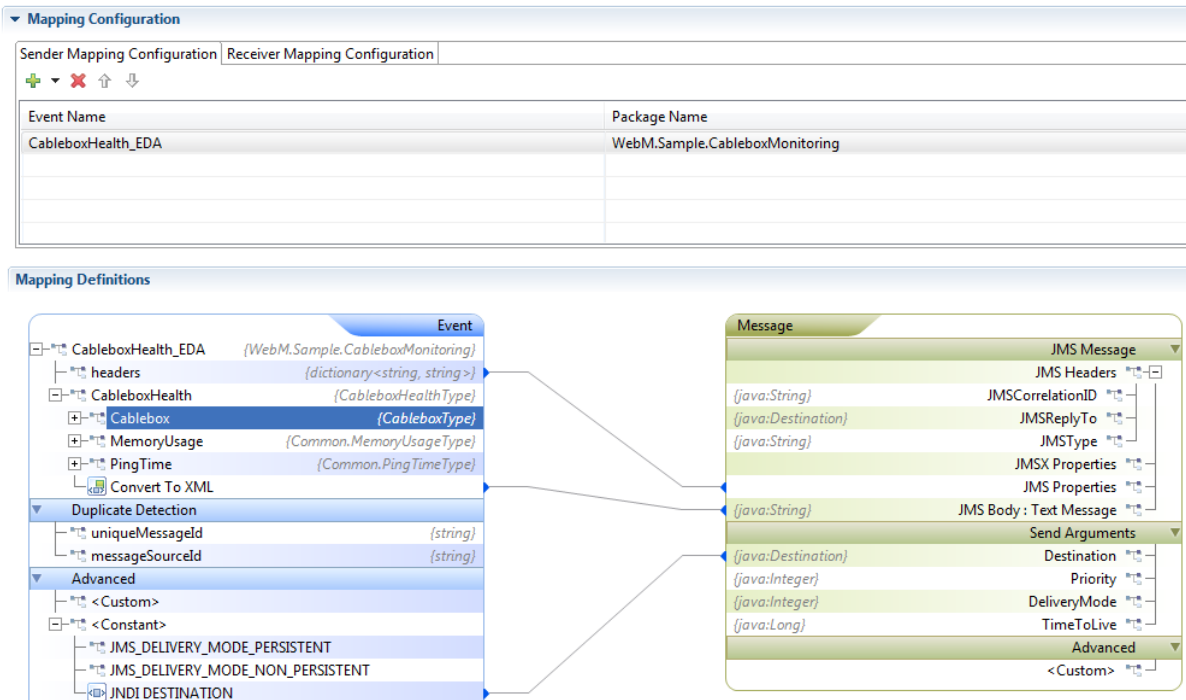
The **Event Type Selection** dialog appears with `*_EDA` in the filter field and a list of the EPL event type definitions that correspond to EDA event types.

5. Select one or more event types for which you want to configure sender mappings and click **OK**.

For each selected event, Apama Studio generates three mappings from Apama EPL event type definitions to JMS messages that will publish EDA events:

- `headers` is mapped to `JMS Properties`.
- `Convert to XML` is mapped to `JMS Body`. Apama Studio automatically turns on **Apply EDA rules** for the `Convert to XML` node. This means that Apama Studio applies a defined set of conventions for mappings between EPL event types that represent EDA events and the JMS messages that publish and consume the EDA events. See ["About convention-based EDA mapping" on page 171](#).

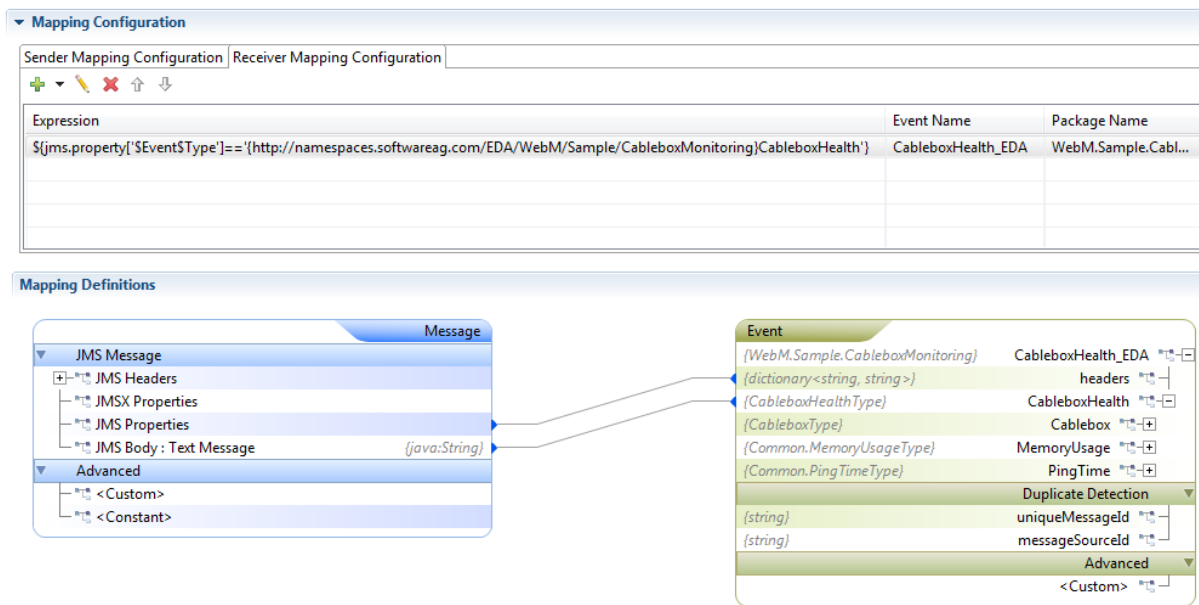
- Constant node item `JNDI DESTINATION` is mapped to `Destination`. The `Destination` is the JNDI name, which follows the convention of the EDA event. You should create a JMS topic and map it to the JNDI name according to EDA conventions before using Apama to send or receive an EDA event. For example:



6. Click the Receiver Mapping Configuration tab.

Apama Studio automatically specifies that the `$Event$Type` property of the JMS message should match the expected name of the EDA event. When the expression evaluates to true for the JMS property value of an incoming event it triggers the mapping of the incoming JMS message to an Apama EPL event type. Apama Studio also automatically generates two mappings from JMS messages that consume EDA events to Apama EPL event type definitions:

- JMS Properties to headers.
- JMS Body to the main text node in the event type definition. This mapping converts the EDA XML to Apama events according to the conventions of EDA mapping. For details, see ["About convention-based EDA mapping" on page 171](#). For example:



Using EDA events in Apama applications

Manually mapping configurations for EDA events

In most cases, you can use Apama Studio to automatically map EPL events and JMS messages that consume and publish EDA events. However, in the following situations, you might find that you need to manually configure these mappings:

- You already defined EPL event types that you want to use with EDA.
- Automatic convention-based mapping provided by Apama Studio is not sufficient.

A correlator-integrated adapter for JMS supports various approaches for manually mapping EPL events and JMS messages. See ["Mapping Apama events and JMS messages" on page 164](#).

The following steps are required to manually map Apama events and JMS messages that consume and publish EDA events.

1. Create Apama event type definitions to hold the EDA payload data.
2. Configure mappings for sending JMS messages that publish EDA events. Do the following in the correlator-integrated messaging for JMS adapter editor Event Mappings tab, with the Sender Mapping Configuration tab selected:
 - a. Pick the Apama event that is the placeholder event for the EDA event type in Apama to do all the associations.
 - b. Map all standard EDA headers with proper EDA header names to JMS properties.
 - c. Map the destination information either as a constant or from some Apama field value.
 - d. Create EDA XML from the selected Apama event and map the XML to JMS Body.

As discussed in ["Mapping Apama events and JMS messages" on page 164](#), there are a number of ways to create EDA XML. You can use any approaches or any combination of approaches described in that section. For example, you can use template-based mapping with convention-based mapping. Some part of the template can be generated by using convention-

based mapping, some part can be hardcoded, and some part can be directly mapped from simple type EPL fields.

3. Configure mappings for receiving JMS messages that consume EDA events. Do the following in the correlator-integrated messaging for JMS adapter editor Event Mappings tab, with the Receiver Mapping Configuration tab selected:
 - a. Pick the Apama event that is the placeholder event for the EDA event type in Apama to do all the associations.
 - b. In the Expression column, update the expression to invoke the mapping for a specific JMS message. Usually, the expression should match the `$Event$Type` JMS property to the name of the expected EDA event.
 - c. Map standard EDA headers from JMS properties. You can either map all JMS properties to a dictionary or map a specific JMS property to an Apama event field.
 - d. Map JMS Body to the Apama event.

As discussed in ["Mapping Apama events and JMS messages" on page 164](#), there are a number of ways to create Apama events from EDA XML. Also, multiple approaches can be combined. For example, you can apply XPATH to the payload XML to obtain just the XML that can be converted into an Apama event by using convention-based mapping. Alternatively, you can apply XSLT to transform the payload XML and then use convention-based mapping to generate Apama events.

Using EDA events in Apama applications

Handling binary data

JUEL mapping expressions can use the following methods on binary data.

The `byteArrayToBase64()` method takes a byte array as the input parameter and returns the Base64 encoded string. Following is an example of mapping byte array data to a Base64-encoded string, which can be mapped to an Apama `string` field.

```
${byteArrayToBase64(jms.body.bytesmessage)}
```

The `base64ToByteArray()` method takes a Base64-encoded string as an input parameter and returns a byte array. For example:

```
${base64ToByteArray(apamaEvent['body'])}
```

The `javaObjectToByteArray()` method takes a serializable Java object as the input parameter and returns a serialized byte array. In the following example, the body of `jms.objectmessage` is serialized into a byte array, the byte array is then encoded into a Base64-encoded string, and that string can be mapped to a `string` field in an Apama event.

```
${byteArrayToBase64(javaObjectToByteArray(jms.body.objectmessage))}
```

The `byteArrayToJavaObject()` method takes a byte array as the input parameter and returns a deserialized Java object. In the following example, the `string` field of an Apama event, which is a Base64-encoded string, is converted into a byte array. The byte array is then deserialized into a Java object and can be mapped, for example, to the body of `jms.objectmessage`.

```
${byteArrayToJavaObject(base64ToByteArray(apamaEvent['body']))}
```

These binary methods can be used by creating custom expressions. In the Event Mappings tab, right-click a `<Custom>` node and select Add Node to display the Add Node dialog:

Note: In these binary methods, if the argument passed to the method is null or empty then the method returns null. If a null value would be set for a field in an Apama event then the field is set to the default value for the field's type, for example, a `string` field is set to the empty string, `""`.

Mapping Apama events and JMS messages

Using custom EL mapping extensions

Apama mapping tools use an expression-based mapping layer to map between Apama events and external message payloads. The expressions use Java Unified Expression Language (EL) resolvers and methods, which must be registered to the mapping layer. Apama includes a set of EL resolvers and EL methods that are registered for you and that you can use in mapping expressions. If you want you can register your own EL resolvers and EL methods and then use them as custom mapping extensions.

See ["Using custom EL mapping extensions" on page 91](#).

Mapping Apama events and JMS messages

JUEL mapping expressions reference for JMS

The expressions that can be used to get or set elements of a JMS message are listed below, along with the set of Apama field types that are recommended for use when mapping when sending or receiving JMS messages:

JMS message element	JMS EL expression	Compatible Apama field type(s) when sending	Compatible Apama field type(s) when receiving

Dictionary of all message headers	jms.headers	dictionary<string, string>	dictionary<string, string>
JMSDestination	jms.header['JMSDestination']	string (with jndi:/topic:/queue: prefix)	string (with topic:/queue: prefix)
JMSReplyTo	jms.header['JMSReplyTo']	string (with jndi:/topic:/queue: prefix)	string (with topic:/queue: prefix)
JMSCorrelationID	jms.header['JMSCorrelationID']	string	string
JMSType	jms.header['JMSType']	string	string
JMSPriority	jms.header['JMSPriority']	integer, string	integer, string
JMSDeliveryMode	jms.header['JMSDeliveryMode']	integer, string (must be a number (though display string can be used (only) when mapping a constant value in tooling); 1=NON_PERSISTENT, 2=PERSISTENT)	integer, string
JMSTimeToLive	jms.header['JMSTimeToLive']	integer, string (in milliseconds from the time JMS sends the message)	N/A when receiving
JMSExpiration	jms.header['JMSExpiration']	N/A when sending	integer, string (in milliseconds since the epoch)
JMSMessageID	jms.header['JMSMessageID']	N/A when sending	boolean, string
JMSTimestamp	jms.header['JMSTimestamp']	N/A when sending	integer, string (in milliseconds since the epoch)
JMSRedelivered	jms.header['JMSRedelivered']	N/A when sending	string
Dictionary of all message properties	jms.properties	dictionary<string, string>	dictionary<string, string>

String Message Property	jms.property['propName']	string	string
Boolean Message Property	jms.property['propName']	boolean	boolean, string
Long Message Property	jms.property['propName']	integer	integer, string
Double Message Property	jms.property['propName']	float	float, string
Byte Message Property	jms.property['propName']	Not supported	string
Short Message Property	jms.property['propName']	Not supported	string
Integer Message Property	jms.property['propName']	Not supported	string
Float Message Property	jms.property['propName']	Not supported	string
JMSX Property	jms.xproperty['propName']	same as other properties	same as other properties
Dictionary of all JMSX properties	jms.xproperties	dictionary<string, string>	dictionary<string, string>
TextMessage Body	jms.body.textmessage	string, event ^[1]	string, event ^[1]
MapMessage Body	jms.body.mapmessage	dictionary<string, string>	dictionary<string, string>
MapMessage Body Entry	jms.body.mapmessage['mapKey']	string	string
ObjectMessage Body with a serializable java.util.Map<Object, Object>	jms.body.objectmessage	dictionary<string, string>	dictionary<string, string>
ObjectMessage Body with a serializable java.util.List<Object>	jms.body.objectmessage	sequence<string>	sequence<string>

ObjectMessage Body with any serializable Object	jms.body. objectmessage	N/A	string
BytesMessage Body	jms.body.bytesmessage	string, sequence<string>, dictionary<string, string>, event	string, sequence<string>, dictionary<string, string>
TextMessage, MapMessage, BytesMessage, ObjectMessage, Message	jms.body.type	string	string

^[1] If a string from the JMS message is mapped to an event, the string should be either of:

- An Apama event string (as generated by the Apama Event Parser), whose event type matches the type of the field it is being mapped to in the source/target Apama event.
- An XML document starting with a < character, whose structure matches what is implied by the event type definition it is being mapped to (see ["Convention-based XML mapping" on page 167](#) for more information)

Note, the JMS headers JMSMessageID, JMSRedelivered and JMSDeliveryMode are supported for completeness but will not normally be required by Apama applications, since built-in duplicate detection based on application-level unique identifiers replaces the first two, and rather than overriding the per-message delivery mode it is usually best to use the default `PERSISTENT/NON_PERSISTENT` setting implied by the sender's `senderReliability` value.

Resolver expressions for obtaining ids

The following tables describe resolver expressions for obtaining sender, receiver, and connection IDs. You cannot use these expressions to set ids.

For sending messages	Description
<code>\${jmsSender['senderId']}</code>	Get the sender id of the sender that is sending the event from your Apama application to a JMS broker.
<code>\${jmsSender['connectionId']}</code>	Get the connection id of the sender that is sending the event from your Apama application to a JMS broker.
For receiving messages	Description
<code>\${jmsReceiver['receiverId']}</code>	Get the receiver id of the receiver in your Apama application that received the JMS message from a JMS broker.
<code>\${jmsReceiver['connectionId']}</code>	Get the connection id of the receiver in your Apama application that received the JMS message from a JMS broker.

String methods in mapping expressions

In JUEL mapping expressions, you can use certain string methods in the parts of the mapping expressions that evaluate to `string` types. The table below describes the `string` methods you can use. These methods use the same-named `java.lang.String` methods. The mapping expressions are evaluated first to obtain a result string and then any specified string method is applied. You use these functions in the following way:

```
${some_expression.substring(5)}
```

In the previous format, `some_expression` is an expression that evaluates to a `string`. In the following examples, `f1` is a field of type `string`:

```
${apamaEvent['f1'].toString().contains('in')}  
${jms.body.textmessage.toString().startsWith('sample')}
```

String method	Description
<code>equalsIgnoreCase('str')</code>	Returns a boolean value that indicates whether the result string is equal to the specified string, ignoring case.
<code>contains('str')</code>	Returns a boolean value that indicates whether the result string contains the specified string.
<code>matches('regex')</code>	Returns a boolean value that indicates whether the result string matches the specified Java regular expression.
<code>startsWith('str')</code>	Returns a boolean value that indicates whether the result string starts with the specified string.
<code>endsWith('str')</code>	Returns a boolean value that indicates whether the result string ends with the specified string.
<code>toLowerCase()</code>	Converts the result string to lowercase and returns it.
<code>toUpperCase()</code>	Converts the result string to uppercase and returns it.
<code>concat('str')</code>	Appends the result string with the specified string and returns this result.
<code>replaceAll('regex', 'regexReplacement')</code>	<p>In the result string, for each substring that matches <code>regex</code>, this method replaces the matching substring with <code>regexReplacement</code>. The string with replacement values is returned.</p> <p>The <code>regexReplacement</code> string may contain backreferences to matched regular expression subsequences using the <code>\</code> and <code>\$</code> characters, as described in the Oracle API documentation for <code>java.util.regex.Matcher.replaceAll()</code>. If a literal <code>\$</code> or</p>

String method	Description
	\ character is required in <code>regexReplacement</code> be sure to escape it with a backslash, for example: <code>"\\$"</code> or <code>"\\"</code> .
<code>substring(startIndex, endIndex)</code>	Returns a new string, which is a substring of the result string. The returned substring includes the character at <code>startIndex</code> and subsequent characters up to but not including the character at <code>endIndex</code> .
<code>substring(startIndex)</code>	Returns a new string, which is a substring of the result string. The returned string includes the character at <code>startIndex</code> and subsequent characters including the last character in the string.
<code>trim()</code>	Returns a copy of the result string with leading and trailing whitespace removed.

Binary methods in mapping expressions

In JUEL mapping expressions, you can use certain binary methods in the parts of the mapping expressions that evaluate to binary data. The table below describes the binary methods you can use.

Binary method	Description
<code>byteArrayToBase64(byteArray)</code>	Encodes a byte array to a Base64-encoded string.
<code>base64ToByteArray(string)</code>	Decodes a Base64-encoded string to a byte array.
<code>javaObjectToByteArray(object)</code>	Serializes the serializable Java object to a byte array.
<code>byteArrayToJavaObject(base64String)</code>	Deserializes the byte array to a serializable Java object.

Mapping Apama events and JMS messages

Implementing a custom Java mapper

If the mapping tools provided with Apama do not meet your needs then you can implement your own Java class to map between Apama event strings and JMS message objects. A custom mapper can handle some event types and delegate handling of other event types to the mapping tools provided with Apama or to other custom mapping tools. Typically, you will want to use the `SimpleAbstractJmsMessageMapper` class, which is in the `com.apama.correlator.jms.config.api.mapper` package. This topic provide a general description of how to implement a custom mapper. See the Javadoc for details.

API overview

The `SimpleAbstractJmsMessageMapper` class is a helper class for implementing simple mappings between JMS messages and Apama events. This class is the recommended way to implement a stateless, bi-

directional mapper, with trivial implementations of methods that most implementors will not need to be concerned with. Most implementations will need to override only the following methods:

- The `mapApamaToJmsMessage()` method converts an Apama event string and (possibly null) unique ids for elimination of duplicate messages to a `JmsSenderMessageHolder` object that contains the message and message-sending parameters.

```
abstract JmsSenderMessageHolder mapApamaToJmsMessage(  
    JmsSenderMapperContext context, MappableApamaEvent event)
```

- The `mapJmsToApamaMessage()` method converts a JMS message object to an Apama event string. It can also add the Apama unique message id (if it is available) into the `Message` object for elimination of duplicate messages.

```
abstract MappableApamaEvent mapJmsToApamaMessage(  
    JmsReceiverMapperContext context, javax.jms.Message message)
```

Typically, the mapper class would contain `com.apama.event.parser.EventType` fields for each of the Apama event types the mapper can handle. Your custom mapper methods will use these fields to parse and generate Apama events. Both methods use `MappableApamaEvent`, which wraps an Apama event string plus optional duplicate detection information.

A `JmsSenderMessageHolder` object wraps a JMS message object in addition to JMS send parameters such as the JMS destination.

The `JmsReceiverMapperContext` and `JmsSenderMapperContext` objects give mappers access to helper methods for

- Converting between strings and JMS destinations
- Performing JNDI lookups if JNDI is configured
- Obtaining other contextual information that may be needed during mapping such as the `receiverId` or `senderId`

The `SimpleAbstractJmsMessageMapper` class also provides optional `senderMapperDelegate/receiverMapperDelegate` bean properties that identify another mapper to use for any messages that this mapper does not handle. The associated methods are used for the XML configuration but should not be called by subclasses.

If your custom mapper requires configuration properties to be specified in the XML configuration file then define the properties as standard Java bean get/set public methods. Include any logic required to validate parameter values in the overridden `JmsSenderMessageHolder.init()` method.

For more complex needs, do not use the `SimpleAbstractJmsMessageMapper` class. Instead, implement the factory interfaces directly to create separate classes for the sender and receiver mappers. This is particularly important when the mapping operations are stateful. For example, if they rely on a cache that should not be shared across all the mapper instances created by the factory to avoid thread-safety concerns or costly and unnecessary synchronization. For the receiver side (sender side is identical) there are two interfaces:

- `JmsReceiverMapperFactory` is the interface that must be implemented by the Java bean, holding any required configuration information. This class will be referenced in the XML configuration file. It provides get/set methods for configuration properties, an `init()` method to perform any validation and a factory method to create `JmsReceiverMapper` instances.
- `JmsReceiverMapper` is the interface that is responsible for actually mapping the objects. A new instance will be created for each receiver and for each thread on which mapping occurs, so this instance can hold any required caches or state without the need for costly locking/

synchronization. A `destroy()` method is provided in case there are resources that need to be cleared or closed when the associated receiver is shut down is removed.

Configuring a custom mapper

To configure a JMS connection to use a custom mapper class, edit the connection's XML configuration file as follows:

- Add a bean definition for the sender and/or receiver mapper factory class, with any associated configuration, and an `id` attribute that will be used to identify this mapper bean in the rest of the configuration. Usually the simplest way to specify the classpath for the custom mapper's classes is to put the mapper bean definition inside a new `<jms:classpath>` element.
- Under the `jms:connection` element, set the `receiverMapper` and/or `senderMapper` properties to point to this mapper, typically you use a `ref="beanid"` attribute to do this. If these properties are not specified explicitly, the default is that the connection uses the Apama-provided mapper, assuming it is the only mapper defined in the configuration.

Following is an example of a configuration that specifies custom mappers:

```
<jms:classpath classpath="mycp">
  <bean id="myCustomMapper" class="MyMapper">
    <!-- if this uses SimpleAbstractJmsMessageMapper, optionally specify the
    factory bean to delegate to for messages this mapper does not handle. -->
    <property name="senderMapperDelegate" ref="standardMapper"/>
    <property name="receiverMapperDelegate" ref="standardMapper"/>
    <!-- mapper-specific configuration could go here -->
  </bean>
</jms:classpath>
<jms:classpath classpath="...">
  <jms:connection id="myConnection">
    ...
    <property name="senderMapper" ref="myCustomMapper"/>
    <property name="receiverMapper" ref="myCustomMapper"/>
  </jms:connection>
</jms:classpath>
```

Any mapper that subclasses `SimpleAbstractJmsMessageMapper` also supports the optional properties `senderMapperDelegate` and `receiverMapperDelegate`. These properties can be used to specify a fallback mapper (factory bean) to delegate to for message types this mapper does not support. Map methods must return null to indicate such types. For other errors, exceptions should always be thrown.

Mapping Apama events and JMS messages

Dynamic senders and receivers

In addition to specifying static senders and receivers in the adapter's configuration file as introduced in ["Getting started with simple correlator-integrated messaging for JMS" on page 152](#), you can dynamically add and manage senders and receivers using actions on Apama's `JMSConnection` event. (Note, for more information on static senders and receivers, see ["Adding static senders and receivers" on page 199](#).)

The unique identifiers specified when adding dynamic senders or receivers must not clash with the identifiers used for any static senders and receivers in the configuration file. You cannot dynamically remove a sender or receiver that was defined statically in the configuration file; only dynamically added senders and receivers can be removed.

It is currently valid to send events to the channel associated with a newly created dynamic sender as soon as the add action has returned. In this case, the correlator ensures that the events get sent to the JMS broker eventually. However, best practice is to add a listener for `JMSSenderStatus` events and wait

for the `OK` status before beginning to send to a dynamic sender. It is valid to send events to an existing sender's channel at any point until its removal is requested by calling the `remove()` action. It is not valid to send any events to that channel after `remove()` has been called, and any events sent after this point are in doubt and could be ignored without any error being logged. Applications that make use of multiple contexts may need to coordinate across contexts to ensure that no `send` or other operations are performed on senders that have been removed in another context.

For more information on dynamically adding senders and receivers, see the `JMSConnection` event documentation in the ApamaDoc documentation.

The example Apama application located in the `APAMA_HOME\samples\correlator_jms\dynamic-event-api` directory demonstrates how to use the event API to dynamically add and remove JMS senders and receivers. In addition, it shows how to monitor senders and receivers for errors and availability.

Using Correlator-Integrated Messaging for JMS

Durable topics

JMS durable topic subscriptions are supported for both static and dynamic receivers. This lets Apama applications persistently register interest in a topic's messages with the JMS broker. If the correlator is down then messages sent to the topic will be held ready for delivery when the correlator recovers.

Statically configured durable topic subscriptions cannot be removed. When a dynamic receiver using a durable topic subscription is removed, the JMS subscription to the topic will be removed at the same time, before the `REMOVED` receiver status notification event is sent. A consequence of this is that the removal of a receiver will not be completed until the JMS connection is up, in order that the subscription can be removed from the JMS broker. Note that durable topic subscriptions cannot be created using `BEST_EFFORT` receivers.

The preferred method of subscribing to a durable topic is to use the `addReceiverWithDurableTopicSubscription` (or `addReceiverWithConfiguration`) action on the `com.apama.correlator.jms.JMSConnection` event. For more information on these actions, see the `JMSConnection` event documentation in the ApamaDoc documentation.

Using Correlator-Integrated Messaging for JMS

Receiver flow control

It is possible to give an EPL application control over the rate at which events are taken from the JMS queue or topic by each JMS receiver. To enable this option, set the `receiverFlowControl` property to `true` in the `JmsReceiverSettings` bean. The configuration for this bean is found in the `jms-global-spring.xml` file. To display the file in Apama Studio, select the Advanced tab in the adapter configuration editor.

Once `receiverFlowControl` has been enabled, use the `com.apama.correlator.jms.JMSReceiverFlowControlMarker` event to enable receiving events from each receiver, by specifying a non-zero window size. For example, to ensure that each receiver will never add more than 5000 events to the input queue of each public context, add the following EPL code

```
using com.apama.correlator.jms.JMSReceiverFlowControlMarker;
...
JMSReceiverFlowControlMarker flowControlMarker;
```

```
on all JMSReceiverFlowControlMarker(): flowControlMarker
{
    flowControlMarker.updateFlowControlWindow(5000);
}
```

A flow control marker is an opaque event object that is always sent to the correlator's public contexts when a new receiver is first added and during recovery of a persistent correlator. The message is also sent regularly as new messages are received and mapped, which typically happens at the end of each received batch, for example, at least once every 1000 successfully-mapped events if the default setting for `maxBatchSize` is used. The marker event indicates a specific point in the sequence of events sent from each receiver, and the application must always respond by calling the `updateFlowControlWindow` action on this marker event. This sets the size of the window of new events the receiver is allowed to take from the JMS queue or topic, relative to the point indicated by the marker.

More advanced applications that need to block JMS receivers until asynchronous application-specific operations arising from the processing of received messages (such as database writes and messaging sending) have completed can factor the number of pending operations into the flow control window. To reliably do this, it is necessary to stash the marker events for each receiver in a dictionary and add logic to call `updateFlowControlWindow` when the number of pending operations falls, so that any receivers that were blocked due to those operations can resume receiving. It is the application's responsibility to ensure that receivers do not remain permanently blocked, by calling `updateFlowControlWindow` sufficiently often. For an example of how receiver flow control can be used together with asynchronous per-event operations, see the `flow-control` sample application in `APAMA_HOME\samples\correlator_jms`.

Applications must make sure that they listen for all `JMSReceiverFlowControlMarker` events, and that their listener for the flow control markers is set up before `JMS.onApplicationInitialized` is called. Any stale or invalid `JMSReceiverFlowControlMarker` event, for example, from before a persistent correlator was restarted, cannot be used to update the flow control window, and any calls on such stale events will simply be ignored.

Documentation is available for the `com.apama.correlator.jms.JMSReceiverFlowControlMarker` event along with documentation for the rest of the API for correlator-integrated messaging for JMS. See the [ApamaDoc](#) documentation.

The current window size for all receivers is indicated by the `rWindow` item in the "JMS Status" lines that are periodically logged by the correlator, and this may be a useful debugging aid if receivers appear to be blocked indefinitely.

Using Correlator-Integrated Messaging for JMS

Monitoring correlator-integrated messaging for JMS status

Apama applications often need to monitor the status of JMS connections, senders, and receivers when the application needs to wait for a receiver or sender to be available (status "OK") before using it, and, conversely, to detect and report error conditions.

The main way to monitor status is to simply set up EPL listeners for the `JMSConnection`, `JMSReceiverStatus`, and `JMSSenderStatus` events which are sent to all public correlator contexts automatically, both on startup and whenever the status of these items changes. Note that there is no need to 'subscribe' to receive these events — provided `JMS.onApplicationInitialized()` was called, these events will be sent automatically, so all that is required is to set up listeners.

Some applications, especially those built using scenarios rather than EPL, prefer to monitor status using the standardized event API defined by `StatusSupport.mon`. The `CorrelatorJMSStatusManager` monitor, which is part of the correlator-integrated messaging for JMS bundle, acts as a bridge between the JMS-specific status events and this API, to allow Apama applications to monitor the status of JMS connections, senders and receivers using the standard Status Support interface. To use this interface:

1. Send a `com.apama.statusreport.SubscribeStatus` event, which is defined as:

```
event SubscribeStatus {
    string serviceID;
    string object;
    string subServiceID;
    string connection;
}
```

The fields for the `SubscribeStatus` event are:

- `serviceID` - This should be set to `CORRELATOR_JMS`.
 - `object` - Can be `CONNECTION`, `RECEIVER`, or `SENDER`, or "" (empty string). If "" is specified, the application will subscribe to status events for all connections, receivers, and senders.
 - `subServiceID` - The name of a specific receiver or sender if `RECEIVER` or `SENDER` is specified in the `object` field. If the `object` field specifies `RECEIVER` or `SENDER`, the `subServiceID` field must have a valid, non-empty value. If the `object` field specifies `CONNECTION` this field must be "".
 - `connection` - The name of a specific connection. If the `object` field specifies a value, the `connection` field must have a valid, non-empty value.
2. Create listeners for `com.apama.statusreport.Status` events (and optionally for `StatusError` events which are sent if the status subscription failed due to an invalid identifier being specified).
 3. To unsubscribe, send an `UnsubscribeStatus` event with field values that match the corresponding `SubscribeStatus` event.

For more information on monitoring correlator-integrated messaging for JMS connections, receivers, and senders, see the descriptions of the `JMSConnectionStatus`, `JMSReceiverStatus`, and `JMSSenderStatus` events in the ApamaDoc documentation.

Using Correlator-Integrated Messaging for JMS

Logging correlator-integrated messaging for JMS status

The correlator writes status information to its log file every five seconds or at an interval that you set with the `--logQueueSizePeriod` option. In addition to the standard correlator status information described in , correlators that are configured for integrated messaging log JMS status information. This information is logged in the following form:

```
INFO [20032] - Correlator Status: sm=2 nctx=1 ls=4 rq=0 eq=0 iq=0 oq=0 rx=8
tx=6 rt=0 nc=1 vm=251384 rung=0
INFO [20032:Status] - JMS Status: s=1 tx=6 sRate=1,200 sOutst=9,005 r=2
rx=4 rRate=1,180 rWindow=-1 rRedel=3 rMaxDeliverySecs=2.1
rDupsDet=2 rDupIds=2,005,023 connErr=2 jvmMB=49
```

Status information logged by correlators that are configured for integrated messaging is described in the following sections.

Table 1. Correlator status log fields related to JMS

Field	Full name	Description
s	Number of senders	The current number of JMS senders (both static and dynamic) on all JMS connections.
tx	Sent events	The total number of events sent to all JMS sender channels and which have been fully processed (either sent to JMS or exhausted the maximum failure retry limit). Includes events sent to dynamic senders that have since been removed, but does not include events sent before the correlator was restarted.
sRate	Send throughput rate	The total number of events sent per second across all senders, calculated over the interval since the last status line (typically 5 seconds).
sOutst	Outstanding sent events	The total number of events that have been sent by EPL but are still queued waiting to be sent to JMS.
r	Number of receivers	The current number of JMS receivers (both static and dynamic) on all JMS connections.
rx	Received messages	The total number of messages received from JMS, including messages received but not yet mapped to Apama events and added to the input queue of each public context. Includes events received by dynamic receivers that have now been removed, but does not include events received before the correlator was restarted, nor does it include any <code>JMSReceiverFlowControlMarker</code> events enqueued when the <code>receiverFlowControl</code> is enabled.
rRate	Received throughput rate	The total number of JMS messages received per second across all receivers, calculated over the interval since the last status line (typically 5 seconds).
rWindow	Receiver flow control window size	If <code>receiverFlowControl</code> is disabled for all receivers, this has the special value "-1". If any receivers have flow control enabled, <code>rWindow</code> gives a measure of the number of events that can be received before all flow controlled receivers will block, calculated as the sum of all the non-negative receiver window sizes. Note that even if this value is greater than 0 there could still be one or more receivers which have exhausted their own windows and are blocked, so consider

Field	Full name	Description
		enabled <code>logDetailedStatus</code> if <i>per-receiver</i> flow control diagnostics are required.
<code>rRedel</code>	Redelivered messages	The total number of JMS messages received with the <code>JMSRedelivered</code> flag set to true, indicating they are in-doubt and may have already been delivered in the past. For many JMS providers this flag is not always set reliably/consistently, but it does at least provide an indication of whether redeliveries may be taking place.
<code>rMaxDeliverySecs</code>	Maximum delivery time	<p>The highest time taken by the JMS broker to deliver a message, based on the difference between the time when each message is received and the value of the <code>JMSTimestamp</code> message header field which indicates the time when it was sent. This is likely to be a low number during normal operation, but will rise during failure modes such as loss of network connectivity or machine crashes as the JMS broker attempts to redeliver messages.</p> <p>This value is useful for understanding the redelivery behavior of the JMS provider in use and for choosing a sensible time expiry window if <code>EXACTLY_ONCE</code> duplicate detection is being used (see <code>dupDetectionExpiryTimeSecs</code> property). A high <code>rMaxDeliverySecs</code> value during testing may indicate that messages remaining on a JMS queue or durable topic from a previous test run may be interfering with the current test run. Note that any difference in the system time on the sending and receiving hosts will add an error to this value, which can result in negative values.</p>
<code>rDupsDet</code>	Duplicate messages detected	The total number of duplicate messages detected by <code>EXACTLY_ONCE</code> receivers and suppressed because their <code>uniqueMessageId</code> was already present in the duplicate detector. Does <i>not</i> include dynamic receivers that have now been removed.
<code>rDupIds</code>	Duplicate ids in memory	The total number of <code>uniqueMessageIds</code> being kept in memory for duplicate detection purposes by <code>EXACTLY_ONCE</code> reliable receivers. This is the total of the size of all per-message-source fixed-size expiry queues plus the unbounded time-based expiry queue. If this becomes too large it is possible the correlator could run out of memory.

Field	Full name	Description
connErr	Connection errors	The total number of times a valid JMS connection has gone down. Note that this tracks errors in existing connections and does not include repeated failures to establish a connection.
jvmUsedMB	JVM used memory	The amount of memory used by the JVM in Megabytes (heap plus non-heap), which can be compared with the maximum memory size provided for the JVM to check how much spare memory there is and ensure that the correlator is not close to running out of memory. This is particularly useful for checking peak memory consumption, such as testing when any <code>EXACTLY_ONCE</code> duplicate detectors are fully populated with the maximum likely number of <code>uniqueMessageIds</code> and any JMON applications are running in the same correlator are also near the maximum memory they are likely to use. Note that the memory usage figure reported by the JVM includes both live objects and <i>objects waiting to be garbage collected</i> , so inevitably this will go up and down a certain amount as garbage collections occur.
...	onApplication Initializedindicator	The suffix <code><waiting for onApplicationInitialized></code> will be added to the status lines if the EPL application has not yet called <code>jms.onApplicationInitialized()</code> as a reminder that status or JMS message events cannot be passed into the correlator until this action is invoked.

Detailed JMS status lines

If the `logDetailedStatus` property in an Apama application that uses correlator-integrated messaging for JMS is set to true in the `JmsSenderSettings` or `JmsReceiverSettings` configuration object, then additional lines will also be logged for each sender and receiver and their parent connections, for example.

```
INFO [19276] - Correlator Status: sm=2 nctx=1 ls=4 rq=0 eq=0 iq=0 oq=0
           rx=8 tx=6 rt=0 nc=1 vm=252372 runq=0
INFO [19276:Status] - JMS Status: s=1 tx=6 sRate=0 sOutst=0 r=2 rx=4
           rRate=0 rWindow=1500 rRedel=0
           rMaxDeliverySecs=0.0 rDupsDet=1 rDupIds=3
           connErr=0 jvmMB=67
INFO [19276:Status] - JMSConnection myConnection: s=1 r=2 connErr=0
           sessionsCreated=3
INFO [19276:Status] - JMSSEnder myConnection-default-sender: tx=6
           sRate=0 sOutst=0 msgErrors=2
INFO [19276:Status] - JMSReceiver myConnection-receiver-SampleQ2:
           rx=4 rRate=0 rWindow=1500 rRedel=0
           rMaxDeliverySecs=0.0 msgErrors=1 rDupsDet=1
           perSourceDupIds=3 timeExpiryDupIds=0
INFO [19276:Status] - JMSReceiver myConnection-receiver-SampleT2:
           rx=0 rRate=0 rWindow=-1 rRedel=0
```

```
rMaxDeliverySecs=0.0 msgErrors=0 rDupsDet=0
perSourceDupIds=0 timeExpiryDupIds=0
```

The JMS connector-specific status lines contain:

Table 2. JMS connector status log fields

Field	Full name	Description
s	Number of senders	The current number of JMS senders (both static and dynamic) on all JMS connections.
r	Number of receivers	The current number of JMS receivers (both static and dynamic) on all JMS connections
connErr	Connection errors	The total number of times this JMS connection has gone down, Note that this tracks errors in existing connections and does not include repeated failures to establish a connection.
sessionsCreated	Send/receive sessions created	The total number of JMS Sessions that have been created during the lifetime of this JMS connection. In normal operation a single session is created for each sender or receiver, but if a connection failure or serious sender/receiver error occurs, a new session will be created, causing this counter to be incremented. Note, this counter is not decremented when the previous session is closed.

The JMS sender-specific status lines contain:

Table 3. JMS sender status log fields

Field	Full name	Description
tx	Sent events	The number of events that were sent to this sender's channel and have been fully processed (either sent to JMS, or exhausted the maximum failure retry limit).
sRate	Send throughput rate	The number of events sent per second to this sender, calculated over the interval since the last status line (typically 5 seconds).
sOutst	Outstanding sent events	The number of events that have been sent by EPL but are still queued waiting to be passed to JMS by this sender.
sMsgErrors	Per-message error count	The number of Apama events that could not be sent to JMS due to some error, typically a mapping failure or destination not found error. See the log file for <code>WARN</code> and <code>ERROR</code> messages that will provide more details.

The JMS receiver-specific status lines contain:

Table 4. JMS receiver status log fields

Field	Full name	Description
<code>rx</code>	Received messages	The number of messages received from JMS, including messages received but not yet mapped to Apama events and added to the input queue of each public context.
<code>rRate</code>	Receive throughput rate	The number of JMS messages received per second by this receiver, calculated over the interval since the last status line (typically 5 seconds).
<code>rWindow</code>	Flow control window size	If <code>receiverFlowControl</code> is disabled this has the special value "-1". If it is enabled, this gives the current flow control window size, that is the number of successfully mapped events that can still be received before the receiver will block. A zero value indicates that the flow control window has been exhausted and the application should call <code>JMSReceiverFlowControlMarker.updateFlowControlWindow()</code> to unblock the receiver. A negative value indicates that the window has been updated to a negative value, which has the same effect as a window of 0.
<code>rRedel</code>	Redelivered messages	The number of JMS messages received with the <code>JMSRedelivered</code> flag set to true.
<code>rMaxDeliverySecs</code>	Maximum delivery time	The highest time taken by the JMS broker to deliver a message to this receiver, based on the difference between the time when each message is received and the value of the <code>JMSTimestamp</code> message header field which indicates the time when it was sent.
<code>rMsgErrors</code>	Per-message error count	The number of received JMS messages that could not be passed to the Apama application due to some error, typically a mapping failure. See the log file for <code>WARN</code> and <code>ERROR</code> messages that will provide more details.
<code>rDupsDet</code>	Duplicate messages detected	The number of duplicate messages detected by this <code>EXACTLY_ONCE</code> receiver and suppressed because their <code>uniqueMessageId</code> was already present in this receiver's duplicate detector. Only displayed for <code>EXACTLY_ONCE</code> receivers.
<code>perSourceDupIds</code>	Per-source duplicate ids in memory	The total number of <code>uniqueMessageIds</code> being kept in memory for duplicate detection purposes by all per-message-source fixed-size expiry queues. Only displayed for <code>EXACTLY_ONCE</code> receivers.

Field	Full name	Description
<code>timeExpiryDupIds</code>	Time-based duplicate ids in memory	The total number of <code>uniqueMessageIds</code> being kept in memory for duplicate detection purposes by the unbounded time-based expiry <code>uniqueMessageId</code> queue.

Using Correlator-Integrated Messaging for JMS

JMS configuration reference

This section includes topics relating to the configuration for applications using Apama's correlator-integrated messaging for JMS. It covers configuration files, configuration objects, and the configuration properties that can be set when developing your applications.

Using Correlator-Integrated Messaging for JMS

Configuration files

The correlator-integrated messaging for JMS configuration consists of a set of XML files and `.properties` files.

A correlator that supports JMS has the following two files:

- `jms-global-spring.xml`
- `jms-mapping-spring.xml`

In addition, for each JMS connection added to the configuration, there will be an additional XML and `.properties` file:

- `connectionId-spring.xml`
- `connectionId-spring.properties`

When the correlator is started with the `--jmsConfig configDir` argument, it will load all XML files matching `*-spring.xml` in the specified configuration directory, and also all `*.properties` files in the same directory. (Note, the correlator will not start unless the specified directory contains at least one configuration file.)

Global configuration that is shared across all a correlator's connections is stored in `jms-global-spring.xml`, the rules for mapping between JMS messages and Apama events are stored in `jms-mapping-spring.xml`, and the `connectionId-spring.xml` files contain the configuration for each JMS broker connection added to the configuration. Each XML file can contain `${...}` property placeholders, whose values come from the `*.properties` files. This provides a way for the configuration to be defined once in the XML files, then customized for development, UAT, and different deployment scenarios by creating separate copies of the `.properties` files.

When using Apama Studio, all these files are generated automatically. A new `connectionId-spring.xml` and `connectionId-spring.properties` file is created when the **JMS Configuration Wizard** is used to add a JMS connection, and the most commonly used settings can be changed at any time using the correlator-integrated messaging for JMS instance editor window (which rewrites the `.properties` file whenever the configuration is changed). Apama Studio makes it easy to set and edit basic configuration options with the adapter editor. In addition, the `jms-global-spring.xml` and `connectionId-`

spring.xml files can be edited manually in Apama Studio to customize more advanced configuration aspects such as advanced sender/receiver settings, logging of messages, etc. To edit the XML, open the correlator-integrated messaging for JMS editor and click on the Advanced tab; the various configuration files can be accessed through the hyperlinks on this tab. Once the editor for an XML file has been opened, you can switch between the Design and Source views using the tabs at the bottom of the editor window.

Note that unlike the other XML files, Apama does not support manual editing of the `jms-mapping-spring.xml` file in this release, and the format of that file may change at any time without notice. We recommend using Apama Studio for all mapping configuration tasks.

JMS configuration reference

XML configuration file format

The correlator-integrated messaging for JMS configuration files use the Spring XML file format, which provides an open-source framework for flexibly wiring together the different parts of an application, each of which is represented by a *bean*. Each bean is configured with an associated set of *properties*, and has a unique identifier which can be specified using the `id` attribute.

For example:

```
<bean id="globalReceiverSettings"
      class="com.apama.correlator.jms.config.JmsReceiverSettings">
  <property name="logJmsMessages" value="true"/>
  <property name="logProductMessages" value="false"/>
</bean>
```

Or:

```
<jms:receiver id="myReceiver1">
  <property name="destination" value="queue:SampleQ1"/>
</jms:receiver>
```

It is not necessary to have a detailed knowledge of Spring to configure correlator-integrated messaging for JMS, but some customers may wish to explore the [Spring 3.0.5](#) documentation to obtain a deeper understanding of what is going on and to leverage some of the more advanced functionality that Spring provides.

The key beans making up the Apama configuration are `jms:connection`, `jms:receiver` and `jms:sender`, plus additional beans that are usually stored in the `jms-global-spring.xml` file and shared across all configured connections, such as the reliable receive database and the advanced sender/receiver settings beans.

Bean ids

All receiver, sender, connection, and other configuration beans have an `id` attribute that specifies a unique identifier. These identifiers are used in log messages, when monitoring status from EPL applications, and, when necessary, for references between different Spring beans in the XML configuration files. It is important that all identifiers are completely unique, for example the same `id` cannot be used for senders and receivers in different connections, or for both a sender and a receiver, even if they located in different XML files.

Setting property values

Most bean properties have primitive values (such as string, number, boolean) which are set like this:

```
<property name="propName" value="my value"/>
```

However, there are also a few properties that reference other beans, such as the `reliableReceiveDatabase` property on `.jms:connection` and the `receiverSettings` property on `.jms:receiver`. These property values can be set by specifying the id of a top-level bean like this (where it is assumed that `globalReceiverSettings` is the id of a `JmsReceiverSettings` bean):

```
<property name="receiverSettings" ref="globalReceiverSettings"/>
```

Any top-level bean may be referenced in this way, that is, any bean that is a child of the `<beans>` element and not nested inside another bean. Referencing a bean that is defined in a different configuration file is supported, and the `jdbc-global-spring.xml` file is intended as a convenient place to store top-level beans that should be shared across many different JMS connections.

Instead of referencing a shared bean, it is also possible to configure a bean property by creating an 'inner' configuration bean nested inside the property value like this:

```
<property name="receiverSettings">
  <bean class="com.apama.correlator.jms.config.JmsReceiverSettings">
    <property name="logJmsMessages" value="true"/>
  </bean>
</property>
```

(Note, advanced users may want to exploit Spring's property inheritance by using the `parent=` attribute on an inner bean to inherit most properties from a standard top-level bean while overriding some specific subset of properties or by type-based 'auto-wiring' - any non-primitive property of `.jms:connection/receiver/sender` for which no value is explicitly set will implicitly reference a top-level bean of the required type. This is how `.jms:connection` beans get a reference to the `reliableReceiveDatabase` and `defaultSender/ReceiverSettings` beans. Most configuration can just ignore this detail and use the automatically wired property values, and the bean representing the Apama-provided mapper, but if desired the defaults for individual connections/senders/receivers can be customized independently of each other by specifying the property values explicitly.)

Adding static senders and receivers

For simple cases where detailed configuration of receivers is not required, it is possible to configure static receivers using a simple semicolon-delimited list of JMS destinations, for example:

```
<property name="staticReceiverList"
  value="topic:MyTopic;jndi:/sample/some-jndiqueuename" />
```

The `staticReceiverList` bean property is represented by a placeholder in the `connectionId-spring.properties` file, and can be edited using Apama Studio.

For more advanced receiver configuration, it is necessary to edit the `connectionId-spring.xml` file manually, and provide a list of `.jms:receiver` beans as the value of the `staticReceivers` property:

```
<property name="staticReceivers">
  <list>
    <jms:receiver id="myReceiver1">
      <property name="destination"
        value="queue:SampleQ1"/>
    </jms:receiver>
    <jms:receiver id="myReceiver2">
      <property name="destination"
        value="jndi:/sample/my-jndi-topic-name"/>
      <property name="durableTopicSubscriptionName"
        value="MyTopicSubscription"/>
    </jms:receiver>
  </list>
</property>
```

Senders may be configured in the same way, for example:

```
<property name="staticSenders">
  <!-- each static sender results in a correlator channel
```



```

        called "jms:senderId" -->
<list>
  <jms:sender id="MyConnection-default-sender">
  </jms:sender>
  <jms:sender id="myReliableSender">
    <property name="senderReliability" value="EXACTLY_ONCE"/>
  </jms:sender>
  <jms:sender id="myUnreliableSender">
    <property name="senderReliability" value="BEST_EFFORT"/>
  </jms:sender>
</list>
</property>

```

If a sender list is not explicitly configured, a single sender with id `connectionId-default-sender` will be created.

JMS configuration reference

JXML configuration bean reference

This topic lists the various configuration objects (beans) and the supported properties for each bean.

See also ["Using custom EL mapping extensions" on page 91](#).

jms:connection

This bean defines the information needed to establish a JMS Connection to a single JMS broker instance. Its required properties are: `connectionFactory` or `connectionFactory.jndiName`, and (if JNDI is used to locate the connection factory), `jndiContext`.

Example:

```

<jms:connection id="MyConnection">
  <property name="staticReceiverList"
    value="${staticReceiverList.MyConnection}" />
  <property name="defaultReceiverReliability"
    value="${defaultReceiverReliability.MyConnection}"/>
  <property name="defaultSenderReliability"
    value="${defaultSenderReliability.MyConnection}"/>
  <property name="connectionFactory.jndiName"
    value="${connectionFactory.jndiName.MyConnection}" />
  <property name="jndiContext.environment">
    <value>
      ${jndiContext.environment.MyConnection}
    </value>
  </property>
  <property name="connectionAuthentication.username"
    value="${connectionAuthentication.username.MyConnection}" />
  <property name="connectionAuthentication.password"
    value="${connectionAuthentication.password.MyConnection}" />
</jms:connection>

```

Supported properties:

- `connectionFactory.jndiName` - the JNDI lookup name for the `ConnectionFactory` object that should be used for this `jms:connection`.
- `connectionFactory` - a JMS provider bean that implements the JMS `ConnectionFactory` interface, if the `ConnectionFactory` is to be instantiated directly by the Spring framework (rather than using JNDI to lookup the `ConnectionFactory`). The bean value that is provided will usually require properties and/or constructor arguments to be specified in order to fully initialize it.

- `connectionAuthentication.username` - the name of the user/principal to be used for the JMS connection (note that this is often different from the username/password needed to login to the JNDI server, which is part of the JNDI environment configuration). Default value is "".
- `connectionAuthentication.password` - the password/credentials to be used for the JMS connection.
- `jndiContext.environment` - the set of properties that specify the environment for initializing access to the JNDI store. Typically includes some standard JNDI keys such as `java.naming.factory.initial`, `java.naming.provider.url`, `java.naming.security.principal` and `java.naming.security.credentials`, and maybe also some provider-specific keys. The usual way to specify a properties map value is key=value entries delimited by newlines and surrounded by the `<value>` element, e.g. `<property name="jndiContext.environment"><value>...</value></property>`.
- `clientId` - the JMS client ID which uniquely identifies each connected JMS client to the broker. Default value is "" although some JMS providers may require this to be set, especially when using durable topics.
- `defaultReceiverReliability` - the Apama reliability mode to use for all this connection's receivers unless overridden on a per-receiver basis; valid values are `BEST_EFFORT`, `AT_LEAST_ONCE`, `EXACTLY_ONCE`, `APP_CONTROLLED`. Default value is `BEST_EFFORT`.
- `defaultSenderReliability` - the Apama reliability mode to use for all this connection's senders unless overridden on a per-sender basis; valid values are `BEST_EFFORT`, `AT_LEAST_ONCE`, `EXACTLY_ONCE`. Default value is `BEST_EFFORT`.
- `staticReceiverList` - a list of destinations to receive from, delimited by semi-colons. Each destination must begin with "queue:", "topic:" or "jndi:". This property provides a simple way to add static receivers when the more advanced configuration options provided by the `staticReceivers` property are not needed. `staticReceiverList` receivers are always added in addition to any receivers specified by `staticReceivers`. The `staticReceiverList` property cannot contain duplicate destination entries (see the `staticReceivers` property if this is required). Default value is "".
- `staticReceivers` - a list of `jms:receiver` beans specifying JMS receivers to create for this connection. The `jms:receiver` elements are wrapped in a `<list>` element, for example, `<property name="staticReceivers"><list>...</list></property>`. Default value is an empty list.
- `staticSenders` - a list of sender beans specifying JMS senders to create for this connection. The `jms:sender` elements are wrapped in a `<list>` element, for example, `<property name="staticSenders"><list>...</list></property>`. Default value is a single sender called "default".
- `defaultReceiverSettings` (advanced users only) - a reference to a `JmsReceiverSettings` bean, which provides access to advanced settings that are usually shared across all configured receivers for this connection. Default value is a reference to the `JmsReceiverSettings` bean instance defined in the `jms-global-spring.xml` file (this uses Spring's `byType` auto-wiring; if multiple top-level `JmsReceiverSettings` beans exist in the configuration then the reference must be specified explicitly in each `jms:connection`).
- `defaultSenderSettings` (advanced users only) - a reference to a `JmsSenderSettings` bean, which provides access to advanced settings that are usually shared across all configured senders for this connection. Default value is a reference to the `JmsSenderSettings` bean instance defined in the `jms-global-spring.xml` file (this uses Spring's `byType` auto-wiring; if multiple top-level `JmsSenderSettings` beans exist in the configuration then the reference must be specified explicitly in each `jms:connection`).

- `reliableReceiveDatabase` (advanced users only) - a reference to a `ReliableReceiveDatabase` bean, which is required for implementing the `AT_LEAST_ONCE` or `EXACTLY_ONCE` reliability modes for any receivers added to this `jms:connection`. Default value is a reference to the single `DefaultReliableReceiveDatabase` bean instance defined in the `jms-global-spring.xml` file (this uses Spring's `byType` auto-wiring; if multiple top-level `DefaultReliableReceiveDatabase` beans exist in the configuration then the reference must be specified explicitly in each `jms:connection`). The only reason for changing this property would be to use separate databases or different `jms:connection` which could in some advanced cases provide a performance advanced, depending on the application architecture and the configuration of the `jms:connection` and disk hardware.
- `connectionRetryIntervalMillis` - Specifies how long to wait between attempts to establish the JMS connection. Default value is 1000 ms.
- `receiverMapper` - points to a custom mapper. Typically you use a `ref="beanid"` attribute to do this. If this property is not specified, the default is that the connection uses the Apama-provided mapper, assuming it is the only mapper defined in the configuration.
- `senderMapper` - points to a custom mapper. Typically you use a `ref="beanid"` attribute to do this. If this property is not specified, the default is that the connection uses the Apama-provided mapper, assuming it is the only mapper defined in the configuration.

jms:receiver

This bean defines a single-threaded context for receiving events from a single JMS destination. Its only required property is `destination`.

Example:

```
<jms:receiver id="myReceiver">
  <property name="destination" value="topic:SampleT1"/>
</jms:receiver>
```

Supported properties:

- `destination` - the JMS queue or topic to receive from. Must begin with the prefix `"queue:"`, `"topic:"` or `"jndi:"`. A JMS queue or topic name can be specified with the `"queue:"` or `"topic:"` prefixes, or if the queue or topic should be looked up using a JNDI name then the `"jndi:"` prefix should be used instead.
- `receiverReliability` - the Apama reliability mode to use when JMS messages are received; valid values are `BEST_EFFORT`, `AT_LEAST_ONCE`, `EXACTLY_ONCE`, `APP_CONTROLLED`. Default value is provided by the parent `jms:connection`'s `defaultReceiverReliability` setting.
- `durableTopicSubscriptionName` - if specified, a durable topic subscriber will be created (instead of a queue/topic consumer), and registered with the specified subscription name. Default value is `""`, which means do not create a durable topic subscription. Note that some providers will require the connection's `clientId` property to be specified when using durable topics.
- `messageSelector` - a JMS message selector string that will be used by the JMS provider to filter the messages pulled from the queue or topic by this receiver, based on the header and/or property values of the messages. Default value is `""` which means that no selector is in operation and all messages will be received. Message selectors can be used to partition the messages received by multiple receivers on the same queue or durable topic. The JMS API documentation describes the syntax of message selectors in detail; a simple example selector is `"JMSType = 'car' AND color = 'blue' AND weight > 2500"`.

- `noLocal` - an advanced JMS consumer parameter that prevents a connection's receivers from seeing messages that were sent on the same (local) JMS connection. Default value is "false".
- `dupDetectionDomainId` - an advanced Apama setting for overriding the way receivers are grouped together for duplicate detection purposes when using `EXACTLY_ONCE` receive mode. Set this to the same string value for a set of receivers to request detection of duplicate `uniqueMessageIds` across all the messages from those receivers. Default value is "`<connectionId>:<destination>`" (that is, look for duplicates across all receivers for the same queue/topic only within the same `jms:connection`).
- `receiverSettings` - a reference to a `JmsReceiverSettings` bean, which provides access to advanced settings that are usually shared across all configured receivers. Default value is provided by the parent connection's `defaultReceiverSettings` property (which is usually a reference to the `JmsReceiverSettings` bean instance defined in the `jms-global-spring.xml` file).

jms:sender

This bean defines a single-threaded context for sending events to a JMS destination, and results in the creation of a correlator output channel called `jms:senderId`. It has no required properties.

Example:

```
<jms:sender id="mySender">
  <property name="senderReliability" value="BEST_EFFORT"/>
  <property name="messageDeliveryMode" value="PERSISTENT"/>
  <property name="senderSettings" ref="globalSenderSettings"/>
</jms:sender>
```

Supported properties:

- `senderReliability` - the Apama reliability mode to use when events are sent to JMS. Valid values are `BEST_EFFORT`, `AT_LEAST_ONCE`, `EXACTLY_ONCE`. Default value is provided by the parent `jms:connection`'s `defaultSenderReliability` setting.
- `messageDeliveryMode` - this property applies to a sender that is using the `BEST_EFFORT` reliability mode to deliver messages to a JMS broker. The default is the JMS `NON_PERSISTENT` delivery mode. You can change the value of this property to `PERSISTENT` mode. While `PERSISTENT` mode is slower, it causes the JMS broker to write messages to disk to protect against crashes of the JMS broker node. The only possible values for the `messageDeliveryMode` property are `PERSISTENT` and `NON_PERSISTENT`. This property is ignored for other reliable senders.
- `senderSettings` - a reference to a `JmsSenderSettings` bean, which provides access to advanced settings that are usually shared across all configured senders. Default value is provided by the parent connection's `defaultSenderSettings` property (which is usually a reference to the `JmsSenderSettings` bean instance defined in the `jms-global-spring.xml` file).

ReliableReceiveDatabase

This bean defines a database used by Apama to implement reliable receiving. It has no required properties. Typically all connections in a correlator will share the same receive database; if the correlator is not started with the `-P` (persistence enabled) flag, this bean will be ignored.

Example:

```
<bean id="myReliableReceiveDatabase"
  class="com.apama.correlator.jms.config.DefaultReliableReceiveDatabase">
  <property name="storePath" value="jms/my-receive.db"/>
  <!-- either absolute path, or path relative to correlator store location -->
</bean>
```

Supported property:

- `storePath` - the path where the message store database should be created. Default value is `jms-receive-persistence.db`. Use an absolute path, or a path relative to the store location specified for use by the correlator state persistence store on the correlator command line.

JmsSenderSettings

This bean defines advanced settings for message senders. It has no required properties. Typically all senders in all connections will share the same `JmsSenderSettings` bean, but it is also possible to use different settings for individual senders.

Example:

```
<bean id="globalSenderSettings"
      class="com.apama.correlator.jms.config.JmsSenderSettings">
  <property name="logJmsMessages" value="false"/>
  <property name="logJmsMessageBodies" value="false"/>
  <property name="logProductMessages" value="false"/>
</bean>
```

Supported properties

- `logJmsMessages` - if true, log information about all JMS messages that are sent (but not the entire body) at `INFO` level. Default value is "false".
- `logJmsMessageBodies` - if true, log information about all JMS messages that are sent, including the entire message body at `INFO` level. Default value is "false".
- `logProductMessages` - if true, log information about all Apama events that are sent at `INFO` level. Default value is "false".
- `logDetailedStatus` - Enables logging of a dedicated `INFO` status line for each sender and a summary line for each parent connection. The default value is "false" (detailed logging is disabled), which results in a single summary line covering all senders and connections.
- `logPerformanceBreakdown` - Enables periodic logging of a detailed breakdown of how much time is being taken by the different stages of mapping, sending, and disk operations for each sender. By default, the messages are logged every minute at the `INFO` level. The interval can be changed if desired. The default is false, and Apama recommends disabling this setting in production environments to prevent the gathering of the performance information from reducing performance.
- `logPerformanceBreakdownIntervalSecs` - Specifies the interval in seconds over which performance throughput and timings information will be gathered and logged. Default is 60.
- `sessionRetryIntervalMillis` - Specifies how long to wait between attempts to create a valid JMS session and producer for this sender either after a serious error while using the previous session or after a previous failed attempt to create the session. However, if the underlying JMS connection has failed the `connectionRetryIntervalMillis` is used instead. Default value is 1000 ms

JmsReceiverSettings

This bean defines advanced settings for message receivers. it has no required properties. Typically all receivers in all connections will share the same `JmsReceiverSettings` bean, but it is also possible to use different settings for individual receivers.

Example:

```
<bean id="globalReceiverSettings"
      class="com.apama.correlator.jms.config.JmsReceiverSettings">
  <property name="dupDetectionPerSourceExpiryWindowSize" value="2000"/>
  <property name="dupDetectionExpiryTimeSecs" value="120"/>
</bean>
```

```

    <property name="logJmsMessages" value="false"/>
    <property name="logJmsMessageBodies" value="false"/>
    <property name="logProductMessages" value="false"/>
</bean>

```

Supported properties:

- `logJmsMessages` - if true, log information about all JMS messages that are received (but not the entire body) at `INFO` level. Default value is "false".
- `logJmsMessageBodies` - if true, log information about all JMS messages that are received, including the entire message body at `INFO` level. Default value is "false".
- `logProductMessages` - if true, log information about all Apama events that are received at `INFO` level. Default value is "false".
- `logDetailedStatus` - Enables logging of a dedicated `INFO` status line for each receiver and a summary line for each parent connection. The default value is "false" (detailed logging is disabled), which results in a single summary line covering all receivers and connections.
- `logPerformanceBreakdown` - Enables periodic logging of a detailed breakdown of how much time is being taken by the different stages of mapping, receiving, and disk operations for each receiver. By default, the messages are logged every minute at the `INFO` level. The interval can be changed if desired. The default is false, and Apama recommends disabling this setting in production environments to prevent the gathering of the performance information from reducing performance.
- `logPerformanceBreakdownIntervalSecs` - Specifies the interval in seconds over which performance throughput and timings information will be gathered and logged. Default is 60.
- `dupDetectionPerSourceExpiryWindowSize` - used for `EXACTLY_ONCE` receiving, and specifies the number of messages that will be kept in each duplicate detection domain per `messageSourceId` (if `messageSourceId` is set on each message by the upstream system - messages without a `messageSourceId` will all be grouped together into one window for the entire `dupDetectionDomainId`). Default value is "2000". It can be set to 0 to disable the fixed-size per-sender expiry window.
- `dupDetectionExpiryTimeSecs` - used for `EXACTLY_ONCE` receiving, and specifies the time for which `uniqueMessageIds` will be remembered before they expire. Default value is "120". It can be set to 0 to disable the time-based expiry window.
- `maxExtraMappingThreads` - Specifies the number of additional (non-receiver) threads to use for mapping received JMS messages to Apama events. The default value is 0. Using a value of 1 means all mapping is performed on a separate thread to the thread receiving messages from the bus; a value greater than 1 provides additional mapping parallelism. This setting cannot be used if `maxBatchSize` has been set to 1. Using multiple separate threads for mapping may improve performance in situations where mapping of an individual message is a heavyweight operation (for example, for complex XML messages) and where adding separate receivers is not desired (because they involve the overhead of additional JMS sessions and reduced ordering guarantees). Note that strictly speaking JMS providers do not have to support multi-threaded construction of JMS messages (since all JMS objects associated with a receiver's Session are meant to be dedicated to a single thread), so although in practice it is likely to be safe, it is important to verify that this setting does not trigger any unexpected errors in the JMS provider being used.

The order in which mapped events are added to the correlator input queue (of each public context) is not changed by the use of extra mapping threads, as messages from all mapping threads on a given receiver are put back into the original receive order at the end of processing each receive batch.

- `sessionRetryIntervalMillis` - Specifies how long to wait between attempts to create a valid JMS session and consumer for this receiver either after a serious error while using the previous session, or after a previous failed attempt to create the session. However, if the underlying JMS connection has failed the `connectionRetryIntervalMillis` is used instead). Default value is 1000 ms.
- `receiverFlowControl` - Specifies whether application-controlled flow is enabled for each receiver. When set to true application-controlled flow control is enabled for each receiver, by listening for the `com.apama.correlator.jms.JMSReceiverFlowControlMarker` event and responding by calling the `updateFlowControlWindow()` action as appropriate. Default value is `false`.

JMS configuration reference

Advanced configuration bean properties

The following properties are *advanced* tuning parameters, for use only when really necessary to improve performance or work around a JMS provider bug. Since these are advanced properties, it is possible that the default values may change in any future release or that new tuning parameters may be added that could alter the semantics of the existing ones, so be sure to carefully check the *Release Notes* when upgrading, if you use any of these properties.

JMSSenderSettings

- `maxBatchSize` - The maximum (and target) number of events to be batched together for sending inside a JMS local (non-XA) transaction (which improves performance on many JMS providers). The `maxBatchSize` indicates the target number of events that will normally be sent in a single batch unless the `maxBatchIntervalMillis` timeout expires first. The `maxBatchSize` must be greater than 0 and the special value of 1 is used to indicate that a non-transacted JMS session should be used instead. Note that the same batching algorithm and parameters are used for both reliable and non-reliable senders. The default value in this release is 500.
- `maxBatchIntervalMillis` - The maximum time a sender will wait for more events on its channel (and for reliable senders, also included in a correlator persist cycle) before timing out and sending the events ready to be sent in the batch, even if the batch size is less than `maxBatchSize`. The default value in this release is 500 ms.

JMSReceiverSettings

- `receiveTimeoutMillis` - The timeout that will be passed to the JMS provider's `MessageConsumer.receive(timeout)` method call to indicate the maximum time it should block for when receiving the next message before returning control to the correlator. The default value in this release is 300 ms. Some providers may require this timeout to be increased to ensure that messages can be successfully received in high-latency network conditions, although well-behaved providers should always work correctly with the default value. Reducing this timeout may improve receive latency (due to reduced time waiting for the batch to complete) on some providers; although note that many JMS providers do not strictly obey the timeout specified here so the real time spent blocking while no messages are available may be significantly higher.
- `maxBatchSize` - The maximum (and target) number of JMS messages to be received before the batch is committed to the receive-side database (if receiver is using `AT_LEAST_ONCE` or `EXACTLY_ONCE` reliability mode) and then added to the input queues of public contexts and acknowledged to the JMS broker (whether reliable or not). The `maxBatchSize` indicates the target number of messages that will normally be received in a single batch unless the `maxBatchIntervalMillis` timeout expires first. The `maxBatchSize` must be greater than 0 and for `BEST_EFFORT` (non-reliable) receivers the special value of 1 is used to indicate that an `AUTO_ACKNOWLEDGE` session will be used instead of the

default `CLIENT_ACKNOWLEDGE` session (though for reliable receivers `CLIENT_ACKNOWLEDGE` is always used even if `maxBatchSize` is 1). The default value in this release is 1000.

The batch size becomes particularly important when using the `APP_CONTROLLED` reliability mode. In this case, you might need to tweak the batch size to improve throughput based on how long the application takes between suspending and acknowledging each batch of messages.

- `maxBatchIntervalMillis` - the maximum time a receiver will attempt to wait for more messages to be received (and mapped) before timing out and processing the messages already received as a single batch, even if the size of that batch is less than `maxBatchSize`. The default value in this release is 500 ms. Note that in practice, when no messages are available, many JMS providers seem to block for longer than the specified `receiveTimeoutMillis` before returning, which may lead to the true maximum batch interval being significantly longer than the value specified here.

[!XML configuration bean reference](#)

Designing and implementing applications for correlator-integrated messaging for JMS

This section describes guidelines for designing and implementing applications that make use of correlator-integrated messaging for JMS.

[Using Correlator-Integrated Messaging for JMS](#)

Using correlator persistence with correlator-integrated messaging for JMS

Correlator-integrated messaging for JMS can be used with or without the correlator's state persistence feature. In a persistent correlator, all reliability modes can be used (both reliable and unreliable messaging), but in a non-persistent correlator only `BEST_EFFORT` (unreliable) messaging is supported, and attempts to add senders or receivers using any other reliability mode will result in an error.

In a persistent correlator, information about all senders and receivers is always stored in the recovery datastore. This includes unreliable ones as well as reliable ones and statically defined ones as well as dynamic ones. This means that persistent Apama applications never need to re-create previously-added JMS senders and receivers after recovery. This will happen automatically, even for `BEST_EFFORT` (unreliable) senders and receivers. For reliable senders and receivers no messages or duplicate detection information will be lost after a crash or restart.

Because sender and receiver information is stored in the database, it is not permitted to shut down a persistent correlator and then make changes such as removing static senders and receivers from the configuration file before restarting. If the ability to remove senders and receivers is required, they must be added dynamically using EPL rather than from the configuration file. However, you can add new senders and receivers to the configuration files between restarts, provided the identifiers do not clash with any previously defined static or dynamic sender or receiver.

It is never possible to change the configuration of dynamic senders or receivers after they are created. For static senders and receivers this is also mostly prohibited, with the exception that the destination of a static receiver defined explicitly in the configuration file can be changed between restarts of the correlator (provided the `receiverId` and `dupDetectionDomainId` remain the same).

To retain maximum flexibility, Apama recommends that customers follow the industry standard practice of using JNDI names for queues and topics. This means that it is always possible to configure any necessary redirections to allow the same logical (JNDI) name to be used in different deployment environments, such as production and deployment (for dynamic as well as static receivers).

There is no restriction on changing the connection factory or JNDI server details between restarts of a persistent correlator. By using the same JNDI names (or if necessary, queue and topic names) in all environments, but different isolated JMS and JNDI servers for production and testing, it is possible to avoid unintended interactions between the production and test environments. At the same time, this keeps the two configurations very similar and allows production datastores to be examined in the test environment if necessary.

Designing and implementing applications for correlator-integrated messaging for JMS

How reliable JMS sending integrates with correlator persistence

This topic describes the details of how JMS sending integrates with correlator persistence. This information is intended for advanced users.

When sending JMS messages in a persistent correlator, all events sent to a JMS sender are queued inside the correlator until the next persist cycle begins. The events cannot be passed to JMS until the EPL application state changes that caused them to be sent have been persisted, otherwise the downstream receiver might see an inconsistent set of events in the case of a failure and recovery. Note that to avoid potentially inconsistent output in the event of a failure, this is true not only for reliable `AT_LEAST_ONCE`, `EXACTLY_ONCE` and `APP_CONTROLLED` messages but also for `BEST_EFFORT` messages in a persistent correlator. The only differences between `BEST_EFFORT` reliable sending mode and the `AT_LEAST_ONCE` or `EXACTLY_ONCE` reliable sending mode (other than whether the messages use the JMS `PERSISTENT` delivery mode flag), is that for `AT_LEAST_ONCE` or `EXACTLY_ONCE` the messages are guaranteed to remain on disk until they have been successfully sent to the JMS broker, whereas for `BEST_EFFORT` this is not the case.

Unique identifiers are generated and assigned to each message when they are sent, and persisted with the events to allow downstream receivers to perform `EXACTLY_ONCE` duplicate detection if desired (note, this assumes the `uniqueMessageId` is mapped into the JMS message in some fashion).

Once the next persist cycle has completed and both the events and the application state that caused them has been committed to disk, the events can be sent to JMS. After messages have been successfully sent to the JMS broker they are lazily removed from the correlator's in-memory and on-disk data structures. The latency of sent messages is therefore dependent on the time taken for the correlator to perform a persist cycle (including the persist interval, the time required to take a snapshot the correlator's state and commit it to disk, and any retries if the correlator cannot take a snapshot for the state immediately), plus any time spent waiting to fill the batch of events to be sent (although this is usually relatively small). Note that if a message send fails and it is not due to the JMS connection being lost then after a small number of retries it will be dropped with an `ERROR` message in the log. If a send fails because the connection is down, the correlator simply waits for it to come up again in all cases.

Using correlator persistence with correlator-integrated messaging for JMS

How reliable JMS receiving integrates with correlator persistence

This topic, for advanced users, describes how JMS receiving integrates with correlator persistence.

When receiving in `AT_LEAST_ONCE` or `EXACTLY_ONCE` mode, messages are taken from the JMS queue or topic in batches (using `JMS_CLIENT_ACKNOWLEDGE` mode). The resulting Apama events are persisted in the reliable receive datastore (which is separate from the correlator's recovery datastore) and then acknowledged back to JMS before the next batch of messages is received. After a batch of events finishes being asynchronously committed to the datastore, it is added to the input queue of each context. When the correlator next completes a persist cycle, all events that had at least been added to the input queue by the beginning of the persist cycle have been (or will be) reliably passed to the application. This means that in `AT_LEAST_ONCE` mode they can be removed from the receive datastore immediately.

If `EXACTLY_ONCE` is being used and the event was mapped with a non-empty `uniqueMessageId` from the JMS message, the `uniqueMessageId` and other metadata are stored both in memory and in the on-disk reliable datastore, and are kept there until the associated `uniqueMessageId` is expired from the duplicate detector. Note however, that as an optimization, because the persisted event strings are no longer needed once the event has been included in the correlator state database, any particularly long event strings may become null in the database. The latency of received messages is therefore dependent on the time spent waiting for other messages to be received to fill the batch, and the time taken to commit the batch to the receive datastore.

When a persistent correlator is restarted and recovers its state from the recovery datastore, no new JMS messages will be received from the broker until recovery is complete. Specifically, until the correlator calls the `onConcludeRecovery()` action on all EPL monitors that have defined this action. It is possible that EPL monitors will see a small number of JMS messages that were received and added to the input queue before the correlator was restarted. To be safe, any required listeners in non-persistent monitors should be set up in `onBeginRecovery()`.

Since a batch of messages is acknowledged to the JMS broker as soon as they have been written to the Apama reliable receive datastore, there is no relationship between JMS message acknowledgment to the broker and when the correlator begins or completes a correlator state datastore persistence cycle. The maximum number of messages that may be received from the JMS broker but not yet acknowledged is limited by the configured `maxBatchSize` (typically this is 1000 messages).

[Using correlator persistence with correlator-integrated messaging for JMS](#)

Sending and receiving reliably without correlator persistence

Apama applications that receive JMS messages can prevent message loss without using correlator persistence by controlling when the application acknowledges the received messages. See ["Receiving messages with APP_CONTROLLED acknowledgements" on page 209](#).

Apama applications that use JMS senders with `BEST_EFFORT` reliability can prevent message loss without using correlator persistence by waiting for acknowledgements that all messages sent to a JMS sender context have been sent to the JMS broker. See ["Sending messages reliably with application flushing notifications" on page 211](#).

[Designing and implementing applications for correlator-integrated messaging for JMS](#)

Receiving messages with APP_CONTROLLED acknowledgements

Apama applications that receive JMS messages can prevent message loss without using correlator persistence by controlling when the application acknowledges the received messages. To do this, use `APP_CONTROLLED` reliability mode. With `APP_CONTROLLED` reliability mode, an application can tie the sending of the JMS acknowledgement to application-defined strategies for preserving the effect of the messages. For example, an application might need to ensure JMS messages are not

acknowledged to the broker until any output resulting from them has been written to a database, a distributed MemoryStore, a downstream JMS destination, or a connected correlator.

An alternative to using `APP_CONTROLLED` reliability mode is to use correlator persistence with reliability mode set to `AT_LEAST_ONCE`. See ["Using correlator persistence with correlator-integrated messaging for JMS" on page 207](#)

When reliability mode is set to `APP_CONTROLLED` applications are still entirely responsible for handling duplicate messages as well as any message re-ordering that occurs. Applications must be able to cope with any message duplication or reordering caused by the JMS provider implementation or failures in the sender, receiver or broker.

In an Apama application, a receiver that is using `APP_CONTROLLED` reliability mode goes through the following cycle:

1. **Receive** a batch of messages. Typically, there are several hundred in a batch. The number is controlled by the `maxBatchSize` and `maxBatchIntervalMillis` receiver settings. See ["Advanced configuration bean properties" on page 206](#).
2. **Suspend** operation at the end of the batch. After suspending, the receiver sends a `JMSAppControlledReceivingSuspended` event to the context that is handling the messages.
3. **Application commits** the received messages or commits the results of received messages, such as state changes or output messages to other systems. For example, the received messages might have caused messages to be sent to a database, a distributed MemoryStore, a downstream JMS destination or a connected correlator. These operations may involve a synchronous plug-in call, or sending a request and then listening for an asynchronous event to indicate completion or acknowledgement.
4. **Acknowledge** receipt of the batch of messages to the JMS broker. After application-specific commit operations for this message batch are complete, the messages no longer need to be retained by the JMS broker. The application calls `JMSReceiver.appControlledAcknowledgeAndResume()` to acknowledge the message batch and resume receiving. The cycle then starts again.

Following is a simple example of the application logic for responding to `JMSAppControlledReceivingSuspended` events and allowing the message batch to be acknowledged after the messages have been suitably handed off to another system:

```
on all JMSAppControlledReceivingSuspended(receiverId="myReceiver")
{
    on MyFinishedPersistingReceivedEvents(requestId=persistReceivedEventsSomehow())
    {
        jms.getReceiver("myReceiver").appControlledAckAndResume();
    }
}
```

The code below shows an example of using `APP_CONTROLLED` receiving, together with flush acknowledgements from the JMS sender. See ["Sending messages reliably with application flushing notifications" on page 211](#). With this strategy, received JMS messages are acknowledged to the JMS broker only after the context gets an acknowledgement from the JMS sender that all the associated output messages have been sent to the JMS broker.

```
on all JMSAppControlledReceivingSuspended(receiverId="myReceiver")
{
    on JMSSenderFlushed(requestId = jmsConnection.getSender("mySender").requestFlush()) {
        jms.getReceiver("myReceiver").appControlledAckAndResume();
    }
}
```

It is important to use the same context to process the messages from a given receiver and to call `appControlledAckAndResume()`.

To improve the throughput of an `APP_CONTROLLED` receiver, try adjusting the `maxBatchSize` and `maxBatchIntervalMillis` receiver settings. The goal is to balance the time spent receiving JMS messages and the time spent committing the results. If the batches are too small then throughput can decrease. If the batches are too large then latency can increase and the JMS broker could use excessive memory to hold the unacknowledged messages.

It is possible to use the `APP_CONTROLLED` reliability mode for a receiver in a persistence-enabled correlator. In this case, process the messages and call `appControlledAckAndResume()` from a non-persistent monitor. Acknowledgements cannot be controlled from a persistent monitor because the JMS acknowledgement would get out of sync with the monitor state after recovery. If you try to call `appControlledAckAndResume()` from a persistent monitor an exception will be thrown.

Note: JMS messages that result in mapping failures cannot be handled by the EPL application so they are usually acknowledged automatically.

[Sending and receiving reliably without correlator persistence](#)

Sending messages reliably with application flushing notifications

Applications that use `BEST_EFFORT` reliability to send JMS messages can prevent message loss without using persistent monitors. To do this, each time an application sends a message to the JMS sender channel it also keeps the state required to re-generate the message. Periodically, the application requests the JMS sender to flush a batch of messages to the JMS broker. After all messages in this batch are sent to a JMS broker, the JMS sender sends a flush acknowledgement to the context that requested flushing. When the application receives the flush acknowledgement it executes an application-defined strategy for clearing state associated with the messages that have been sent to the JMS sender channel. This protects the application against failure of the correlator host.

Note: Messages are still asynchronously sent to the JMS broker even when no flushing has been requested. Requesting a flush simply gives the application the ability to be notified when the messages have been handed off to the JMS broker.

The typical behavior of an application that sends messages reliably without using correlator persistence is as follows:

1. Continuously send messages to the JMS sender channel.

At the same time, the application must keep track of the messages that have been sent to the sender channel but not yet flushed to the JMS broker. These are referred to as outstanding messages.

Also, the applications must reliably keep whatever state is required to re-generate each message. It is important to ensure that the application would not lose data if the outstanding messages were lost due to failure of the correlator node. This is typically achieved by delaying acknowledgement of the incoming JMS messages, Apama events or database/MemoryStore transactions that are generating the sent messages.

2. Request JMS sender to flush outstanding messages.

Periodically, for example, for every 1000 outstanding messages, the application requests that the sender flush the outstanding messages to the JMS broker. This is accomplished by invoking the `JMSSender.requestFlush()` action. After sending the messages to the JMS broker, this action sends a `JMSSenderFlushed` acknowledgement event to the context that requested flushing.

The application should set up a listener for the `JMSSenderFlushed` event whose `requestId` field is equal to the `requestId` generated by the `requestFlush()` action. Also, this listener needs a reference to whatever state corresponds to this batch of outstanding messages. For example, this might be a transaction id.

You must determine how many messages to send before flushing the batch. Flushing each message is not advised as it would add a noticeable performance overhead. However, you do not want to flush messages so infrequently that excessive memory or buffer space is required to hold the state associated with the outstanding messages.

Be sure to implement any required mechanism for downstream receivers to deal with duplicate messages. Typically, an application does this by adding a unique id to each message.

3. Continue sending events to the JMS sender channel.

In many cases, it is fine to continue sending new events to the sender channel while waiting for acknowledgement that previous batches have been flushed. That is, it is okay to have multiple batches in flight to the JMS broker at any one time. This improves throughput but is more complicated to implement. Whether it is possible to have multiple flushes in flight simultaneously in your specific application depends on what the application needs to do when it receives a `JMSSenderFlushed` acknowledgement event.

4. Application receives a flush notification event.

When the JMS sender has finished processing all events in a batch that is being flushed to a JMS broker, it sends a `JMSSenderFlushed` event to the context that invoked the `requestFlush()` action. At this point, the messages are the responsibility of the JMS broker and they are safe from loss even if the correlator or other nodes fail.

The application should now remove any state associated with the messages in this batch. For example, the application can acknowledge the incoming messages that generated the messages sent to the JMS broker, or commit a database or `MemoryStore` transaction, or send an event that allows some other component to clear associated state from its buffers.

While this feature allows a well-designed application to prevent message loss in the case of a correlator failure, it cannot prevent message loss due to invalid mapping rules or non-existent JMS destinations. Such failures are recorded in the correlator log, but any messages associated with these failures are still included in the next flush acknowledgement, even though sending them to the JMS broker resulted in a failure. This behavior

- Prevents failure of one message indefinitely blocking the sending of subsequent messages
- Applies only to application bugs that would not benefit from retrying

If a recoverable failure occurs, such as loss of connection to the JMS broker, Apama keeps trying to send the messages until the connection is restored. While this might result in a long delay before the flush acknowledgement can be sent, no messages are lost. The flush acknowledgement is therefore an indication that the message batch has been fully processed by the correlator's JMS sender to the best of its ability. The flush notification is not a guarantee that every message in the batch was successfully delivered to the broker. For example, problems in the application or in the mapping configuration might have prevented successful delivery to the JMS broker.

Sending messages reliably without using correlator persistence is available only for senders that are using `BEST_EFFORT` reliability mode. Senders that are using `AT_LEAST_ONCE` or `EXACTLY_ONCE` reliability mode use the correlator's persistence feature and so have no need for manual send notifications.

A call to the `requestFlush()` action in a persistent monitor throws an exception. Allowing this call would cause the JMS acknowledgement state to be out of sync with the monitor state after recovery.

The code below provides an example of sending messages reliably with flushing acknowledgements.

```
using com.apama.correlator.jms.JMSSender;
using com.apama.correlator.jms.JMSConnection;

monitor FlushMessagesToJMSBroker {
    . . .
    // Each time the application sends an event to the JMS sender
    // channel, increment the number of messages sent but not flushed.
    send MyEvent() to.jmsConnection.getSender("mySender").getChannel();
    sendsSinceLastFlush := sendsSinceLastFlush + 1;
    if sendsSinceLastFlush = 1000 then {
        // Stash state needed to re-send messages in case of correlator
        // failure. After receiving a flush acknowledgement, this state can
        // be cleared. In this example, keep a transaction id for a database.
        integer transactionAssociatedWithFlushRequest := currentTransaction;

        // Optionally, allow multiple flushes to be in flight concurrently.
        currentTransaction := startNewTransaction();

        // Request JMS sender to flush messages to the JMS broker.
        // Listen for flush acknowledgement event and ensure that state
        // that was saved can be cleared when the listener fires.
        on JMSSenderFlushed(requestId =.jmsConnection.getSender("mySender").requestFlush()) {
            commitTransaction(transactionAssociatedWithFlushRequest);
        }
    }
}
```

When using sender flushing, an application can optionally set the JMS sender `messageDeliveryMode` property to `PERSISTENT`. This ensures that the messages are protected from loss by the JMS broker. See `jms:sender` properties in ["XML configuration bean reference" on page 200](#).

[Sending and receiving reliably without correlator persistence](#)

Using the correlator input log with correlator-integrated messaging for JMS

The correlator input log can be used in applications that use most correlator-integrated messaging for JMS features including sending, receiving and listening for status events. The input log will include a record of all events that were received from JMS so there is no need for JMS to be explicitly enabled with the `--jmsConfig` option when performing replay. Instead, the resulting input log can be extracted and used in the normal way, without the `--jmsConfig` option. Attempting to perform replay with correlator-integrated messaging for JMS is not supported and is likely to fail, especially with reliable receivers in a persistent correlator.

Note that the "dynamic" capabilities of correlator-integrated messaging for JMS do not currently work in a replay correlator (because a correlator plug-in is used behind the scenes), so if you need to retain the possibility of using an input log you must not use dynamic senders and receivers or call the `JMSSender.getOutstandingEvents()` method.

[Designing and implementing applications for correlator-integrated messaging for JMS](#)

Reliability considerations when using JMS

When using the `EXACTLY_ONCE`, `AT_LEAST_ONCE` or `APP_CONTROLLED` reliability mode, Apama's correlator-integrated messaging for JMS provides a "reliable" way to send messages into and out of the correlator such that in the event of a failure, any received messages whose effects were not persisted

to stable storage will be redelivered and processed again, and that the events received from the correlator by external systems are consistent with the persisted and recovered state.

- Correlator-integrated messaging for JMS guarantees no message loss, assuming there is stable storage and that the JMS broker behaves reliably. Also, there must be no fatal message mapping errors.
- When using `EXACTLY_ONCE` reliability mode, correlator-integrated messaging for JMS guarantees no message duplication within a specifically configured window size. The window size, for example, might be set to the last 2000 events or events received in the last two minutes. Note that even with the help of Apama's `EXACTLY_ONCE` functionality, JMS message duplicate detection is not a simple or automatic process and requires careful design. Customers are strongly encouraged to architect their applications to be tolerant of duplicate messages and use the simpler `AT_LEAST_ONCE` reliability mode instead of `EXACTLY_ONCE` when possible. (Using the `APP_CONTROLLED` reliability mode for receivers is an advanced alternative.)
- Apama's correlator-integrated messaging for JMS provides a *best effort* correct message ordering but this is not guaranteed. The exact message ordering behavior is broker-specific. Correlator-integrated messaging does not make ordering guarantees in the event of a broker or client failure. Occasionally, some JMS brokers reorder messages unexpectedly. If your application requires correct message order, it may be possible to set the `JMSXGroupSeq` and `JMSXGroupID` message properties to request the chosen JMS provider implementation to provide ordering for a group of related messages. It is not possible to provide ordering across all messages without forcing use of a single consumer, which would reduce throughput scalability.

Care must be taken when designing, configuring and testing the application to ensure it can cope with significant fluctuations in message rates, as well as serious failures such as network or component failures that lasts for several minutes, hours or days. Consider using JMS message expiry to avoid flooding queues with unnecessary or stale messages on recovery after a long period of down time.

Designing and implementing applications for correlator-integrated messaging for JMS

Duplicate detection when using JMS

Apama provides an `EXACTLY_ONCE` receiver reliability setting that allows a finite number of duplicate messages to be detected and dropped before they get to the correlator. This setting can be used to reduce the chance of duplicates; however with JMS, duplicate detection is a complex process. Therefore, customers are strongly encouraged to architect their applications to be tolerant of duplicate messages and use the simpler `AT_LEAST_ONCE` reliability mode instead of `EXACTLY_ONCE` when possible.

Configuring duplicate detection is an inexact science given that it depends considerably on the behavior of the sender(s) for a queue, and requires careful architecture and sizing to ensure robust operation in normal use and expected error cases. Moreover it is not possible to guarantee duplicate messages will never be seen without an infinite buffer of duplicates. Give particular attention to architectures where multiple sender processes are writing to the same queue, especially if it is possible that one sender may send a duplicate message it has taken off another failed sender that has not recorded the fact that it is already processed and sent out a given message.

Duplicate detection is a trade-off between probability of an old duplicate not being recognized as such, and the amount of memory and disk required, which will also have an impact on latency and throughput.

Selecting the right value for the `dupDetectionExpiryTimeSecs` is a very important aspect of ensuring that the duplicate detection process will operate reliably — detecting duplicates where necessary

without running out of memory when something goes wrong. The expiry time used for the duplicate detector should take into account how the JMS provider will deal with several consecutive process or connection failures on the receive side, especially if the JMS provider temporarily holds back messages for failed connections in an attempt to work around temporary network problems. Be sure to consult the documentation for the JMS provider being used to understand how it handles connection failures. It is a good idea to conduct tests to see what happens when the connection between the JMS broker and the correlator goes down. When testing, consider using the `rMaxDeliverySecs=` value from the `"JMS Status:"` line in the correlator log to help understand the minimum expiry time needed to catch redelivered duplicates. Note, however, this is only useful if the JMS provider reliably sets the `JMSRedelivered` flag when performing a redelivery. A good rule of thumb is to use an expiry time of two to three times the broker's redelivery timeout.

Note that although space within the reliable receive (duplicate detection) datastore is reclaimed and reused when older duplicates expire, the file size will not be reduced. There is currently no mechanism for reducing the amount of disk space used by the database, so the on-disk size may grow, bounded by the peak duplicate detector size, but will not shrink.

Messages that are subject to duplicate detection contain:

- `uniqueMessageId` - an application-level identifier which functions as the key for determining whether a message is functionally equivalent (or identical) to a message already processed, and should therefore be ignored.
- `messageSourceId` - an optional application-specific string which acts as a key to uniquely identify upstream message senders. This could be a standard GUID (globally unique identifier) string. If provided, the `messageSourceId` is used to control the expiry of `uniqueMessageIds` from the duplicate detection cache, allowing `dupDetectionPerSourceExpiryWindowSize` messages to be kept per `messageSourceId`. This massively improves the reliability of the duplicate detection while keeping the window size relatively small, since if one sender fails then recovers several hours later, there is no danger of another (non-failed) sender filling up the duplicate detection cache in the meantime and expiring the ids of the first sender causing its duplicates to go undetected.

The key configuration options for duplicate detection are:

- `dupDetectionPerSourceExpiryWindowSize` - The number of messages that will be kept in each duplicate detection domain per `messageSourceId` (if `messageSourceId` is set on each message by the upstream system - messages without a `messageSourceId` will all be grouped together into one window for the entire `dupDetectionDomainId`). This property is specified on the global `JmsReceiverSettings` bean. It is usually configured based on the characteristics of the upstream JMS sender, and the maximum number of in-doubt messages that it might resend in the case of a failure. The default value in this release is 2000. It can be set to 0 to disable the fixed-size per-sender expiry window.
- `dupDetectionExpiryTimeSecs` - The time for which `uniqueMessageIds` will be remembered before they expire. This property is specified on the global `JmsReceiverSettings` bean. The default value in this release is 2 minutes. It can be set to 0 to disable the time-based expiry window (which makes it easier to have a fixed bound on the database size, though this is not an option if the JMS provider itself causes duplicates by redelivering messages after a timeout due to network problems).
- `dupDetectionDomainId` - An application-specific string which acts as a key to group together receivers that form a duplication detection domain, for example, a set of receivers that must be able to drop duplicate messages with the same `uniqueMessageId` (which may be from one, or multiple upstream senders). This property is specified on the `jms:receiver` bean. By default, the duplicate detection domain is always the same as the JMS destination name and `connectionId`, so cross-receiver duplicate detection would happen only if multiple receivers in the same connection are concurrently listening to the same queue; duplicates would not be detected if

sent to a different queue name, or if sent to the same queue name on a different connection, or if JNDI is used to configure the receiver but the underlying JMS name referenced by the JNDI name changes. Also note that if the message streams processed by each receiver were being partitioned using message selector, unnecessary duplicate detection would be performed in this case. The duplicate detection domain name can be specified on a per-receiver basis to increase, reduce or change the set of receivers across which duplicate detection will be performed. Common values are:

- `dupDetectionDomainId=connectionId+"."+jmsDestinationName` - the default for queues.
- `dupDetectionDomainId=jmsDestinationName` - if using receivers to access the same queue from multiple separate connections.
- `dupDetectionDomainId=jndiDestinationName` - if using JNDI to configure receiver names, and needing the ability to change the queue or topic that the JNDI name points to.
- `dupDetectionDomainId=connectionId+"."+receiverId` - the default for topics; also used if each receiver should check for duplicates independently of other receivers. This is useful if receivers are already using message selectors to partition the message stream, which implies that cross-receiver duplicates are not possible.
- `dupDetectionDomainId=<application-defined-name>` - if using multiple receivers per selector-partitioned message stream. The name is likely to be related to the message selector expression.

Duplicate detection only works if the upstream JMS sender has specified a `uniqueMessageId` for each message (the `uniqueMessageId` is typically as a message property, but could alternatively be embedded within the message body if the mapper is configured to extract it). Any messages that do not have this identifier will not be subject to duplicate detection. The `uniqueMessageId` string is expected to be unique across all messages within the configured `dupDetectionDomainId` (for example, `queue`), including messages with different `messageSourceIds`. By default, sent JMS messages would have a `uniqueMessageId` of `seqNo:messageSourceId`, where `seqNo` is a contiguous sequence number that is unique for the sender, for example:

```
uniqueMessageId=1:mymachinename1.domain:1234:567890:S01
uniqueMessageId=2:mymachinename1.domain:1234:567890:S01
uniqueMessageId=3:mymachinename1.domain:1234:567890:S01
uniqueMessageId=1:mymachinename2.domain:4321:987654:S01
uniqueMessageId=2:mymachinename2.domain:4321:987654:S01
uniqueMessageId=1:mymachinename2.domain:4321:987654:S02
uniqueMessageId=2:mymachinename2.domain:4321:987654:S02
...
```

To reliably perform duplicate detection if there are multiple senders writing to the same queue (without the Apama receiver having to configure a very large and therefore costly time window to prevent premature expiry of ids from a sender that has failed and produces no messages for a while then recovers, possibly sending duplicates as it does so), the upstream senders should be configured to send with a globally-unique `messageSourceId` identifying the message source/sender, which should also be configured in the mapping layer of the receiver.

Apama's duplicate detection involves a set of fixed-size per-sourceId queues, and when the queue is full the oldest items are expired to a shared queue ordered by timestamp (time received by the correlator's JMS receiver) whose items are expired based on a time window. So the receiver settings controlling duplicate detection window sizes are:

- `dupDetectionPerSourceExpiryWindowSize`
- `dupDetectionExpiryTimeSecs`

`uniqueMessageIds` are expired from the per-source queue (and moved to the time-based queue) when it is full of newer ids, or when a newer message with the same `uniqueMessageId` already in the queue for that source is received.

`uniqueMessageIds` are expired from the time-based queue (and removed from the database permanently) when they are older than the newest item in the time-based queue by more than `dupDetectionExpiryTimeSecs`.

Designing and implementing applications for correlator-integrated messaging for JMS

Performance considerations when using JMS

When designing an application that uses correlator-integrated messaging for JMS it may be relevant to consider the following topics that relate to performance issues.

There are no guarantees about maximum latency. Persistent JMS messages inevitably incur significant latency compared to unreliable messaging, and Apama's support for JMS is focused around throughput rather than latency. Messages can be held up unexpectedly by many factors such as: the JMS provider; by connection failures; by waiting a long time for the receive-side commit transaction; by the broker `acknowledge()` call taking a long time; or by waiting a long time for the correlator to do an in-memory copy of its state.

Multiple receivers on the same queue may improve performance. But consider that "For PTP, JMS does not specify the semantics of concurrent `QueueReceivers` for the same `Queue`; however JMS does not prohibit a provider from supporting this. Therefore, message delivery to multiple `QueueReceivers` will depend on the JMS provider's implementation. Applications that depend on delivery to multiple `QueueReceivers` are not portable".

- If performance is an issue, be sure to check the correlator log for `WARN` and `ERROR` messages, which may indicate your application or configuration has a connection problem that may be responsible for the performance problem.
- Ensure that the correlator is not running with `DEBUG` logging enabled or is logging all messages. Either of these will obviously cause a big performance hit. Apama recommends running the correlator at `INFO` log level; this avoids excessive logging, but still retains sufficient information that may be indispensable for tracking problems.
- In practice, most performance problems are caused by mapping, especially when XML is used. Whenever possible, Apama recommends avoiding the use of XML in JMS messages due to the considerable overhead that is always added by using such a complex message format. For example, use `MapMessage` or a `TextMessage` containing an Apama event string.
- If you are receiving several different event types, ensure that the conditional expressions used to select which mapping to execute are as simple as possible. In particular, there will be a significant performance improvement if JMS message properties are used to distinguish between different message types instead of XML content inside the message body itself because JMS message properties were designed in part for this purpose.
- Use the Correlator Status lines in the log file to check whether the bottleneck is the JMS runtime or in the EPL application itself. A full input queue ("`iq=`") is a strong indicator that the application may not be consuming messages fast enough from JMS.
- Consider enabling the `logPerformanceBreakdown` setting in `JmsSenderSettings` and `JmsReceiverSettings` to provide detailed low-level information about which aspects of sending and receiving are the most costly. This may indicate whether the main bottleneck, and hence the main optimization target, is in the message mapping or in the actual sending or receiving of messages. If mapping is

not the main problem, it may be possible to achieve an improvement by customizing some of the advanced sender and receiver properties such as `maxBatchSize` and `maxBatchIntervalMillis`.

- Consider using `maxExtraMappingThreads` to perform the mapping of received JMS messages on one or more separate threads. This is especially useful when dealing with large or complex XML messages.
- Take careful measurements. The key to successful performance optimization is taking and accurately recording good measurements, along with the precise configuration changes that were made between each measurement. It is also a good idea to take multiple measurements over a period of at least several minutes (at least), and take account of the amount of variation or error in the measurements (by recording minimum, mean, and maximum or calculating the standard deviation). In this way it is possible to notice configuration changes that have made a real and significant impact on the performance, and distinguish them from random variation in the results. Note that many JMS providers are observed to behave badly and exhibit poor performance when overloaded (for example, when sending so fast that queues inside the broker fill up and things begin to block). For this reason, the best way to test maximum steady-state performance is usually to create a way for the process that sends messages to be notified by the receiving process about how far behind it is. For example, if the sender and receiver are both correlators, `engine_connect` can be used to create a fast channel from the receiver back to the sender, and the test system can be set to send Apama events to the sender channel every 0.5 seconds so it knows how many events have been received so far. This allows better performance testing with a bound on the maximum number of outstanding messages (sent but not yet received) to prevent the broker being overwhelmed.
- Be careful when measuring performance using a virtual machine rather than dedicated hardware. VMs often have quite different performance characteristics to physical hardware. Take particular care when using VMs running on a shared host, which may be impacted by spikes in the disk/memory/CPU/network of other unrelated VMs running on the same host that belong to different users.

Designing and implementing applications for correlator-integrated messaging for JMS

Performance logging when using JMS

The `JmsSenderSettings` and `JmsReceiverSettings` configuration objects both contain a property called `logPerformanceBreakdown` which can be set to true to enable measurement of the time taken to perform the various operations required for sending and receiving, with messages logged periodically at `INFO` level with a summary of measurements taken since the last log message. The default logging interval is once per minute.

Although this property should not be enabled in a production system where performance is a priority because the gathering of the performance data adds unnecessary overhead, it can be indispensable during development and testing for demonstrating what each sender and receiver thread is spending its time doing. To produce more useful statistics, note that the first batch of messages sent or received after connection may be ignored (which will affect all statistics logged, including the number of messages received and throughput). All times are measured using the standard Java `System.nanoTime()` method, which should provide the most accurate time measurements the operating system can achieve, though not usually to nano second accuracy. For more information on the `logPerformanceBreakdown` property, see "[XML configuration bean reference](#)" on page 200.

Performance considerations when using JMS

Receiver performance when using JMS

Each receiver performance log message has a low-level breakdown of the percentage of thread time spent on various aspects of processing each message and message batch, as well as a summary line stating the (approximate) throughput rate over the previous measurement interval, and an indication of the minimum, mean (average) and maximum number of events in each batch that was received.

The items that may appear in the detailed breakdown are:

- **RECEIVING** - time spent in the JMS provider's `MessageConsumer.receive()` method call for each message received.
- **MAPPING** - time spent mapping each JMS message to the corresponding Apama event. If `maxExtraMappingThreads` is set to a non-zero value then this is the time spent waiting for remaining message mapping jobs to complete on their background thread(s) at the end of each batch.
- **DB_WAIT** - (only for reliable receive modes) time spent waiting for background reliable receive database operations (writes, deletes, etc) to complete, per batch.
- **DB_COMMIT** - (only for reliable receive modes) time spent committing (synching) received messages to disk at the end of each batch.
- **APP_CONTROLLED_BLOCKING** - for receivers that are using `APP_CONTROLLED` reliability mode, this is the time spent waiting for the EPL monitor to call the `appControlledAckAndResume()` action. A monitor calls this action after it finishes processing a batch of messages from the receiver.
- **ENQUEUEING** - (only for `BEST_EFFORT` and `APP_CONTROLLED` receive mode) time spent adding received messages to each public context's input queue.
- **JMS_ACK** - time spent in the JMS provider's `Message.acknowledge()` method call at the end of processing each batch of messages.
- **R_TIMEOUTS** - the total time spent waiting for JMS provider to complete `MessageConsumer.receive()` method calls that timed out without returning a message from the queue or topic, per batch. Indicates either that Apama is receiving messages faster than they are added to the queue or topic or that the JMS provider is not executing the receive (timeout) call very efficiently or failing to return control at the end of the requested timeout period.
- **FLOW_CONTROL** - the total time spent (before each batch) blocking until the EPL application increases the flow control window size by calling `JMSReceiverFlowControlMarker.updateFlowControlWindow(...)`. In normal usage, this should be negligible unless some part of the system has failed or the application is not updating the flow control window correctly.
- **TOTAL** - aggregates the total time taken to process each batch of received messages.

Performance considerations when using JMS

Sender performance when using JMS

Each sender performance log message has a low-level breakdown of the percentage of thread time spent on various aspects of processing each message and message batch, as well as a summary line stating the approximate throughput rate over the previous measurement interval, and an indication of the minimum, mean (average) and maximum number of events in each batch that was sent.

The items that may appear in the detailed breakdown are:

- **MAPPING** - time spent mapping each Apama event to the corresponding JMS message. This includes the time spent looking up any JMS queue, topic, or JNDI destination names, unless cached.
- **SENDING** - time spent in the JMS provider's `MessageProducer.send()` method call for each message.
- **JMS_COMMIT** - time spent in the JMS provider's `Session.commit()` method call for each batch of sent messages (only if a JMS `TRANSACTIONAL_SESSION` is being used to speed up send throughput).
- **WAITING** - the total time spent waiting for the first Apama event to be passed from EPL to the JMS runtime for sending, per batch. This is affected by what the EPL code is doing, and for reliable sender modes, also by the (dynamically tuned) period of successful correlator persist cycles.
- **BATCHING** - the total time spent waiting for enough Apama events to fill each send batch, after the first event has been passed to the JMS runtime.
- **TOTAL** - aggregates the total time taken to process each batch of sent messages.

Performance considerations when using JMS

Configuring Java options and system properties when using JMS

Sometimes it is necessary to specify Java system properties to configure a JMS provider's client library, or to change JVM options such as the maximum memory heap size. Because these settings inevitably affect all JMS providers that the correlator is connecting to, in addition to any JMon applications in the correlator, Java options must be specified on the correlator command line rather than in a JMS connection's configuration file.

Each Java option to be passed to the correlator should be prefixed with `-J` on the command line, for example, `-J-Dpropname=propvalue -J-Xmx512m`. To set Java options when starting the correlator from Apama Studio, edit the Apama launch configuration for your project as follows:

1. In the Project Explorer right-click the project name and select **Run As > Run Configurations**. The **Run Configurations** dialog is displayed.
2. In **Run Configurations** dialog, in the Project field, make sure the your project is selected.
3. On **Run Configurations** dialog's Components tab, select the correlator to use and click Edit. The **Correlator Configuration** dialog is displayed.
4. In the **Correlator Configuration** dialog, in the Extra command line arguments field, add the system property, for example, `-J-Dpropname=propvalue -J-Xmx512m` and click OK.

Designing and implementing applications for correlator-integrated messaging for JMS

Diagnosing problems when using JMS

This topic contains several approaches for diagnosing JMS issues you may encounter.

- Consider contacting the vendor of the JMS provider that is being used. JMS brokers are complex pieces of software with many configuration options. JMS providers often maintain on-line databases of known bugs and issues. Software AG is not in a position to provide detailed support or performance tuning for JMS brokers provided by a third party, but the provider may be able to suggest useful changes to configuration options that can affect performance and reliability trade-

offs and provide further assistance tracking down crashes, hangs, performance, disconnection and flow control problems.

- Check the correlator log file for `WARN` and `ERROR` messages that may indicate the underlying problem. Also check for any log lines "Longest delay between a JMS message being sent and the broker delivering it to this receiver is now", especially after an unexpected disconnection or when testing a correlator/broker machine or network failure. This will give an indication of how your broker redelivers in-doubt messages, and may affect the size of the duplicate detection time-based expiry window.
- Check the JMS broker's log files and console for error messages or warnings.
- Consider temporarily using `logJMSMessages` and `logProductMessages` to display all messages being sent and received. This is particularly useful for problems related to mapping; on the other hand it is not useful for diagnosing performance-related issues.
- Use the "JMS Status:" lines to understand what is going on in more detail. Consider setting `logDetailedStatus=true` to get more in-depth per-sender and per-receiver status lines.
- Check for any log lines "Longest delay between a JMS message being sent and the broker delivering it to this receiver is now" ... which may indicate that the broker is behaving strangely or that queued messages from a previous test run are unexpectedly being received, perhaps causing mapping failures or performance problems.
- If further assistance from Software AG is required to track down a problem, it is essential to provide a copy of the full correlator log file and the JMS configuration being used to ensure that all the required information is available.
 - To capture the correlator log output, edit the launch configuration as follows:
 1. Right-click the project and select Run As > Run Configurations from the pop-up menu.
 2. Ensure the configuration for this project is selected.
 3. Select the Components tab,
 4. Edit the DefaultCorrelator setting by adding extra command line arguments: `--logfile logs/correlator.log`.
 5. Optionally add `--truncate` to clear the log file at start up to eliminate confusion with output from previous runs.

Note, simply copying lines from the Console view is usually *not* adequate for support purposes (for example, status lines are missing and in some cases header information is missing as well).

- To collect the essential JMS configuration files.
 1. Right-click the project and select Properties from the pop-up menu.
 2. In the Resource section, note the directory information listed in the Location field. (Copy the information if desired.)
 3. In the file system, navigate to that directory. (Paste the directory information into the Run command of the Windows Start menu.)
 4. Zip up the contents of the `bundle_instance_files\Correlator-Integrated_JMS` sub-directory.

Using Correlator-Integrated Messaging for JMS

JMS failures modes and how to cope with them

Apama provides many features that simplify the integration process, but JMS brokers are complex pieces of software performing a complex task and successfully designing a truly reliable application built on JMS requires careful thought and testing, as well as a full understanding of the behavior and configuration of your chosen JMS provider.

The following list highlights some of the things that can go wrong, and how they are handled by Apama along with suggestions for how they might be handled by a solution architect.

- **Failure of connection between the correlator and the JMS-broker** (due to machine failure or network problems) – Apama handles this by writing an `ERROR` to the correlator log and sending `JMSConnectionStatus`, `JMSSenderStatus`, and `JMSReceiverStatus` events detailing the error to all affected connections, senders, and receivers. An application can use these events to display the problem on a dashboard or send an email or text message to notify an administrator. Once the connection has gone down Apama will repeatedly try to re-establish it, at a rate determined by the `connectionRetryIntervalMillis` property of `jms:connection` (once per second by default). As soon as the connection has been re-established, all associated senders and receivers will create a session using the new connection and begin to send and receive again. Note that occasionally some third party JMS libraries have been observed to hang after a network problem, preventing successful reconnection, especially when there is a mismatch between the .jar versions used on the client and server; it is worth testing to ensure this does not affect your deployment. During the period when the connection is down, the JMS sender will be unable to send events to the JMS broker, so all such events will be queued in memory - see the "**Sending messages too fast**" failure mode for more details.
- **Sending messages too fast** (because the connection is down; because the broker's queue is exceeded due to a downstream JMS client receiving blocking; or simply because the attempted send rate is too high) – A bounded number of unsent messages will be held in a Java buffer until sent to JMS, but if the number of outstanding events exceeds that buffer they will be queued in C++ code. It is possible the correlator could fail with a C++ out of memory error in rare cases where too many events are sent to a reliable sender between persistence cycles. However in most cases the behavior will be that the JMS runtime acts as an Apama 'slow consumer' and in time causes correlator contexts to block when calling `send` until the messages can be processed. In time this may also cause the input queue to fill up, to prevent an out of memory error occurring. All of this behavior can be avoided if necessary by using the `JMSSender.getOutstandingEvents()` action to keep track of the number of outstanding events and take some policy-based action when this number gets too high. Typical responses might be to page some out to a database, notify an administrator, or begin to drop messages. Also note that many JMS providers have built in support for 'paging' or 'flow to disk' that, when enabled, allows messages to be buffered on disk client-side if the broker cannot yet accept them. In some cases this may be more desirable than causing the correlator to block.
- **Receiving messages too fast** – In a well-designed system an Apama application will usually be able to keep up with the rate of messages arriving from JMS. However it is important to consider the possibility of a large number of messages being received quickly on startup or after a period of downtime (for example, due to hardware failure), or from a backlog of input messages building up when downstream systems such as databases or JMS destinations that the application needs to use to complete processing of input messages become unresponsive.

If messages are received too fast for the Apama application's listeners to synchronously process them, the input queue will fill up, after which the JMS receivers will be blocked from sending more messages until the backlog is cleared. However, if the listeners for the input messages complete quickly but kick off asynchronous operations for each input message (for example, event listeners for database requests, or adding the messages to EPL data structures) then it is possible that the correlator could instead run out of memory if messages continue to be received faster than they can be fully processed. The correlator's support for JMS provides a feature called *receiver flow control* to deal with these situations, which allows an EPL application to set a window size representing the number of events that each JMS receiver can take from the broker, thereby putting a finite bound on the number of outstanding events and operations. See ["Receiver flow control" on page 189](#) for more information about receiver flow control. Another approach to avoid a very large warm-up period when dealing with old messages during startup is to make use of the JMS message time-to-live header when sending messages. This ensures that older messages can be deleted from the queue by the JMS broker once they are no longer useful. Some JMS providers may also have configuration options to enable throttling of message rates.

- **JMS destination not found for a receiver** (when the JMS connection is still up) – This could be a transient problem such as a situation where a JMS server is up but a JNDI server is down, where or a JNDI name has not yet been configured. The failure could also be a permanent one such as a destination name that is invalid. Apama handles this case by writing an `ERROR` log message, sending a `JMSReceiverStatus` event with status of `"DESTINATION_NOT_FOUND"` or possibly `"ERROR"`), then backing off for the configured `sessionRetryIntervalMillis` (1 second by default), before retrying. If it is expected that destination names may often be invalid, it might be best to use dynamic rather than static receivers. This allows the Apama application to take a policy-based decision on whether to give up trying to look up the destination and remove the receiver after a timeout period.
- **JMS error sending message** (when the JMS connection is still up) – This could be a transient problem such as a situation where the JMS server has a problem but the connection's exception listener not yet triggered. The failure could be permanent one such as a case where a JMS message is invalid for some reason. Apama writes an `ERROR` log message when this happens. If the error is specific to this message such as `MessageFormatException` OR `InvalidDestinationException` then the message is simply dropped. In other error cases, Apama will back off for the configured `sessionRetryIntervalMillis` (1 second by default) then close and recreate the session and `MessageProducer` before retrying once. After two failed attempts Apama stops trying to send the message to avoid the sender getting stuck. If a number of messages are being sent in a transacted batch for performance reasons, when a failure occurs Apama retries each message in the batch one by one in their own separate transactions to ensure that problems with one message do not affect other messages.
- **JMS destination not found when sending a message** (when the JMS connection is still up) – This could be a transient problem such as a JMS server being up but with a JNDI server down, or a JNDI name not configured yet. It could be a permanent failure such as a destination name that is invalid. Apama handles this case in a fashion similar to the way it handles the **"JMS error sending message"** case mentioned above, except that it does not attempt to retry sending if it determines that a destination not found error was the cause, since it is unlikely to work a second time after an initial failure, and other messages being sent to different destinations would get held up if it did.
- **Exception while a mapping message** (during sending or receiving; typically caused by invalid mapping rules, invalid conditional expressions, or malformed messages, such as an unexpected XML schema) – If the mapping error is so serious that the message cannot be mapped at all (for example, receiving a message that did not map any of the defined conditional mapping

expressions), an `ERROR` is logged and the message is dropped. If the error affects only one of the field mapping rules, then an `ERROR` is logged and the field will be given a default value such as `""`, `0`, `null`, etc. Note that a large batch of badly formed messages can result in a large number of messages and stack traces being written to the log, so care should be taken to avoid this by comprehensive testing and careful writing of conditional expressions.

- **Error parsing received event type** (due to mismatch between mapping rules and injected event types, or failure to inject the required types) – The correlator logs a `WARN` message when events are received that do not match any injected event type; the log file should be checked during integration testing to ensure this is not happening.
- **EXACTLY_ONCE duplicate detector fails to detect duplicates** – Correctly detecting all duplicate messages involves ensuring that the upstream JMS client (if not a correlator) is correctly putting truly unique identifiers into all the messages it sends, and that the receiving JMS client is configured with a sufficiently large window of duplicate identifiers to catch all likely cases in which duplicates might be sent. When configuring the receiver's duplicate detector, it is particularly important to understand the circumstances under which your JMS provider will redeliver messages — some providers will redeliver messages several minutes after they were originally sent especially in the event of a failure, which means the duplicate detector time window needs to be at least two or three times larger than the redelivery window. If messages are being put onto the bus from multiple senders, it is an extremely good idea to set a `messageSourceId` on each message to allow correlator-integrated messaging for JMS to maintain a separate duplicate detection window for each message source. In some applications it may be useful to set a time-to-live on sent messages to place a bound on the maximum delay between sending a message and having it received and successfully recognized as a duplicate, in those situations where it is better to risk dropping potentially non-duplicate older messages than to risk re-processing duplicate older messages.
- **EXACTLY_ONCE duplicate detector out of memory** – It is important to ensure that there is enough memory on the machine and enough allocated to the correlator's JVM to hold the all of the duplicate detection information required for both normal usage and exceptional cases; if this memory is exceeded then the correlator process will fail with an out of memory error. Note that this only applies to reliable receivers using `EXACTLY_ONCE` reliability; due to the additional complexity arising from duplicate detection, customers are advised to use this feature only when really needed — in many cases it is possible to architect an application so that it is tolerant of duplicate messages (idempotent) which completely avoids the need for all design, sizing and testing work that `EXACTLY_ONCE` mode entails. If duplicate detection is enabled, the total amount of memory required by the duplicate detector for each `dupDetectionDomainId` is a function of the average message size, the number of distinct `messageSourceIds` (per `dupDetectionDomainId`), and the configuration parameters `dupDetectionPerSourceExpiryWindowSize` and `dupDetectionExpiryTimeSecs`. It is not practical to accurately estimate the exact memory requirements of the duplicate detector in advance; instead, it is recommended that applications with high reliability requirements are carefully tested to determine how much memory is required with the peak likely memory usage, and to ensure that the correlator's JVM is configured with a sufficiently high maximum memory limit to accommodate this (for example on the command line set `-J-Xmx2048m` for a 2GB heap). The most important parameter to watch is the `dupDetectionExpiryTimeSecs`, since the time-based expiry queue does not have a bounded number of items, so if it is set to be too large or a lot of messages are received unexpectedly in a very short space of time it could grow to a very large size. The "JMS Status" lines that the correlator periodically logs provide invaluable information about the number of duplicate detection ids being stored at any time, as well as the amount of memory the JVM is currently using. Enabling the `logDetailedStatus` receiver settings flag will turn on additional information for each receiver that includes a breakdown of the number of duplicate detection identifiers stored in each part of the duplicate detector.

-
- **Disk errors/corruption** – Both correlator persistence and the reliable receive functionality of correlator-integrated messaging for JMS depend on the disk subsystem they are written to. It is important to use some form of storage that is reliable such as a NAS/Network-Attached Storage device or SAN/storage-area network and which is guaranteed to not introduce corruption in the event of a failure such as a power failure. Apama also relies on the file system to implement correct file locking; if this is not the case or if the device is not correctly configured, then it is possible that messages could be lost or the correlator could fail, either in normal operation or in the event of an error.
 - **JMS provider bugs** – A number of widely used enterprise JMS providers have bugs that might result in message loss, reordering, or unexpected re-deliveries (causing duplication). In other cases some bugs manifest as broker or client-side hangs, Java deadlocks, thread and memory leaks, or other unexpected failures. These are especially common when a JMS client like the correlator has been disconnected uncleanly from the JMS broker, perhaps due to the process or network connection being forcibly killed. Correlator-integrated messaging for JMS includes workarounds for many known third-party bugs in the JMS providers that Apama supports to make life easier for customers. However, it is not possible to find workarounds for all problems. Therefore Apama encourages customers to familiarize themselves with the release notes and outstanding bugs lists published by their JMS vendor — ideally before selecting a vendor — and to conduct sufficient testing early in the application development process to allow for a change of JMS vendor if required.

Using Correlator-Integrated Messaging for JMS

Chapter 6: Using Universal Messaging in Apama Applications

■ Overview of using UM in Apama applications	226
■ Setting up UM for use by Apama	233
■ Starting correlators that use UM	235
■ Configuring adapters to use UM	235
■ EPL and UM channels	237
■ Defining UM properties for Apama applications	238
■ Monitoring Apama application use of UM	239

Universal Messaging (UM) is Software AG's middleware service that delivers data across different networks. It provides messaging functionality without the use of a web server or modifications to firewall policy. In Apama applications, you can configure and use the connectivity provided by UM.

Only UM channels can be used with Apama. UM queues and datagroups are not supported in this Apama release.

Apama 5.3 supports only the 9.8 release of Universal Messaging. The [Apama 5.3 Supported Platforms](#) document lists supported releases for all Apama components.

Using Message Services

Overview of using UM in Apama applications

In an Apama application, correlators and adapters can connect to UM realms or clusters. A correlator or adapter connected to a UM realm or cluster uses UM as a message bus for sending Apama events between Apama components. Connecting a correlator or adapter to UM is an alternative to

- Specifying a connection between two correlators by executing the `engine_connect` correlator utility. This is the main reason to use UM.
- Defining connections between an adapter and particular correlators in the `<apama>` element of an adapter configuration file. This is a secondary reason to use UM. You might find that having some adapters connected to UM is a good fit for your application.

Using UM can simplify an Apama application configuration. Instead of specifying many point-to-point connections you specify only the address (or addresses) of the UM realm or cluster. Apama components connected to the same UM realm can use UM channels to send and receive events. (UM channels are equivalent to JMS topics.) Connections to UM are automatically made as needed, giving extra flexibility in how the application is architected.

When an Apama application uses UM a correlator automatically connects to the required UM channels. There is no need to explicitly connect UM channels to individual correlators. A correlator automatically receives events on UM channels that monitors subscribe to and automatically sends events to UM channels.

Comparison of Apama channels and UM channels

In an Apama application configured to use UM, when an event is sent and a channel name is specified the default behavior is that Apama determines whether there is a UM channel with that name. If there is then Apama uses the UM message bus and the specified UM channel to deliver the event to any subscribers. Subscribed contexts can be in either the originating correlator or other correlators connected to the UM broker.

If a UM channel with the specified name does not exist then the default is that the channel is an Apama channel. An event sent on an Apama channel is delivered to any contexts that are subscribed to that channel.

Regardless of whether the channel is a UM channel or an Apama channel, events are delivered directly to receivers that are connected directly to the correlator.

See ["Enabling automatic creation of UM channels" on page 230](#) to learn about how you can change the default behavior.

The following table compares the behavior of Apama channels and UM channels.

Apama channels	UM channels
<p>Configuration of multiple point-to-point connections.</p> <p>Each execution of <code>engine_connect</code> specifies the correlator to connect to. Each adapter configuration specifies each correlator that adapter connects to.</p> <p>Correlators and adapters require explicitly set connections to communicate with each other.</p>	<p>Specification of the same UM realm address or addresses.</p> <p>Startup options for connected correlators specify the same UM realm to connect to. Each adapter configuration specifies the same address for connecting to UM.</p> <p>Correlators and adapters automatically connect to UM to communicate with each other.</p>
Configuration changes are required when an Apama component is moved to a different host.	No configuration change needed when an Apama component is moved to a different host if both hosts are connected to the same UM realm.
Outside a correlator, channel subscriptions can be from only explicitly connected Apama components.	Outside a correlator, channel subscriptions can be from any Apama component connected to the same UM realm.
Events sent on an Apama channel go directly to subscribers.	Events sent on a UM channel go to the UM broker and then to subscribers.
Connection configurations must be synchronized with application code.	Connection to a UM realm is independent of application code.
Less efficient for sending the same event to many Apama components.	More efficient for sending the same event to many Apama components.

Apama channels	UM channels
More efficient when sending an event to a context in the same correlator. The event stays inside the correlator.	Less efficient when sending an event to a context in the same correlator. The event leaves the correlator, enters the UM realm, and then returns to the correlator.
Default channel, the empty string, is allowed.	No default channel.

Overview of using UM in Apama applications

Choosing when to use UM channels and when to use Apama channels

Typically, you want to

- Use UM channels to send events from one correlator to another correlator, from adapters to correlators, or from correlators to external receivers. You also might want to use UM channels when your application needs the flexibility for a monitor or context to be moved to another correlator. With UM channels, you can move monitors or contexts among the correlators connected to the same UM realm without any re-configuration.
- Use Apama channels to send events from one context to one or more contexts in the same correlator.

Consider the case of multiple correlators connected to the same UM realm. Specification of a UM channel lets events pass between a context sending events on the channel and a context subscribed to that channel regardless of whether the two contexts are

- In the same correlator
- In different correlators on the same host
- In different correlators on different hosts

A UM channel gives flexibility in allowing the channel to be shared across multiple correlators. A deployment could start with monitors running in the same correlator and later re-deploy the monitors to run in separate correlators. The only re-configuration required is which correlators to start and where to inject the monitors.

The first time a channel is used the default behavior is that Apama determines whether it is a UM channel or an Apama channel and the designation is cached. After the first use, the presence or not of the channel in the UM broker is cached, so further use of the channel is not impacted. See ["Enabling automatic creation of UM channels" on page 230](#) to learn about changing this default behavior.

Using UM channels lets you take advantage of some UM features:

- Using a UM cluster can guard against failure of an individual UM realm server. See ["Universal Messaging Clusters: An Overview" on page 230](#) in the UM documentation.
- UM provides access control lists and other security features such as client identity verification by means of certificates and on the wire encryption. Using these features, you can control the components that each component is allowed to send events to.

Using a UM channel rather than an Apama channel can have a lower throughput and higher latency. If there is a UM channel that contexts and plug-ins send to and that other contexts and plug-ins in

the same correlator (or in different correlators) subscribe to, all events sent on that UM channel are delivered by means of the UM broker. In some cases, this might mean that events leave a correlator and are then returned to the same correlator. In this case, using an Apama channel is faster because events would be delivered directly to the contexts and plug-ins subscribed to that channel.

Overview of using UM in Apama applications

General steps for using UM in Apama applications

Before you perform the steps required to use UM in an Apama application, consider how your application uses channels. You should know which components need to communicate with each other, which events travel outside a correlator, and which events stay in a single correlator. Understand what channels you need and decide which channels should be UM channels and which, if any, should be Apama channels.

For an Apama application to use UM, the tasks you must accomplish are:

1. Install and configure UM separately from Apama.
2. Make UM libraries available to Apama.
See [Universal Messaging Documentation](#) and ["Setting up UM for use by Apama" on page 233](#).
3. In your UM installation, create UM channels for use in your Apama application. Alternatively, see ["Enabling automatic creation of UM channels" on page 230](#).
4. Start each correlator with specification of UM options. See ["Starting correlators that use UM" on page 235](#).
5. Optionally, configure adapters to connect to UM. See ["Configuring adapters to use UM" on page 235](#).
6. In your EPL code, subscribe to receive events delivered on UM channels.
See *Subscribing to channels* in *Developing Apama Applications*.
7. In your EPL code, specify UM channels when sending events.
See *Generating events with the send command* in *Developing Apama Applications*.
8. Optionally, create a UM properties file for use when starting a correlator. See ["Defining UM properties for Apama applications" on page 238](#).
9. Monitor the Apama application's use of UM. See ["Monitoring Apama application use of UM" on page 239](#).

Overview of using UM in Apama applications

About events transported by UM

When correlators and adapters pass events over the UM bus the events are in their string form. This is the same form as used by the `engine_send` and `engine_receive` utilities. These strings are encoded as UTF-8 bytes, with a null terminator character and are carried in the event data of UM ["nConsumeEvents" on page 230](#) objects.

The event properties are not set or used. Thus it is not possible to use UM to filter events.

It is possible to use the UM client libraries (available for Java, C#, C++ and other languages) to send events to or receive events from Apama components. The events must have a null terminator character at the end of the string form of the event. Once the null terminator character is removed,

the Apama event parsers available in the Java, C# or C client libraries can be used to handle events from Apama components, or construct event strings to which a null terminator character must be appended.

[Overview of using UM in Apama applications](#)

Using UM channels instead of `engine_connect`

When you are using UM channels in an Apama application you connect multiple correlators by specifying the same UM realm when you start each correlator. By using UM channels, you probably do not need to use `engine_connect` at all.

While it is possible to configure an Apama application to use both UM channels and `engine_connect`, it is not recommended.

[Overview of using UM in Apama applications](#)

Using UM channels instead of configuring adapter connections

In an Apama application, you can use UM as the communication mechanism between an adapter and one or more correlators. If you do then keep in mind the following:

- Adapters must send events on named channels; adapters cannot use the default (empty string) channel.
- A service monitor that communicates with an adapter should either be run on only one correlator, or be correctly designed to use multiple correlators. See "[Considerations for using UM channels](#)" on page 232.

When an adapter needs to communicate with only one correlator, which is often the case for a service monitor, an Apama channel might be a better choice than a UM channel. However, even in this situation, it is possible and might be preferable to use a UM channel. See "[Comparison of Apama channels and UM channels](#)" on page 227.

See also: "[Configuring adapters to use UM](#)" on page 235.

[Overview of using UM in Apama applications](#)

Enabling automatic creation of UM channels

For an Apama application to use UM channels, the default behavior requires you to use UM Enterprise Manager or UM client APIs to create those channels. You can change the default behavior so that a UM channel can be automatically created if it does not already exist when an Apama application needs to use it.

To enable automatic creation of UM channels, create a UM configuration properties file. In Apama Studio, the default name of this file is `um-config.properties`. In the properties file, specify the following properties:

- Set the `um.channels.prefix` property to a string. The default is `"UM_"`.
- Set the `um.channels.mode` property to `autocreate` or `mixed`.

When set to `autocreate`, Apama looks up only channels whose names begin with the specified prefix. If the channel does not exist it is created. For example, if the default prefix is used, channel names must start with `UM_` for the channel to be a UM channel.

When set to `mixed`, Apama looks up each channel to determine if it is a UM channel. If the channel does not exist then it is created only if it has the prefix specified by the `um.channels.prefix` property.

An advantage of specifying `autocreate` is that Apama does not look up channel names that do not begin with the specified prefix. This can improve performance especially if non-UM channel names are generated automatically, which might create many channels.

Setting the `um.channels.mode` property to `mixed` can be beneficial when you want to have two sets of channels. One set of channels, perhaps one per user, would be automatically created when needed and would be managed by Apama. The names of these channels would all start with the specified prefix. The other set of channels would be externally managed. This set might require channel attributes that are different from the attributes of the Apama-managed channels. Each channel in this set could have attributes that are different from any other channel.

If you do not specify the `um.channels.prefix` property and you set the `um.channels.mode` property to `autocreate` or `mixed` then channels whose names begin with `UM_` can be automatically created.

By default, the `um.channels.mode` property is set to `precreate`, which requires UM channels to be created by using UM Enterprise Manager or UM client APIs, for example, see `nSession.createChannel` in the *Enterprise Client API for Java* section of the [Universal Messaging documentation](#). Apama looks up all channels (except the default `""` channel) to determine whether they are UM channels. If a channel does not exist as a UM channel it is not created.

The following table compares the behavior of the settings for `um.channels.mode`:

Sample Channel Name	<code>precreate</code>	<code>mixed</code>	<code>autocreate</code>
<code>UM_myChannel</code>	Look up. Never create.	Look up. Can create.	Look up. Can create.
<code>myChannel</code>	Look up. Never create.	Look up. Never create.	Never look up. Never create.

After you create a UM configuration properties file with the desired settings for the `um.channels.mode` and `um.channels.prefix` properties, do the following:

- When you start a correlator that uses UM, specify the `--umConfigFile` option with the name of your UM configuration properties file name. Do not specify the `--rnames` option when you start the correlator.
- For an IAF adapter that you want to use UM, in its configuration file, in the `<universal-messaging>` element, set the `um-properties` attribute to the name of your UM configuration properties file.

The recommendation is to use the same UM configuration properties file for all Apama components.

When a UM channel is automatically created it has the attributes described in ["Setting up UM for use by Apama" on page 233](#). If you want a UM channel to have any other attributes then you must create the channel in UM before any Apama component sends to or subscribes to the channel.

After Apama looks up a channel name to determine whether it is a UM channel, Apama caches the result and does not look it up again. Consequently, the following situation is possible:

1. You use UM interfaces to create channels.

2. You start a correlator with `um.channels.mode` set to `precreate`.
3. Apama looks up, for example, `channelA` and determines that it is not a UM channel.
4. You use UM interfaces to create, for example, `channelA`.

For Apama to recognize `channelA` as a UM channel, the correlator must be restarted.

Overview of using UM in Apama applications

Considerations for using UM channels

When using UM channels in an Apama application, consider the following:

- Injecting EPL affects only the correlator it is injected into. Be sure to inject into each correlator the event definition for each event that correlator processes. If a correlator sends an event on a channel or receives an event on a channel the correlator must have a definition for that event.
- The UM message bus can be configured to throttle or otherwise limit events, in which case not all events sent to a channel will be processed. See ["Starting correlators that use UM" on page 235](#).
- Only events can be sent or received by means of UM. You cannot use UM for EPL injections, delete requests, engine send, receive, watch or inspection utilities, nor `engine_management -r` requests.
- If you want events to go to only a single correlator it is up to you to design your deployment to accomplish that. If one or more contexts in a particular correlator are the only subscribers to a particular UM channel then only that correlator receives events sent on that channel. However, there is no automatic enforcement of this. In this situation, using the `engine_send` correlator utility might be a better choice than using a UM channel.
- UM channels can be configured for fixed capacity, and that is the default configuration used if the correlator creates a UM channel. This does mean that if a context is sending to a channel while the same context is subscribed to that channel, then if the output queue, channel capacity and the context's input queue are all full the send can deadlock, as the send will hold up processing the next event, but not complete if all queues are full. Similarly, avoid a cycle of contexts and UM channels creating a deadlock.
- It is possible to use the UM client libraries (available for Java, C#, C++ and other languages) to send events to or receive events from Apama correlators and adapters.
- UM is not used by the following:
 - Apama client library connections
 - Correlator utilities such as `engine_connect`, `engine_send` and `engine_receive`
 - Adapter-to-correlator connections defined in the `<apama>` element of an adapter configuration file

While it is not recommended, it is possible to specify the name of a UM channel when you use these Apama interfaces. Even though you specify the name of a UM channel, UM is not used. Events are delivered only to the Apama components that they are directly sent to. This can be useful for diagnostics but mixing connection types for a single channel is not recommended in production.

- It is possible for third-party applications to use UM channels to send events to and receive events from Apama components. See ["About events transported by UM" on page 229](#).

However, the UM configuration described here is intended for communication among Apama correlators and adapters. If third-party applications are using UM to transfer events, then it may be more appropriate to access those events by means of correlator-integrated messaging for JMS, which allows mapping between Apama events and XML payloads on a JMS message bus. If formats other than XML are used, an adapter that translates from those message formats to Apama events can be used. See ["Using Correlator-Integrated Messaging for JMS" on page 150](#).

- The name of an Apama channel can contain any UTF-8 character. However, the name of a UM channel is limited to the following character set:

0-9

a-z

A-Z

/

#

_ (underscore)

- (hyphen)

Consequently, some escaping is required if UM needs to work with an Apama channel name that contains characters that are not supported in UM channel names.

When writing EPL you do not need to be concerned about escape characters in channel names. Apama takes care of this for you.

When interfacing directly with UM, for example in a UM client application for Java, you will need to consider escaping.

When creating UM channels to be used by an Apama application you might need to consider escaping. For example, you might already be using Apama channels whose names contain characters that are unsupported in UM channel names. To use those same channels with UM, you need to create the channels in UM and when you do you must escape the unsupported characters.

The escape sequence is the pound (hash) symbol, followed by the UTF-8 character number in hexadecimal (lowercase), followed by the pound (hash) symbol. For example, the following sequence would be used to escape a period in a channel name:

#2e#

Suppose that in UM you want to create a channel whose name in Apama is `My.Channel`. In UM, you need to create a channel with the following name:

`My#2e#Channel`

Overview of using UM in Apama applications

Setting up UM for use by Apama

For Apama to use the UM message bus, there are some required UM tasks. These steps will be familiar to experienced UM users.

Plan and implement the configuration of the UM cluster that Apama will use. The recommendation is to have at least three UM realms in a cluster because this supports UM quorum rules for ensuring that there is never more than one master in a cluster. However, if you can have only two UM realms you can use the `isPrime` flag to correctly configure a two-realm cluster. For details about configuring a UM cluster see the following topics in the [Universal Messaging documentation](#):

- *Universal Messaging Clusters: Quorum*
- *Universal Messaging Clusters with Sites* which describes an exception to the quorum rule.

To set up UM for use by Apama, do the following for each UM realm to be used by Apama:

1. Install Universal Messaging.
2. Start a UM server.
3. Use UM's Enterprise Manager or UM client APIs to set the access control lists of the UM server to allow the user that the correlator is running on. See UM documentation for details.
4. If you will use the default channel mode, `precreate`, use UM's Enterprise Manager or client APIs to add the channels that Apama will use. For a description of `precreate` behavior, see "[Enabling automatic creation of UM channels](#)" on page 230.

When you add channels set the channel attributes as follows. Together, these attributes provide behavior similar to that provided by using the Apama correlator utility, `engine_connect`.

- Set Channel Capacity to 20000 or some suitable number, at least 2000 events. A number higher than 20,000 would allow larger bursts of events to be processed before applying flow control but would not affect overall throughput.
- Select Use JMS engine. See [Engine Differences](#) in the Universal Messaging documentation.
- Set Honour Capacity when channel is full to true.

These channel attributes provide automatic flow control. If a receiver is slow then event publishers block until the receivers have consumed events.

If you use the `mixed` or `autocreate` channel modes then any channels created by Apama have these attributes.

Other channel attributes are allowed. However, it is possible to set UM channel attributes in a way that might prevent all events from being delivered to all intended receivers, which includes correlators. For example, UM can be configured to conflate or throttle the number of events going through a channel, which might cause some events to not be delivered. Remember that delivery of events is subject to the configuration of the UM channel. Consult the UM documentation for more details before you set channel attributes that are different from the recommended attributes.

5. Ensure that Apama can locate the UM libraries. Do either of the following, whichever is easiest for your deployment.
 - In your UM configuration properties file, set the `um.install.dir` property to the location in the UM installation of the libraries for that platform. This is the properties file that you can specify when you start a correlator (`-UMconfig` option) or in an adapter configuration file (`um-properties` attribute). For example, if you installed UM on a Windows 64 machine in the default location, you would specify:

```
um.install.dir=C:/SoftwareAG/UniversalMessaging/cplusplus/lib/x86_64
```

You can use forward or back slashes on Windows, but if you use back slashes you must escape each one with a back slash: `\\`.

- Before running the correlator, IAF, or an Apama deployment script, add the location of the UM libraries for your platform to your environment's `PATH` or `LD_LIBRARY_PATH`. You can do this either in the environment in which you start the Apama components in or in the system's environment. For example, on Windows you can select Computer > Properties > Advanced System Settings > Environment Variables.

Using Universal Messaging in Apama Applications

Starting correlators that use UM

For a correlator to use UM, you must specify one of the following options when you start the correlator:

- `--rnames list`

Specifies one or more UM realm names (`RNAMES`) separated by commas or semicolons.

Commas indicate that you want the correlator to try to connect to the UM realms in the order in which you specify them here. For example, if you have a preferred local server you could specify its associated `RNAME` first and then use commas as separators between specifications of other `RNAMES`, which would be connected to if the local server is down.

Semicolons indicate that the correlator can try to connect to the specified UM realms in any order. For example, use semicolons when you have a cluster of equally powered machines in the same location and you want to load balance a large number of clients.

You can specify multiple UM realms only when they are connected in a single UM cluster. That is, all `RNAMES` you specify must belong to the same UM cluster. Since channels are shared across a cluster, connecting to more than one UM realm lets you take advantage of UM's failover capability.

You can specify `-r` in place of `--rnames`.

Additional information is available in the

[Universal Messaging documentation](#) in the topics about *Universal Messaging Communication Protocols and RNAMES* and *Universal Messaging Clusters: An Overview*.

- `--umConfigFile path`

Specifies the name or path to a properties file that defines the UM configuration settings for the Apama correlator you are starting. See ["Defining UM properties for Apama applications" on page 238](#).

You can specify `-UMconfig` in place of `--umConfigFile`.

Using Universal Messaging in Apama Applications

Configuring adapters to use UM

If you are configuring your Apama application to use Software AG's Universal Messaging, you can configure an adapter to use the UM message bus to send and receive events. To do this, add a `<universal-messaging>` element to your adapter configuration file. A `<universal-messaging>` element can replace or follow the `<apama>` element.

A `<universal-messaging>` element contains:

- Required specification of the `realms` attribute OR the `um-properties` attribute.
- Required specification of the `<subscriber>` element.
- Optional specification of the `defaultChannel` attribute.
- Optional specification of the `enabled` attribute, which indicates whether the deployed adapter uses the configuration specified in this `<universal-messaging>` element.

Specification of realms or um-properties attribute

Specification of the `realms` attribute OR the `um-properties` attribute is required.

The `realms` attribute can be set to a list of `RNAMEs` (UM realm names) to connect to. You can use commas or semicolons as separators.

Commas indicate that you want the adapter to try to connect to the UM realms in the order in which you specify them here. Semicolons indicate that the adapter can try to connect to the specified UM realms in any order. See ["Starting correlators that use UM" on page 235](#) for more information.

If you specify more than one `RNAME`, each UM realm you specify must belong to the same UM cluster. Specification of more than one UM realm lets you benefit from failover features. See "Communication Protocols and RNAMEs" in the [Universal Messaging documentation](#).

The `um-properties` attribute can be set to the name or path of a properties file that contains UM configuration settings. See ["Defining UM properties for Apama applications" on page 238](#).

Specification of subscriber element

Specification of the `<subscriber>` element is required. The `<subscriber>` element must specify the `channels` attribute. Set the `channels` attribute to a string that specifies the names of the UM channels this adapter receives events from. Use a comma to separate multiple channel names.

Specification of defaultChannel attribute

Specification of the `defaultChannel` attribute is optional. If specified, set the `defaultChannel` attribute to the name of a UM channel. You cannot specify an empty string. In other words, the value of the `defaultChannel` attribute cannot be the default Apama channel, which is the empty string.

An adapter that uses UM must send each event to a named channel. An adapter that is configured to use UM identifies the named channel to use as follows:

1. If the `transportChannel` attribute is set for an event type (in an `<event>` or `<unmapped>` element) then this is the channel the adapter uses for that event type.
2. If the `transportChannel` attribute is not set for an event type but the `presetChannel` attribute is set then this is the channel the adapter uses for that event type.
3. If neither `transportChannel` nor `presetChannel` is set for an event type then the adapter uses the channel set by the `defaultChannel` attribute in the `<universal-messaging>` element.

4. If neither `transportChannel` nor `presetChannel` is set and you did not explicitly set `defaultChannel` and you used Apama Studio to create the adapter configuration file then the `defaultChannel` attribute is set to `"adapter_nameadapter_instance_id"`. For example: `"File Adapter instance 3"`.
5. If none of `transportChannel`, `presetChannel`, or `defaultChannel` are set and if you did not use Apama Studio to create the adapter configuration file then the adapter fails if it tries to use UM.

All events sent by the adapter on channels that are UM channels are delivered to those channels.

Specification of a value for the `defaultChannel` attribute affects events that are sent from this adapter to Apama engine clients, and from this adapter to correlators when the adapter connects to that correlator by means of the `engine_connect` correlator utility.

Specification of the enabled attribute

Optional specification of the `enabled` attribute, which indicates whether the deployed adapter uses the configuration specified in this `<universal-messaging>` element. The default is that the `enabled` attribute is set to `"true"`. If the `enabled` attribute is not specified or if it is set to `"true"` then the configuration specified in the `<universal-messaging>` element is used.

If `enabled` is set to `"false"` then the deployed adapter ignores the `<universal-messaging>` element and does not use UM. The deployed adapter uses only its explicitly set connections.

Subscribing to receive events from an adapter that is using UM

In each context, in any correlator, that is listening for events from an adapter that is using UM, at least one monitor instance must subscribe to the channel or channels on which events are sent from the adapter. For example, if you are using an ADBC adapter, you must include a `monitor.subscribe(channelName)` command for the corresponding instance of the ADBC adapter. Note that not all adapter service monitors support access from multiple correlators. If this is the case, then only one correlator should run the service monitors for that adapter.

Adapter configuration examples

Following are some examples of `<universal-messaging>` elements:

```
<universal-messaging
  realms="nsp://localhost:5629"
  defaultChannel="orders"
  enabled="true">
  <subscriber channels="UK, US, GER"/>
</universal-messaging>
<universal-messaging um-properties="UM-config.properties">
  <subscriber channels="signal,forward"/>
</universal-messaging>
```

Using Universal Messaging in Apama Applications

EPL and UM channels

In an Apama application that is configured to use UM, you write EPL code to subscribe to channels and to send events to channels as you usually do. The only difference is that you cannot specify the default channel (the empty string) when you want to use a UM channel. You must specify a UM channel name to use UM.

A monitor that subscribes to a UM channel causes its containing context to receive events delivered to that channel. There is nothing special you need to add to your EPL code.

Using UM channels makes it easier to scale an application across multiple correlators because UM channels can automatically connect parts of the application as required. If you use the EPL `integer.getUnique()` method remember that the return value is unique for only a single correlator. If a globally unique number is required you can concatenate the result of `integer.getUnique()` with the correlator's physical ID. Obtain the physical ID from the Apama Management interface correlator plug-in with a call to the `getComponentPhysicalId()` method.

See *Using the Management interface in Developing Apama Applications*.

Using Universal Messaging in Apama Applications

Defining UM properties for Apama applications

When you start a correlator and you want that correlator to use UM, instead of specifying the `--rnames` option you might want to specify a file that defines UM properties for your Apama application. This is an Apama file (a standard Java properties file) that lists UM configuration details. You can specify a properties file with the `-UMconfig` option when you start a correlator. See ["Starting correlators that use UM" on page 235](#). This file is separate from the service configuration file that you might specify for the `-Xconfig` option at correlator start-up.

Another place where you can specify a UM properties file is in the `<universal-messaging>` element of an adapter configuration file. See ["Configuring adapters to use UM" on page 235](#).

Apama provides the `UM-config.properties` template file in the `etc` folder of your Apama installation directory. The template is for a standard Java properties file. When you use Apama Studio to add UM configuration to a project, Apama Studio copies the `UM-config.properties` file to the `config` folder in your project. The recommendation is to use one properties file for all Apama components.

A UM properties file for Apama can contain entries for the following properties:

Property name	Description	Default
<code>um.channels.mode</code>	Indicates whether UM channels can be dynamically created. Specify <code>autocreate</code> , <code>mixed</code> , or <code>precreate</code> . See "Enabling automatic creation of UM channels" on page 230 .	<code>precreate</code>
<code>um.channels.prefix</code>	Specifies a prefix for channel names. Channel names must have this prefix to allow dynamic creation.	<code>UM_</code>
<code>um.install.dir</code>	Location in the UM installation of the UM libraries for the platform this properties file is being used on.	None
<code>um.realms</code>	List of <code>RNAME</code> values (URLs). This is the same value you might specify for the <code>--rnames</code> option when you start a correlator. You can use commas or semicolons as separators.	Required

Property name	Description	Default
	<p>Commas indicate that you want the adapter to try to connect to the UM realms in the order in which you specify them here. Semicolons indicate that the adapter can try to connect to the specified UM realms in any order. See "Starting correlators that use UM" on page 235 for more information.</p> <p>Every <code>RNAME</code> you specify must belong to the same UM cluster.</p>	
<code>um.security.certificatefile</code>	Security certificate used to connect to UM.	None
<code>um.security.certificatepassword</code>	Password for the specified security certificate file.	None
<code>um.security.truststorefile</code>	Certificate authority file for verifying server certificate.	None
<code>um.security.user</code>	User name supplied to the UM realm.	Current user name from the operating system

For example, a UM properties file for an Apama installation running on Windows 64 might contain the following:

```
um.realms=nsp://localhost:5629
um.security.user=ckent
um.channels.mode=autocreate
um.install.dir=C:/SoftwareAG/UniversalMessaging/cplus/lib/x86_64
```

The UM configuration file for Apama is encoded in UTF-8.

Using Universal Messaging in Apama Applications

Monitoring Apama application use of UM

You can use UM Enterprise Manager or UM APIs to find out about

- Which correlators are subscribed to which UM channels
- The number of events flowing through a UM channel
- The contents of the events going through a UM channel

See [Universal Messaging documentation](#) for Enterprise Manager.

To monitor and manage Apama components, you must use Apama tools and APIs.

Using Universal Messaging in Apama Applications

III

Developing Custom Adapters

This documentation describes how to use the Apama Integration Adapter Framework (IAF). The IAF is a middleware-independent and protocol-neutral adapter tailoring framework designed to provide for the easy and straightforward creation of software adapters to interface Apama with middleware buses and other message sources. It provides facilities to generate adapters that can communicate with third-party messaging systems, extract and decode self-describing or schema-formatted messages, and flexibly transform them into Apama events. Vice-versa, Apama events can be transformed into the proprietary representations required by third-party messaging systems. It provides highly configurable and maintainable interfaces and semantic data transformations. An adapter generated with the IAF can be re-configured and upgraded at will, and in many cases, without having to restart it. Its dynamic plug-in loading mechanism allows a user to customize it to communicate with proprietary middleware buses and decode message formats.

The information in this book is organized as follows:

- ["The Integration Adapter Framework" on page 241](#) describes the architecture of the IAF.
- ["Using the IAF" on page 247](#) describes how to start, stop, and manage the IAF runtime. It also describes the IAF sample applications.
- ["C/C++ Transport Plug-in Development" on page 277](#) presents the C/C++ Transport Plug-in Development Specification and describes how to implement transport layer plug-ins in C and C++.
- ["C/C++ Codec Plug-in Development" on page 291](#) presents the C/C++ Codec Plug-in Development Specification and describes how to implement codec layer plug-ins in C and C++.
- ["C/C++ Plug-in Support APIs" on page 311](#) describes programming interfaces provided with the Apama software that may be useful in implementing transport layer and codec plug-ins for the IAF.
- ["Transport Plug-in Development in Java" on page 317](#) presents the Transport Plug-in Development Specification for Java and describes how to implement transport layer plug-ins in Java.
- ["Java Codec Plug-in Development" on page 326](#) presents the Codec Plug-in Development Specification for Java and describes how to implement codec layer plug-ins in Java.
- ["Plug-in Support APIs for Java" on page 339](#) describes the Java interface for recording status and error log messages from the IAF runtime and any plug-ins loaded within it.
- ["Monitoring Adapter Status" on page 344](#) describes how to provide status information about IAF adapters.
- ["Out of Band Connection Notifications" on page 358](#) describes the set of out of band events that can be used to notify adapters when sender and receiver components connect to or disconnect from the IAF.
- ["The Event Payload" on page 363](#) describes how to add a payload field to an Apama event type. A payload field is used to accommodate data that does not comply with the rigid structure of an Apama event.

Chapter 7: The Integration Adapter Framework

■ Overview	241
■ Architecture	242
■ The transport layer	244
■ The codec layer	244
■ The Semantic Mapper layer	245
■ Contents of the IAF	246

There are two ways of integrating with Apama through software. The first is to use the low-level Client Software Development Kits (SDKs) for C, C++ or Java to write your own custom software interface. The second is to instantiate an adapter with the higher-level Integration Adapter Framework (IAF). The information in *Developing Custom Adapters* describes how to use the IAF.

For information on using the client SDKs, see "[Developing Custom Clients](#)" on page 365.

Developing Custom Adapters

Overview

The IAF is a middleware-independent and protocol-neutral adapter tailoring framework. It is designed to allow easy and straightforward creation of software adapters to interface Apama with middleware buses and other message sources. It provides facilities to generate adapters that can communicate with third-party messaging systems, extract and decode self-describing or schema-formatted messages, and flexibly transform them into Apama events. In the opposite direction, Apama events can be transformed into the proprietary representations required by third-party messaging systems. It provides highly configurable and maintainable interfaces and semantic data transformations. An adapter generated with the IAF can be re-configured and upgraded at will, and in many cases, without having to restart it. Its dynamic plug-in loading mechanism allows a user to customize it to communicate with proprietary middleware buses and decode message formats.

On the other hand, the SDKs provide lower-level client application programming interfaces that allow one to directly connect to Apama and transfer Apama Event Processing Language (EPL) code and events in and out. (The Apama Event Processing Language is the new name for MonitorScript.) The SDKs provide none of the abstractions and functionality of the IAF, and hence their use is only recommended when a developer needs to write a highly customized and very high performance software client, or wishes to integrate existing client code with Apama in process.

The IAF is available on all platforms supported by Apama, although not all adapters will work on all platforms. For the most up-to-date information about supported platforms and compilers, see Software AG's Knowledge Center in Empower at <https://empower.softwareag.com>.

Architecture

The first step in integrating Apama within a user environment is to connect the correlator to one or more message/event sources and/or sinks. In the majority of cases the source or provider of messages will be some form of middleware message bus although it could also be a database or other storage based message source, as well as an alternative network-based communication mechanism, like a trading system. The same applies for the sink or consumer of messages.

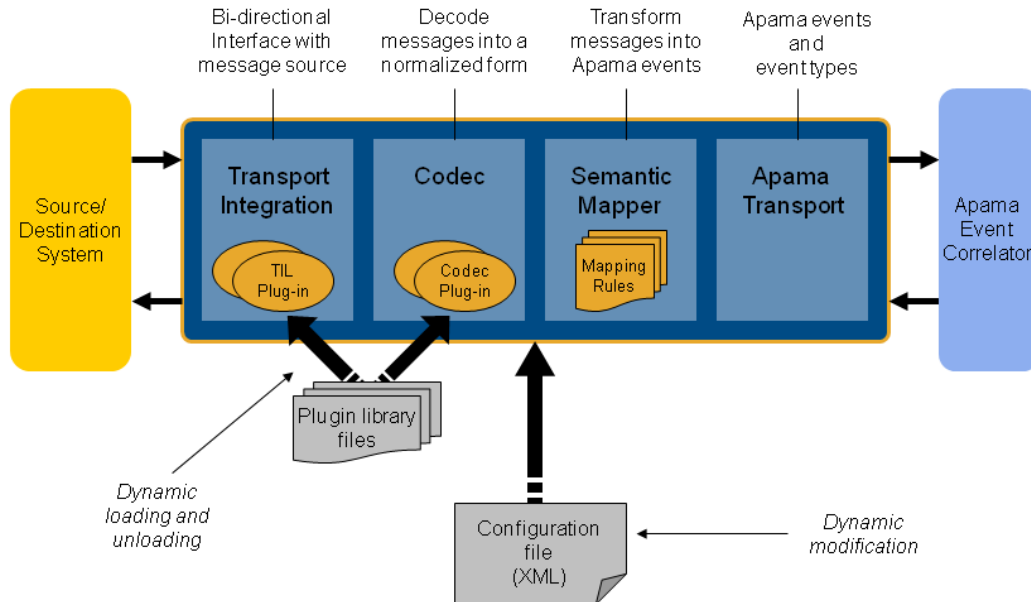
These message sources and/or sinks vary extensively. Typically each comes with its own proprietary communications paradigm, communications protocol, message representation and programming interfaces. Interfacing any software like Apama with a source/sink of messages like a message bus therefore requires the writing of a specialized software adapter or connector, which needs to be maintained should the messaging environment or the message representation change. The adapter needs to interface with the messaging middleware or message source, receive messages from it and decode them, and then transform them into events that Apama can understand and process. The latter transformation is not always straightforward, as the message representation might vary from message to message and require semantic understanding.

Conversely, the events generated by Apama need to be processed in the inverse direction and eventually end up back on the message bus.

Note: In the Apama documentation, a message traveling from a message source through the IAF and into Apama is described as traveling *downstream*, whereas a message output as an alert from Apama and progressing back out through the IAF towards a message sink is described as traveling *upstream*.

In order to facilitate development of software adapters, Apama provides the IAF. In contrast to the SDKs, the IAF is not a programming library. The IAF is effectively a customizable, middleware independent, generic adapter that can be adapted by a user to communicate with their middleware and apply their specific semantic transformations to their messages.

Figure 2. The architecture of an IAF adapter



As illustrated above, the IAF acts as the interface between the messaging middleware and the event correlator. There are four primary components to the IAF:

- *The Transport Layer.* This is the layer that communicates with the user's message source/sink. Its functionality is defined through one or more Apama or user-provided message source/sink specific transport plug-ins, written in C, C++ or Java.
- *The Codec Layer.* The codec layer translates messages from any custom representation into a normalized form and vice-versa. The transformation is carried out by one of its codec plug-ins. These can be provided by Apama or by the user, and may be written in C, C++ or Java. Note that Java codec plug-ins may only communicate with Java transport plug-ins, and C/C++ codec plug-ins with C/C++ transport plug-ins.
- *The Semantic Mapper.* This layer provides functionality to transform the messages received from the message source into Apama events. The Semantic Mapper is a standard component that is configured for use with a particular adapter by means of a set of semantically rich translation mapping rules. These rules define both how to generate Apama events from externally generated messages and how user-custom messages for an external destination may be generated from Apama events.

An adapter can be configured to bypass this kind of mapping in the Semantic Mapper. Used this way, the Semantic Mapper converts the string form of an Apama event directly to a normalized form and vice-versa.

- *The Apama Interface layer.* This layer abstracts away communication with Apama's event correlator. It injects EPL definitions and event instances into the event correlator and asynchronously receives back events from it.

Additionally, the `engine_send` and `engine_receive` tools can be run against the IAF simply by supplying the port on which the IAF is running. For example, running

```
engine_receive -p 16903
```

connects to the IAF running on the default port and receives all event emitted by it.

The next sections explore the transport, codec and Semantic Mapper layers in more detail.

The transport layer

The *transport layer* is the front-end of the IAF. The transport layer's purpose is to communicate with an external message source and/or sink, extracting downstream messages from the message source ready for delivery to the codec layer, and sending Apama events already encoded by the codec layer on to the message sink.

This layer interfaces with the middleware message bus or message source/sink through the latter's custom programming interface. It receives and dispatches messages from and to it as well as carrying out other proprietary operations. Depending on the nature of the message bus or message source in use, these operations could include opening a database file and running SQL queries on it, registering interest in specific message topics with a message bus, or providing security credentials for authentication. Note that if the IAF is also being used to output messages back to a message sink, then it must also carry out the operations required to enable this; for example, opening and writing to a database file, or dispatching messages onto a message bus.

As this functionality depends entirely on the message source and/or sink the IAF needs to interface with, the transport layer's functionality is loaded dynamically through a custom transport plug-in.

Although Apama provides a set of standard transport plug-ins for popular messaging systems, the user may develop new transport plug-ins. See "[C/C++ Transport Plug-in Development](#)" on [page 277](#) and "[Transport Plug-in Development in Java](#)" on [page 317](#) for the Transport plug-in Development Specifications for C/C++ and Java, which describe how custom transport plug-ins may be developed in the C, C++ and Java programming languages.

The transport layer can contain one or more transport layer plug-ins. These are loaded when the adapter starts, and the set of loaded plug-ins can be changed while the adapter is running. In addition, a loaded plug-in may be re-configured at any time using the IAF Client tool. If a transport plug-in requires startup or re-configuration parameters, these need to be supplied in the IAF configuration file as documented in "[The IAF configuration file](#)" on [page 255](#).

Because a transport layer plug-in effectively implements a custom message transport, this manual uses the terms *transport layer plug-in* and *event transport* interchangeably.

The codec layer

While the transport layer communicates with the custom message sources and sinks and extracts messages, such as stock trade data, from them, the responsibility of the codec layer is to correctly interpret and decode each message into a 'normalized' format on which the semantic mapping rules can be run; similarly in the upstream direction a codec may be responsible for encoding a normalized message in an appropriate format for transmission by particular transport(s).

Message sources like middleware message buses typically use proprietary representation of messages. Messages might appear as strings (possibly human readable or otherwise encoded) or sequences of binary characters. Messages might also be self-describing (possibly in XML or through

some other proprietary descriptive format) or else be structured according to a schema available elsewhere.

Producing a universal generic normalized format from these messages requires the codec layer to understand the particular format of the messages. In the upstream direction the codec layer needs to encode the messages correctly according to the destination message sink.

As with the transport layer, in order to enable this custom functionality, the IAF is designed to dynamically load codec plug-ins that are capable of decoding and encoding the messages being received, when supplied with any required configuration properties. Apama provides some generic codec plug-ins, such as the `StringCodec` codec. This can decode most string based name-value representations of messages once it is configured with the syntactic elements used to delimit the elements in a message. In addition the user may develop proprietary codec plug-ins. See "[C/C++ + Codec Plug-in Development](#)" on page 291 and "[Java Codec Plug-in Development](#)" on page 326 for the Codec plug-in Development Specifications for C/C++ and for Java, which describe how custom codec plug-ins may be developed using the C, C++ and Java programming languages.

An adapter can load multiple codec plug-ins (to deal with different message types). These are loaded at startup and the set of loaded codecs can be changed while the adapter is running. Individual codec plug-ins may also be re-configured at any time. If a codec plug-in requires startup or re-configuration parameters, these need to be supplied in the IAF configuration file as documented in "[The IAF configuration file](#)" on page 255.

This manual uses the terms *codec layer plug-in* and *event codec* interchangeably.

The Semantic Mapper layer

The Semantic Mapper maps and transforms incoming messages into Apama events that can be passed into the event correlator. Conversely, it can accept incoming Apama events and map them into messages that can be sent upstream on the user's message sink.

Apama events are rigidly defined. Every event must be structured according to a well-defined type definition. Therefore all events are of a specific named type, where this defines the number of fields (or parameters) in the event, their order, the name of each field, and its type. Furthermore it is possible to define which fields are relevant for querying in EPL event expressions, and which are not. See *Developing Apama Applications* for further information on event type definitions and EPL event expressions. This rigorous format permits the event correlator to be highly optimized and contributes towards Apama's scalable performance.

The source messages that are to be passed into Apama as events (or the sink messages that Apama needs to generate) might match this specification, in which case the mapping will be straightforward. However, they might also differ in several ways, some of which are listed here:

- The messages might be self-describing, and need to be parsed in order to deduce what fields they contain.
- The fields contained in every message might appear in varying order.
- Some messages of different types and with differing sets of fields might reflect the same information but in a different format (e.g. trade events from different markets or news headlines from different sources).
- The set of fields contained in messages might differ even if the messages are all of the same type.

- The messages might not be of an obvious type, and their nature (e.g. a trade event or a news headline) might need to be deduced from their contents.
- The set of fields might be enhanced over time to capture additional information.
- Some messages might have fields that are completely irrelevant.
- Some messages might have fields that are irrelevant for matching on but might be useful otherwise.

In order to address these conditions and allow meaningful Apama events to be created from external messages, the Semantic Mapper supports a semantically rich set of translation and transformation rules. These need to be expressed in the IAF configuration file.

The rules available are described in ["Event mappings configuration" on page 259](#).

You can configure an adapter so that some events bypass this kind of mapping in the Semantic Mapper. Instead of mapping each field in an incoming event to a field in an Apama event or the converse, the entire event is treated as a string in a single field.

Contents of the IAF

The Integration Adapter Framework is included when you select Developer or Server during the Apama installation.

The Integration Adapter Framework contains the following components:

- Core files – these include the IAF Runtime, the management tools and the libraries they require.
- Example adapters written in C and Java – this includes the complete sources of the `FileTransport/JFileTransport` transport layer plug-ins and the `StringCodec/JStringCodec` plug-ins, sample configuration files, a file with a set of input messages, an EPL file with a sample application, and a set of reference result messages.

There is also a set of Market examples written in C and Java – these provide access to streaming prices for `Depth` and `Tick` and a facility to place orders, on which executions are reported. The adapter also reports its status (whether it is connected or not, or if the IAF process has been stopped). This is used by the subscription and order management services. A sample server implemented in Python is included. It requires Python 2.4 or later. Refer to the README file in the `samples\iaf_plugin\market` directory for more information and instructions on how to build and run the Market examples.

- A suite of development materials – these include the C/C++ header files and Java API sources required to develop transport and codec layer plug-ins for both languages. Also included is a skeleton transport and codec plug-in in C, the IAF configuration file XML Document Type Definition (DTD), a makefile for use with GNU Make on UNIX, and a 'workspace' file for use with Microsoft's Visual Studio.NET on Microsoft Windows.

Chapter 8: Using the IAF

■ The IAF runtime	247
■ IAF Management – Managing a running adapter I	251
■ IAF Client – Managing a running adapter II	254
■ IAF Watch – Monitoring running adapter status	255
■ The IAF configuration file	255
■ IAF samples	274

This section describes how to start and manage the Integration Adapter Framework and how to specify an adapter's configuration file.

Developing Custom Adapters

The IAF runtime

Once installed, running the IAF is straightforward. As already stated, the IAF is not a development library but a generic adapter framework whose functionality can be tailored according to a user's requirements through loading of the appropriate plug-ins.

In order to create an adapter with the IAF, one must supply a configuration file. This file – described in "[The IAF configuration file](#)" on page 255 – specifies which plug-ins to load and what parameters to configure them with, defines the translation and transformation rules of the Semantic Mapper, and configures communication with Apama.

The adapter can then be started as follows:

```
> iaf configuration.xml
```

IAF library paths

In order for the IAF to successfully locate and load C/C++ transport layer and codec plug-ins, the location(s) of these must be added to the environment variable `LD_LIBRARY_PATH` on UNIX, or `PATH` on Windows.

A transport or codec plug-in library may depend on other dynamic libraries, whose locations should also be added to the `LD_LIBRARY_PATH` or `PATH` environment variable as appropriate for the platform. The documentation for a packaged adapter will state which paths should be used for the adapter's plug-ins. Note that on the Windows platform, the IAF may generate an error message indicating that it was unable to load a transport or codec plug-in library, when in fact it was a dependent library of the plug-in that failed to load. On UNIX platforms the IAF will correctly report exactly which library could not be loaded.

When using the IAF with a Java adapter the location of the Java Virtual Machine (JVM) library is determined in the same way. On UNIX systems the `LD_LIBRARY_PATH` environment variable will be searched for a library called `libjvm.so`, and on Windows the IAF will search for `jvm.dll`, first in the

`third_party\jre\bin\server` and `third_party\jre\bin` directories of the Apama installation referenced by the `APAMA_HOME` environment variable, then in any other directories on the `PATH` environment variable. Using a JVM other than the one shipped with Apama is not supported and Technical Support will generally request that any Java-related problems with the IAF are reproduced with the supported JVM.

To develop, build, and test an Apama application, the recommendation is that you use Oracle JDK 8. The minimum you can use is JDK 7.

To deploy an Apama application, the recommendation is that you use Oracle JRE 8, which is the version that Apama provides. Use of any JRE other than the one that Apama ships with is discouraged.

See ["Java configuration \(optional\)" on page 273](#) for information about how the location of Java plug-in classes are determined.

IAF command line options

The complete usage information for the executable `iaf` (on UNIX) or `iaf.exe` (on Windows) is as follows. This can be displayed at any time by launching the IAF with the `--help` option.

```
Usage: iaf [ options ] [ config.xml ]
Where config.xml is the name of a configuration file using the format
described in the Integration Adapter Framework documentation.
Options include:
  -V | --version          Print program version info
  -h | --help             This message
  -p | --port <port>     Port to listen for commands on (default is 16903)
  -f | --logfile <file>  Log to named file (default is stderr)
  -l | --loglevel <level> Set logging verbosity. Available levels
                        are TRACE, DEBUG, INFO, WARN, ERROR, FATAL, CRIT and OFF.
                        The default logging level is INFO.
  -t | --truncat         Truncate the log file
  -N | --name <name>     Set the component name
  -e | --events           Dump event definitions to stdout then exit
  --logQueueSizePeriod <p> Send info to log every <p> seconds
  -Xconfig | --configFile <file> Use service configuration file <file>
```

Note that a configuration file must be provided unless the `-h` or `-V` options are used.

Unless `-e` or `--events` are used, the above will generate and start a custom adapter, load and initialize the plug-ins defined in the configuration file, connect to Apama, and start processing incoming messages.

When the `-e` or `--events` command line switches are used, `iaf` generates event definitions that can be saved to a file and injected during your application's startup sequences as specified by Apama Studio, the Enterprise Management and Monitoring console (EMM), or Apama command line tools. If either of these switches is used, the IAF will load the IAF configuration file, process it, generate the event definitions and print them out onto `stdout` (standard output) and promptly exit. A valid configuration file must be supplied with either of these switches. The output definitions are grouped by package, with interleaved comments between each set. If all the event types in the configuration are in the same package, the output will be valid EPL code that can be injected directly into the correlator. Otherwise, it will have to be split into separate files for each package. The IAF can be configured to automatically inject event definitions into a connected correlator, but this is not the default behavior. The event definitions generated by the `-e` or `--events` options are exactly what the IAF would inject into the correlator, if configured to do so.

The `-Xconfig` and `--configFile` are reserved for usage under guidance by Apama support. For more information about the service configuration file, see .

If the `--logfile` and `--loglevel` command line switches are provided, any logging settings set in the IAF configuration file (`<logging>` and `<plugin-logging>`) will be ignored.

If the IAF cannot write to the log file specified either with the `--logfile` option or in the adapter's configuration file, the IAF will fail to start.

IAF log file status messages

The IAF sends status information to its log file every 5 seconds (the default behavior) or at time intervals you specify with the `--logQueueSizePeriod` option when you start the IAF. IAF status information contains the following elements:

Status line element	Description
ApEvRx	Number of Apama events received since the IAF started. These events were received from the correlator that the IAF is connected to.
ApEvTx	Number of Apama events sent since the IAF started. These events were sent to the correlator that the IAF is connected to.
TrEvRx	Number of events received by all transports in the IAF since the IAF started. These events were received from user-defined sources outside the correlator.
TrEvTx	Number of events sent from all transports in the IAF since the IAF started. These events were sent to user-defined targets outside the correlator.
vm	Number of kilobytes of virtual memory being used by the IAF process.
si	The rate (pages per second) at which pages are being read from swap space.
so	The rate (pages per second) at which pages are being written to swap space.

For example:

```
Status: ApEvRx=589 ApEvTx=2056000 TrEvRx=2056008 TrEvTx=587 vm=407200 si=0.0 so=0.0
```

IAF log file rotation

Rotating the IAF log file refers to closing the IAF log file while the IAF is running and opening a new log file. This lets you archive log files and avoid log files that are too large to easily view.

Each site should decide on and implement its own IAF log rotation policy. You should consider

- How often to rotate log files
- How large an IAF log file can be
- What IAF log file naming conventions to use to organize log files.

There is a lot of useful header information in the log file being used when the IAF starts. If you need to provide log files to Apama technical support, you should be able to provide the log file that was in use when the IAF started, as well as any other log files that were in use before and when a problem occurred.

On Windows, to automate log file rotation, you can set up scheduled tasks that run the following utilities:

- The following command instructs the IAF to close the log file it is using and start using a log file that has the name you specify. When you run this request to rotate the log file the IAF log file has a new name each time you rotate it. This is because Windows does not let you change the name of a file that is being used. Be sure to enclose the argument after `-r` in quotation marks.

```
iaf_management -r "setLogFile new-log-filename"
```

- You can configure the IAF to log to two separate files. Each command instructs the IAF to start using the specified log file for either the IAF core processes (generic IAF information such as status messages) or the IAF plug-in processes (transports and codecs being used). Be sure to enclose the argument after `-r` in quotation marks.

```
iaf_management -r "setCoreLogFile new-log-filename"
iaf_management -r "setPluginLogFile new-log-filename"
```

Consider using two IAF log files when you need to focus on diagnosing something specific to your application, for example, you need to easily spot authentication messages. If you do use separate log files you might want to rotate them at the same time so that they stay in sync with each other.

On UNIX, to automate log file rotation, you can write a `cron` job that periodically does any of the following:

- Set log file name

```
iaf_management -r "setLogFile new-log-filename"
```

- Set core log file and plug-in log file

```
iaf_management -r "setCoreLogFile new-log-filename"
iaf_management -r "setPluginLogFile new-log-filename"
```

- Reopen the log

```
iaf_management -r "reopenLog"
```

Move the IAF log file before you execute the `reopenLog` request. Since UNIX allows you to rename a file that is in use, the IAF processes will log to the renamed log file. When you then request the IAF to reopen its log file the IAF creates a new log file with the same name. For example, suppose you move `iaf_current.log` to `iaf_archive_2014_01_31.log` and then send a `reopenLog` request. The IAF creates `iaf_current.log`, opens it, and begins sending any log messages to it. Be sure to enclose the argument after `-r` in quotation marks.

If you are using two IAF log files, the `reopenLog` request applies to both of them. Consequently, you want to move both log files before you issue the `reopenLog` request.

- Send a `SIGHUP` signal

You can write a `cron` job that sends a `SIGHUP` signal to IAF processes. The standard UNIX `SIGHUP` mechanism causes IAF processes to re-open their log files.

The `cron` job should first rename log files. Since UNIX allows you to rename a file that is in use, the IAF processes will log to the renamed log files until the `cron` job sends a `SIGHUP` to IAF processes. The `SIGHUP` signal makes the processes re-open their log files and so they open files that

have the old names and begin using them. Of course, these files are initially empty because the IAF must re-create them.

Sending a `SIGHUP` signal does the same thing as the `reopenLog` request. Also, a `SIGHUP` signal forces the IAF configuration file to be reloaded and this reload stops and starts the transports and codecs.

If you instruct the IAF to open a named log file and the IAF cannot open that log file or cannot write to that log file, the IAF sends log messages to `stderr` but does not generate an error.

Apama does not support automatic log file rotation based on time of day or log file size.

IAF Management – Managing a running adapter I

The IAF Management tool is provided for performing generic component management operations on a running adapter. It can be used to shut down a running adapter, request the process ID of a running adapter, or check that an adapter process is running and acknowledging communications. Any output information is displayed on `stdout`.

See also ["IAF Client – Managing a running adapter II" on page 254](#) for IAF-specific management information, as opposed to this generic component management tool.

The executable for interfacing with the IAF's management interface status is `iaf_management.exe` (on Windows) or `iaf_management` (on UNIX). When it is run with the `-h` command line option it displays the following usage information:

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

IAF Management options

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

Table 5. IAF Management options

<code>-V</code>	Displays version information for the <code>iaf_management</code> tool.
<code>-h</code>	Displays usage information for running the <code>iaf_management</code> tool.
<code>-v</code>	Displays information in a more verbose manner.
<code>-n host</code>	Name of the host of the component that you want to connect to. The default is <code>localhost</code> . You cannot use non-ASCII characters in the host name.
<code>-p port</code>	Port on which the adapter is listening. The default is <code>16903</code> .
<code>-w</code>	Instructs the <code>iaf_management</code> tool to wait for the component to start. This option is similar to the <code>-W</code> option, except that the <code>-w</code> option instructs the <code>iaf_management</code> tool to wait forever. The <code>-W</code> option lets you specify how many seconds to wait. See the information for the <code>-W</code> option for an example.
<code>-W num</code>	Instructs the <code>iaf_management</code> tool to wait <i>num</i> seconds for the component to start. If the component is not ready before the specified number of seconds has elapsed, the <code>iaf_management</code> tool terminates with an exit code of 1.
<code>-N</code>	Displays the name of the component.
<code>-T</code>	Displays the type of the component that the <code>iaf_management</code> tool connects to.
<code>-Y</code>	Displays the physical ID of the component. This can be useful if you are looking at log information that identifies components by their physical IDs.
<code>-L</code>	Displays the logical ID of the component. This can be useful if you are looking at log information that identifies components by their logical IDs.
<code>-O</code>	Displays the log level of the component. The returned value is one of the following: <code>TRACE</code> , <code>DEBUG</code> , <code>INFO</code> , <code>WARN</code> , <code>ERROR</code> , <code>CRIT</code> , <code>FATAL</code> , or <code>OFF</code> .
<code>-C</code>	Displays the version of the component.
<code>-R</code>	Displays the product version of the component. For example, when the tool connects to a correlator, it displays the version of the Apama software that is running.
<code>-B</code>	Displays the build number of the component. This information is helpful if you need technical support. It indicates the exact software contained by the component you connected to.

-F	Displays the build platform of the component. This information is helpful if you need technical support. It indicates the set of libraries required by the component you connected to.
-P	Displays the process ID of the correlator you are connecting to. This can be useful if you are looking at log information that identifies components by their process ID.
-H	Displays the host name of the component. When debugging connectivity issues, this option is helpful for obtaining the host name of a component that is running behind a proxy or on a multihomed system.
-U	Displays the user name of the component. On a multiuser machine, this is useful for determining who owns a component.
-D	Displays the working (current) directory of the component. This can be helpful if a plug-in writes a file in a component's working directory.
-E	Displays the port of the component.
-c	This option is for use by technical support. It displays all the connections to the component.
-a	Displays all information for the component.
-xs	Disconnect sender with physical id <i><id></i>
-xr	Disconnect receiver with physical id <i><id></i>
-i <i>category</i>	This option is for use by technical support. It displays component-specific information for the specified category.
-d	Ping the component. This confirms that the component process is running and acknowledging communications.
-l <i>level</i>	Sets the amount of information that the component logs. In order of decreasing verbosity, you can specify <code>TRACE</code> , <code>DEBUG</code> , <code>INFO</code> , <code>WARN</code> , <code>ERROR</code> , <code>FATAL</code> , <code>CRIT</code> , or <code>OFF</code> .
-r <i>req</i>	This option is for use by technical support. It sends a component-specific request.
-s <i>why</i>	Instructs the component to shut down and specifies a message that indicates the reason for termination. The component inserts the string you specify in its log file with a <code>CRIT</code> flag, and then shuts down.

IAF Management exit status

The following exit values are returned:

Key	Value
0	All status requests were processed successfully
1	No connection to the adapter was possible or the connection failed
2	Other error(s) occurred while requesting/processing status
3	Deep ping failed

IAF Client – Managing a running adapter II

The IAF Client tool is provided for performing IAF-specific management operations on a running adapter. It can be used to stop a running adapter, to temporarily pause sending of events from an adapter into the event correlator, and to request an adapter to dynamically reload its configuration.

The executable for this tool is `iaf_client` (on UNIX) or `iaf_client.exe` (on Windows). The complete usage information (as displayed when it is run with the `--help` option) is as follows:

```
Usage: iaf_client [ options ]
Where options include:
  -h | --help           This message
  -n | --hostname <host> Connect to an IAF on <host>
  -p | --port <port>    IAF is listening on <port>
  -r | --reload         Tell the IAF to reload its config
  -s | --suspend        Tell the IAF to suspend event sending
  -t | --resume         Tell the IAF to resume event sending
  -q | --quit           Tell the IAF to shut down
  -v | --verbose        Be more verbose
  -V | --version        Print program version info
  One of --reload, --quit, --suspend, -q, -r or -s must be specified
```

Note that if the adapter is listening for control connections on a non-standard port (specified with the `--port` option to the `iaf` program), you must pass the same port number to `iaf_client`.

Also, can only use ASCII characters to specify the name of the host.

Suspend and resume

The `--suspend` and `--resume` options control the sending of events to the correlator, not to the external transport.

Adapter reload

The `--reload` option is worthy of further explanation. Using `--reload` it is possible to dynamically reconfigure a running adapter from a changed configuration file without restarting the IAF.

When the IAF is started, it loads all the transport and codec plug-ins defined in its configuration file, and initializes them with any plug-in-specific properties provided.

When an adapter is reconfigured using `--reload`, the IAF will:

- Pass the current set of `<property>` names and values in the configuration file to each loaded transport and codec layer plug-in.

Note: Although plug-in authors will support dynamic reconfiguration of properties wherever possible, it is important to be aware that there may be some properties that by the nature can not be

changed while the adapter is still running. These should be detailed in the documentation for the transport or codec plug-in. Some transport and codec plug-ins may not support configuration file reloading at all; this should be documented by the specific plug-ins.

- Load and initialize any new transport and codec layer plug-ins that have been listed in the `<transports>` and `<codecs>` sections of the configuration file.
- Unload any transport and codec layer plug-ins that are no longer listed in the `<transports>` and `<codecs>` sections of the configuration file.

Changing the `name` of a running plug-in and performing a reload is equivalent to unloading the plug-in and then loading it again. It is important to realize that this will result in any runtime state stored in memory by the plug-in being lost.

Note: It is not possible to dynamically change a loaded plug-in's C/C++ `library` filename or Java `className`, nor to change a C/C++ plug-in into a Java one (or vice-versa).

If an adapter is reconfigured to use a different log file and the IAF cannot write to the new log file when reloaded, the IAF uses the log file the adapter was using before reconfiguring. If the IAF cannot use the original log file, it writes to `stderr`.

IAF Watch – Monitoring running adapter status

The IAF Watch tool allows one to monitor the live status of a running adapter. This is available as `iaf_watch` (on UNIX) or `iaf_watch.exe` (on Windows).

The usage information for this tool, when run with the `--help` option, is as follows:

```
Usage: iaf_watch [ options ]
Where options include:
  -h | --help           This message
  -n | --hostname <host> Connect to an IAF on <host>
  -p | --port <port>    IAF is listening on <port>
  -i | --interval <ms> Poll every <ms> milliseconds
  -f | --filename <file> Write to <file> instead of to standard out
  -r | --raw            Raw (i.e. parser friendly) output
  -t | --title          Add header to output (Raw mode only)
  -o | --once           Poll once then exit
  -v | --verbose        Be more verbose
  -u | --utf8           Write output in utf8
  -V | --version        Print program version info
```

By default the tool will collect status from the adapter once per second and display this in a human-readable form. Note that if the adapter is listening for control connections on a non-standard port (specified with the `--port` option to the `iaf` program), you must pass the same port number to `iaf_watch`.

Note: You should only use ASCII characters to specify the name of a host.

The IAF configuration file

An IAF configuration file is an essential part of any adapter generated with the IAF.

The configuration file must be formatted in XML, and the Document Type Definition (DTD) for it is `iaf_4_0.dtd`, which is located in the `etc` directory of your installation. You may wish to use this DTD in conjunction with your XML editor to assist you in writing a correctly formatted configuration file.

The configuration file is loaded and processed when the adapter process starts. A running adapter can be signaled to reload and reprocess the configuration file at any time, by running the `iaf_client` tool with the `-r` or `--reload` option.

On UNIX platforms a `SIGHUP` signal sent to the IAF process re-opens the log file.

The root element in the configuration file is `<adapter-config>`. This must always be defined. Within it a single instance of the following elements must exist:

- `<transports>` - This element defines the transport layer plug-in(s) to be loaded.
- `<codecs>` - Defines the codec layer plug-in(s) to be loaded.
- `<mapping>` - Defines the mapping rules for the Semantic Mapper layer, which are used in the conversion between codec layer normalized messages and correlator events.

Following those elements, there must be at least one of the following elements. It is also possible to specify one of each of these elements:

- `<apama>` - Defines how the IAF connects to the Apama event correlator(s).
- `<universal-messaging>` - Defines how the IAF connects to Software AG's Universal Messaging message bus.

There are also three optional elements that can appear before these required elements (in order):

- `<logging>` - Defines the log file and logging level used by the IAF.
- `<plugin-logging>` - Defines the log file and logging level used by the transport and codec layer plug-ins.
- `<java>` - Defines the environment of the embedded Java Virtual Machine (JVM) in which any Java codec and transport plug-ins will run.

Each of these elements is discussed in more detail in the following sections.

Including other files

The adapter configuration file supports the XML `XInclude` extension so you can reference other files from the configuration file. This makes it possible, for example, to keep the transport properties in one file and the mapping properties in another. For more information on XML Inclusions, see <https://www.w3.org/TR/xinclude/>. The standard adapters packaged with the Apama installation use this scheme. For example, the Apama ODBC adapter specifies its transport properties in the `adapters\config\ODBC.xml.dist` file and its mapping properties in the `adapters\config\ODBC-static.xml`. For more information on the standard Apama adapters, see ["Using the Apama File Adapter" on page 106](#).

In order to match the DTD, the `xmlns:xi` attribute must be placed either on the `<adapter-config>` element (as the name `xmlns:xi`) or on the `<xi:include>` element. Apama strongly recommends that you use only relative filenames instead of URLs to remote servers.

For example:

```
<adapter-config xmlns:xi="http://www.w3.org/2001/XInclude">
  <transports>
    <transport name="testmarket1" library="protocol-transport">
      <property name="host" value="localhost"/>
      <property name="port" value="12000"/>
    </transport>
  </transports>
</adapter-config>
```

```

    </transport>
</transports>
<xi:include href="market-static.xml"
    xpointer="xpointer(/static/codecs)"/>
<xi:include href="market-static.xml"
    xpointer="xpointer(/static/mapping)"/>

```

Transport and codec plug-in configuration

The adapter configuration file requires both a `<transports>` and a `<codecs>` element.

The `<transports>` element defines the transport layer plug-in(s) to be loaded, and contains one or more nested `<transport>` elements, one for each plug-in.

The syntax of the `<codecs>` element mirrors the `<transports>` element precisely, and contains one or more nested `<codec>` elements, each of which defines a codec layer plug-in to be loaded.

The `<transport>` and `<codec>` elements

The transport or codec layer plug-in that should be loaded is defined by the attributes of the `<transport>` or `<codec>` elements:

- To load a C or C++ plug-in, there must be a `library` attribute, whose value is the filename of the library in which the plug-in is implemented. The extension and library name prefix will be deduced automatically based on the platform the IAF is running on. For example, on Windows `library="FileTransport"` would reference a file called `FileTransport.dll`; on a UNIX system the library filename would be `libFileTransport.so`.
- To load a Java plug-in, instead provide a `className` attribute, whose value is the fully qualified name of the Java class that implements the plug-in.

If the optional `jarName` attribute is also provided, the plug-in class will be loaded from the Java archive (`.jar`) that it specifies; otherwise the IAF will use the usual classpath searching mechanism to locate the class. See ["Java configuration \(optional\)" on page 273](#) for more information about setting a classpath for use with the IAF.

- All `<transport>` and `<codec>` elements must also have a `name` attribute. The name is an arbitrary string used to reference the plug-in within the IAF, and must be unique within the configuration file. Even if the same plug-in was to be loaded more than once inside the same IAF, the corresponding `<transport>` or `<codec>` elements would still need to have different names.
- The `<transport>` and `<codec>` elements can include an optional `recordTimestamps` attribute. This attribute supports the latency framework feature. The value of the attribute determines the values of the `recordUpstream` and `recordDownstream` members of the `IAF_TimestampConfig` object (C/C++ plug-ins) or `TimestampConfig` object (Java plug-ins) object passed to the semantic mapper. The attribute takes one of the following values:
 - `none` — Do not record any timestamps
 - `upstream` — Record timestamps for upstream events only
 - `downstream` — Record timestamps for downstream events only
 - `both` — Record timestamps for both upstream and downstream events

The default, if the `recordTimestamps` attribute is not present, is `none`.

- The `<transport>` and `<codec>` elements can include an optional `logTimestamps` attribute. This attribute supports the latency framework feature. The attribute takes a space- or comma-separated list of keywords for its value. Supported keywords are:

- `upstream` — Log latency for upstream events
- `downstream` — Log latency for downstream events
- `roundtrip` — Log roundtrip latency for all events, in a plug-in-specific way
- `logLevel` — Set the logging level for timestamp logging. Any of the standard Apama log levels are accepted for this keyword.

The value of this attribute determines the values of the `logUpstream`, `logDownstream`, `logRoundtrip` and `logLevel` members of the `IAF_TimestampConfig` object (C/C++ plug-ins) or `TimestampConfig` object (Java plug-ins) passed to the semantic mapper. If the `logTimestamps` attribute is not present, the log level defaults to `INFO` and the other timestamp logging parameters default to `false`.

Note: Although C/C++ and Java transport and codec layer plug-ins may coexist in the same IAF, using a C/C++ codec plug-in with a Java transport plug-in or vice-versa is not permitted.

Plug-in <property> elements

<transport> and <codec> elements may also contain any number of <property> elements, which are the mechanism by which plug-in-specific options are configured.

Each <property> element has two attributes: `name` and `value`. The syntax of the `name` and `value` is entirely determined by the plug-in author. Typically the `name`, `value` pairs are not ordered, and there is no constraint on the uniqueness of the names. Most plug-ins treat the `name` attribute in a case-sensitive manner.

In most (though not all) cases, plug-in authors allow properties to be changed dynamically without restarting the IAF, by using the IAF Client tool to request a reload of properties from the IAF configuration file. See ["IAF Management – Managing a running adapter I" on page 251](#) for more information about using the IAF Client.

Example

The transport/codec definition section of an IAF configuration file might look as follows for a C/C++ transport plug-in implemented on Windows by `FileTransport.dll` with a codec in `StringCodec.dll`:

```
...
<transports>
  <transport name="File" library="FileTransport">
    <!-- Transport-specific configuration property -->
    <property name="input" value="simple-feed.evt"/>
    <property name="output" value="output.evt"/>
  </transport>
</transports>
<codecs>
  <codec name="String" library="StringCodec">
    <!-- Codec-specific configuration property -->
    <property name="NameValueSeparator" value="="/>
    <property name="FieldSeparator" value=", "/>
    <property name="Terminator" value=";"/>
  </codec>
</codecs>
...
```

Similarly the configuration section for the equivalent Java plug-ins, both packaged inside `FileAdapter.jar`, would be:

```
...
<transports>
  <transport
    name="File"
```

```

    jarName="JFileAdapter.jar"
    className="com.apama.iaf.transport.file.JFileTransport"
  >
    <!-- Transport-specific configuration property -->
    <property name="input" value="simple-feed.evt"/>
    <property name="output" value="output.evt"/>
  </transport>
</transports>
<codecs>
  <codec
    name="String"
    jarName="JFileAdapter.jar"
    className="com.apama.iaf.codec.string.JStringCodec"
  >
    <!-- Codec-specific configuration property -->
    <property name="NameValueSeparator" value="="/>
    <property name="FieldSeparator" value=", "/>
    <property name="Terminator" value=";" />
  </codec>
</codecs>
...

```

You are advised to peruse the `iaf_4_0.dtd` file as it represents the complete syntactic reference to the correct structure of the configuration file.

The topic ["Event mappings configuration" on page 259](#) describes the semantic translation and transformation rules. ["IAF samples" on page 274](#) illustrates an example configuration file.

Event mappings configuration

The adapter configuration file requires a `<mapping>` element, which configures the adapter's Semantic Mapper layer.

The `<mapping>` element may contain the following optional attributes to support the latency framework:

- An optional `recordTimestamps` attribute. The value of the attribute determines the values of the `recordUpstream` and `recordDownstream` members of the `IAF_TimestampConfig` object (C/C++ plug-ins) or `TimestampConfig` object (Java plug-ins) object passed to the transport or codec. The attribute takes one of the following values:
 - `none` — Do not record any timestamps
 - `upstream` — Record timestamps for upstream events only
 - `downstream` — Record timestamps for downstream events only
 - `both` — Record timestamps for both upstream and downstream events

The default, if the `recordTimestamps` attribute is not present, is `none`.
- An optional `logTimestamps` attribute. This attribute takes a space- or comma-separated list of keywords for its value. Supported keywords are:
 - `upstream` — Log latency for upstream events
 - `downstream` — Log latency for downstream events
 - `roundtrip` — Log roundtrip latency for all events
 - `log level` — Set the logging level for timestamp logging. Any of the standard Apama log levels are accepted for this keyword.

The value of this attribute determines the values of the `logUpstream`, `logDownstream`, `logRoundtrip` and `logLevel` members of the `IAF_TimestampConfig` object (C/C++ plug-ins) or `TimestampConfig` object (Java plug-ins) passed to the transport or codec. If the `logTimestamps` attribute is not present, the log level defaults to `INFO` and the other timestamp logging parameters default to `false`.

The `<mapping>` element may contain the following (in order):

- An optional `<logUnmappedDownstream>` element, which specifies a file to which unmapped downstream messages should be logged.
- An optional `<logUnmappedUpstream>` element, which specifies a file to which unmapped upstream Apama events should be logged.
- One or more `<event>` elements, specifying the mapping between Apama correlator events and external messages. Setting up the correct `<event>` elements is the main part of configuring the Semantic Mapper.
- One or more `<unmapped>` elements, which specify events that will bypass the Semantic Mapper, using a string representation of the entire Apama event.

Note: The order in which `<event>` and `<unmapped>` elements appear can be mixed.

Each of these will be discussed in more detail below, after a brief explanation of the operation of the Semantic Mapper.

Semantic Mapper operation

The IAF Semantic Mapper takes as input a set of rules that specify when and how an Apama event can be generated from an external message, and similarly how suitable messages of the correct external format should be constructed from Apama events. These rules are termed an event mapping.

All Apama events must belong to a named event type that defines their structure. On startup, the IAF will parse each `<event>` element, derive the structure of the event being described, and optionally inject an EPL event definition for it into the event correlator. The event mappings are therefore organized by Apama event type, as `<event>` elements.

When an external message is received from a codec plug-in, the Semantic Mapper will run it past each event mapping sequentially, in the order provided in the configuration file. First it checks whether it matches a set of conditions specified within that mapping. If it does, it proceeds to transform and translate it according to the mapping rules provided. If it does not match the conditions, the Semantic Mapper will move on to the next event mapping. If the message matches against several `<event>` mappings, only the first mapping is executed unless the `breakDownstream` attribute is set to `false`. When this attribute is set to `false`, all mappings that match are executed.

In the upstream direction, when the Semantic Mapper receives an Apama event, it will already know the type of the event, because this information is part of each event sent out by the event correlator. However, it is possible to specify multiple upstream mappings from the same Apama event type; therefore, just as with downstream mappings, the Semantic Mapper will check the incoming Apama event against the conditions defined in each of the mappings for that event type. Just as for downstream mappings, the first matching mapping will be used, unless the `breakUpstream` attribute is set to `false`. When this attribute is set to `false`, all mappings that match are executed.

In both directions it is possible for an incoming event not to match against any event mapping, and in which case no mapping is executed. The `<logUnmappedDownstream>` and `<logUnmappedUpstream>` elements allow such messages and events to be logged.

The <logUnmappedDownstream> and <logUnmappedUpstream> elements

These optional elements enable the logging of Apama events and codec messages that were not matched by any of the configured mapping rules, before they are discarded by the adapter. This can be useful for debugging and diagnostics.

The <logUnmappedDownstream> element turns on logging of messages from an external event source that were not mapped onto an Apama correlator event, after being received from a codec plug-in; the <logUnmappedUpstream> element enables logging of events from the correlator that did not match the conditions necessary for mapping to an external message that could be passed on to a codec plug-in.

Both elements have a single attribute called `file`, which is used to specify the filename that the log should be written to.

For example:

```
<mapping>
...
  <logUnmappedDownstream file="unmapped_from_adapter.log"/> <logUnmappedUpstream
    file="unmapped_from_Correlator.log"/> ...
</mapping>
```

Note: Due to buffering of files in the operating system, the contents of the log files on disk may not be complete until the IAF is shutdown or reconfigured.

The <event> element

The <mapping> section contains one or more <event> elements, each of which specifies a mapping between an Apama correlator event type and a kind of external message. Setting up the correct <event> elements is the main part of configuring the Semantic Mapper.

Each <event> element can have the following attributes:

- `name` – This is the name of this Apama correlator event type, and is required.
- `package` – This optional attribute specifies the EPL package of the Apama event; if it is not provided, the default package is used. See *Developing Apama Applications* for information about packages.
- `direction` – This optional attribute defines whether this event mapping is to be used solely for downstream mapping (from incoming external messages to Apama events), upstream mapping (from Apama events to outgoing messages) or for mapping in both directions. The allowed values for the attribute are `upstream`, `downstream` and `both`. The default value if the attribute is undefined is `both`.
- `encoder` – Required for a mapping that can be used in the upstream direction (`direction` = "upstream" or "both"), but ignored when processing downstream messages. In the upstream direction the attribute specifies the codec plug-in that should be used to process the message, once the translation process is complete. The name supplied here must match the `name` provided in the <codec> element.
- `copyUnmappedToDictionaryPayload` – This optional Boolean attribute defines what the Semantic Mapper should do with any fields in the incoming messages that do not match with any field mapping. If `copyUnmappedToDictionaryPayload` is `false` (and `copyUnmappedToPayload`, below, is also `false` or not present), then any unmapped fields are discarded. If it is set to `true` however, they will be packaged into a special field called `__payload`, implicitly added as the last field of the Apama event type. Fields in normalised events with a value of null will be included in the dictionary with the value set to an empty string. If this attribute is undefined its value defaults to `false`.

Using `copyUnmappedToDictionaryPayload` puts all the payload fields in a standard EPL dictionary that is efficient and easy to access. See ["Creating a payload field" on page 363](#) for more information about the payload field.

- `copyUnmappedToPayload` – As of Release 4.1, this option is deprecated; use `copyUnmappedToDictionaryPayload` instead. This optional Boolean attribute defines what the Semantic Mapper should do with any fields in the incoming messages that do not match with any field mapping. If `copyUnmappedToPayload` is `false` (and `copyUnmappedToDictionaryPayload`, above, is also `false` or not present), then any unmapped fields are discarded. If it is set to `true` however, they will be packaged into a special field called `__payload`, implicitly added as the last field of the Apama event type. The default value if this attribute is undefined is `false`. See ["Creating a payload field" on page 363](#) for more information about the payload field.

- `breakUpstream` – This optional Boolean attribute defines what the Semantic Mapper should do when it matches an upstream Apama event to an event mapping.

When set to `true`, the semantic mapper stops the evaluating process and begins executing the actions specified by that mapping and then goes on to evaluate the next event. This is the default behavior when the `breakUpstream` attribute is not specified.

When set to `false`, the semantic mapper executes the actions specified in the mapping and then keeps evaluating the same event against the other event mappings.

This attribute only affects upstream event processing.

- `breakDownstream` – This optional Boolean attribute defines what the Semantic Mapper should do when it matches an incoming downstream message to an event mapping.

When set to `true`, the semantic mapper stops the evaluating process and begins executing the actions specified by that mapping and then goes on to evaluate the next message. This is the default behavior, when the `breakDownstream` attribute is not specified.

When set to `false`, the semantic mapper executes the actions specified in the mapping and then keeps evaluating the same message against the other event mappings.

This attribute only affects downstream event processing.

- `inject` – Determines whether the IAF will automatically inject the event definitions that are implicitly defined by this event mapping into the correlator. The default is `false`. Injecting events from the configuration files is deprecated. Instead, you should use the `-e` or `--events` switches to `iaf` to generate the EPL code for the event definitions and then inject the events (and monitors) during the application's start-up sequence with the tool that you use to start the correlator, such as Apama Studio, the Enterprise Management and Monitoring console (EMM) or the Apama command line tools.
- `copyTimestamps` – This is an optional attribute. If set to `true` (the default is `false`), the semantic mapper will add an additional field to the generated event definition to hold timestamp information. The new field will be called `__timestamps` and will be of type `dictionary<integer,float>` where the dictionary keys are timestamp indexes and the values are the corresponding timestamps. The timestamp field is inserted before the payload field, so it may be the last or next to last field in the event definition. If timestamp copying is enabled for an event type, all timestamps present in the `__timestamps` field of a matching upstream event will be copied into a new `AP_TimestampSet/TimestampSet` object and passed to the upstream codec. Likewise, any timestamps passed to the semantic mapper by the codec will be copied into the `__timestamps` field of the outgoing downstream event and thus made available to the correlator.

- `transportChannel` — optional. If present, then for upstream events (events leaving the correlator), the channel is put in the `NormalisedEvent` using the value of the `transportChannel` attribute.

If present, then for downstream events (events going into the correlator), if the value of the `transportChannel` attribute is in the `NormalizedEvent`, then that value from the `NormalizedEvent` is used as the channel name. It is possible that a subsequent `<map>` element with an identical `transport` attribute value could override it.

- `presetChannel` - optional. If present, then for downstream events (events going into the correlator), if no channel has been set by the `transportChannel` attribute, then the value of `presetChannel` is used as the channel name.

If `transportChannel` is set, then that value in the `NormalisedEvent` can still be used for a normal `<map>` rule, but it will not appear in the `unmappedDictionary` (if present).

Thus, it is possible to define either a default channel name per type, or a `NormalisedEvent` field that the transport will send and receive, and this could be re-using a `NormalisedEvent` field used by a `<map>` element.

A typical bidirectional `<event>` element might look like the following. For downstream, if "CHANNEL" (from `transportChannel`) is in the `NormalisedEvent`, then the value of the "CHANNEL" entry is used as the channel name, otherwise "channelB" from `presetChannel` is used. For upstream, the channel name is placed in the "CHANNEL" entry in the `NormalisedEvent`.

```
<event name="Tick"
  direction="both"
  encoder="String"
  copyUnmappedToDictionaryPayload="true"
  inject="false"
  presetChannel="channelB"
  transportChannel="CHANNEL">
  <id-rules>
    ...
  </id-rules>
  <mapping-rules>
    ...
  </mapping-rules>
</event>
```

The `<id-rules>` and `<mapping-rules>` elements are described below.

The `<event>` mapping conditions

The `<id-rules>` element defines a set of conditions that must be satisfied by an incoming message for it to trigger the mapping to an Apama event, or to decide how to map an incoming Apama event back to a normalized message. The `<id-rules>` element contains `<upstream>` and `<downstream>` sub-elements, which in turn contain the mapping conditions to be used when the Semantic Mapper is searching for a mapping to use in the upstream or downstream direction, respectively. Each condition is encoded in an `<id>` element.

Note: Conditions are only required for the directions that the mapping can operate in. For example, a mapping with `direction="downstream"` does not need any `<upstream>` id rules, while a mapping with `direction="both"` must specify both `<upstream>` and `<downstream>` id rules. The `<id-rules>`, `<upstream>` and `<downstream>` elements themselves must exist though.

`<id>` - Each `<id>` sets a condition on a set of fields contained in the normalized message or Apama event. This element takes up to three attributes; `fields`, which defines the fields that the condition must apply to; `test`, which specifies the condition; and `value`, which provides a value to compare the field value with. The value attribute is only required for relational tests. For example:

```
<id fields="Stock, Exchange, Price" test="exists"/>
```

specifies that the `Stock`, `Exchange` and `Price` fields must exist if the condition is to be satisfied and the mapping proceed. No `value` is needed to perform the test in this case.

However, the following example:

```
<id fields="Exchange" test="==" value="LSE"/>
```

specifies that the `Exchange` field must exist and have the value "LSE" for the condition to be satisfied.

Note: The value for the `fields` attribute is a list of fields, delimited by spaces or commas. This means, for example that `<id fields="Exchange EX,foo" test="==" value="LSE"/>` will successfully match a field called "Exchange", "EX" or "foo", but *not* a field called "Exchange EX,foo". You should keep this in mind when you assign field names for normalized events. While fields with spaces or commas in their names may be included in a payload dictionary in upstream or downstream directions, they cannot be referenced directly in mapping or id rules.

The following test conditions may be specified. The first four tests are unary operators that do not need a `value` attribute. These tests can be applied to multiple fields in the same `<id>` rule:

- `exists` - The fields exist, but do not necessarily have a value
- `notExists` - The fields do not exist
- `anyExists` - One or more of the fields exist, not necessarily with values
- `hasValue` - At least one of the fields exists and has a value. To test that multiple fields all exist and all have values, use multiple `hasValue` conditions, one for each field, such as

```
<id fields="symbol" test="hasValue"/>  
<id fields="newLimit" test="hasValue"/>
```

The remaining tests are binary relational operators that do require a `value` attribute. Furthermore, these tests can only be applied to single fields:

- `==` - Case-sensitive string equality
- `!=` - Case-sensitive string inequality
- `===` - Case-insensitive string equality
- `~!=` - Case-insensitive string inequality

While the equality tests will fail on fields with missing values, the inequality tests will pass.

All `<id>` conditions in an `<id-rules>` element must be satisfied for the mapping to proceed.

Note: A mapping with no `<id>` elements will always match. This allows a catch-all mapping to be specified. This should be the last definition.

The id rules that test transport field values function in isolation from each other, that is, as soon as a test id rule fails, the mapper stops looking at subsequent rules. This means there is no way to group together tests against the same field name with an OR condition to see if any of them match. Any type of OR value testing needs to be implemented at the codec or EPL layers, or by creating copies of the entire `<event>` element for each value to test.

The following is an example of a valid `<id-rules>` element for a bidirectional mapping. Note that the upstream rules are empty, so this mapping will match any incoming Apama event of the appropriate type:

```
<id-rules>  
  <downstream>
```



```

<id fields="Stock, Price" test="exists"/>
<id fields="Exchange" test="==" value="LSE"/>
</downstream>
<upstream/>
</id-rules>

```

The <event> mapping rules

The <mapping-rules> element defines a set of mappings that describe how to create an Apama event from an incoming message. Conversely they define the mapping from an Apama event to an outgoing message. Each mapping must be defined in a <map> element, which has the following attributes:

- apama** – This is the Apama event field name to copy the value into (downstream) or to take the value from (upstream). This attribute is optional. In an upstream direction, if the `apama` attribute is not specified or is provided empty, a field will be created within the external message and set to the value specified by `default`. Not specifying the `apama` attribute has no significance in a downstream direction — this line of the mapping will be ignored.

Note: The IAF does not know what types are injected into the correlator, and will drop events with a `Failed to parse` warning if the *types* and *order* of the elements with an `apama=` attribute do not match the event definition that was injected into the correlator. Mapping rules that do not specify an `apama=` attribute are not affected by this.

- transport** – This defines the external message's field name to copy the value from (downstream) or to copy the value into (upstream). For a downstream mapping it is possible to define more than one value here; in which case the first encountered is used. This attribute is optional. In a downstream direction, if the `transport` attribute is not specified or is provided empty, a field will be created within the Apama event and set to the value specified by `default`. Not specifying the `transport` attribute has no significance in an upstream direction — this line of the mapping will be ignored.
- type** – The type of the field in the Apama event type. Any simple correlator type is valid here (`string`, `integer`, `decimal`, `float`, `boolean` and `location`); for complex correlator reference types such as `sequence<...>` and `dictionary<...,>`, specify `reference` instead. If a field is of reference type, the `referenceType` attribute can be supplied if needed to define the type (see below). When a field is of reference type, the Semantic Checker passes its string form to and from the codec untouched, and performs no checking upon the validity of the value. Note that fields in the external message are always un-typed character strings, regardless of any type they may have had on the external transport that produced them. Furthermore, the Semantic Mapper does not perform any type "casting" or "coercion" when converting a character string in an external event field to the appropriate Apama type, meaning that the Apama event produced might be invalid and be rejected by the correlator. Conversions in the upstream direction, from the Apama field to a string in the external event, will always succeed. Codec and transport plug-ins should be aware of these rules when working with events that will be, or have been, processed by the Semantic Mapper.
- referenceType** – This is an optional attribute, but it must be supplied if the attribute `type="reference"` and the event of which this field is a member has the attribute `inject="true"` (which is now deprecated) or if the IAF will be run with the `-e` option in order to generate a EPL file with event definitions. This EPL file is then injected to the correlator (this is the recommended method of injecting events). The value of this attribute must be a valid correlator type. This attribute is only used for the process of constructing the event definitions that are to be injected into a correlator. Note that since this is an XML attribute value, some characters such as angle brackets or quotation marks must be correctly encoded using their XML entity name.

For example, a `sequence<string>` must be written as `referenceType="sequence <string>".` Note also that when nesting sequences within sequences, a space must be present between the angle brackets to prevent the correlator from parsing this as a bitwise shift.

- `default` – The default value to set the Apama field to if the external field specified in `transport` is missing. Note that the value provided must be of the type specified in `type`. For example, a valid string is "test" or "", a valid integer is "0", a valid decimal is "0.0" or "0.0d", a valid float is "0.0", a valid boolean is "true" or "false", and a valid location is "(0.0, 0.0, 0.0, 0.0)".
- `defaultIfEmpty` – Optionally, sets the default value to assign to the field in the Apama event if the external field specified in `transport` is present but has no value defined. The same type considerations apply as for the `default` attribute. If this attribute is not defined the value specified by the `default` attribute will apply for this condition as well. Bearing in mind angled brackets and quotes have to be written as XML entities (see above), for a nested event you need to write `default="InnerEvent ("")"` if you want the default value to be `InnerEvent ("")`.

Note that at least one of `apama` and `transport` must be specified in a mapping.

The following is an example of a valid `<mapping-rules>` element:

```
<mapping-rules>
  <map apama="stockName" transport="Stock" type="string" default=""/>
  <map apama="stockPrice" transport="Price" type="float" default="0.0"/>
  <map apama="stockVolume" transport="Volume, TradingVolume,
    CombinedVolume" type="float" default="0.0"/>
</mapping-rules>
```

In a downstream direction, this specifies that the `Stock` field must be copied over into `stockName`, `Price` must be copied into `stockPrice`, and the first encountered of `Volume`, `TradingVolume` or `CombinedVolume` must be copied into `stockVolume`. First encountered means the first such instance when the event is parsed left to right.

In an upstream direction, the Apama field values would be copied into external message fields of the names specified. A given field in an upstream message can be generated from several different sources. These are evaluated in the following order:

1. A `<map>` rule mapping from a named Apama event field to the transport field.
2. A value for the transport field in the event payload. See ["The Event Payload" on page 363](#) for more details on using the event payload.
3. Any default value available from a `<map>` rule with no corresponding Apama event field.

You may have noticed that while an Apama event must always have its full complement of fields defined and with type-valid values, the same is not assumed of external events.

Note: If multiple upstream mappings for the same Apama type exist, they must all specify all of the fields in the type in the same order, with the same type values.

Note: Tips for writing a codec when using reference types:

- A codec is responsible for constructing the string form of any value. This means that if your event contains a `sequence<string>` then the codec must generate an entry in the normalized event whose value is of the form:
`["string value 1", "string value 2", "Value with a \" and backslash \\\"]`
- If the codec generates an event that the correlator cannot parse, the correlator will drop the event and the codec will have no way of knowing. Be careful constructing the event strings.

- Similarly, events from the correlator will contain a normalized event entry whose value is the string from of the field's value, as in the example above. The codec is responsible for parsing these strings.
- When writing adapters in Java, Apama suggests you use the classes in the `com.apama.event.parser` package to parse and construct the strings to send to the Semantic Mapper. If you are writing a C/C++ adapter, the corresponding functions for parsing and constructing strings to send to the Semantic Mapper are found in the `AP_EventParser.h` and `AP_EventWriter.h` header files.

For more information on the Java classes, see ["Working with normalized events" on page 334](#) as well as the Apama `Javadoc`. For more information on the C/C++ functions, see ["Codec utilities" on page 303](#).

- If nesting other events in the fields of an event, caution must be exercised regarding package namespaces. Always use the fully qualified event name when referencing it in the string form. Also always ensure that the correlator has the enclosed event type defined before the enclosing event type.

The <unmapped> element

The <mapping> section may contain one or more <unmapped> elements, each of which specifies a mapping between the string representation of an Apama event type and a normalized event.

Each <unmapped> element can have the following attributes:

- `name` – This optional attribute specifies the name of the Apama correlator event type to match. If omitted, matches all Apama event types.
- `package` – This optional attribute specifies the EPL package of the Apama event. This attribute can be specified only if the `name` attribute is also supplied.
- `transport` – This attribute is required; it specifies the field in the `NormalisedEvent` to map to.
- `direction` – This optional attribute defines whether this event mapping is to be used solely for downstream mapping (from incoming external messages to Apama events), upstream mapping (from Apama events to outgoing messages) or for mapping in both directions. The allowed values for the attribute are `upstream`, `downstream` and `both`. The default value if the attribute is undefined is `both`.
- `encoder` – Required for a mapping that can be used in the upstream direction (`direction` = "upstream" or "both"), but ignored when processing downstream messages. In the upstream direction the attribute specifies the codec plug-in that should be used to process the message, once the translation process is complete. The name supplied here must match the `name` provided in the <codec> element.
- `breakUpstream` – This optional Boolean attribute defines what the Semantic Mapper should do when it matches an upstream Apama event to an event mapping.

When set to `true`, the semantic mapper stops the evaluating process and begins executing the actions specified by that mapping and then goes on to evaluate the next event. This is the default behavior when the `breakUpstream` attribute is not specified.

When set to `false`, the semantic mapper executes the actions specified in the mapping and then keeps evaluating the same event against the other event mappings.

This attribute only affects upstream event processing.

- `breakDownstream` – This optional Boolean attribute defines what the Semantic Mapper should do when it matches an incoming downstream message to an event mapping.

When set to `true`, the semantic mapper stops the evaluating process and begins executing the actions specified by that mapping and then goes on to evaluate the next message. This is the default behavior, when the `breakDownstream` attribute is not specified.

When set to `false`, the semantic mapper executes the actions specified in the mapping and then keeps evaluating the same message against the other event mappings.

This attribute only affects downstream event processing.

- `transportChannel` — optional. If present, then for upstream events (events leaving the correlator), the channel is put in the `NormalisedEvent` using the value of the `transportChannel` attribute.

If present, then for downstream events (events going into the correlator), if the value of the `transportChannel` attribute is in the `NormalisedEvent`, then that value from the `NormalisedEvent` is used as the channel name. It is possible that a subsequent `<map>` element with an identical `transport` attribute value could override it.

- `presetChannel` - optional. If present, then for downstream events (events going into the correlator), if no channel has been set by the `transportChannel` attribute, then the value of `presetChannel` is used as the channel name.

If `transportChannel` is set, then that value in the `NormalisedEvent` can still be used for a normal `<map>` rule, but it will not appear in the `unmappedDictionary` (if present).

Thus, it is possible to define either a default channel name per type, or a `NormalisedEvent` field that the transport will send and receive, and this could be re-using a `NormalisedEvent` field used by a `<map>` element.

In the following example, for downstream, if `"CHANNEL"` (from `transportChannel`) is in the `NormalisedEvent`, then the value of the `"CHANNEL"` entry is used as the channel name, otherwise `"channelB"` from `presetChannel` is used. For upstream, the channel name is placed in the `"CHANNEL"` entry in the `NormalisedEvent`.

```
<unmapped
  name="Unmapped"
  direction="both"
  package="com.apama.sample"
  transport="Apama"
  encoder="$CODEC$"
  presetChannel="channelB"
  transportChannel="CHANNEL">
  <id-rules>
    <downstream>
      <id fields="Apama" test="exists"/>
    </downstream>
  </id-rules>
</unmapped>
```

The `<unmapped>` mapping conditions

An `<unmapped>` element must have an `<id-rules>` element that defines a set of conditions an incoming must satisfy in order to trigger the mapping to an Apama event. If the value of the `direction` attribute of an `<unmapped>` element is `"both"` or `"downstream"`, the `<id-rules>` element must contain a `<downstream>` sub-element. The `<downstream>` sub-element contains conditions to be used by the Semantic Mapper when the message is moving downstream direction. Each condition is encoded in an `<id>` element.

Each `<id>` sets a condition on a set of fields contained in the normalized message or Apama event. This element takes up to three attributes; `fields`, which defines the fields that the condition must apply to; `test`, which specifies the condition; and `value`, which provides a value to compare the field value with. The value attribute is only required for relational tests.

The `<unmapped>` entries behave in the same way as `<event>` entries — the IAF processes `<event>` and `<unmapped>` entries in order, translating events with any that match, and ending at the first entry that has `breakUpstream` or `breakDownstream` set to true or not specified (they both default to true).

Apama event correlator configuration

The adapter configuration file requires an `<apama>` element, which configures how the IAF connects to the Apama event correlator(s). An `<apama>` element can contain the following elements in the following order:

- `<sinks>` — This element lists the Apama correlators that the IAF needs to connect with in order to inject EPL event type definitions and events. You can specify the following attribute in a `<sink>` element:

`parallelConnectionLimit` — optional — The default behavior is that the IAF limits itself to an internally set number of connections with each specified sink. This number scales according to the number of CPUs that the IAF detects on the host that is running the IAF. While this number is usually sufficient, there are some situations in which you might want to change it. For example, if you are trying to conserve resources you might want to limit the number of connections to 1, or if you want to prevent multiple threads from sharing a connection you might allow a higher number of connections than the default allows. See the information below about multiple connections from IAF to correlator.

Each correlator is defined in its own `<sink>` element:

- `<sink>` — This element defines an event correlator that the IAF must send events to. You can define more than one `<sink>` element. All sinks specified will be injected with any EPL event type definitions that are defined in `<event>` elements in the configuration file. The following attributes are allowed in `<sink>` elements:

`host` — required — Defines the name or address of the host machine where the correlator is running.

`port` — required — Specifies the port that this correlator can be contacted on.

`sendEvents` — optional — The default behavior is that all sinks receive all events generated by the Semantic Mapper. To prevent the Semantic Mapper from sending all events to a particular correlator, add `sendEvents="false"` to the `<sink>` element that defines that correlator. No events will be sent to that correlator regardless of any channel settings.

- `<sources>` — This element lists the Apama components (usually event correlators) from which the IAF can receive events. Each component is defined in its own `<source>` element:

- `<source>` — This element defines an Apama component that the IAF needs to register with as an event consumer. This enables the IAF to receive any alerts generated by the specified component. The following attributes are allowed in `<source>` elements:

`host` — required — Defines the name or address of the host machine where the Apama component is running.

`port` — required — Specifies the port that this Apama component can be contacted on.

`channels` — required — Specifies the channels that the IAF should listen on to receive events. An empty string indicates that the IAF receives all generated events. To receive events on only particular channels, specify a comma-separated list of channel names. Do not include any spaces. For example, `channels="UK,USA,GER"`.

`disconnectable` — Specifies whether or not the IAF can be disconnected if it is slow. If set to `yes` or `true` (case insensitive), the IAF can be disconnected. Any other setting specifies that it cannot be disconnected.

It is possible to define the IAF as having only sinks, or only sources, or both, or neither. If the IAF has been started with no sinks and no sources, you would use the `engine_connect` tool to connect it to a correlator or another IAF.

For complete information on `engine_connect`, see "Event correlator pipelining" in *Deploying and Managing Apama Applications*.

Disabling Apama messaging and using UM instead

A deployed adapter can use Software AG's Universal Messaging (UM) message bus in place of connections specified in the `<apama>` element. When you want to use UM instead of explicitly set connections do the following:

- Add a `<universal-messaging>` element in place of or after the `<apama>` element. See ["Configuring adapters to use UM" on page 235](#).
- Add the `enabled` attribute to the `<apama>` element, if you have one, and set it to `false`. For example:

```
<apama enabled="false">.....</apama>
```

Alternatively, you can remove the `<apama>` element.

When the `enabled` attribute is set to `"false"` then the entire `<apama>` element is ignored. In other words, the deployed adapter does not use any connections specified in the `<apama>` element. Instead, the deployed adapter uses the UM configuration specified in the `<universal-messaging>` element.

The default is that the `enabled` attribute is set to `true`. If the `enabled` attribute is not specified or if it is set to `true` then the connections specified in the `<apama>` element are used.

While specifying both an `<apama>` element and a `<universal-messaging>` element in an adapter configuration file is permitted, it is not recommended.

See also "Using Universal Messaging" in *Connecting Apama Applications to External Components*.

Multiple connections from IAF to correlator

To improve performance, an IAF transport might use multiple threads to send events to the codec and thus to the Semantic Mapper. If more than one thread is sending events downstream (IAF to correlator) then for each thread, the IAF creates a new connection to each `<sink>` defined in the configuration file, up to the defined limit. Thus, multiple threads can deliver events in parallel to the same sink. In combination with the `channelTransport` attribute on events (defined in `<event>` elements), threads can deliver events to different channels to be received by different contexts. For optimal parallel event delivery, each IAF transport thread should send events on a distinct set of channels. There are no ordering guarantees when different threads deliver events to the same sink.

There is a limit on how many connections to each sink the IAF can create. The IAF logs the limit for the number of connections in the startup stanza. If events are sent on more threads than the number of allowed connections, then the IAF re-uses existing connections, which means that some threads share connections. If a thread terminates, the connection it is using is not closed since it might be

in use by another thread. See the information above for the `parallelConnectionLimit` attribute on the `<sink>` element.

Example

Following is an example of an `<apama>` element:

```
<adapter-config>
...
  <apama>
    <sinks parallelConnectionLimit="1">
      <sink host="localhost" port="15903"/>
    </sinks>
    <sources>
      <source host="localhost" port="15903" channels="MY_ADAPTER"/>
    </sources>
  </apama>
</adapter-config/>
```

Configuring adapters to use UM

If you are configuring your Apama application to use Software AG's Universal Messaging, you can configure an adapter to use the UM message bus to send and receive events. To do this, add a `<universal-messaging>` element to your adapter configuration file. A `<universal-messaging>` element can replace or follow the `<apama>` element.

A `<universal-messaging>` element contains:

- Required specification of the `realms` attribute OR the `um-properties` attribute.
- Required specification of the `<subscriber>` element.
- Optional specification of the `defaultChannel` attribute.
- Optional specification of the `enabled` attribute, which indicates whether the deployed adapter uses the configuration specified in this `<universal-messaging>` element.

Specification of realms or um-properties attribute

Specification of the `realms` attribute OR the `um-properties` attribute is required.

The `realms` attribute can be set to a list of `RNAMES` (UM realm names) to connect to. You can use commas or semicolons as separators.

Commas indicate that you want the adapter to try to connect to the UM realms in the order in which you specify them here. Semicolons indicate that the adapter can try to connect to the specified UM realms in any order. See ["Starting correlators that use UM" on page 235](#) for more information.

If you specify more than one `RNAME`, each UM realm you specify must belong to the same UM cluster. Specification of more than one UM realm lets you benefit from failover features. See "Communication Protocols and RNAMES" in the [Universal Messaging documentation](#).

The `um-properties` attribute can be set to the name or path of a properties file that contains UM configuration settings. See ["Defining UM properties for Apama applications" on page 238](#).

Specification of subscriber element

Specification of the `<subscriber>` element is required. The `<subscriber>` element must specify the `channels` attribute. Set the `channels` attribute to a string that specifies the names of the UM channels this adapter receives events from. Use a comma to separate multiple channel names.

Specification of defaultChannel attribute

Specification of the `defaultChannel` attribute is optional. If specified, set the `defaultChannel` attribute to the name of a UM channel. You cannot specify an empty string. In other words, the value of the `defaultChannel` attribute cannot be the default Apama channel, which is the empty string.

An adapter that uses UM must send each event to a named channel. An adapter that is configured to use UM identifies the named channel to use as follows:

1. If the `transportChannel` attribute is set for an event type (in an `<event>` or `<unmapped>` element) then this is the channel the adapter uses for that event type.
2. If the `transportChannel` attribute is not set for an event type but the `presetChannel` attribute is set then this is the channel the adapter uses for that event type.
3. If neither `transportChannel` nor `presetChannel` is set for an event type then the adapter uses the channel set by the `defaultChannel` attribute in the `<universal-messaging>` element.
4. If neither `transportChannel` nor `presetChannel` is set and you did not explicitly set `defaultChannel` and you used Apama Studio to create the adapter configuration file then the `defaultChannel` attribute is set to `"adapter_nameadapter_instance_id"`. For example: `"File Adapter instance 3"`.
5. If none of `transportChannel`, `presetChannel`, or `defaultChannel` are set and if you did not use Apama Studio to create the adapter configuration file then the adapter fails if it tries to use UM.

All events sent by the adapter on channels that are UM channels are delivered to those channels.

Specification of a value for the `defaultChannel` attribute affects events that are sent from this adapter to Apama engine clients, and from this adapter to correlators when the adapter connects to that correlator by means of the `engine_connect` correlator utility.

Specification of the enabled attribute

Optional specification of the `enabled` attribute, which indicates whether the deployed adapter uses the configuration specified in this `<universal-messaging>` element. The default is that the `enabled` attribute is set to `"true"`. If the `enabled` attribute is not specified or if it is set to `"true"` then the configuration specified in the `<universal-messaging>` element is used.

If `enabled` is set to `"false"` then the deployed adapter ignores the `<universal-messaging>` element and does not use UM. The deployed adapter uses only its explicitly set connections.

Subscribing to receive events from an adapter that is using UM

In each context, in any correlator, that is listening for events from an adapter that is using UM, at least one monitor instance must subscribe to the channel or channels on which events are sent from the adapter. For example, if you are using an ADBC adapter, you must include a `monitor.subscribe(channelName)` command for the corresponding instance of the ADBC adapter. Note that not all adapter service monitors support access from multiple correlators. If this is the case, then only one correlator should run the service monitors for that adapter.

Adapter configuration examples

Following are some examples of `<universal-messaging>` elements:

```
<universal-messaging
  realms="nsp://localhost:5629"
  defaultChannel="orders"
  enabled="true">
  <subscriber channels="UK, US, GER"/>
```



```

</universal-messaging>
<universal-messaging um-properties="UM-config.properties">
  <subscriber channels="signal,forward"/>
</universal-messaging>

```

Logging configuration (optional)

The `<logging>` and `<plugin-logging>` optional elements define the logging configuration used by the adapter. If present, they must appear as the first elements nested in the `<adapter-config>` element.

The `<logging>` element configures the logging for the IAF itself, whereas the `<plugin-logging>` element configures logging for the transport and codec layer plug-ins in the adapter.

Both elements have two attributes:

- The `level` attribute sets the logging verbosity level; it must be one of the strings `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, `CRIT`, or `OFF` (with the same case).
- The `file` attribute determines the file that logging messages will be written to. This should be a file path relative to the directory that the adapter was started from, or one of the special values `stdout` or `stderr` which to log to standard output or standard error, respectively.

If the IAF cannot write to the specified log file, the adapter will fail to start.

Note: If the `--logfile` and `--loglevel` command line switches are passed to the IAF executable when this is run, the `<logging>` and `<plugin-logging>` elements are ignored.

An example of these elements is provided below:

```

<adapter-config>
  <logging level="INFO" file="iaf.log" /> <plugin-logging
level="DEBUG" file="plugins.log" /> ...
</adapter-config>

```

If logging is not configured explicitly in the configuration file or with command line options, logging defaults to the `INFO` level on the standard error stream.

The `<logging>` element accepts two optional sub-elements, `<upstream-events>` and `<downstream-events>` that can be configured to log details of all events sent to and received from a connected correlator, without needing to set the entire IAF to `DEBUG` level logging. These sub-elements each take a single attribute (`level`) whose values specifies the logging level to be used to log upstream and downstream events, respectively. If this log level is equal to or greater than the IAF logging level, details of the events will appear in the IAF log file. For example:

```

<logging level="WARN">
  <upstream-events level="ERROR"/>
  <downstream-events level="INFO"/>
</logging>

```

In this configuration, upstream events will be logged (because `ERROR` is greater than `WARN`) but downstream events will not (because `INFO` is less than `WARN`). If either of the upstream or downstream event logging levels is not explicitly set, it will default to `DEBUG` (so events will not be logged by default, unless the IAF is explicitly configured for `DEBUG` logging)

Java configuration (optional)

Transport and codec plug-ins written in Java are executed by the IAF inside an embedded Java Virtual Machine (JVM). The optional `<java>` element allows the environment of this JVM to be configured.

The `<java>` element may contain zero or more of the following nested elements:

- `<classpath>` – This element adds a single entry onto the JVM classpath, which is the list of paths used by Java to locate classes. Each `<classpath>` element has a single `path` attribute that specifies a directory or Java Archive file (`.jar`) to add to the classpath.

The full classpath used by the IAF's JVM is made up by concatenating (in order):

1. the contents of the `APAMA_IAF_CLASSPATH` environment variable if one is defined,
2. each of the path entries specified by `<classpath>` elements, in the order they appear in the configuration file, OR if there are none, the contents of the `CLASSPATH` environment variable,
3. the path of the `lib/JPlugin_internal.jar` file used internally by the IAF.

Additionally, if a `jarName` attribute is used in the `<codec>` or `<transport>` element that defines a plug-in (as in "[Transport and codec plug-in configuration](#)" on page 257), the plug-in will be loaded using a new classloader with access to the specified Java Archive in addition to the JVM classpath.

You should make sure that all shared classes are in a separate jar that is specified by a `<classpath>` element. The shared classes are then loaded by the parent classloader. This ensures that when a codec or transport references a shared class, they both agree it is the same class.

- `<jvm-option>` - This element allows arbitrary JVM command line options to be specified. The JVM option should be placed between the start and end `jvm-option` tags, e.g.

```
<jvm-option>-Xmx256m</jvm-option>
```

See the usage screen of the JVM's Java executable for a full list of supported options.

- `<property>` - This element specifies a Java system property that should be passed to the JVM. It has `name` and `value` attributes, such that using:

```
<property name="propName" value="propValue"/>
```

is a shorthand equivalent to:

```
<jvm-option>-DpropName=propValue</jvm-option>
```

See "[The IAF runtime](#)" on page 247 for a description of how the IAF selects the JVM library to use.

The properties specified in the `<java>` element cannot be changed once the JVM has been loaded by the IAF. This will occur when the IAF reads a configuration file that specifies a Java transport or codec plug-in. If the same IAF process is later reconfigured to use only C/C++ transports, the JVM will *not* be unloaded. The IAF will log a warning message if a reconfiguration of the IAF process attempts to change the previously configured JVM properties.

IAF samples

Your distribution contains two complete examples that demonstrate how the IAF can be used in practice – C and Java implementations of a text file adapter, including build scripts and complete source code.

See "[Using Adapter Plug-ins](#)" on page 123 for information about how the sample plug-ins could be used in practice.

The C example

The C example is available in `samples\iaf_plugin\c\simple` and contains the following:

- The complete source code of the `FileTransport` transport layer plug-in and the `StringCodec` codec plug-in, in the `FileTransport.c` and `StringCodec.c` files.
 - The `FileTransport` transport layer plug-in can read and write messages from and to a text file. This makes it a useful tool in testing the IAF and the event correlator with files of sample messages.
 - The `StringCodec` codec plug-in can decode messages represented as strings containing a list of field names and values. The configuration properties for the plug-in allow customization of the syntactic characters used as field, name, and message separators (e.g. `"", "=", ";"`).
- A `Makefile` for compiling the plug-in sources with GNU Make on UNIX. This builds `libFileTransport.so` and `libStringCodec.so`, the plug-in binaries.
- A 'workspace' file and `dsp` folder for compiling the plug-in sources with Microsoft's Visual Studio .NET on Microsoft Windows. The `make.bat` batch file can be used to build the Windows plug-in binaries, `FileTransport.dll` and `StringCodec.dll`.
- A sample configuration file, `config.xml`. This is an example of an IAF configuration that loads C plug-ins, configures them with plug-in properties, injects a specific EPL file into the event correlator, provides a simple event mapping, and configures the IAF for sending and receiving events to and from the event correlator.
- A simple EPL file, `simple.mon`. This defines a monitor that examines the incoming events and selectively emits some back out to the IAF.
- A text file, `simple-feed.evt`, with some test input messages that can be loaded by the File Transport plug-in, parsed by the String Codec plug-in, translated into Apama events by the Semantic Mapper, and then injected into the event correlator.
- A reference file, `simple-ref.evt`, which shows the expected output file generated when the adapter is run.

To run the example, follow the steps outlined in the `README.txt` file provided in the `samples\iaf_plugin\c\simple` folder.

Note: Plug-ins need to be placed in a location where they can be picked up by the event correlator:

- On Windows you either need to copy the `.dll` into the `bin` folder, or else place it somewhere which is on your 'path', that is a location that is referenced by the `PATH` environment variable.
- On UNIX you either need to copy the `.so` into the `lib` directory, or else place it somewhere which is on your 'library path', that is a directory that is referenced by the `LD_LIBRARY_PATH` environment variable.

The Java example

The Java example is in the `samples\iaf_plugin\java\simple` directory, which contains the files:

- The complete source code of the `JFileTransport` transport layer plug-in and the `JStringCodec` codec plug-in, in the `src` directory.

- The `JFileTransport` transport layer plug-in can read and write messages from and to a text file. This makes it a useful tool in testing the IAF and the event correlator with files of sample messages.
- In normal operation, `JFileTransport` sends `String` objects on to the codec for decoding; however by setting the `upstreamNormalised` plug-in property it is possible to use the transport plug-in in a different mode in which it also performs the functionality that the codec usually performs (in this case by calling the `JStringCodec` class directly). In this mode the transport passes IAF normalized event messages on to the codec plug-in, demonstrating the use of the pass-through `JNullCodec` plug-in provided with the Apama distribution.
- The `JStringCodec` codec plug-in can convert between normalized events and messages represented as strings containing a list of field names and values. The configuration properties for the plug-in allow customization of the syntactic characters used as field, name, and message separators (e.g. `"", "=", ";"`).
- An Apache Ant `build.xml` file is included, for compiling the `JFileAdapter.jar` binary that contains both plug-ins (and works on all platforms).
- A sample configuration file, `config.xml`. This is an example of an IAF configuration that loads Java transport and codec plug-ins, configures them with plug-in properties, provides an event mapping, and configures the IAF for sending and receiving events to and from the event correlator.
- This configuration file also includes several optional configuration options for logging, custom JVM options, logging of unmapped events/messages, and use of non-standard correlator event batching.
- A second configuration file, `config-no-codec.xml` that demonstrates how the standard `JNullCodec` plug-in can be used with a transport plug-in that incorporates codec functionality itself and produces normalized events directly.
- A simple EPL file, `simple.mon`. This is identical to the file included with the C sample, and defines a monitor that examines the incoming events and selectively emits some back out to the IAF.
- A text file, `simple-feed.evt`. This is identical to the file included with the C sample, and contains some test input messages that can be loaded by the File Transport plug-in, parsed by the String Codec plug-in, translated into Apama events by the Semantic Mapper, and then injected into the event correlator.
- A reference file, `simple-ref.evt`, which shows the expected output file generated when the adapter is run. Note that this file is (only trivially) different to the reference file for the C plug-ins.

To run the example, follow the steps outlined in the `README.txt` file provided in the `samples\iaf_plugin\java\simple` folder.

Chapter 9: C/C++ Transport Plug-in Development

■ The C/C++ transport plug-in development specification	277
■ Transport Example	288
■ Getting started with transport layer plug-in development	289

The *transport layer* is the front-end of the IAF. The transport layer's purpose is to abstract away the differences between the programming interfaces exposed by different middleware message sources and sinks. It consists of one or more custom plug-in libraries that extract *downstream* messages from external message sources ready for delivery to the codec layer, and send Apama events already encoded by the codec layer *upstream* to the external message sink. See "[The Integration Adapter Framework](#)" on page 241 for a full introduction to transport plug-ins and the IAF's architecture.

An adapter should send events to the correlator only after its `start` function is called and before the `stop` function returns.

This topic includes the C/C++ Transport Plug-in Development Specification and additional information for developers of event transports using C/C++. "[Transport Plug-in Development in Java](#)" on page 317 provides information about developing transport plug-ins in Java.

To configure the build for a transport plug-in:

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

```
iafcore$(APAMA_LIBRARY_VERSION).lib;apcommon$(APAMA_LIBRARY_VERSION).lib
```

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

Developing Custom Adapters

The C/C++ transport plug-in development specification

A C/C++ transport layer plug-in is implemented as a dynamic shared library. In order for the IAF to be able to load and use it, it must comply with Apama's Transport Plug-in Development Specification. This Specification describes the structure of a transport layer plug-in, and the C/C++ functions it needs to implement so that it can be used with the IAF. The Specification also provides a mechanism for startup and configuration parameters to be passed to the plug-in from the IAF's configuration file.

Property names and values used by transport plug-ins must be in UTF-8 format.

A transport layer plug-in implementation must include the C header file `EventTransport.h`. It also needs to include `EventCodec.h`, to allow the event transport to pass messages to codecs within the IAF codec layer.

C/C++ Transport Plug-in Development

Transport functions to implement

`EventTransport.h` provides the definition for a number of functions whose implementation needs to be provided by the event transport author.

These functions are as follows:

updateProperties

```
/**
 * Update the configuration of the transport. The transport may assume
 * that stop(), flushUpstream() and flushDownstream() have all been called
 * before this function is invoked. The recommended procedure for
 * updating properties is to first compare the new property set with the
 * existing stored properties -- if there are no changes, no action should
 * be taken. Any pointer to the old property set becomes invalid as soon
 * as this function returns; any such pointers should therefore be
 * discarded in favour of the supplied new properties.
 *
 * @param transport The event transport instance
 * @param properties The new transport property set derived from the IAF
 * configuration file
 * @param timestampConfig Timestamp recording/logging settings
 * @return Event transport error code. If this is not
 * AP_EventTransport_OK, the getLastError() function should be called to
 * get a more detailed description of what went wrong.
 */
AP_EventTransportError (*updateProperties)(
    struct AP_EventTransport* transport,
    AP_EventTransportProperties* properties,
    IAF_TimestampConfig* timestampConfig);
```

sendTransportEvent

```
/**
 * Called by an event encoder to send a message to the external transport.
 * Ownership of the message is transferred to the transport when this
 * function is called. It is assumed that the encoder and transport share
 * the same definition of the content of the event, so that the transport
 * can effectively interpret the event and free any dynamically-allocated
 * memory.
 *
 * @param transport The event transport instance
 * @param event The event to be sent on the external transport
 * @param timeStamp Timestamps associated with this event
 * @return Event transport error code. If this is not
 * AP_EventTransport_OK, the getLastError() function should be called to
 * get a more detailed description of what went wrong.
 */
AP_EventTransportError (*sendTransportEvent)(
    struct AP_EventTransport* transport,
    AP_TransportEvent event,
    AP_TimestampSet* timeStamp);
```

addEventDecoder

```
/**
 * Add a named event decoder to the set of decoders known to the
 * transport. If the named decoder already exists, it should be
```

```

* replaced.
*
* @param transport The event transport instance
* @param name The name of the decoder to be added
* @param decoder The decoder object itself
*/
void (*addEventDecoder)(struct AP_EventTransport* transport,
    const AP_char8* name,
    struct AP_EventDecoder* decoder);

```

removeEventDecoder

```

/**
 * Remove a named event decoder from the set of decoders known to the
 * transport. If the named decoder does not exist, the function should do
 * nothing.
 *
 * @param transport The event transport instance
 * @param name The decoder to be removed
 */
void (*removeEventDecoder)(struct AP_EventTransport* transport,
    const AP_char8* name);

```

flushUpstream

```

/**
 * Flush any pending normalized events onto the external transport. The
 * transport may assume that the stop() function has been called before
 * this function, so in many cases no action will be required to complete
 * the flushing operation.
 *
 * @param transport The event transport instance
 * @return Event transport error code. If this is not
 * AP_EventTransport_OK, the getLastError() function should be called to
 * get a more detailed description of what went wrong.
 */
AP_EventTransportError (*flushUpstream)(
    struct AP_EventTransport* transport);

```

flushDownstream

```

/**
 * Flush any pending transport events into the decoder. The transport may
 * assume that the stop() function has been called before this function,
 * so in many cases no action will be required to complete the flushing
 * operation. Under no circumstances should any events be sent to the
 * Correlator after flushDownstream() has returned.
 *
 * @param transport The event transport instance
 * @return Event transport error code. If this is not
 * AP_EventTransport_OK, the getLastError() function should be called to
 * get a more detailed description of what went wrong.
 */
AP_EventTransportError (*flushDownstream)(
    struct AP_EventTransport* transport);

```

start

```

/**
 * Establish a connection and start processing incoming data from the
 * external transport.
 *
 * An adapter should send events to the correlator only after its start()
 * method is called and before the stop() method returns. Therefore we
 * strongly recommend that a transport should not change to a state where
 * it is possible to receive events from any external transport until the
 * start() method has been called. In many cases, adapters will also need
 * to communicate with service monitors in the correlator to ensure that
 * the required monitors and event definitions are injected before they
 * begin to process messages from the external system. This is necessary in

```

```

* order to avoid events from the adapter being lost if the correlator is
* not yet ready to parse and process them.
*
* @param transport The event transport instance
* @return Event transport error code. If this is not
* AP_EventTransport_OK, the getLastError() function should be called to
* get a more detailed description of what went wrong.
*/
AP_EventTransportError (*start)(struct AP_EventTransport* transport);

```

When the `start` function is invoked the event transport is effectively signaled to start accepting incoming messages and pass them onto a codec. Events should not be sent to the correlator until the `start` function is called.

It is up to the event transport to determine which codec to communicate with from the list of codecs made available to it through `addEventDecoder` and `removeEventDecoder`. Typically a configuration property would be used to specify the codec to be used. If a handle to the desired codec had been stored in a variable called `decoder` (of type `AP_EventDecoder*`) when `addEventDecoder` was called, an event could be passed on to the codec using:

```
decoder->functions->sendTransportEvent(decoder, event);
```

This codec function is described in ["C/C++ Codec Plug-in Development" on page 291](#).

stop

```

/**
 * Stop processing incoming data from the external transport, typically
 * by pausing or closing down connections.
 *
 * Adapter authors must ensure that no events are sent to the Correlator
 * after stop() has returned (the only exception being rare cases where the
 * transport sends buffered events in the Correlator in the
 * flushDownstream() method, which is called by the IAF after stop()).
 * If necessary any messages that are unavoidably received from the
 * transport after stop() has returned should be blocked, queued or simply
 * dropped.
 *
 * @param transport The event transport instance
 * @return Event transport error code. If this is not
 * AP_EventTransport_OK, the getLastError() function should be called to
 * get a more detailed description of what went wrong.
*/
AP_EventTransportError (*stop)(struct AP_EventTransport* transport);

```

Events should not be sent to the correlator after the `stop` function has returned. The `stop` method must wait for any other threads sending events to complete before the `stop` method returns.

getLastError

```

/**
 * getLastError
 *
 * Return the transport's stored error message, if any. The message
 * string is owned by the transport so should not be modified or freed by
 * the caller.
 *
 * @param transport The event transport instance
 * @return The last error message generated by the transport
 */
const AP_char8* (*getLastError)(struct AP_EventTransport* transport);
getStatus/**
 * getStatus
 *
 * Fill in the supplied AP_EventTransportStatus structure with up-to-date
 * status information for the transport. Note that any data pointed to by
 * the returned structure (such as strings) remains owned by the
 * transport. The caller must copy this data if it wishes to modify it.
 */

```



```

* @param codec The event transport instance
* @param status The status structure to be filled in
*/
void (*getStatus)(struct AP_EventTransport* transport,
    AP_EventTransportStatus* status);

```

The `AP_EventTransportStatus` structure contains four fields. The first field is a free-form text string that the transport can use to report any custom status information it might have. The `iaf_watch` tool will display the contents of this string. Note that the length of the status string is limited, currently to 1024 characters. Longer strings will be silently truncated. The next two fields report the total number of events received and sent by the transport. The last field, a pointer to an `AP_NormalisedEvent`, can contain custom information such as the state of the adapter.

The C/C++ transport plug-in development specification

Defining the transport function table

The `EventTransport.h` header file provides a definition for an `AP_EventTransport_Functions` structure. This defines a function table whose elements must be set to point to the implementations of the above functions. Its definition is as follows:

```

/**
 * AP_EventTransport_Functions
 *
 * Table of client visible functions exported by a transport library
 * instance. These functions declare the only operations that may be
 * performed by users of a transport.
 *
 * Note that all of these functions take an initial AP_EventTransport*
 * argument; this is analogous to the (hidden) 'this' pointer passed to
 * a C++ object when a member function is invoked on it.
 */
struct AP_EventTransport_Functions {
/**
 * updateProperties
 *
 * Update the configuration of the transport. The transport may assume
 * that stop(), flushUpstream() and flushDownstream() have all been called
 * before this function is invoked. The recommended procedure for
 * updating properties is to first compare the new property set with the
 * existing stored properties -- if there are no changes, no action should
 * be taken. Any pointer to the old property set becomes invalid as soon
 * as this function returns; any such pointers should therefore be
 * discarded in favour of the supplied new properties.
 *
 * @param transport The event transport instance
 * @param properties The new transport property set derived from the IAF
 * configuration file
 * @param timestampConfig Timestamp recording/logging settings
 * @return Event transport error code. If this is not
 * AP_EventTransport_OK, the getLastError() function should be called to
 * get a more detailed description of what went wrong.
 */
    AP_EventTransportError (*updateProperties)(
        struct AP_EventTransport* transport,
        AP_EventTransportProperties* properties,
        IAF_TimestampConfig* timestampConfig);
/**
 * sendTransportEvent
 *
 * Called by an event encoder to send a message to the external transport.
 * Ownership of the message is transferred to the transport when this
 * function is called. It is assumed that the encoder and transport share
 * the same definition of the content of the event, so that the transport
 * can effectively interpret the event and free any dynamically-allocated
 * memory.
 */

```

```

* @param transport The event transport instance
* @param event The event to be sent on the external transport
* @param timeStamp Timestamps associated with this event
* @return Event transport error code. If this is not
* AP_EventTransport_OK, the getLastError() function should be called to
* get a more detailed description of what went wrong.
*/
AP_EventTransportError (*sendTransportEvent)(
    struct AP_EventTransport* transport,
    AP_TransportEvent event,
    AP_TimestampSet* timeStamp);
/**
* addEventDecoder
*
* Add a named event decoder to the set of decoders known to the
* transport. If the named decoder already exists, it should be
* replaced.
*
* @param transport The event transport instance
* @param name The name of the decoder to be added
* @param decoder The decoder object itself
*/
void (*addEventDecoder)(struct AP_EventTransport* transport,
    const AP_char8* name, struct AP_EventDecoder* decoder);
/**
* removeEventDecoder
*
* Remove a named event decoder from the set of decoders known to the
* transport. If the named decoder does not exist, the function should do
* nothing.
*
* @param transport The event transport instance
* @param name The decoder to be removed
*/
void (*removeEventDecoder)(struct AP_EventTransport* transport,
    const AP_char8* name);
/**
* flushUpstream
*
* Flush any pending normalized events onto the external transport. The
* transport may assume that the stop() function has been called before
* this function, so in many cases no action will be required to complete
* the flushing operation.
*
* @param transport The event transport instance
* @return Event transport error code. If this is not
* AP_EventTransport_OK, the getLastError() function should be called to
* get a more detailed description of what went wrong.
*/
AP_EventTransportError (*flushUpstream)(
    struct AP_EventTransport* transport);
/**
* flushDownstream
*
* Flush any pending transport events into the decoder. The transport may
* assume that the stop() function has been called before this function,
* so in many cases no action will be required to complete the flushing
* operation. Under no circumstances should any events be sent to the
* Correlator after flushDownstream() has returned.
*
* @param transport The event transport instance
* @return Event transport error code. If this is not
* AP_EventTransport_OK, the getLastError() function should be called to
* get a more detailed description of what went wrong
*/
AP_EventTransportError (*flushDownstream)(
    struct AP_EventTransport* transport);
/**
* start
*

```

```

* Establish a connection and start processing incoming data from the
* external transport.
*
* An adapter should send events to the correlator only after its start()
* method is called and before the stop() method returns. Therefore we
* strongly recommend that a transport should not change to a state where
* it is possible to receive events from any external transport until the
* start() method has been called. In many cases, adapters will also need
* to communicate with service monitors in the correlator to ensure that
* the required monitors and event definitions are injected before they
* begin to process messages from the external system. This is necessary in
* order to avoid events from the adapter being lost if the correlator is
* not yet ready to parse and process them.
*
* @param transport The event transport instance
* @return Event transport error code. If this is not
* AP_EventTransport_OK, the getLastError() function should be called to
* get a more detailed description of what went wrong.
*/
AP_EventTransportError (*start)(struct AP_EventTransport* transport);
/**
* stop
*
* Stop processing incoming data from the external transport, typically
* by pausing or closing down connections.
*
* Adapter authors must ensure that no events are sent to the Correlator
* after stop() has returned (the only exception being rare cases where the
* transport sends buffered events in the Correlator in the
* flushDownstream() method, which is called by the IAF after stop()).
* If necessary any messages that are unavoidably received from the
* transport after stop() has returned should be blocked, queued or simply
* dropped.
*
* @param transport The event transport instance
* @return Event transport error code. If this is not
* AP_EventTransport_OK, the getLastError() function should be called to
* get a more detailed description of what went wrong.
*/
AP_EventTransportError (*stop)(struct AP_EventTransport* transport);
/**
* getLastError
*
* Return the transport's stored error message, if any. The message
* string is owned by the transport so should not be modified or freed by
* the caller.
*
* @param transport The event transport instance
* @return The last error message generated by the transport
*/
const AP_char8* (*getLastError)(struct AP_EventTransport* transport);
/**
* getStatus
*
* Fill in the supplied AP_EventTransportStatus structure with up-to-date
* status information for the transport. Note that any data pointed to by
* the returned structure (such as strings) remains owned by the
* transport. The caller must copy this data if it wishes to modify it.
*
* @param codec The event transport instance
* @param status The status structure to be filled in
*/
void (*getStatus)(struct AP_EventTransport* transport,
    AP_EventTransportStatus* status);
};

```

Note that the order of the function pointers within the function table is critical to the reliable operation of the IAF. However, the order that the function definitions appear within the plug-in source code, and indeed the names of the functions, are not important. Apama recommends that the

functions be declared `static`, so that they are not globally visible and can only be accessed via the function table.

It is therefore not obligatory to implement the functions documented above with the same names as per the definitions, as long as the mapping is performed correctly in an instantiation of `AP_EventTransport_Functions`. A definition in an event transport implementation would look as follows:

```
/**
 * Function table for the AP_EventTransport interface. The address of this
 * structure must be placed in the 'functions' field of the AP_EventTransport
 * object.
 */
static struct AP_EventTransport_Functions EventTransport_Functions
= {
    updateProperties,
    sendTransportEvent,
    addEventDecoder,
    removeEventDecoder,
    flushUpstream,
    flushDownstream,
    start,
    stop,
    getLastError,
    getStatus
};
```

The function table created above needs to be placed in an `AP_EventTransport` object. The definition of this structure is as follows:

```
/**
 * External (client-visible) interface to an IAF TIL plugin library. The
 * AP_EventTransport struct contains a table of function pointers, declared in
 * the AP_EventTransport_Functions struct above. The implementation of these
 * functions is private. Users of the transport library should invoke
 * functions on it as in the following example (transport is of type
 * AP_EventTransport*):
 *
 * i = transport->functions->sendTransportEvent(mapper, event, timestamp);
 */
struct AP_EventTransport {
    void* reserved;
    struct AP_EventTransport_Functions* functions;
};
```

and one such object needs to be created for every plug-in within its constructor function. Its first element, `reserved`, is a placeholder for any private data that the transport layer requires.

The C/C++ transport plug-in development specification

The transport constructor, destructor and info functions

Every event transport needs to implement a constructor function, a destructor function and an 'info' function. These methods are called by the IAF to (respectively) instantiate the event transport, to clean it up during unloading, and to provide information about the plug-in's capabilities.

`EventTransport.h` provides the following definition for a pointer to the constructor function:

AP_EventTransportCtorPtr

```
/**
 * Pointer to the constructor function for the transport library. Each TIL
 * plugin library must export a function with this signature, named using the
 * AP_EVENTTRANSPORT_CTOR_FUNCTION_NAME macro.
 *
 * Constructs a new instance of the event transport. This function will be
 * called by the adapter main program when the adapter starts up. The
 * transport should be created in a 'stopped' state, such that it is not
```

```

* actively processing data from the external transport. A call to the
* start() function is required to begin processing.
*
* Note that the input parameters (name, properties, timestampConfig) are
* owned by the caller. The IAF framework guarantees that properties and
* timestampConfig will remain valid until after a subsequent call to
* updateProperties(), so it is safe to hold a pointer to this structure.
* You should, however, copy the name string if you wish to keep it. The
* contents of the output parameter errMsg belongs to the transport.
*
* @param name The name of this transport instance
* @param properties Transport property set derived from the IAF config file
* @param err 'Out' parameter for error code if constructor fails
* @param errMsg 'Out' parameter for error message if constructor fails
* @param timestampConfig Timestamp recording/logging settings
* @return Pointer to a new transport instance or NULL if the constructor
* fails for some reason. In the case of failure, err and errMsg should be
* filled in with an appropriate error code and message describing the
* failure.
*/
typedef AP_EVENTTRANSPORT_API AP_EventTransport* (
    AP_EVENTTRANSPORT_CALL* AP_EventTransportCtorPtr)(AP_char8* name,
        AP_EventTransportProperties* properties, AP_EventTransportError* err,
        AP_char8** errMsg, IAF_TimestampConfig* timestampConfig);

```

Typically part of the work of this constructor would be a call to `updateProperties`, in order to set up the initial configuration of the plug-in.

The destructor function's related definition is as follows:

AP_EventTransportDtorPtr

```

/**
* Pointer to the destructor function for the transport library. Each TIL
* plugin library must export a function with this signature, named using
* the AP_EVENTTRANSPORT_DTOR_FUNCTION_NAME macro.
*
* Destroys an instance of the event transport that was previously created
* by AP_EventTransport_ctor. The transport may assume that it has been
* stopped and flushed before the destructor is called.
*
* @param transport The event transport object to be destroyed
*/
typedef AP_EVENTTRANSPORT_API void (
    AP_EVENTTRANSPORT_CALL* AP_EventTransportDtorPtr)(
        AP_EventTransport* obj);

```

while the info function is defined as:

AP_EventTransportInfoPtr

```

/**
* Pointer to the info function for the transport library. Each transport
* plugin library must export a function with this signature, named using
* the AP_EVENTTRANSPORT_INFO_FUNCTION_NAME macro.
*
* Returns basic information about the transports implemented by this
* shared library, along with the transport API version it was built
* against.
*
* @param version 'Out' parameter for version number of transport library.
* This should be the value of AP_EVENTTRANSPORT_VERSION that was defined
* in EventTransport.h when the transport was built. The IAF will check
* the version number to ensure that it can support all the capabilities
* offered by the transport
* @param capabilities 'Out' parameter for the capabilities of the
* transports provided by this library. Currently no such capabilities
* are defined, so the value of this field will be ignored by the IAF.
* However, all bits of the capabilities value are reserved for future use,
* so current transports should ensure that they set this value to zero.
*/

```

```
typedef AP_EVENTTRANSPORT_API void (
    AP_EVENTTRANSPORT_CALL* AP_EventTransportInfoPtr)(AP_uint32* version,
    AP_uint32* capabilities);
```

The IAF will search for these functions by the names `AP_EventTransport_ctor`, `AP_EventTransport_dtor` and `AP_EventTransport_info` when the library is loaded, so you must use these exact names when implementing them in a transport layer plug-in.

The C/C++ transport plug-in development specification

Other transport definitions

`EventTransport.h` also provides some additional definitions that the event transport author needs to be aware of.

First of these is the set of error codes that can be returned by the transport's functions:

```
/**
 * AP_EventTransportError
 *
 * Error codes that can be returned by transport library functions. The
 * enumeration values follow the normal UNIX convention of zero == OK and
 * non-zero == error.
 */
typedef enum {
    AP_EventTransport_OK = 0,           /* Everything is fine */
    AP_EventTransport_InternalError,    /* Some unspecified internal error occurred */
    AP_EventTransport_TransportFailure, /* Trouble reading/writing the external transport */
    AP_EventTransport_DecodingFailure,  /* Trouble sending transport event to decoder */
    AP_EventTransport_BadProperties,     /* Transport was passed an invalid property set */
    AP_EventTransport_CantStart         /* Transport could not start correctly */
} AP_EventTransportError;
```

Next is a definition for a configuration property. This corresponds to the properties that can be passed in as initialization or re-configuration parameters from the configuration file of the IAF.

```
/**
 * AP_EventTransportProperty
 *
 * A single transport property, corresponding to a <property> element
 * in the adapter configuration file.
 */
typedef struct {
    AP_char8* name;
    AP_char8* value;
} AP_EventTransportProperty;
```

Properties are passed to the event transport within an `AP_EventTransportProperties` structure:

```
/**
 * AP_EventTransportProperties
 *
 * Properties for the transport, extracted from the <transport> element
 * in the adapter configuration file.
 */
typedef struct {
    AP_char8* name;
    AP_EventTransportProperty** properties;
} AP_EventTransportProperties;
```

Finally, the status of a transport is reported in an `AP_EventTransportStatus` structure:

```
/**
 * AP_EventTransportStatus
 *
 * Transport status information structure, filled in by the getStatus call.
 */
typedef struct {
    AP_char8* status;
```

```

    AP_uint64 totalReceived;
    AP_uint64 totalSent;
    AP_NormalisedEvent* statusDictionary;
} AP_EventTransportStatus;

```

The C/C++ transport plug-in development specification

Transport utilities

The header files `AP_EventParser.h` and `AP_EventWriter.h` provide definitions for the Event Parser and Event Writer utilities. These utilities allow parsing and writing of the string form of reference types that are used by any `<map type="reference">` elements in the adapters configuration file. From the header files:

AP_EventParser

```

/**
 * Event Parser - a utility to parse events from strings
 *
 * An EventParser struct is created from a C string (NULL terminated). The
 * type of event can then be interrogated (the type member), though both
 * integers and floats are labelled as floats. The number of elements is
 * the number of fields in an event, entries in a sequence, or key,value
 * pairs in a dictionary.
 *
 * Individual fields can then be retrieved, either as a basic type, or as a
 * reference type - basic types retrieved as reference types have a single
 * element which must be retrieved using the relevant method. getRefValue
 * will return NULL if there are no more elements, and the basic type
 * methods will return 0/ false/ empty string.
 *
 * Dictionary entries must be obtained via the getDictPair function, which
 * sets two pointers to the key and value as reference types.
 *
 * No type checking is performed on the get methods; the behaviour when
 * there is a type mismatch is undefined.
 *
 * There are two variants of the get methods; the Next set maintain a
 * counter while those with an idx argument retrieve a specific element,
 * which does not affect the next counter. The next counter cannot be reset.
 *
 * Note that the implementation is lazy - it only parses stuff if requested
 * to. (though getNumberElements will parse all of that entity). Parsing is
 * cached, so calling getNumberElements repeatedly is cheap, and
 * getRefValue/ getString/ getDictPair will return the same objects if
 * called repeatedly with the same index.
 *
 * Where methods return a pointer (getString and getRefValue), that
 * pointer is valid until the top-level AP_EventParser is destroyed by
 * AP_EventParser_dtor (no need to destroy any of its sub- objects). A
 * well-formed program should have a matching number of AP_EventParser_ctor
 * and AP_EventParser_dtor calls, and only use string or refValue pointers
 * before AP_EventParser_dtor is called.
 */
AP_EventWriter
/**
 * Event Writer - a utility to create string forms of events.
 *
 * A utility class that can build string forms of events, sequences and
 * dictionaries.
 *
 * An AP_EventWriter object holds a number of fields, each of which may be a
 * basic type (string, int, float, boolean), or a reference type (event,
 * sequence, dictionary). Once the structure has been built up by adding
 * values, the toString method can be called. This returns a string that
 * the caller is responsible for freeing.
 *
 * Calling the destructor on an object will destroy all AP_EventWriter
 * objects it refers to in a recursive manner - thus, only the top-level
 * object should be destroyed, and it is not possible to share

```

```

* AP_EventWriter objects between two different containing events.
*
*/

```

The C/C++ transport plug-in development specification

Communication with the codec layer

If a transport layer plug-in is to be able to receive messages and then pass them on to the codec layer it must be able to communicate with appropriate *decoding codecs*. A *decoding* codec is one that can accept messages from the transport layer and parse them (decode them) into the normalized event format accepted by the Semantic Mapper.

When a codec is loaded into the IAF its details are passed to all transport layer plug-ins by calling their `addEventDecoder` function. This tells the transport layer plug-in the name of the decoding codec and provides a reference to its `AP_EventDecoder` structure.

The reference to `AP_EventDecoder` gives the transport layer plug-in access to the following functions:

sendTransportEvent

```

/**
 * Called by the event transport to decode an event and send it on to the
 * Semantic Mapper. Ownership of the message is transferred to the
 * decoder when this function is called. It is assumed that the encoder
 * and transport share the same definition of the content of the event, so
 * that the transport can effectively interpret the event and free any
 * dynamically-allocated memory.
 *
 * @param decoder The event decoder instance
 * @param event The event to be decoded
 * @param timeStamp Timestamps associated with this event
 * @return Event codec error code. If this is not AP_EventCodec_OK, the
 * getLastError() function of the decoder should be called to get a more
 * detailed description of what went wrong.
 */
AP_EventCodecError (*sendTransportEvent)(struct AP_EventDecoder* decoder,
    AP_TransportEvent event, AP_TimestampSet* timeStamp);

```

getLastError

```

/**
 * getLastError
 *
 * Return the decoder's stored error message, if any. The message string
 * is owned by the decoder so should not be modified or freed by the
 * caller.
 *
 * @param decoder The event decoder instance
 * @return The last error message generated by the decoder
 */
const AP_char8* (*getLastError)(struct AP_EventDecoder* decoder);

```

Assuming the reference to the `AP_EventDecoder` structure has been stored in a variable called `decoder`, the functions can be called as follows:

```

errorCode = decoder->functions->sendTransportEvent(decoder, event);
errorMessage = decoder->functions->getLastError(decoder);

```

The C/C++ transport plug-in development specification

Transport Example

As part of the IAF distribution Apama includes the `FileTransport` transport layer plug-in, implemented in the `samples\iaf_plugin\c\simple\FileTransport.c` source file.

The `FileTransport` plug-in can read and write messages from and to a text file, and it is recommended that developers examine this sample to see what a typical transport plug-in implementation looks like.

See ["IAF samples" on page 274](#) for more information about this sample. ["The File Transport plug-in" on page 125](#) describes how the `FileTransport` plug-in can be used in practice.

C/C++ Transport Plug-in Development

Getting started with transport layer plug-in development

In order to facilitate quick development of a transport layer plug-in your distribution includes a transport plug-in skeleton.

This file, called `skeleton-transport.c`, implements a complete transport layer plug-in that complies with the transport layer Plug-in Development Specification but where all the custom message source specific functionality is missing. The file is located in the `samples\iaf_plugin\c\skeleton` directory of your installation.

The skeleton starts a background thread to do the actual message reading. This is the only approach suitable, unless the external transport is able to call back into the transport layer plug-in.

In order to turn the skeleton into a fully operational message source specific transport layer plug-in, the plug-in author needs to fill in the gaps within the `updateProperties`, `sendTransportEvent`, `addEventDecoder`, `removeEventDecoder`, `flushUpstream`, `flushDownstream`, `start`, `stop`, `getLastError` and `getStatus` functions. These must implement their specified functionality in the context of the custom message source. The constructor, destructor and info functions are also likely to require adaptation.

The skeleton defines a structure, called `EventTransport_Internals`, to store all its private data, and this structure is placed within the `reserved` field of the `AP_EventTransport` object created within the constructor method. It is likely that this structure will need to be modified to contain additional data that the adapter might require.

Any custom initialization and communications code, such as code to connect and register with a message bus, or opening a database, etc., can either be placed in the constructor or in the primary worker thread's `run` method. Alternatively, one might need to place such code in the `updateProperties` method, which is called by the IAF at initialization time as well as whenever it is requested to reload the configuration file and thus resend the plug-in's properties.

The distribution also includes a `Makefile` (for use with GNU Make on UNIX) as well as a workspace file and `dsp` folder, for use with Microsoft's Visual Studio .NET on Microsoft Windows, for this skeleton, which can be adapted to compile your transport layer plug-in and link it against any custom libraries required.

Once a plug-in is built, it needs to be placed in a location where it can be picked up by the IAF.

This means that on Windows you either need to copy the `.dll` into the `bin` folder, or else place it somewhere which is on your 'path', that is a location that is referenced by the `PATH` environment variable.

On UNIX you either need to copy the `.so` into the `lib` directory, or else place it somewhere which is on your 'library path', that is a directory that is referenced by the `LD_LIBRARY_PATH` environment variable.

[C/C++ Transport Plug-in Development](#)

Chapter 10: C/C++ Codec Plug-in Development

■ The C/C++ codec Plug-in Development Specification	291
■ Transport Example	309
■ Getting Started with codec layer plug-in development	309

The *codec layer* is a layer of abstraction between the transport layer and the IAF's Semantic Mapper. It consists of one or more plug-in libraries that perform message *encoding* and/or *decoding*. Decoders translate *downstream* messages retrieved by the transport layer into the standard 'normalized event' format on which the Semantic Mapper's rules run. Encoders work in the opposite direction, encoding *upstream* normalized events into an appropriate format for transport layer plug-ins to send on. See "[The Integration Adapter Framework](#)" on page 241 for a full introduction to codec plug-ins and the IAF's architecture.

This topic includes the C/C++ Codec Plug-in Development Specification and additional information for developers of C/C++ event codecs. "[Java Codec Plug-in Development](#)" on page 326 provides analogous information about developing codec plug-ins in Java.

Before developing a new codec plug-in, it is worth considering whether one of the standard Apama IAF plug-ins could be used instead. "[Using Adapter Plug-ins](#)" on page 123 provides more information on the standard IAF codec plug-ins: `StringCodec` and `NullCodec`. The `StringCodec` plug-in codes normalized events as formatted text strings. The `NullCodec` plug-in is useful in situations where it does not make sense to decouple the codec and transport layers, and allows transport plug-ins to communicate with the Semantic Mapper directly using normalized events.

To configure the build for a codec plug-in:

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

```
iafcore$(APAMA_LIBRARY_VERSION).lib;apcommon$(APAMA_LIBRARY_VERSION).lib
```

Finally, select Project Properties > Linker > General > Additional Library Directories, and add

```
$(APAMA_HOME)\lib.
```

Developing Custom Adapters

The C/C++ codec Plug-in Development Specification

A codec plug-in needs to be structured as a dynamic shared library. In order for the IAF to be able to load and use it, it must comply with Apama's Codec Plug-in Development Specification. This

describes the overall format of a codec plug-in and the C/C++ functions it needs to implement so that its functionality is accessible by the IAF. The Specification also provides a mechanism for startup and configuration parameters to be passed to the plug-in from the IAF's configuration file.

Property names and values used by codec plug-ins must be in UTF-8 format.

A codec plug-in implementation must include the C header file `EventCodec.h`. This file can be found in the `include` directory. As a codec also needs to communicate both with a transport layer plug-in (or event transport) and with the Semantic Mapper, `EventTransport.h` and `SemanticMapper.h` also need to be included.

C/C++ Codec Plug-in Development

Codec functions to implement

`EventCodec.h` provides the definition for a number of functions whose implementation needs to be provided by the event transport author.

However, in contrast to the Transport Layer Plug-in Development Specification, the set of functions that need to be implemented varies depending on whether the codec is to implement only a message decoder, only a message encoder, or a bidirectional encoder/decoder.

In all cases implementations need to be provided for the following functions:

updateProperties

```
/**
 * Update the configuration of the codec. The codec may assume that
 * flushUpstream() and flushDownstream() have been called before this
 * function is invoked. The recommended procedure for updating properties
 * is to first compare the new property set with the existing stored
 * properties -- if there are no changes, no action should be taken. Any
 * pointer to the old property set becomes invalid as soon as this
 * function returns; any such pointers should therefore be discarded in
 * favour of the supplied new properties.
 *
 * @param codec The event codec instance
 * @param properties The new codec property set derived from the IAF
 * configuration file
 * @param timestampConfig Timestamp recording/logging settings
 * @return Event codec error code. If this is not AP_EventCodec_OK, the
 * getLastError() function of the codec should be called to get a more
 * detailed description of what went wrong.
 */
AP_EventCodecError (*updateProperties)(struct AP_EventCodec* codec,
    AP_EventCodecProperties* properties,
    IAF_TimestampConfig* timestampConfig);
```

It is recommended that `updateProperties` is invoked by the codec constructor.

getLastError

```
/**
 * Return the codec's stored error message, if any. The message string is
 * owned by the codec so should not be modified or freed by the caller.
 *
 * @param codec The event codec instance
 * @return The last error message generated by the codec
 */
const AP_char8* (*getLastError)(struct AP_EventCodec* codec);
```

getStatus

```
/**
 * Fill in the supplied AP_EventCodecStatus structure with up-to-date
 * status information for the codec. Note that any data pointed to by the
```

```

* returned structure (such as strings) remains owned by the codec. The
* caller must copy this data if it wishes to modify it.
*
* @param codec The event codec instance
* @param status The status structure to be filled in
*/
void (*getStatus)(struct AP_EventCodec* codec, AP_EventCodecStatus* status);

```

The `AP_EventCodecStatus` structure contains four fields. The first field is a free-form text string that the transport can use to report any custom status information it might have. The `iaf_watch` tool will display the contents of this string. Note that the length of the status string is limited, currently to 1024 characters. Longer strings will be silently truncated. The next two fields report the total number of events encoded and decoded by the codec. The last field, a pointer to an `AP_NormalisedEvent`, can contain custom information such as the state of the adapter.

The C/C++ codec Plug-in Development Specification

Codec encoder functions

If the codec is to implement an encoder, implementations need to be provided for the following functions:

```

sendNormalisedEvent
/**
 * Called by the Semantic Mapper to encode an event and send it on to the
 * event transport. Ownership of the message is transferred to the
 * encoder when this function is called.
 *
 * @param encoder The event encoder instance
 * @param event The event to be encoded
 * @param timeStamp Timestamps associated with this event
 * @return Event codec error code. If this is not AP_EventCodec_OK, the
 * getLastError() function of the encoder should be called to get a more
 * detailed description of what went wrong.
 */
AP_EventCodecError (*sendNormalisedEvent)(struct AP_EventEncoder* encoder,
    AP_NormalisedEvent* event, AP_TimestampSet* timeStamp);

```

```

flushUpstream
/**
 * Flush any pending normalized events into the attached event transport.
 * If event processing in the encoder is synchronous (as it usually will
 * be) this function need not do anything except return AP_EventCodec_OK.
 *
 * @param encoder The event encoder instance
 * @return Event codec error code. If this is not AP_EventCodec_OK, the
 * getLastError() function of the encoder should be called to get a more
 * detailed description of what went wrong.
 */
AP_EventCodecError (*flushUpstream)(struct AP_EventEncoder* encoder);

```

```

getLastError
/**
 * Return the encoder's stored error message, if any. The message string
 * is owned by the encoder so should not be modified or freed by the
 * caller.
 *
 * @param encoder The event encoder instance
 * @return The last error message generated by the encoder
 */
const AP_char8* (*getLastError)(struct AP_EventEncoder* encoder);

```

```

addEventTransport
/**
 * Add a named event transport to the set of transports known to the

```

```

* encoder.
*
* If the named transport already exists, it should be replaced.
*
* @param encoder The event encoder instance
* @param name The name of the transport to be added.
* @param transport The transport object itself
*/
void (*addEventTransport)(struct AP_EventEncoder* encoder,
    const AP_char8* name, struct AP_EventTransport* transport);

```

removeEventTransport

```

/**
* Remove a named event transport from the set of transports known to the
* encoder.
*
* If the named transport does not exist, the function should do nothing.
*
* @param encoder The event encoder instance
* @param name The name of the transport to be removed
*/
void (*removeEventTransport)(struct AP_EventEncoder* encoder,
    const AP_char8* name);

```

Codec functions to implement

Codec decoder functions

If the codec is to provide a decoder, implementations need to be provided for the following functions:

sendTransportEvent

```

/**
* Called by the event transport to decode an event and send it on to the
* Semantic Mapper. Ownership of the message is transferred to the
* decoder when this function is called. It is assumed that the encoder
* and transport share the same definition of the content of the event, so
* that the transport can effectively interpret the event and free any
* dynamically-allocated memory.
*
* @param decoder The event decoder instance
* @param event The event to be decoded
* @param timeStamp Timestamps associated with this event
* @return Event codec error code. If this is not AP_EventCodec_OK, the
* getLastError() function of the decoder should be called to get a more
* detailed description of what went wrong.
*/
AP_EventCodecError (*sendTransportEvent)(struct AP_EventDecoder* decoder,
    AP_TransportEvent event, AP_TimestampSet* timeStamp);

```

setSemanticMapper

```

/**
* Set the Semantic Mapper object to be used by the decoder. Currently
* only a single Semantic Mapper is supported in each adapter instance.
*
* @param decoder The event decoder instance
* @param mapper The Semantic Mapper to be used
*/
void (*setSemanticMapper)(struct AP_EventDecoder* decoder,
    AP_SemanticMapper* mapper);

```

flushDownstream

```

/**
* Flush any pending transport events into the attached Semantic Mapper.
* If event processing in the decoder is synchronous (as it usually will
* be) this function need not do anything except return AP_EventCodec_OK.
*

```



```

* @param decoder The event decoder instance
* @return Event codec error code. If this is not AP_EventCodec_OK, the
* getLastError() function of the decoder should be called to get a more
* detailed description of what went wrong.
*/
AP_EventCodecError (*flushDownstream)(struct AP_EventDecoder* decoder);

```

getLastError

```

/**
* Return the decoder's stored error message, if any. The message string
* is owned by the decoder so should not be modified or freed by the
* caller.
*
* @param decoder The event decoder instance
* @return The last error message generated by the decoder
*/
const AP_char8* (*getLastError)(struct AP_EventDecoder* decoder);

```

Codec functions to implement

Defining the codec function tables

In a transport layer plug-in, the plug-in author needs to provide a function table that tells the IAF which functions to call to invoke specific functionality.

The Codec Development Specification follows this model but depending on whether the codec being developed is an encoder, a decoder or an encoder/decoder, up to three function tables may need to be defined.

Note that the order of the function pointers within each function table is critical to the reliable operation of the IAF. However, the order that the function definitions appear within the plug-in source code, and indeed the names of the functions, are not important. Apama recommends that the functions be declared `static`, so that they are not globally visible and can only be accessed via the function table.

The C/C++ codec Plug-in Development Specification

The codec function table

Every codec needs to define a generic codec function table. The header file provides a definition for this as an `AP_EventCodec_Functions` structure. Its definition is as follows:

```

/**
* AP_EventCodec_Functions
*
* Table of client visible functions exported by a codec library instance.
* These functions declare the only operations that may be performed by users
* of a codec (but note that separate function tables exist for encoding-
* and decoding-specific functions).
*
* Note that all of these functions take an initial AP_EventCodec*
* argument; this is analogous to the (hidden) 'this' pointer passed to a
* C++ object when a member function is invoked on it.
*/
struct AP_EventCodec_Functions {
    /**
    * updateProperties
    *
    * Update the configuration of the codec. The codec may assume that
    * flushUpstream() and flushDownstream() have been called before this
    * function is invoked. The recommended procedure for updating properties
    * is to first compare the new property set with the existing stored
    * properties -- if there are no changes, no action should be taken. Any
    * pointer to the old property set becomes invalid as soon as this
    * function returns; any such pointers should therefore be discarded in

```

```

* favour of the supplied new properties.
*
* @param codec The event codec instance
* @param properties The new codec property set derived from the IAF
* configuration file
* @param timestampConfig Timestamp recording/logging settings
* @return Event codec error code. If this is not AP_EventCodec_OK, the
* getLastError() function of the codec should be called to get a more
* detailed description of what went wrong.
*/
AP_EventCodecError (*updateProperties)(struct AP_EventCodec* codec,
    AP_EventCodecProperties* properties,
    IAF_TimestampConfig* timestampConfig);
/**
* getLastError
*
* Return the codec's stored error message, if any. The message string is
* owned by the codec so should not be modified or freed by the caller.
*
* @param codec The event codec instance
* @return The last error message generated by the codec
*/
const AP_char8* (*getLastError)(struct AP_EventCodec* codec);
/**
* getStatus
*
* Fill in the supplied AP_EventCodecStatus structure with up-to-date
* status information for the codec. Note that any data pointed to by the
* returned structure (such as strings) remains owned by the codec. The
* caller must copy this data if it wishes to modify it.
*
* @param codec The event codec instance
* @param status The status structure to be filled in
*/
void (*getStatus)(struct AP_EventCodec* codec,
    AP_EventCodecStatus* status);
};

```

where the library functions `updateProperties`, `getLastErrorCodec` and `getStatus` are being defined as being the implementations of the Codec Development Specification's `updateProperties`, `getLastError` and `getStatus` function definitions respectively.

Defining the codec function tables

The codec encoder function table

If the codec being implemented is to act as an encoder it needs to implement the encoder functions listed previously and map them in an encoder function table. This structure is defined in `EventCodec.h` as an `AP_EventEncoder_Functions` struct:

```

/**
* AP_EventEncoder_Functions
*
* Table of client visible functions exported by the encoder part of a
* codec library instance. These functions declare the only operations
* that may be performed by users of an encoder.
*
* Note that all of these functions take an initial AP_EventEncoder*
* argument; this is analogous to the (hidden) 'this' pointer passed to
* a C++ object when a member function is invoked on it.
*/
struct AP_EventEncoder_Functions {
    /**
    * sendNormalisedEvent
    *
    * Called by the Semantic Mapper to encode an event and send it on to the
    * event transport. Ownership of the message is transferred to the
    * encoder when this function is called.
    */

```

```

*
* @param encoder The event encoder instance
* @param event The event to be encoded
* @param timeStamp Timestamps associated with this event
* @return Event codec error code. If this is not AP_EventCodec_OK, the
* getLastError() function of the encoder should be called to get a more
* detailed description of what went wrong.
*/
AP_EventCodecError (*sendNormalisedEvent)(struct AP_EventEncoder* encoder,
    AP_NormalisedEvent* event, AP_TimestampSet* timeStamp);
/**
* flushUpstream
*
* Flush any pending normalized events into the attached event transport.
* If event processing in the encoder is synchronous (as it usually will
* be) this function need not do anything except return AP_EventCodec_OK.
*
* @param encoder The event encoder instance
* @return Event codec error code. If this is not AP_EventCodec_OK, the
* getLastError() function of the encoder should be called to get a more
* detailed description of what went wrong.
*/
AP_EventCodecError (*flushUpstream)(struct AP_EventEncoder* encoder);
/**
* getLastError
*
* Return the encoder's stored error message, if any. The message string
* is owned by the encoder so should not be modified or freed by the
* caller.
*
* @param encoder The event encoder instance
* @return The last error message generated by the encoder
*/
const AP_char8* (*getLastError)(struct AP_EventEncoder* encoder);
/**
* addEventTransport
*
* Add a named event transport to the set of transports known to the
* encoder.
*
* If the named transport already exists, it should be replaced.
*
* @param encoder The event encoder instance
* @param name The name of the transport to be added.
* @param transport The transport object itself
*/
void (*addEventTransport)(struct AP_EventEncoder* encoder,
    const AP_char8* name, struct AP_EventTransport* transport);
/**
* removeEventTransport
*
* Remove a named event transport from the set of transports known to the
* encoder.
*
* If the named transport does not exist, the function should do nothing.
*
* @param encoder The event encoder instance
* @param name The name of the transport to be removed
*/
void (*removeEventTransport)(struct AP_EventEncoder* encoder,
    const AP_char8* name);
};

```

In the implementation of an encoding codec, this function table could be implemented as follows:

```

/**
* Function table for the AP_EventEncoder interface. The address of this
* structure will be placed in the 'functions' field of the embedded
* AP_EventEncoder object.
*/

```

```
static struct AP_EventEncoder_Functions EventEncoder_Functions = {
    sendNormalisedEvent,
    flushUpstream,
    getLastErrorEncoder,
    addEventTransport,
    removeEventTransport
};
```

This time, the library functions `sendNormalisedEvent`, `flushUpstream`, `getLastErrorEncoder`, `addEventTransport` and `removeEventTransport` are being defined as the implementations of the Codec Development Specification's `sendNormalisedEvent`, `flushUpstream`, `getLastError`, `addEventTransport` and `removeEventTransport` function definitions respectively.

Defining the codec function tables

The codec decoder function table

If the codec being implemented is to act as a decoder it needs to implement the decoder functions listed previously and map them in a decoder function table. This structure is defined in `EventCodec.h` as an `AP_EventDecoder_Functions` structure:

```
/**
 * AP_EventDecoder_Functions
 *
 * Table of client visible functions exported by the decoder part of a
 * codec library instance. These functions declare the only operations that
 * may be performed by users of a decoder.
 *
 * Note that all of these functions take an initial AP_EventDecoder*
 * argument; this is analogous to the (hidden) 'this' pointer passed to
 * a C++ object when a member function is invoked on it.
 */
struct AP_EventDecoder_Functions {
    /**
     * sendTransportEvent
     *
     * Called by the event transport to decode an event and send it on to the
     * Semantic Mapper. Ownership of the message is transferred to the
     * decoder when this function is called. It is assumed that the encoder
     * and transport share the same definition of the content of the event, so
     * that the transport can effectively interpret the event and free any
     * dynamically-allocated memory.
     *
     * @param decoder The event decoder instance
     * @param event The event to be decoded
     * @param timeStamp Timestamps associated with this event
     * @return Event codec error code. If this is not AP_EventCodec_OK, the
     *         getLastError() function of the decoder should be called to get a more
     *         detailed description of what went wrong.
     */
    AP_EventCodecError (*sendTransportEvent)(struct AP_EventDecoder* decoder,
        AP_TransportEvent event, AP_TimestampSet* timeStamp);
    /**
     * setSemanticMapper
     *
     * Set the Semantic Mapper object to be used by the decoder. Currently
     * only a single Semantic Mapper is supported in each adapter instance.
     *
     * @param decoder The event decoder instance
     * @param mapper The Semantic Mapper to be used
     */
    void (*setSemanticMapper)(struct AP_EventDecoder* decoder,
        AP_SemanticMapper* mapper);
    /**
     * flushDownstream
     *
     * Flush any pending transport events into the attached Semantic Mapper.
     */
}
```

```

* If event processing in the decoder is synchronous (as it usually will
* be) this function need not do anything except return AP_EventCodec_OK.
*
* @param decoder The event decoder instance
* @return Event codec error code. If this is not AP_EventCodec_OK, the
* getLastError() function of the decoder should be called to get a more
* detailed description of what went wrong.
*/
AP_EventCodecError (*flushDownstream)(struct AP_EventDecoder* decoder);
/**
* getLastError
*
* Return the decoder's stored error message, if any. The message string
* is owned by the decoder so should not be modified or freed by the
* caller.
*
* @param decoder The event decoder instance
* @return The last error message generated by the decoder
*/
const AP_char8* (*getLastError)(struct AP_EventDecoder* decoder);
};

```

In the implementation of a decoding codec, this function table could be implemented as follows:

```

/**
* EventDecoder_Functions
*
* Function table for the AP_EventDecoder interface. The address of this
* structure will be placed in the 'functions' field of the embedded
* AP_EventDecoder object.
*/
static struct AP_EventDecoder_Functions EventDecoder_Functions = {
    sendTransportEvent,
    setSemanticMapper,
    flushDownstream,
    getLastErrorDecoder
};

```

As before, this definition defines a number of library functions as the implementations of the function definitions specified in the Codec Development Specification.

Defining the codec function tables

Registering the codec function tables

The encoding and decoding function tables created above need to be placed in the relevant object, `AP_EventEncoder` and `AP_EventDecoder`. These, together with the generic function table, need to be placed in an `AP_EventCodec` object. The definitions of these structures are as follows:

```

/**
* AP_EventEncoder
*
* External (client-visible) interface to the encoder part of a codec
* plugin library. The AP_EventEncoder struct contains a table of
* function pointers, declared in the AP_EventEncoder_Functions struct
* above. The implementation of these functions is private. Users of
* the encoder should invoke functions on it as in the following example
* (encoder is of type AP_EventEncoder*):
*
* i = encoder->functions->sendNormalisedEvent(encoder, event, timestamps);
*/
struct AP_EventEncoder {
    /**
    * Pointer to private internal data.
    */
    void* reserved;
    /**
    * Function table of encoder operations. See documentation for
    * AP_EventEncoder_Functions for details.
    */
};

```

```

    */
    struct AP_EventEncoder_Functions* functions;
};
/**
 * AP_EventDecoder
 *
 * External (client-visible) interface to the decoder part of a codec
 * plugin library. The AP_EventDecoder struct contains a table of
 * function pointers, declared in the AP_EventDecoder_Functions struct
 * above. The implementation of these functions is private. Users of the
 * decoder should invoke functions on it as in the following example (
 * decoder is of type AP_EventDecoder*):
 *
 * i = decoder->functions->sendTransportEvent(decoder, event, timestamps);
 */
struct AP_EventDecoder {
    /**
     * Pointer to private internal data.
     */
    void* reserved;
    /**
     * Function table of decoder operations. See documentation for
     * AP_EventDecoder_Functions for details.
     */
    struct AP_EventDecoder_Functions* functions;
};
/**
 * AP_EventCodec
 *
 * External (client-visible) interface to an IAF codec plug-in library. The
 * AP_EventCodec struct contains pointers to the encoder and decoder parts
 * of the codec (either of these may be NULL if the codec operates in one
 * direction only) and a table of function pointers, declared in the
 * AP_EventCodec_Functions struct above. The implementation of these
 * functions is private. Users of the codec should invoke functions on it
 * as in the following example (codec is of type AP_EventCodec*):
 *
 * i = codec->functions->updateProperties(codec, properties, timestampConfig);
 */
struct AP_EventCodec {
    /**
     * Pointer to private internal data.
     */
    void* reserved;
    /**
     * Function table of codec operations. See documentation for
     * AP_EventCodec_Functions for details.
     */
    struct AP_EventCodec_Functions* functions;
    /**
     * Pointer to embedded encoder.
     */
    AP_EventEncoder* encoder;
    /**
     * Pointer to embedded decoder.
     */
    AP_EventDecoder* decoder;
};

```

The first element of each structure, `reserved`, is a placeholder for any private data that the encoder, decoder, or the generic codec code requires.

An `AP_EventCodec` object needs to be created for every plug-in within its constructor function. As indicated in the descriptive text, the encoder and decoder fields in it may be set to `NULL` if the codec does not implement the respective functionality, although clearly it is meaningless to have both set to `NULL`.

Defining the codec function tables

The codec constructor, destructor and info functions

Every event codec needs to implement a constructor function, a destructor function and an ‘info’ function. These methods are called by the IAF to (respectively) to instantiate the event codec, to clean it up during unloading, and to provide information about the plug-in’s capabilities.

EventCodec.h provides the following definition for a pointer to the constructor function:

AP_EventCodecCtorPtr

```
/**
 * Pointer to the constructor function for the codec library. Each codec
 * plugin library must export a function with this signature, named using
 * the AP_EVENTCODEC_CTOR_FUNCTION_NAME macro.
 *
 * Constructs a new instance of the event codec. This function will be
 * called by the adapter main program when the adapter starts up.
 *
 * Note that the input parameters (name, properties, timestampConfig) are
 * owned by the caller. The IAF framework guarantees that properties and
 * timestampConfig will remain valid until after a subsequent call to
 * updateProperties(), so it is safe to hold a pointer to this structure.
 * You should, however, copy the name string if you wish to keep it. The
 * contents of the output parameter errMsg belongs to the codec.
 *
 * @param name The name of this codec instance, also found in properties
 * @param properties Codec property set derived from the IAF config file
 * @param err 'Out' parameter for error code if constructor fails
 * @param errMsg 'Out' parameter for error message if constructor fails
 * @param timestampConfig Timestamp recording/logging settings
 * @return Pointer to a new codec instance or NULL if the constructor fails
 * for some reason. In the case of failure, err and errMsg should be filled
 * in with an appropriate error code and message describing the failure.
 */
typedef AP_EVENTCODEC_API AP_EventCodec* (
    AP_EVENTCODEC_CALL* AP_EventCodecCtorPtr)(
        AP_char8* name, AP_EventCodecProperties* properties,
        AP_EventCodecError* err, AP_char8** errMsg,
        IAF_TimestampConfig* timestampConfig);
```

while the destructor function definition is as follows:

AP_EventCodecDtorPtr

```
/**
 * Pointer to the destructor function for the codec library. Each codec
 * plugin library must export a function with this signature, named using
 * the AP_EVENTCODEC_DTOR_FUNCTION_NAME macro.
 *
 * Destroys an instance of the event codec that was previously created by
 * AP_EventCodec_ctor. The codec may assume that it has been flushed
 * before the destructor is called.
 *
 * @param codec The event codec object to be destroyed
 */
typedef AP_EVENTCODEC_API void
    (AP_EVENTCODEC_CALL* AP_EventCodecDtorPtr)(AP_EventCodec* codec);
```

The IAF will search for these functions by the names `AP_EventCodec_ctor` and `AP_EventCodec_dtor` when the library is loaded, so you must use these exact names when implementing a codec plug-in.

In addition, every codec needs to implement an ‘information’ function. This is called by the IAF to obtain information as to the capabilities (encoder/decoder) of the codec. The definition is:

AP_EventCodecInfoPtr

```
/**
 * Pointer to the info function for the codec library. Each codec plugin
 * library must export a function with this signature, named using the
```



```

* AP_EVENTCODEC_INFO_FUNCTION_NAME macro.
*
* Returns basic information about the codecs implemented by this shared
* library, specifically whether they are encoders, decoders or
* bidirectional codecs.
*
* @param version 'Out' parameter for version number of codec library. This
* should be the value of AP_EVENTCODEC_VERSION that was defined in
* EventCodec.h when the codec was built. The IAF will check the version
* number to ensure that it can support all the capabilities offered by the
* codec.
* @param capabilities 'Out' parameter for the capabilities of the codecs
* provided by this library. This should be the sum of the appropriate
* AP_EVENTCODEC_CAP_* constants found in EventCodec.h.
*/
typedef AP_EVENTCODEC_API void
    (AP_EVENTCODEC_CALL* AP_EventCodecInfoPtr) (AP_uint32* version,
        AP_uint32* capabilities);

```

The IAF will search for and call this function by the name `AP_EventCodec_info`.

The C/C++ codec Plug-in Development Specification

Other codec definitions

`EventCodec.h` also provides some additional definitions that the codec author needs to be aware of.

First of these are the codec capability bits. These are returned by the ‘information’ function previously illustrated to define whether the codec can decode or encode messages.

```

#define AP_EVENTCODEC_CAP_ENCODER 0x0001
#define AP_EVENTCODEC_CAP_DECODER 0x0002

```

Next is the set of error codes that can be returned by the codec’s functions:

```

/**
 * AP_EventCodecError
 *
 * Error codes that can be returned by codec library functions. The
 * enumeration values follow the normal UNIX convention of zero == OK and
 * non-zero == error.
 */
typedef enum {
    AP_EventCodec_OK = 0,           /* Everything is fine */
    AP_EventCodec_InternalError,    /* Some unspecified internal error occurred */
    AP_EventCodec_EncodingFailure,  /* Couldn't encode an incoming normalized event */
    AP_EventCodec_DecodingFailure,  /* Couldn't decode an incoming customer event */
    AP_EventCodec_TransportFailure, /* Trouble sending encoded event to transport */
    AP_EventCodec_MappingFailure,   /* Trouble sending decoded event to Semantic Mapper */
    AP_EventCodec_BadProperties      /* Codec was passed an invalid property set */
} AP_EventCodecError;

```

Next is a definition for a configuration property. This corresponds to the properties that can be passed in as initialization or re-configuration parameters from the configuration file of the IAF.

```

/**
 * AP_EventCodecProperty
 *
 * A single codec property, corresponding to a <property> element in the
 * adapter configuration file.
 */
typedef struct {
    AP_char8* name;
    AP_char8* value;
} AP_EventCodecProperty;

```

Properties are passed to the event transport within an `AP_EventCodecProperties` structure:

```

/**
 * AP_EventCodecProperties

```

```

*
* Properties for the codec, extracted from the <codec> element in the
* adapter configuration file.
*/
typedef struct {
    AP_char8* name;
    AP_EventCodecProperty** properties;
} AP_EventCodecProperties;

```

Finally, the status of a codec is reported in an `AP_EventCodecStatus` structure:

```

/**
 * AP_EventCodecStatus
 *
 * Codec status information structure, filled in by the getStatus call.
 */
typedef struct {
    AP_char8* status;
    AP_uint64 totalDecoded;
    AP_uint64 totalEncoded;
    AP_NormalisedEvent* statusDictionary;
} AP_EventCodecStatus;

```

You are advised to peruse `EventCodec.h` for the complete definitions. `EventTransport.h` and `SemanticMapper.h` are also relevant as they define the functions that a codec author can invoke within the transport layer and the Semantic Mapper, respectively.

The C/C++ codec Plug-in Development Specification

Codec utilities

The header files `AP_EventParser.h` and `AP_EventWriter.h` provide definitions for the Event Parser and Event Writer utilities. These utilities allow parsing and writing of the string form of reference types that are used by any `<map type="reference">` elements in the adapters configuration file. From the header files:

AP_EventParser

```

/**
 * Event Parser - a utility to parse events from strings
 *
 * An EventParser struct is created from a C string (NULL terminated). The
 * type of event can then be interrogated (the type member), though both
 * integers and floats are labelled as floats. The number of elements is
 * the number of fields in an event, entries in a sequence, or key,value
 * pairs in a dictionary.
 *
 * Individual fields can then be retrieved, either as a basic type, or as a
 * reference type - basic types retrieved as reference types have a single
 * element which must be retrieved using the relevant method. getRefValue
 * will return NULL if there are no more elements, and the basic type
 * methods will return 0/ false/ empty string.
 *
 * Dictionary entries must be obtained via the getDictPair function, which
 * sets two pointers to the key and value as reference types.
 *
 * No type checking is performed on the get methods; the behaviour when
 * there is a type mismatch is undefined.
 *
 * There are two variants of the get methods; the Next set maintain a
 * counter while those with an idx argument retrieve a specific element,
 * which does not affect the next counter. The next counter cannot be reset.
 *
 * Note that the implementation is lazy - it only parses stuff if requested
 * to. (though getNumberElements will parse all of that entity). Parsing is
 * cached, so calling getNumberElements repeatedly is cheap, and
 * getRefValue/ getString/ getDictPair will return the same objects if
 * called repeatedly with the same index.

```

```

*
* Where methods return an pointer (getString and getRefValue), that
* pointer is valid until the top-level AP_EventParser is destroyed by
* AP_EventParser_dtor (no need to destroy any of its sub- objects). A
* well-formed program should have a matching number of AP_EventParser_ctor
* and AP_EventParser_dtor calls, and only use string or refValue pointers
* before AP_EventParser_dtor is called.
*
*/AP_EventWriter

/**
* Event Writer - a utility to create string forms of events.
*
* A utility class that can build string forms of events, sequences and
* dictionaries.
* An AP_EventWriter object holds a number of fields, each of which may be
* a basic type (string, int, float, boolean), or a reference type (event,
* sequence, dictionary). Once the structure has been built up by adding
* values, the toString method can be called. This returns a string that
* the caller is responsible for freeing.
* Calling the destructor on an object will destroy all AP_EventWriter
* objects it refers to in a recursive manner - thus, only the top-level
* object should be destroyed, and it is not possible to share
* AP_EventWriter objects between two different containing events.
*
*/

```

The C/C++ codec Plug-in Development Specification

Communication with other layers

A decoding codec plug-in's role is to decode messages from a transport layer plug-in into a normalized format that can be processed by the Semantic Mapper. To achieve this it needs to be able to communicate with the Semantic Mapper. The accessible Semantic Mapper functionality is presented in `SemanticMapper.h`.

When a decoding codec starts it is passed a handle to an `AP_SemanticMapper` object through its `setSemanticMapper` function. This object is defined in `SemanticMapper.h` as follows:

```

**
* AP_SemanticMapper
*
* External (client-visible) interface to the Semantic Mapper component of
* an IAF instance. The AP_SemanticMapper struct contains contains a table
* of function pointers, declared in the AP_SemanticMapper_Functions struct
* above. The implementation of these functions is private. Users of the
* Semantic Mapper should invoke functions on it as in the following
* example (mapper is of type AP_SemanticMapper*):
*
* i = mapper->functions->sendNormalisedEvent(mapper, event);
*/
struct AP_SemanticMapper{
    /**
     * Pointer to private internal data.
     */
    void* reserved;
    /**
     * Function table of Semantic Mapper operations. See documentation for
     * AP_SemanticMapper_Functions for details.
     */
    struct AP_SemanticMapper_Functions* functions;
};
typedef struct AP_SemanticMapper AP_SemanticMapper;

```

where `functions`, (of type `AP_SemanticMapper_Functions*`) points to the definitions for two functions: `sendNormalisedEvent` and `getLastError`:

```

/**

```

```

* AP_SemanticMapper_Functions
*
* Table of client visible functions exported by a Semantic Mapper
* instance. These functions declare the only operations that may be
* performed by users of a Semantic Mapper.
*
* Note that all of these functions take an initial AP_SemanticMapper*
* argument; this is analogous to the (hidden) 'this' pointer passed to a
* C++ object when a member function is invoked on it.
*/
struct AP_SemanticMapper_Functions {
    /**
     * sendNormalisedEvent
     *
     * Send a customer-specific event to the Semantic Mapper. The event will
     * be translated into a single Apama event that will be queued for
     * injection into the Engine.
     *
     * @param mapper The Semantic Mapper to send the event to.
     * @param event The event to send, represented as a set of name-value
     * pairs.
     * @param timeStamp Timestamps associated with this event
     * @return Semantic Mapper error code. If this is not
     * AP_SemanticMapper_OK, the getLastError() function should be called to
     * get a more detailed description of what went wrong.
     */
    AP_SemanticMapperError (*sendNormalisedEvent)(
        struct AP_SemanticMapper* mapper, AP_NormalisedEvent* event,
        AP_TimestampSet* timeStamp);
    /**
     * getLastError
     *
     * Return the Semantic Mapper's stored error message, if any. The message
     * string is owned by the mapper so should not be modified or freed by the
     * caller.
     *
     * @param mapper The Semantic Mapper instance
     * @return The last error message generated by the mapper
     */
    const AP_char8* (*getLastError)(struct AP_SemanticMapper* mapper);
};

```

Code inside a decoding codec that calls these functions on the Semantic Mapper looks as follows. Assuming that `mapper` holds a reference to the `AP_SemanticMapper` object:

```
errorCode = mapper->functions->sendNormalisedEvent(mapper, NormalisedEvent);
```

and likewise for `getLastError`.

The error codes that may be returned by `sendNormalisedEvent` are as follows:

```

/**
 * AP_SemanticMapperError
 *
 * Error codes that can be returned by operations on the Semantic Mapper. The
 * enumeration values follow the normal UNIX convention of zero == OK and
 * non-zero == error.
 */
typedef enum {
    AP_SemanticMapper_OK = 0,           /* Everything is fine */
    AP_SemanticMapper_InternalError,    /* Some unspecified internal error occurred */
    AP_SemanticMapper_MappingFailure,   /* Couldn't convert customer event to Apama event */
    AP_SemanticMapper_InjectionFailure /* Couldn't queue converted event for injection into Engine */
} AP_SemanticMapperError;

```

On the other hand, an encoding codec plug-in's role is to encode messages in normalized format into some specific format that can then be accepted by a transport layer plug-in for transmission to an external message sink (like a message bus). To achieve this it needs to be able to communicate with a transport layer plug-in loaded in the IAF.

When an encoding codec starts its `addEventTransport` function will be called once for each available transport. For each, it is passed a handle to an `AP_EventTransport` object. This object is defined in `EventTransport.h` and was described in detail in ["C/C++ Transport Plug-in Development" on page 277](#). As stated in ["Defining the codec function tables" on page 295](#), it contains a pointer to `AP_EventTransport_Functions`, which in turn references the functions available in the transport layer plug-in. Of these, only two are relevant to the author of an encoding codec:

sendTransportEvent

```
AP_EventTransportError (*sendTransportEvent)(
    struct AP_EventTransport* transport, AP_TransportEvent event,
    AP_TimestampSet* timeStamp);
```

getLastError

```
const AP_char8* (*getLastError)(struct AP_EventTransport* transport);
```

Code inside an encoding codec that calls these functions on the transport layer plug-in looks as follows. Assuming that `transport` holds a reference to the `AP_EventTransport` object:

```
errorCode = transport->functions->sendTransportEvent(transport, event);
```

and likewise for `getLastError`.

The C/C++ codec Plug-in Development Specification

Working with normalized events

The function of a decoding codec plug-in is to convert incoming messages into a standard normalized event format that can be processed by the Semantic Mapper. Events sent upstream to an encoding codec plug-in are provided to the plug-in in this same format.

Normalized events are essentially dictionaries of name-value pairs, where the names and values are both character strings. Each name-value pair nominally represents the name and content of a single field from an event, but users of the data structure are free to invent custom naming schemes to represent more complex event structures. Names must be unique within a given event. Values may be empty or `NULL`.

Some examples of normalized event field values for different types are:

- `string` `"a string"`
- `integer` `"1"`
- `float` `"2.0"`
- `decimal` `"100.0d"`
- `sequence<boolean>` `"[true,false]"`
- `dictionary<float,integer>` `"{2.3:2,4.3:5}"`
- `SomeEvent` `"SomeEvent(12)"`

Note: When assigning names to fields in normalized events, keep in mind that the `fields` and `transport` attributes for event mapping conditions and event mapping rules both use a list of fields delimited by spaces or commas. This means, for example that `<id fields="Exchange EX,foo" test="==" value="LSE"/>` will successfully match a field called "Exchange", "EX" or "foo", but *not* a field called "Exchange EX,foo". While fields with spaces or commas in their names may be included in a payload dictionary in upstream or downstream directions, they cannot be referenced directly in mapping or id rules.

To construct strings for the normalized event fields representing container types (dictionaries, sequences, or nested events), use the `EventWriter` utility found in the `AP_EventWriter.h` header file, which is located in the `include` directory of the Apama installation. The following examples show how to add a sequence and a dictionary to a normalized event (note the escape character “\” used in order to insert a quotation mark into a string).

```
#include <AP_EventWriter.h>
AP_EventWriter *map, *list;
AP_NormalisedEvent *event;
AP_EventWriterValue key, value;
list=AP_EventWriter_ctor(AP_SEQUENCE, NULL);
list->addString(list, "abc");
list->addString(list, "de\"f");
map=AP_EventWriter_ctor(AP_DICTIONARY, NULL);
    key.stringValue="key1"; value.stringValue="value";
map->addDictValue(map, AP_STRING, key, AP_STRING, value);
key.stringValue="key\"{}2";
value.stringValue="value\"{}2";
map->addDictValue(map, AP_STRING, key, AP_STRING, value);
event=AP_NormalisedEvent_ctor();
event->functions->addQuick(event, "mySequenceField",
event->functions->list->toString(list));
event->functions->event->functions->addQuick(event,
    "myDictionaryField", event->functions->map->toString(map));
AP_EventWriter_dtor(list);
AP_EventWriter_dtor(map);
```

Fields names and values of normalized events are in UTF-8 format. This means that the writer of the codec needs to ensure that downstream events are correctly formed and the codec should expect to handle UTF-8 coming upstream.

The `NormalisedEvent.h` header file defines objects and functions that make up a special programming interface for constructing and examining normalized events.

`NormalisedEvent.h` contains two main structures: `AP_NormalisedEvent` which represents a single normalized event, and `AP_NormalisedEventIterator` which can be used to step through the contents of a normalized event structure.

The C/C++ codec Plug-in Development Specification

The `AP_NormalisedEvent` structure

The `AP_NormalisedEvent` structure has a pointer to a table of client-visible functions exported by the object called `AP_NormalisedEvent_Functions`. This function table provides access to the operations that may be performed on the event object:

- `size` - Return the number of elements (name-value pairs) currently stored by the event.
- `empty` - Check whether the event is empty or not.
- `add` - Add a new name-value pair to the event. The given name and value will be copied into the event. If an element with the same name already exists, it will not be overwritten. This returns an iterator into the events at the point where the new element was added.
- `addQuick` - Add a new name-value pair to the event. The given name and value will be copied into the event. If an element with the same name already exists, it will not be overwritten.
- `remove` - Remove the named element from the normalized event.
- `removeAll` - Remove all elements from the normalized event. The `empty` function will return `AP_TRUE` after this function has been called.

- `replace` - Change the value of a named element in the normalized event. If an element with the given name already exists, its value will be replaced with a copy of the given value. Otherwise, a new element is created just as though `addQuick` had been called.
- `exists` - Check whether a given element exists in the normalized event.
- `find` - Search for a named element in the normalized event. Returns an iterator into the event at the point where the element was located.
- `findValue` - Search for a named element in the normalized event and return its value.
- `findValueAndRemove` - Search for a named element in the normalized event and return its value. If found, the element will also be removed from the event.
- `findValueAndRemove2` - Search for a named element in the normalized event and return its value. If found, the element will also be removed from the event. Unlike `findValueAndRemove` it tells you whether it was able to remove the value, or whether it did not exist.
- `first` - Return an iterator pointing to the first element of the normalized event. Successive calls to the `next` function of the returned iterator will allow you to visit all the elements of the event.
- `last` - Return an iterator pointing to the last element of the normalized event. Successive calls to the `back` function of the returned iterator will allow you to visit all the elements of the event.
- `toString` - Return a printable string representation of the normalized event. The returned string is owned by the caller and should be freed when it is no longer required.
- `char8free` - Free a string returned by the `toString` function; should be called when the string is no longer required.

All of these functions take an initial `AP_NormalisedEvent*` argument; this is analogous to the implicit 'this' pointer passed to a C++ object when a member function is invoked on it.

In addition, the `AP_NormalisedEvent_ctor` constructor function is provided to create a new event instance. `AP_NormalisedEvent_dtor` destroys a normalized event object, and should be when the event is no longer required to free up resources.

The reader is invited to examine the header file for the full definitions of all these functions.

Working with normalized events

The `AP_NormalisedEventIterator` structure

Some of the functions above refer to a normalized event 'iterator'. This object is for stepping through the contents of a normalized event, in forwards or reverse order.

The `AP_NormalisedEventIterator` structure contains a function table defined by `AP_NormalisedEventIterator_Functions`, which includes all of the functions exported by a normalized event iterator:

- `valid` - Check whether the iterator points to a valid element of the normalized event. Typically used as part of the loop condition when iterating over the contents of an event.
- `key` - Return the key (name) associated with the current event element pointed to by the iterator. The returned value is owned by the underlying normalized event and should not be modified or freed by the caller.

- `value` - Return the value associated with the current event element pointed to by the iterator. The returned value is owned by the underlying normalized event and should not be modified or freed by the caller.
- `next` - Move the iterator to the next element of the underlying normalized event instance. The iterator must be in a valid state (such that the `valid` function would return `AP_TRUE`) before this function is called. Note that the order in which elements is returned is not necessarily the same as the order in which they were added. The order may change as elements are added to or removed from the underlying event.
- `back` - Move the iterator to the previous element of the underlying normalized event instance. The iterator must be in a valid state (such that the `valid` function would return `AP_TRUE`) before this function is called. Note that the order in which elements is returned is not necessarily the same as the order in which they were added. The order may change as elements are added to or removed from the underlying event.

Note that all of these functions take an initial `AP_NormalisedEventIterator*` argument; this is analogous to the implicit `'this'` pointer passed to a C++ object when a member function is invoked on it.

`AP_NormalisedEventIterator_dtor` destroys a normalized event iterator object, and should be called when the iterator is no longer required to free up resources. There is no public constructor function; iterators are created and returned only by `AP_NormalisedEvent` functions.

See the `NormalisedEvent.h` header file for more information about the structures and functions introduced in this section.

Working with normalized events

Transport Example

As part of the IAF distribution Apama includes the `FileTransport` transport layer plug-in, implemented in the `samples\iaf_plugin\c\simple\FileTransport.c` source file.

The `FileTransport` plug-in can read and write messages from and to a text file, and it is recommended that developers examine this sample to see what a typical transport plug-in implementation looks like.

See ["IAF samples" on page 274](#) for more information about this sample. ["The File Transport plug-in" on page 125](#) describes how the `FileTransport` plug-in can be used in practice.

C/C++ Codec Plug-in Development

Getting Started with codec layer plug-in development

Note: Before developing a new codec plug-in, it is worth considering whether one of the standard Apama IAF plug-ins could be used instead; see ["Using Adapter Plug-ins" on page 123](#) for more information.

In order to facilitate quick development of new codec plug-ins your distribution includes a codec plug-in skeleton.

This file, called `skeleton-codec.c`, implements a complete codec plug-in that complies with the Codec Plug-in Development Specification but where the entire custom message format encoding/decoding functionality is missing. The file is located in the `samples\iaf_plugin\c\skeleton` directory of your installation.

In order to turn the skeleton into a fully operational message format specific codec plug-in, the plug-in author needs to fill in the gaps within the codec generic, decoding and encoding functions; `updateProperties`, `getLastErrorCodec`, `getStatus`, `sendNormalisedEvent`, `flushUpstream`, `getLastErrorEncoder`, `addEventTransport`, `removeEventTransport`, `sendTransportEvent`, `setSemanticMapper`, `flushDownstream`, and `getLastErrorDecoder`. These must implement their specified functionality in the context of the custom message format. The information, constructor and destructor functions are also likely to require adaptation.

The skeleton defines a structure, called `EventCodec_Internals`, to store all its private data, and this structure is placed within the `reserved` field of the `AP_EventCodec` object created within the constructor method. It is likely that this structure will need to be modified to contain additional data that the adapter might require.

The distribution also contains a `makefile` (for use with GNU Make on UNIX), as well as a workspace file and `dsp` folder, for use with Microsoft's Visual Studio .NET on Microsoft Windows, for this skeleton, which can be adapted to compile your codec plug-in and link it against any custom libraries required.

Once a plug-in is built, it needs to be placed in a location where it can be picked up by the IAF.

This means that on Windows you either need to copy the `.dll` into the `bin` folder, or else place it somewhere which is on your 'path', that is a location that is referenced by the `PATH` environment variable.

On UNIX you either need to copy the `.so` into the `lib` directory, or else place it somewhere which is on your 'library path', that is a directory that is referenced by the `LD_LIBRARY_PATH` environment variable.

C/C++ Codec Plug-in Development

■ Logging from plug-ins in C/C++	311
■ Using the latency framework	313

This section describes other programming interfaces provided with the Apama software that may be useful in implementing transport layer and codec plug-ins for the IAF.

Developing Custom Adapters

Logging from plug-ins in C/C++

This API provides a mechanism for recording status and error log messages from the IAF runtime and any plug-ins loaded within it. Plug-in developers are encouraged to make use of the logging API instead of custom logging solutions so that all the information may be logged together in the same standard format and log file(s) used by other plug-ins and the IAF runtime.

The logging API also allows control of logging verbosity, so that any messages below the configured logging level will not be written to the log. The logging level and file are initially set when an adapter first starts up – see ["Logging configuration \(optional\)" on page 273](#) for more information about the logging configuration.

The C/C++ interface to the logging system is declared in the header file `AP_Logger.h`, which can be found in the `include` directory of the Apama installation. All users of the logging system should include this header file. The types and functions of interest to IAF plug-in writers are:

```
AP_LogLevel
/**
 * Enumeration of logging verbosity levels. In order of increasing verbosity,
 * the levels are: NULL < OFF < CRIT < FATAL < ERROR < WARN < INFO < DEBUG < TRACE.
 * Messages logged at level X will be sent to the log if the current
 * level is >= X.
 */
enum AP_LogLevel {
    AP_LogLevel_NULL,
    AP_LogLevel_OFF,
    AP_LogLevel_CRIT,
    AP_LogLevel_FATAL,
    AP_LogLevel_ERROR,
    AP_LogLevel_WARN,
    AP_LogLevel_INFO,
    AP_LogLevel_DEBUG,
    AP_LogLevel_TRACE
};
```

Note: `AP_LogLevel_NULL` means “no log level has been set” and should be interpreted by IAF and plug-ins as “use the default logging level.”

```
AP_LogTrace
/**
 * Log a message at TRACE level. Note that the message will only appear
 * in the log file if the current logging level is AP_LogLevel_Trace.
 *
 * @param message The message to be logged. This is a standard printf()
```

```

    * format string; any values required by the formatting should be passed
    * as additional arguments to the AP_LogTrace() call.
    */
AP_COMMON_API void AP_COMMON_CALL AP_LogTrace(const char* message, ...);

```

Along with the other logging functions described below, `AP_LogTrace` is based on the standard C library `printf` function. The message parameter may contain `printf` formatting characters that will be filled in from the remaining arguments.

AP_LogDebug

```

/**
 * Log a message at DEBUG level. Note that the message will only appear
 * in the log file if the current logging level is AP_LogLevel_DEBUG or
 * greater.
 *
 * @param message The message to be logged. This is a standard printf()
 * format string; any values required by the formatting should be passed
 * as additional arguments to the AP_LogDebug() call.
 */
AP_COMMON_API void AP_COMMON_CALL AP_LogDebug(const char* message, ...);

```

AP_LogInfo

```

/**
 * Log a message at INFO level. Note that the message will only appear
 * in the log file if the current logging level is AP_LogLevel_INFO or
 * greater.
 *
 * @param message The message to be logged. This is a standard printf()
 * format string; any values required by the formatting should be passed
 * as additional arguments to the AP_LogInfo() call.
 */
AP_COMMON_API void AP_COMMON_CALL AP_LogInfo(const char* message, ...);

```

AP_LogWarn

```

/**
 * Log a message at WARN level. Note that the message will only appear
 * in the log file if the current logging level is AP_LogLevel_WARN or
 * greater.
 *
 * @param message The message to be logged. This is a standard printf()
 * format string; any values required by the formatting should be passed
 * as additional arguments to the AP_LogWarn() call.
 */
AP_COMMON_API void AP_COMMON_CALL AP_LogWarn(const char* message, ...);

```

AP_LogError

```

/**
 * Log a message at ERROR level. Note that the message will only appear
 * in the log file if the current logging level is AP_LogLevel_ERROR or
 * greater.
 *
 * @param message The message to be logged. This is a standard printf()
 * format string; any values required by the formatting should be passed
 * as additional arguments to the AP_LogError() call.
 */
AP_COMMON_API void AP_COMMON_CALL AP_LogError(const char* message, ...);

```

AP_LogCrit

```

/**
 * Log a message at CRIT level. Note that the message will only appear
 * in the log file if the current logging level is AP_LogLevel_CRIT or
 * greater.
 *
 * @param message The message to be logged. This is a standard printf()
 * format string; any values required by the formatting should be passed
 * as additional arguments to the AP_LogCrit() call.

```

```
*/
AP_COMMON_API void AP_COMMON_CALL AP_LogCrit(const char* message, ...);
```

The logging API offers other functions to set and query the current logging level and output file. While these functions are available to plug-in code, it is recommended that plug-ins do not use them. The IAF core is responsible for updating the state of the logging system in response to adapter re-configuration requests.

[C/C++ Plug-in Support APIs](#)

Using the latency framework

The latency framework API provides a way to measure adapter latency by attaching high-resolution timing data to events as they stream into, through, and out of the adapter. Developers can then use these events to compute upstream, downstream, and round-trip latency numbers, including latency across multiple adapters.

The `sendNormalisedEvent()` and `sendTransportEvent()` functions contain an `AP_TimestampSet` parameter that carries the microsecond-accurate timestamps that can be used to compute the desired statistics.

[C/C++ Plug-in Support APIs](#)

C/C++ timestamp

A timestamp is an index-value pair. The index represents the point in the event processing chain at which the timestamp was recorded, for example “upstream entry to semantic mapper” and the value is a floating point number representing the time. The header file `AP_TimestampSet.h` defines a set of standard indexes but a custom plug-in can define additional indexes for even finer-grained measurements. When you add a custom index definition, be sure to preserve the correct order, for example, an index denoting an “entry” point should be less than an one denoting an “exit” point from that component.

Timestamps are relative measurements and are meant to be compared only to other timestamps in the same or similar processes on the same computer. Timestamps have no relationship to real-world “wall time”.

[Using the latency framework](#)

C/C++ timestamp set

A timestamp set is the collection of timestamps that are associated with an event. The latency framework API provides functions that developers can use to add, inspect, and remove timestamps from an event’s timestamp set.

[Using the latency framework](#)

C/C++ timestamp configuration object

Constructors and `updateProperties()` methods for transport and codec plug-ins take the following argument: `IAF_TimestampConfig`.

A timestamp configuration object contains a set of fields that a plug-in can use to decide whether to record and/or log timestamp information. Although timestamp configuration objects are passed to

all transport and codec plug-ins, it is up to the authors of a plug-in to write the code that makes use of them.

The fields in the object are:

- `recordUpstream` — If true, the plug-in should record timestamps for all upstream events it processes, and pass these along to the upstream component, if any.
- `recordDownstream` — If true, the plug-in should record timestamps for all downstream events it processes, and pass these along to the downstream component, if any.
- `logUpstream` — If true, the plug-in should log the latency for all upstream events it processes, at the logging level given by the `logLevel` member. A plug-in may implicitly enable upstream timestamp recording if upstream logging is enabled.
- `logDownstream` — If true, the plug-in should log the latency for all downstream events it processes, at the logging level given by the `logLevel` member. A plug-in may implicitly enable downstream timestamp recording if downstream logging is enabled.
- `logRoundtrip` — If true, the plug-in should log the “round trip” latency for all events it processes in either direction, if possible. At its simplest, the round trip latency can just be the difference between the largest and smallest timestamps passed to the plug-in, or an individual plug-in may choose to present some more plug-in-specific latency number. As with the other logging options, the logging level given by the `logLevel` member should be used.
- `logLevel` — The logging verbosity level to use if any of the timestamp logging options are enabled.

Using the latency framework

C/C++ latency framework API

The C/C++ interface for the latency framework is declared in the header file `AP_TimestampSet.h`. Plug-ins using the latency framework should include this file and also include the `IAF_TimestampConfig.h` header file, which declares the timestamp configuration object.

The functions of interest are the following.

addNow

```
/**
 * Add a new index-time pair to the timestamp. The given index and current
 * time will be copied into the timestamp. If an element with the same index
 * already exists, it will NOT be overwritten.
 *
 * @param timestamp The AP_TimestampSet instance
 * @param index The index of the new element
 */
void (*addNow)(struct AP_TimestampSet* timestamp,
               AP_TimestampSetIndex index);
```

addTime

```
/**
 * Add a new index-time pair to the timestamp. The given index and time will
 * be copied into the timestamp. If an element with the same index already
 * exists, it will NOT be overwritten.
 *
 * @param timestamp The AP_TimestampSet instance
 * @param index The index of the new element
 * @param time The time of the new element
 */
void (*addTime)(struct AP_TimestampSet* timestamp,
```

```
AP_TimestampSetIndex index, AP_TimestampSetTime time);
```

replace

```
/**
 * Change the time of an indexed element in the timestamp. If an
 * element with the given index already exists, its time will be replaced
 * with a copy of the given time. Otherwise, a new element is created
 * just as though addTime() had been called.
 *
 * @param timestamp The AP_TimestampSet instance
 * @param index The index of the element to be updated
 * @param time The new time for the indexed element
 */
void (*replace)(struct AP_TimestampSet* timestamp,
                AP_TimestampSetIndex index, AP_TimestampSetTime newTime);
```

replaceWithNow

```
/**
 * Change the time of an indexed element in the timestamp. If an
 * element with the given index already exists, its time will be replaced
 * with a copy of the given time. Otherwise, a new element is created
 * just as though addNow() had been called.
 *
 * @param timestamp The AP_TimestampSet instance
 * @param index The index of the element to be updated
 * @param time The new time for the indexed element
 */
void (*replaceWithNow)(struct AP_TimestampSet* timestamp,
                      AP_TimestampSetIndex index);
```

findTime

```
/**
 * Search for an indexed element in the timestamp and return its
 * time.
 *
 * @param timestamp The AP_TimestampSet instance
 * @param index The element index to search for
 * @return The time associated with the given index, or NULL if no
 * matching element could be found.
 */
AP_TimestampSetTime (*findTime)(struct AP_TimestampSet* timestamp,
                               AP_TimestampSetIndex index);
```

findTimeAndRemove

```
/**
 * Search for an indexed element in the timestamp and return its
 * time. If found, the element will also be removed from the timestamp.
 *
 * @param timestamp The AP_TimestampSet instance
 * @param index The element index to search for
 * @return The time associated with the given index, or NULL if no
 * matching element could be found. If the element was found, it will be
 * deleted from the timestamp. Note that this function cannot distinguish
 * between an element with a missing time and an element that does not
 * exist - NULL will be returned in either case. The returned object
 * (if any) must be explicitly deleted by the caller after use.
 */
AP_TimestampSetTime (*findTimeAndRemove)(
    struct AP_TimestampSet* timestamp, AP_TimestampSetIndex index);
```

getSize

```
/**
 * Return the number of elements in the AP_TimestampSet
 *
 * @param timestamp The AP_TimestampSet instance
```

```
* @return The number of elements in the timestamp set
*/
AP_uint32 (*getSize)(struct AP_TimestampSet* timestamp);

toString
W
/**
 * Return a printable string representation of the timestamp. The
 * returned string is owned by the caller and should be freed when it is
 * no longer required using the char8free function.
 *
 * @param timestamp The AP_TimestampSet instance
 * @return Printable string representation of the timestamp
 */
AP_char8* (*toString)(struct AP_TimestampSet* timestamp);
```

Using the latency framework

Chapter 12: Transport Plug-in Development in Java

■ The Transport Plug-in Development Specification for Java	317
■ Example	324
■ Getting started with Java transport layer plug-in development	324

The *transport layer* is the front-end of the IAF. The transport layer's purpose is to abstract away the differences between the programming interfaces exposed by different middleware message sources and sinks. It consists of one or more custom plug-in libraries that extract *downstream* messages from external message sources ready for delivery to the codec layer, and send Apama events already encoded by the codec layer *upstream* to the external message sink. See "[The Integration Adapter Framework](#)" on page 241 for a full introduction to transport plug-ins and the IAF's architecture.

An adapter should send events to the correlator only after its `start` function is called and before the `stop` function returns.

This topic includes the Transport Plug-in Development Specification for Java and additional information for developers of Java event transports. "[C/C++ Transport Plug-in Development](#)" on page 277 provides analogous information about developing transport plug-ins using C/C++.

Developing Custom Adapters

The Transport Plug-in Development Specification for Java

A Java transport layer plug-in is implemented as a Java class extending `AbstractEventTransport`. Typically this class would be packaged up, together with any supporting classes, as a Java Archive (`.jar`) file.

To comply with Apama's Transport Plug-in Development Specification, an event transport class must satisfy two conditions:

1. It must have a constructor with the signature:

```
public AbstractEventTransport(  
    String name,  
    EventTransportProperty[] properties,  
    TimestampConfig timestampConfig)  
    throws TransportException
```

This will be used by the IAF to instantiate the plug-in.

2. It must extend the `com.apama.iaf.plugin.AbstractEventTransport` class, correctly implementing all of its abstract methods.

(These methods are mostly directly equivalent to the functions with the same names in the C/C++ Transport Plug-in Development Specification.)

Note that all Java plug-ins are dependent on classes in `jplugin_public5.3.jar`, so this file must always be on the classpath during plug-in development. It is located in the Apama installation's `lib` directory.

Unless otherwise stated, Java classes referred to in this topic are members of the `com.apama.iaf.plugin` package, whose classes and interfaces are contained in this `.jar`.

Transport Plug-in Development in Java

Java transport functions to implement

HTML Javadoc documentation for `AbstractEventTransport` and related classes is provided as part of the Apama documentation set. The Javadoc is located in the Apama installation's `doc\javadoc` directory.

This topic includes the text of the Javadoc documentation for the functions a transport plug-in author needs to implement, in addition to some pointers on the idiomatic way in which such plug-ins are usually written.

The Constructor

```
/**
 * Construct a new instance of AbstractEventTransport. All subclasses MUST
 * provide a constructor with the same signature, which will be used by the
 * IAF to create an instance of the transport class.
 *
 * The AbstractEventTransport implementation does nothing, but subclasses
 * should make use of the arguments to initialize the transport.
 *
 * @param name The transport name, as specified in the IAF config file
 * @param properties The transport property set specified in the IAF
 * configuration file
 * @param timestampConfig Timestamp recording/logging settings from the IAF
 * configuration file
 * @throws TransportException
 */
public AbstractEventTransport(String name,
                               EventTransportProperty[] properties
                               TimestampConfig timestampConfig)
    throws TransportException
```

A typical constructor would create a logger using the plug-in `name` provided (see ["Logging from plug-ins in Java" on page 339](#)), make a call to the `updateProperties` method to deal with the initial property set passed in, and perform any other initialization operations required for the particular transport being developed.

updateProperties

```
/**
 * Update the configuration of the transport. The transport may assume
 * that stop(), flushUpstream() and flushDownstream() have all been called
 * before this function is invoked. The recommended procedure for
 * updating properties is to first compare the new property set with the
 * existing stored properties - if there are no changes, no action should
 * be taken.
 *
 * @param properties The new transport property set specified in the IAF
 * configuration file
 * @param TimestampConfig timestampConfig
 * @throws TransportException
 */
abstract public void updateProperties(EventTransportProperty[] properties,
                                     TimestampConfig timestampConfig)
    throws TransportException;
```

The `properties` array contains an `EventTransportProperty` object for each plug-in property specified in the IAF configuration file (in order). The `getName` and `getValue` methods allow the plug-in to retrieve the name and value of each property as `String` objects.

See the Javadoc documentation for more information about the `EventTransportProperty` class.

addEventDecoder

```
/**
 * Add a named event decoder to the set of decoders known to the
 * transport. If the named decoder already exists, it should be
 * replaced.
 *
 * @param name The name of the decoder to be added
 * @param decoder The decoder object instance
 * @throws TransportException
 */
abstract public void addEventDecoder(String name, EventDecoder decoder)
    throws TransportException;
```

In an adapter in which multiple event codecs could be present, this function would usually be implemented by storing the `<name, decoder>` pair in a Java map, from which the `EventDecoder` could later be retrieved using a plug-in property (e.g. `"decoderName"`) to determine which of the plug-ins in the map should be used.

Alternatively, if this transport plug-in will only ever be used in an adapter with just one codec plug-in, this method can be implemented simply by storing the provided `EventDecoder` object in an instance field.

See ["Communication with the codec layer" on page 321](#) for more information.

removeEventDecoder

```
/**
 * Remove a named event decoder from the set of decoders known to the
 * transport. If the named decoder does not exist, the function should do
 * nothing.
 *
 * @param name The decoder to be removed
 * @throws TransportException
 */
abstract public void removeEventDecoder(String name)
    throws TransportException;
```

This method is usually implemented by removing the named codec plug-in from a map, or nulling out a field holding the previously added `EventCodec`.

flushUpstream

```
/**
 * Flush any pending transport events onto the transport. The transport may
 * assume that the stop() function has been called before this function,
 * so in many cases no action will be required to complete the flushing
 * operation.
 *
 * @throws TransportException
 */
abstract public void flushUpstream() throws TransportException;
```

Usually has a blank implementation, unless there is some kind of upstream buffering.

flushDownstream

```
/**
 * Flush any pending transport events into the decoder. The transport may
 * assume that the stop() function has been called before this function,
 * so in many cases no action will be required to complete the flushing
 * operation.
 *
 * @throws TransportException
 */
abstract public void flushDownstream() throws TransportException;
```

Usually has a blank implementation, unless there is some kind of downstream buffering.

start

```
/**
 * Start processing incoming data from the external transport.
 *
 * @throws TransportException
 */
abstract public void start() throws TransportException;
```

This is where a plug-in should establish a connection to its external message source/sink, often by starting a new thread to process or poll for new downstream messages. Events should not be sent to the correlator until the `start` method has been called.

"[Communication with the codec layer](#)" on page 321 explains how a transport plug-in can pass downstream messages it receives from an external source on to a codec plug-in.

stop

```
/**
 * Stop processing incoming data from the external transport. New events
 * arriving may be blocked, queued or simply dropped, but under no
 * circumstances should any be sent into the adapter until the start()
 * function is called.
 *
 * @throws TransportException
 */
abstract public void stop() throws TransportException;
```

Here, a plug-in may close existing connections and interrupt running threads to ensure that no more messages are passed to the event codec. Events should not be sent to the correlator after the `stop` method has returned. The `stop` method must wait for any other threads sending events to complete before the `stop` method returns.

```
cleanup/**
 * Frees any resources allocated by the transport (useful for resources
 * external to the JVM that were allocated in the constructor). The IAF
 * guarantees to call this method exactly once.
 *
 * @throws TransportException
 */
abstract public void cleanup() throws TransportException;
```

This is where any heavy-weight threads or data structures used for interfacing with the external transport that do not get cleaned up when the plug-in is stopped should be destroyed.

getStatus

```
/**
 * Return a TransportStatus class containing up-to-date status information
 * for the transport.
 *
 * @return An immutable TransportStatus class containing status
 * information.
 */
abstract public TransportStatus getStatus();
```

This method provides the statistics and status message displayed by the IAF Watch tool. A typical plug-in will continuously keep track of the number of messages sent upstream and downstream. Then, when `getStatus` is called, these message counts can simply be packaged up in a new `TransportStatus` object together with a `String` describing the current status of the plug-in (maximum length 1024 characters), and returned.

For example:

```
public TransportStatus getStatus()
```

```
{
    String status = (started) ? "Status: Running" : "Status: Not running";
    return new TransportStatus(status, totalReceived, totalSent);
}
```

See the Javadoc documentation for more information about the `TransportStatus` class.

getAPIVersion

```
/**
 * Return the transport API version that the transport was built against.
 * @return Must be EventTransport.API_VERSION.
 */
public abstract int getAPIVersion();
```

Always return `EventTransport.API_VERSION`.

sendTransportEvent (send upstream)

```
/**
 * Called by an event encoder to send an upstream message to the external
 * transport.
 *
 * It is assumed that the encoder and transport share
 * the same definition of the content of the event, so that the transport
 * can effectively interpret the event.
 *
 * @param event An object representing the event to be sent by the
 * transport, in a format shared by the encoder and transport.
 * @param timestamps A TimestampSet representing the timestamps attached to
 *
 * @throws TransportException Thrown by the transport if any error occurs
 * sending the message.
 */
public abstract void sendTransportEvent(Object event,
    TimestampSet timestamps)
    throws TransportException;
```

This is the method that a codec layer plug-in calls when it receives a translated upstream Apama event that needs to be sent on to an event transport for transmission to an external message sink.

Note that there are no guarantees about which threads might be used to call this method, so plug-in authors will need to consider any thread synchronization issues arising from use of shared data structures here.

See ["Communication with the codec layer" on page 321](#) for more information about processing upstream events received from a codec plug-in.

[The Transport Plug-in Development Specification for Java](#)

Communication with the codec layer

This section discusses how the transport layer communicates with the codec layer in both the upstream and downstream directions.

[The Transport Plug-in Development Specification for Java](#)

Sending upstream messages received from a codec plug-in to a sink

When a codec plug-in has encoded an event ready for transmission by a transport plug-in it will pass it on calling the transport's `sendTransportEvent` method (as defined above). It is then up to the transport plug-in to process the message (which will be of some type agreed by the codec and transport plug-in authors), and send it on to the external sink it provides access to.

Note that there are no guarantees about which threads might call this method, so plug-in authors will need to consider thread synchronization issues carefully.

If there is a problem sending the event on, the transport plug-in should throw a `TransportException`.

Communication with the codec layer

Sending downstream messages received from a source on to a codec plug-in

In order that messages can be easily sent on to a codec plug-in, an event transport will usually have saved a reference to the event codec(s) it will be using before it establishes a connection to the external source.

Typically an event transport will build up a list of registered codec plug-ins from the parameters passed to the `addEventDecoder` and `removeEventDecoder` methods. If this is the case, the `start` method of the plug-in can select one of these plug-ins on the basis of a plug-in property provided in the configuration file (e.g. `<property name="decoderName" value="MyCodec"/>`), and saving it in an instance field (e.g. `currentDecoder`).

Once the plug-in has a reference to the event codec (or codecs) it will use, whenever an external message is received it should be passed on by calling the `sendTransportEvent` method on the codec plug-in (from the `EventDecoder` interface):

```
/**
 * Called by the event transport to decode a downstream event using a Java
 * Codec, which will then send it on to the Semantic Mapper.
 *
 * It is assumed that the encoder and transport share the same definition
 * of the content of the event, so that the transport can effectively
 * interpret the event.
 *
 * @param event An object representing the event to be decoded, in a format
 * shared by the decoder and transport.
 * @param timestamps A TimestampSet representing the timestamps attached to
 * the event.
 *
 * @throws CodecException Thrown by the decoder if the event provided
 * has an invalid format.
 * @throws SemanticMapperException Thrown if an error occurred during
 * processing of the message by the Semantic Mapper.
 */
public void sendTransportEvent(Object event,
    TimestampSet timestamps)
    throws CodecException, SemanticMapperException;
```

For example, part of the event processing code for a transport plug-in might be:

```
MyCustomMessageType message = myCustomMessageSource.getNextMessage();
currentDecoder.sendTransportEvent(message, timestamps);
```

If an error occurs in the codec or Semantic Mapper layers preventing the message from being converted into an Apama event, a `CodecException` or `SemanticMapperException` is thrown. Like all per-message errors, these should be logged at `Warning` level, preferably with a full stack trace logged at `Debug` level too. If necessary, transports may also send messages downstream to the correlator to inform running monitors about the error.

When a transport sends a message to the codec via the `sendTransportEvent` method, it passes an `Object` reference and this allows custom types to be passed between the two plug-ins. However, any custom types should be loaded via the main (parent) classloader, as each plug-in specified in the IAF configuration file is loaded with its own classloader. Consider, for example, the following three classes all loaded into a single jar file, `MyAdapter.jar`, which is used in the IAF configuration file in the `jarName` attribute of the `<transport>` element.:

- `MyTransport.class`

- `MyCodec.class`
- `MyContainer.class` (the container class used in the call to `sendTransportEvent`)

When you load the transport and codec, a new classloader is used for each. This means both have their own copy of the `MyContainer` class. When the transport creates an instance of `MyContainer` and then passes it into the codec, the codec will recognize that the Object `getClass().getName()` is `MyContainer`, but will not be able to cast it to this type as its `MyContainer` class is from a different classloader.

To prevent this from happening, make sure that all shared classes are in a separate jar that is specified by a `<classpath>` element. The shared classes are then loaded by the parent classloader. This ensures that when a codec or transport references a shared class, they will both agree it is the same class.

Note that any codec plug-in called by a Java transport plug-in must also be written in Java.

Communication with the codec layer

Transport exceptions

`TransportException` is the exception class that should be thrown by a transport plug-in whenever the IAF calls one of its methods and an error prevents the method from successfully completing — for example, a message that cannot be sent on to an external sink in `sendTransportEvent`, or a serious problem that prevents the plug-in from initializing when `start` is called.

A `TransportException` object always has an associated `message`, which is a `String` explaining the problem (this may include information about another exception that caused the `TransportException` to be thrown). There is also a `code` field that specifies the kind of error that occurred; the possible codes are defined as constants in the `TransportException` class:

```
/**
 * Some unspecified internal error occurred
 */
public static final int INTERNALERROR = 1;
/**
 * Trouble reading/writing the external transport
 */
public static final int TRANSPORTFAILURE = 2;
/**
 * Trouble sending transport event to decoder
 */
public static final int DECODINGFAILURE = 3;
/**
 * Transport was passed an invalid property set
 */
public static final int BADPROPERTIES = 4;
```

`TransportException` defines a number of constructors, to make it easy to set up the exception's information quickly in different situations:

```
/**
 * Constructs a TransportException from a string message describing the
 * error, and assumes an error code of TRANSPORTFAILURE.
 *
 * @param message The cause of the error.
 */
public TransportException(String message) { ... }
/**
 * Constructs a TransportException from a string message describing the
 * error and a numeric code representing the class of error.
 *
 * @param message The cause of the error.
```

```

    * @param code One of the TransportException error codes.
    */
    public TransportException(String message, int code) { ... }
    /**
     * Constructs a TransportException from a string message describing the
     * error, and an exception object that is the cause of the error. It
     * assumes an error code of TRANSPORTFAILURE.
     *
     * @param message The cause of the error. This message will be suffixed
     * with the message of the 'cause' exception.
     * @param cause The exception object that caused the error.
     */
    public TransportException(String message, Throwable cause) { ... }
    /**
     * Constructs a TransportException from a string message describing the
     * error, a numeric code representing the class of error and an exception
     * object that is the cause of the error.
     *
     * @param message The cause of the error. This message will be suffixed
     * with the message of the 'cause' exception.
     * @param cause The exception object that caused the error.
     * @param code One of the TransportException error codes.
     */
    public TransportException(String message, Throwable cause, int code) { ... }

```

The Transport Plug-in Development Specification for Java

Logging

See ["Logging from plug-ins in Java" on page 339](#) for information about how transport plug-ins should log error, status and debug information.

The Transport Plug-in Development Specification for Java

Example

As part of the IAF distribution, Apama includes the `JFileTransport` transport layer plug-in, in the `samples\iaf_plugin\java\simple\src` directory.

The `JFileTransport` plug-in can read and write messages from and to a text file, and it is recommended that developers examine this sample to see what a typical transport plug-in implementation looks like.

See ["IAF samples" on page 274](#) for more information about this sample. The section ["The File Transport plug-in" on page 125](#) describes how the `JFileTransport` plug-in can be used in practice.

Transport Plug-in Development in Java

Getting started with Java transport layer plug-in development

Your distribution includes a complete 'skeleton' implementation of a transport layer plug-in in order to make development of new plug-ins faster.

This is located in the `samples\iaf_plugin\java\skeleton\src` directory of the installation, in a file called `SkeletonTransport.java`. The `SkeletonTransport` class complies fully with the Transport Plug-in

Development Specification, but contains none of the custom message source/sink functionality that would be present in a full transport plug-in.

The skeleton starts a background thread to do the actual message reading. This is required unless the message source can asynchronously call back into the class that implements the plug-in.

The code contains `TODO`: comments indicating the main changes that need to be made to add support for a specific message source/sink. These include:

- Adding code to `sendTransportEvent` for sending an upstream event received from an event codec on to the external message sink (if supported).
- Adding code to the `run` method of the `MessageProcessingThread` for retrieving downstream messages from the external source and forwarding them on to an event codec (if supported).
- Alternatively, if the external message source works by making asynchronous calls using the listener pattern, the processing thread should usually be removed, and much of the code can be moved directly to the method called by the message source.
- Adding code to start communications with the external messaging system in the `start` method, and to ensure it ceases in the `stop` method.
- Adding code to validate and save any new plug-in properties that are to be supported, in `updateProperties`.
- Adding code to initialize and clean up resources associated with the plug-in's operation. This would usually be done in the `start/stop` methods, in the background processing thread, or in the `updateProperties` and `cleanup` methods.

Depending on your requirements, it may also be necessary to make changes to the other methods – `addEventDecoder`, `removeEventDecoder`, `flushUpstream`, `flushDownstream`, `getStatus`, and the constructor.

The `skeleton` directory includes an Apache Ant build file called `build.xml` that provides a convenient way to build `.jar` files of compiled classes from plug-in source files, ready for use with the IAF.

Chapter 13: Java Codec Plug-in Development

■ The Codec Plug-in Development Specification for Java	326
■ Java Codec Example	337
■ Getting started with Java codec layer plug-in development	337

The *codec layer* is a layer of abstraction between the transport layer and the IAF's Semantic Mapper. It consists of one or more plug-in libraries that perform message *encodinganddecoding*. Decoding involves translating *downstream* messages retrieved by the transport layer into the standard 'normalised event' format on which the Semantic Mapper's rules run; encoding works in the opposite direction, converting *upstream* normalized events into an appropriate format for transport layer plug-ins to send on. Note that unlike the situation with C/C++, in Java codec plug-ins are always both encoders and decoders. See "[The Integration Adapter Framework](#)" on page 241 for a full introduction to codec plug-ins and the IAF's architecture.

This chapter includes the Codec Plug-in Development Specification for Java and additional information for developers of Java event codecs. "[C/C++ Codec Plug-in Development](#)" on page 291 provides analogous information about developing codec plug-ins using C/C++.

Before developing a new codec plug-in, it is worth considering whether one of the standard Apama plug-ins could be used instead. "[Using Adapter Plug-ins](#)" on page 123 provides more information on the standard IAF codec plug-ins: `JStringCodec` and `JNullCodec`. The `JStringCodec` plug-in codes normalized events as formatted text strings. The `JNullCodec` plug-in is useful in situations where it does not make sense to decouple the codec and transport layers, and allows transport plug-ins to communicate with the Semantic Mapper directly using normalized events.

Developing Custom Adapters

The Codec Plug-in Development Specification for Java

A Java codec layer plug-in is implemented as a Java class extending `AbstractEventCodec`. Typically this class would be packaged up, together with any supporting classes, as a Java Archive (`.jar`) file.

To comply with Apama's Codec Plug-in Development Specification, an event codec class must satisfy two conditions:

1. It must have a constructor with the signature:

```
public AbstractEventCodec(  
    String name,  
    EventCodecProperty[] properties,  
    TimestampConfig timestampConfig)  
    throws CodecException
```

This will be used by the IAF to instantiate the plug-in.

2. It must extend the `com.apama.iaf.plugin.AbstractEventCodec` class, correctly implementing all of its abstract methods.

(These methods are mostly directly equivalent to the functions with the same names in the C/C++ Codec Plug-in Development Specification.)

Note that all Java plug-ins are dependent on classes in `jplugin_public5.3.jar`, so this file must always be on the classpath during plug-in development. It is located in the Apama installation's `lib` directory.

Unless otherwise stated, Java classes referred to in this chapter are members of the `com.apama.iaf.plugin` package, whose classes and interfaces are contained in this `.jar`.

Java Codec Plug-in Development

Java codec functions to implement

HTML Javadoc documentation for `AbstractEventCodec` and related classes is provided as part of the Apama documentation set. The Javadoc files are located in the Apama installation's `doc\javadoc` directory.

This topic includes the text of the Javadoc documentation for the functions that a codec plug-in author needs to implement, in addition to some pointers on how such plug-ins are usually written.

The Constructor

```
/**
 * Construct a new instance of AbstractEventCodec. All subclasses MUST
 * provide a constructor with the same signature, which will be used by the
 * IAF to create an instance of the codec class. <P><P>
 *
 * The AbstractEventCodec implementation does nothing, but subclasses
 * should make use of the arguments to initialize the codec.
 *
 * @param name The codec name, as specified in the IAF config file
 * @param properties The codec property set specified in the IAF
 * configuration file
 * @param timestampConfig The timestamp recording/logging settings from the
 * IAF configuration file
 *
 * @throws CodecException
 */
public AbstractEventCodec(String name, EventCodecProperty[] properties,
    TimestampConfig timestampConfig)
    throws CodecException
```

A typical constructor would create a logger using the plug-in `name` provided (see ["Logging from plug-ins in Java" on page 339](#)), make a call to the `updateProperties` method to deal with the initial property set passed in, and perform any other initialization operations required for the particular event codec being developed.

Note that unlike event transports, codec plug-ins do not have start and stop methods

updateProperties

```
/**
 * Update the configuration of the codec. The codec may assume
 * that flushUpstream() and flushDownstream() have been called
 * before this function is invoked. The recommended procedure for
 * updating properties is to first compare the new property set with the
 * existing stored properties - if there are no changes, no action should
 * be taken.
 *
 * @param properties The new codec property set specified in the IAF
 * configuration file
 * @param TimestampConfig timestampConfig
 * @throws CodecException
 */
```

```

abstract public void updateProperties(EventCodecProperty[] properties,
    TimestampConfig timestampConfig)
    throws CodecException;

```

The `properties` array contains an `EventCodecProperty` object for each plug-in property specified in the IAF configuration file (in order). The `getName` and `getValue` methods allow the plug-in to retrieve the name and value of each property as `String` objects.

See the Javadoc documentation for more information about the `EventCodecProperty` class.

addEventTransport

```

/**
 * Add a named event transport to the set of transports known to the
 * codec. If the named transport already exists, it should be
 * replaced.
 *
 * @param name The name of the transport to be added
 * @param transport The transport object instance
 * @throws CodecException
 */
abstract public void addEventTransport(String name, EventTransport transport)
    throws CodecException;

```

In an adapter in which multiple event transports could be present, this function would usually be implemented by storing the `<name, transport>` pair in a Java map, from which the `EventTransport` object could later be retrieved when required by the `sendNormalisedEvent` method, using a plug-in property (e.g. `"transportName"` to determine which of the plug-ins in the map should be used.

Alternatively, if this codec plug-in will only ever be used in an adapter with just one transport plug-in, this method can be implemented simply by storing the provided `EventTransport` object in an instance field.

See ["Communication with other layers" on page 331](#) for more information.

removeEventTransport

```

/**
 * Remove a named event transport from the set of transports known to the
 * codec. If the named transport does not exist, the function should do
 * nothing.
 *
 * @param name The transport to be removed
 * @throws CodecException
 */
abstract public void removeEventTransport(String name)
    throws CodecException;

```

This method is usually implemented by removing the named transport plug-in from a map, or nulling out a field holding the previously added `EventTransport`.

setSemanticMapper

```

/**
 * Set the Semantic Mapper object to be used by the decoder. Currently
 * only a single Semantic Mapper is supported in each adapter instance.
 *
 * @param mapper The Semantic mapper object instance
 * @throws CodecException
 */
abstract public void setSemanticMapper(SemanticMapper mapper)
    throws CodecException;

```

This method is usually implemented by storing the provided `SemanticMapper` object in an instance field, for use when sending on downstream messages.

flushUpstream

```
/**
 * Flush any pending codec events onto the codec. In many cases no action
 * will be required to complete the flushing operation.
 *
 * @throws CodecException
 */
abstract public void flushUpstream() throws CodecException;
```

Usually has a blank implementation, unless there is some kind of upstream buffering.

flushDownstream

```
/**
 * Flush any pending codec events into the decoder. In many cases no action
 * will be required to complete the flushing operation.
 *
 * @throws CodecException
 */
abstract public void flushDownstream() throws CodecException;
```

Usually has a blank implementation, unless there is some kind of downstream buffering.

cleanup

```
/**
 * Frees any resources allocated by the codec (useful for resources
 * external to the JVM that were allocated in the constructor). The IAF
 * guarantees to call this method exactly once.
 *
 * @throws CodecException
 */
abstract public void cleanup() throws CodecException;
```

This is where any external resources used by the event codec should be freed.

getStatus

```
/**
 * Return a CodecStatus class containing up-to-date status information
 * for the codec.
 *
 * @return An immutable CodecStatus class containing status
 * information.
 */
abstract public CodecStatus getStatus();
```

This method provides the statistics and status message displayed by the IAF Watch tool. A typical plug-in will continuously keep track of the number of messages sent upstream and downstream. Then, when `getStatus` is called, these message counts can simply be packaged up in a new `CodecStatus` object together with a `String` describing the current status of the plug-in (maximum length 1024 characters), and returned.

For example:

```
public CodecStatus getStatus()
{
    String status = "Status: OK";
    return new TransportStatus(status, totalReceived, totalSent);
}
```

See the Javadoc documentation for more information about the `CodecStatus` class.

getAPIVersion

```
/**
 * Return the codec API version that the codec was built against.
 * @return Must be EventCodec.API_VERSION.
 */
public abstract int getAPIVersion();
```

Always return `EventCodec.API_VERSION`.

sendTransportEvent

```
/**
 * Called by the event transport to decode a downstream event using a Java
 * Codec, which will then send it on to the Semantic Mapper. It is assumed
 * that the encoder and transport share the same definition
 * of the content of the event, so that the transport can effectively
 * interpret the event.
 *
 * @param event An object representing the event to be decoded, in a format
 * shared by the decoder and transport.
 * @param timestamps A TimestampSet representing the timestamps attached to
 * the event.
 *
 * @throws CodecException Thrown by the decoder if the event provided
 * has an invalid format.
 * @throws SemanticMapperException Thrown if an error occurred during
 * processing of the message by the Semantic Mapper.
 */
public void sendTransportEvent(Object event, TimestampSet timestamps)
    throws CodecException, SemanticMapperException;
```

This is the method that a transport layer plug-in calls when it receives a message that should be decoded and then sent downstream towards the Apama event correlator.

Note that there are no guarantees about which threads might be used to call this method, so plug-in authors will need to consider any thread synchronization issues arising from use of shared data structures here.

See ["Communication with other layers" on page 331](#) for more information about processing downstream messages and passing them on to the Semantic Mapper.

sendNormalisedEvent (send **upstream**)

```
/**
 * Called by the Semantic Mapper to encode a normalized event and send
 * it directly through to the transport.
 *
 * @param event A NormalisedEvent representing the event to be encoded.
 * @param timestamps A TimestampSet representing the timestamps attached to
 * the event.
 *
 * @throws CodecException Thrown by the codec if the event provided
 * has an invalid format.
 * @throws TransportException Thrown if an error occurred in the Transport
 * when sending the message.
 */
public void sendNormalisedEvent(NormalisedEvent event,
    TimestampSet timestamps)
    throws CodecException, TransportException;
```

This is the method that the Semantic Mapper calls when it receives a message that should be encoded and then sent upstream to an event transport.

Note that there are no guarantees about which threads might be used to call this method, so plug-in authors will need to consider any thread synchronization issues arising from use of shared data structures here.

See ["Communication with other layers" on page 331](#) for more information about processing upstream messages and passing them on to a transport plug-in. See ["Working with normalized events" on page 334](#) for help working with `NormalisedEvent` objects.

The Codec Plug-in Development Specification for Java

Communication with other layers

This section discusses how the codec layer communicates with the transport layer and Semantic Mapper in upstream and downstream directions.

The Codec Plug-in Development Specification for Java

Sending upstream messages received from the Semantic Mapper to a transport plug-in

When the Semantic Mapper produces normalized events, it sends them on to the codec layer by calling the codec plug-ins' `sendNormalisedEvent` methods (as defined above). The event codec must then encode the normalized event for transmission by the transport layer.

In order to send messages upstream to an event transport, a codec plug-in must have a reference to the transport plug-in object. Typically, an event codec does this by building up a map of registered transport plug-ins from the parameters passed to the `addEventTransport` and `removeEventTransport` methods. It might then use a property provided in the configuration file (e.g. `<property name="transportName" value="MyTransport"/>`) to determine which event transport to use when the `sendNormalisedEvent` method is called.

Alternatively, if this transport plug-in will only ever be used in an adapter with just one codec plug-in, the `EventTransport` object could be stored in an instance field when it is provided to the `addEventTransport` method.

Once the plug-in has a reference to the event transport (or transports) it will use, it can pass on normalized events it has encoded into transport messages by calling the transport plug-in `sendTransportEvent` method:

```
/**
 * Called by an event encoder to send an upstream message to the external
 * transport.
 *
 * Ownership of the message is transferred to the transport when this
 * function is called. It is assumed that the encoder and transport share
 * the same definition of the content of the event, so that the transport
 * can effectively interpret the event.
 *
 * @param event An object representing the event to be sent by the
 * transport, in a format shared by the encoder and transport.
 * @param timestamps A TimestampSet representing the timestamps attached to
 * the event.
 *
 * @throws TransportException Thrown by the transport if any error occurs
 * sending the message.
 */
public void sendTransportEvent(Object event, TimestampSet timestamps)
    throws TransportException;
```

For example, the implementation of the event codec's `sendNormalisedEvent` could look something like this:

```
// Select EventTransport using saved plugin property value
EventTransport transport = eventTransports.get(currentTransportName);
// Encode message
MyCustomMessageType message = myEncodeMessage(event);
// Send to Transport layer plugin
transport.sendTransportEvent(message, timestamps);
```

If an error occurs in the transport layer a `TransportException` is thrown. Typically such exceptions do not need to be caught by the codec plug-in, unless the codec plug-in is able to somehow deal with the problem.

A `CodecException` should be thrown if there is an error encoding the normalized event.

Note that there are no guarantees about which threads might call the `sendNormalisedEvent` method, so plug-in authors will need to consider any thread synchronization issues arising from use of shared data structures.

Any transport plug-in called by a Java codec plug-in must also be written in Java.

Communication with other layers

Sending downstream messages received from a transport plug-in to the Semantic Mapper

When a transport plug-in configured to work with the event codec receives a messages from its external message source, it will pass it on to the codec plug-in by calling the `sendTransportEvent` method (as defined above). It is then up to the codec plug-in to decode the message from whatever custom format is agreed between the transport and codec plug-ins into a standard normalized event that can be passed on to the Semantic Mapper.

When the message has been decoded it should be sent to the Semantic Mapper using its `sendNormalisedEvent` method:

```
/**
 * Called by the event codec to send a decoded event to the Semantic Mapper.
 *
 * @param event A normalized event to be sent to the semantic mapper
 * @param timestamps A TimestampSet representing the timestamps attached to
 * the event.
 *
 * @throws SemanticMapperException Thrown by the Semantic Mapper if there
 * is a problem mapping the event
 */
public void sendNormalisedEvent(NormalisedEvent event,
    TimestampSet timestamps)
    throws SemanticMapperException;
```

For example, the implementation of the event codec's `sendTranasportEvent` could look something like this:

```
// (Assume there's an instance field: SemanticMapper semanticMapper)
// Decode message
NormalisedEvent normalisedEvent = myDecodeMessage(event);
// Send to Transport layer plug-in
semanticMapper.sendNormalisedEvent(normalisedEvent, timestamps);
```

If an error occurs in the Semantic Mapper, a `SemanticMapperException` is thrown. Typically such exceptions do not need to be caught by the codec plug-in, unless the codec plug-in is able to somehow deal with the problem.

A `CodecException` should be thrown if there is an error decoding the normalized event.

Communication with other layers

Java codec exceptions

`CodecException` is the exception class that should be thrown by a codec plug-in whenever the one of its methods is called and an error prevents the method from successfully completing — for example, a message that cannot be encoded or decoded because it has an invalid format.

A `CodecException` object always has an associated `message`, which is a `String` explaining the problem (this may include information about another exception that caused the `CodecException` to be thrown).

There is also a `code` field that specifies the kind of error that occurred; the possible codes are defined as constants in the `CodecException` class:

```
/**
 * Some unspecified internal error occurred
 */
public static final int INTERNALERROR = 1;
/**
 * Couldn't encode an incoming normalized event
 */
public static final int ENCODINGFAILURE = 2;
/**
 * Couldn't decode an incoming customer event
 */
public static final int DECODINGFAILURE = 3;
/**
 * Trouble sending encoded event to transport
 */
public static final int TRANSPORTFAILURE = 4;
/**
 * Trouble sending decoded event to Semantic Mapper
 */
public static final int MAPPINGFAILURE = 5;
/**
 * Codec was passed an invalid property set
 */
public static final int BADPROPERTIES = 6;
```

Like the `TransportException` object, `CodecException` defines a number of constructors, to make it easy to set up the exception's information quickly in different situations:

```
/**
 * Constructs a CodecException from a string message describing the
 * error, and assumes an error code of TRANSPORTFAILURE.
 *
 * @param message The cause of the error.
 */
public CodecException(String message) { ... }
/**
 * Constructs a CodecException from a string message describing the
 * error and a numeric code representing the class of error.
 *
 * @param message The cause of the error.
 * @param code One of the CodecException error codes.
 */
public CodecException(String message, int code) { ... }
/**
 * Constructs a CodecException from a string message describing the
 * error, and an exception object that is the cause of the error. It
 * assumes an error code of TRANSPORTFAILURE.
 *
 * @param message The cause of the error. This message will be suffixed
 * with the message of the 'cause' exception.
 * @param cause The exception object that caused the error.
 */
public CodecException(String message, Throwable cause) { ... }
/**
 * Constructs a CodecException from a string message describing the
 * error, a numeric code representing the class of error and an exception
 * object that is the cause of the error.
 *
 * @param message The cause of the error. This message will be suffixed
 * with the message of the 'cause' exception.
 * @param cause The exception object that caused the error.
 * @param code One of the CodecException error codes.
 */
public CodecException(String message, Throwable cause, int code) { ... }
```

The Codec Plug-in Development Specification for Java

Semantic Mapper exceptions

Codec plug-ins should never need to construct or throw `SemanticMapperException` objects, but they need to be able to catch them if they are thrown from the `SemanticMapper.sendNormalisedEvent` method when it is called by the event codec.

`SemanticMapperException` has exactly the same set of constructors as the `CodecException` class described above. The only significant different is the set of error codes, which for `SemanticMapperException` are as follows:

```
/**
 * Some unspecified internal error occurred
 */
public static final int INTERNALERROR = 1;
/**
 * Couldn't convert customer event to an Apama event
 */
public static final int MAPPINGFAILURE = 2;
/**
 * Couldn't queue converted event for injection into the Engine
 */
public static final int INJECTIONFAILURE = 3;
```

[The Codec Plug-in Development Specification for Java](#)

Logging

See ["Logging from plug-ins in Java" on page 339](#) for information about how codec plug-ins should log error, status and debug information.

[The Codec Plug-in Development Specification for Java](#)

Working with normalized events

The function of a decoding codec plug-in is to convert incoming messages into a standard normalized event format that can be processed by the Semantic Mapper. Events sent upstream to an encoding codec plug-in are provided to the plug-in in this same format.

Normalized events are essentially dictionaries of name-value pairs, where the names and values are both character strings. Each name-value pair nominally represents the name and content of a single field from an event, but users of the data structure are free to invent custom naming schemes to represent more complex event structures. Names must be unique within a given event. Values may be empty or `null`.

Some examples of normalized event field values for different types are:

- `string "a string"`
- `integer "1"`
- `float "2.0"`
- `decimal "100.0d"`
- `sequence<boolean> "[true,false]"`
- `dictionary<float,integer> "{2.3:2,4.3:5}"`
- `SomeEvent "SomeEvent(12)"`

Note: When assigning names to fields in normalized events, keep in mind that the `fields` and `transport` attributes for event mapping conditions and event mapping rules both use a list of fields delimited by spaces or commas. This means, for example that `<id fields="Exchange EX,foo" test="==" value="LSE"/>` will successfully match a field called "Exchange", "EX" or "foo", but *not* a field called "Exchange EX,foo". While fields with spaces or commas in their names may be included in a payload dictionary in upstream or downstream directions, they cannot be referenced directly in mapping or id rules.

To construct strings for the normalized event fields representing container types (dictionaries, sequences, or nested events), use the event parser/builder found in the `util5.3.jar` file, which is located in the Apama installation's `lib` directory. The following examples show how to add a sequence and a dictionary to a normalized event (note the escape character `"\"` used in order to insert a quotation mark into a string).

```
List<String> list = new ArrayList<String>();
list.add("abc");
list.add("de\"f");
Map<String,String> map = new HashMap<String,String>();
map.put("key1", "value1");
map.put("key\"{}2", "value\"{}2");
final SequenceFieldType STRING_SEQUENCE_FIELD_TYPE =
    new SequenceFieldType(StringFieldType.TYPE);
final DictionaryFieldType STRING_DICT_FIELD_TYPE =
    new DictionaryFieldType(StringFieldType.TYPE, StringFieldType.TYPE);
NormalisedEvent event = new NormalisedEvent();
event.add("mySequenceField",
    STRING_SEQUENCE_FIELD_TYPE.format(list));
event.add("myDictionaryField", STRING_DICT_FIELD_TYPE.format(map));
```

The programming interface for constructing and using normalized events is made up of three Java classes: `NormalisedEvent`, `NormalisedEventIterator` and `NormalisedEventException`. `NormalisedEvent` is the most important part of the interface, and encapsulates the data and operations that can be performed on a single normalized event. Some of these operations return `NormalisedEventIterator` objects, which support the process of stepping through the name-value pairs in the normalized event. Any errors encountered result in instances of `NormalisedEventException` being thrown.

This section provides an overview of the capabilities of the two main classes. See the Javadoc documentation for full information on the normalized event interface.

[The Codec Plug-in Development Specification for Java](#)

The `NormalisedEvent` class

The `NormalisedEvent` class represents a single normalized event. The following methods are provided for examining and modifying the name-value pairs making up the event:

- `size` - Return the number of elements (name-value pairs) currently stored by the event.
- `empty` - Check whether the event is empty or not.
- `add` - Add a new name-value pair to the event. The given name and value will be copied into the event. If an element with the same name already exists, it will not be overwritten. This returns an iterator into the events at the point where the new element was added.
- `addQuick` - Add a new name-value pair to the event. The given name and value will be copied into the event. If an element with the same name already exists, it will not be overwritten.
- `remove` - Remove the named element from the normalized event.

- `removeAll` - Remove all elements from the normalized event. The `empty` function will return `true` after this function has been called.
- `replace` - Change the value of a named element in the normalized event. If an element with the given name already exists, its value will be replaced with a copy of the given value. Otherwise, a new element is created just as though `addQuick` had been called.
- `exists` - Check whether a given element exists in the normalized event.
- `find` - Search for a named element in the normalized event. Returns an iterator into the event at the point where the element was located.
- `findValue` - Search for a named element in the normalized event and return its value.
- `findValueAndRemove` - Search for a named element in the normalized event and return its value. If found, the element will also be removed from the event. Returns `null` if the specified element does not exist or has value `null`.
- `findValueAndRemove2` - Search for a named element in the normalized event and return its value. If found, the element will also be removed from the event. Unlike `findValueAndRemove` it throws an exception if the value is not found.
- `first` - Return an iterator pointing to the first element of the normalized event. Successive calls to the `next` function of the returned iterator will allow you to visit all the elements of the event.
- `last` - Return an iterator pointing to the last element of the normalized event. Successive calls to the `back` function of the returned iterator will allow you to visit all the elements of the event.
- `toString` - Return a printable string representation of the normalized event.

Threading note: Normalised events are not thread-safe. If your code will be accessing the same normalized event object (or associated iterators) from multiple threads, you must implement your own thread synchronization to prevent concurrent modification.

A public zero-argument constructor is provided for creation of new (initially empty) `NormalisedEvent` objects.

Working with normalized events

The `NormalisedEventIterator` class

Several of the `NormalisedEvent` methods return an instance of the `NormalisedEventIterator` class, which provides a way to step through the name-value pairs making up the normalized event, forwards or backwards.

The following public methods are provided:

- `valid` - Check whether the iterator points to a valid element of the normalized event. Typically used as part of the loop condition when iterating over the contents of an event.
- `key` - Return the key (name) associated with the current event element pointed to by the iterator.
- `value` - Return the value associated with the current event element pointed to by the iterator.
- `next` - Move the iterator to the next element of the underlying normalized event instance. The iterator must be in a valid state (such that the `valid` function would return `true`) before this function is called. Note that the order in which elements is returned is not necessarily the same as the order in which they were added. The order may change as elements are added to or removed from the underlying event.

- `back` - Move the iterator to the previous element of the underlying normalized event instance. The iterator must be in a valid state (such that the `valid` function would return `true`) before this function is called. Note that the order in which elements is returned is not necessarily the same as the order in which they were added. The order may change as elements are added to or removed from the underlying event.

There is no public constructor; iterators are created and returned only by `NormalisedEvent` methods.

See the Javadoc documentation for full information about the classes introduced in this section.

Working with normalized events

Java Codec Example

As part of the IAF distribution Apama includes the `JStringCodec` codec layer plug-in, in the `samples\iaf_plugin\java\simple\src` directory.

The `JStringCodec` plug-in converts between normalized events and a text string representation that can be customized using plug-in configuration properties.

Developers are encouraged to explore this sample to see what a typical codec plug-in implementation looks like.

See ["IAF samples" on page 274](#) for more information about this sample. The section ["The String Codec plug-in" on page 127](#) describes how the `JStringCodec` plug-in can be used in practice.

Java Codec Plug-in Development

Getting started with Java codec layer plug-in development

Your distribution includes a complete ‘skeleton’ implementation of a codec layer plug-in in order to make development of new plug-ins faster.

This is located in the `samples\iaf_plugin\java\skeleton\src` directory of the installation, in a file called `SkeletonCodec.java`. The `SkeletonCodec` class complies fully with the Codec Plug-in Development Specification, but contains none of the custom encoding and decoding functionality that would be present in a full codec plug-in.

The code contains `TODO`: comments indicating the main changes that need to be made to develop a useful plug-in. These include:

- Adding code to `sendTransportEvent` to decode a message received from the transport layer into a normalized event (if supported).
- Adding code to `sendNormalisedEvent` to encode a message received from the Semantic Mapper transport into a message that can be sent on by the transport layer (if supported).
- Adding code to validate and save any new plug-in properties that are to be supported, in `updateProperties`.
- Adding code to initialize and clean up resources associated with the plug-in’s operation. This would usually be done in the `updateProperties` and `cleanup` methods.

Depending on your requirements, it may also be necessary to make changes to the other main methods – `addEventTransport`, `removeEventTransport`, `flushUpstream`, `flushDownstream`, `getStatus`, and the constructor.

The `skeleton` directory includes an Apache Ant build file called `build.xml` that provides a convenient way to build `.jar` files of compiled classes from plug-in source files, ready for use with the IAF.

Java Codec Plug-in Development

■ Logging from plug-ins in Java	339
■ Using the latency framework	340

This section describes other programming interfaces provided with the Apama software that may be useful in implementing transport layer and codec plug-ins for the IAF.

Developing Custom Adapters

Logging from plug-ins in Java

This API provides a mechanism for recording status and error log messages from the IAF runtime and any plug-ins loaded within it. Plug-in developers are encouraged to make use of the logging API instead of custom logging solutions so that all the information may be logged together in the same standard format and log file(s) used by other plug-ins and the IAF runtime.

The logging API also allows control of logging verbosity, so that any messages below the configured logging level will not be written to the log. The logging level and file are initially set when an adapter first starts up – see ["Logging configuration \(optional\)" on page 273](#) for more information about the logging configuration.

The Java logging API is based around the `Logger` class.

The recommended way of using the `Logger` class is to have a `private final com.apama.util.Logger` variable, and then create an instance in the transport or codec's constructor based on the plug-in name, such as the following:

```
private final Logger logger;
public MyTransport(String name, ...)
{
    super(...);
    logger = Logger.getLogger(name);
}
```

The `Logger` class supports the following logging levels:

- `FORCE`
- `CRIT`
- `FATAL`
- `ERROR`
- `WARN`
- `INFO`
- `DEBUG`
- `TRACE`

For each level, there are three main methods. For example, for logging at the `DEBUG` level, here are the three main methods:

- `logger.debug(String)` — Logs a message, if this log level is currently enabled.
- `logger.debug(String, Throwable)` — Logs the stack trace and message of a caught exception together with a high-level description of the problem. Apama strongly recommend logging exceptions like this to assist with debugging in the event of problems.
- `logger.isDebugEnabled()` — Determines whether messages at this log level are currently enabled (this depends on the current IAF log level, which may be changed dynamically). Apama strongly recommend checking this method's result (particularly for `DEBUG` messages) before logging messages where constructing the message string may be costly, for example:

```
if (logger.isDebugEnabled())
    logger.debug("A huge message was received, and the string
        representation of it is: "+thing.toString()+
        " and here is some other useful info: "+foo+", "+bar);
```

Note that there is no point using the `*Enabled()` methods if the log message is a simple string (or string plus exception), such as:

```
logger.debug("The operation completed with an error: ", exception);
```

To make it easier to diagnose any errors that may occur, Apama recommends one of the following methods to log the application's stack trace:

- `errorWithDebugStackTrace(java.lang.String msg, java.lang.Throwable ex)` — Logs the specified message at the `ERROR` level followed by the exception's message string, and then logs the exception's stack trace at the `DEBUG` level.
- `warnWithDebugStackTrace(java.lang.String msg, java.lang.Throwable ex)` — Logs the specified message at the `WARN` level followed by the exception's message string, and then logs the exception's stack trace at the `DEBUG` level.

See the [Javadoc](#) documentation for more information about the `Logger` class.

[Plug-in Support APIs for Java](#)

Using the latency framework

The latency framework API provides a way to measure adapter latency by attaching high-resolution timing data to events as they stream into, through, and out of the adapter. Developers can then use these events to compute upstream, downstream, and round-trip latency numbers, including latency across multiple adapters.

The `sendNormalisedEvent()` and `sendTransportEvent()` methods contain a `TimestampSet` parameter that carries the microsecond-accurate timestamps that can be used to compute the desired statistics.

HTML Javadoc documentation for `com.apama.util.TimestampSet` and `com.apama.util.TimestampConfig` classes is provided as part of the Apama documentation set. The Javadoc is located in the Apama installation's `doc\javadoc` directory.

[Plug-in Support APIs for Java](#)

Java timestamp

A timestamp is an index-value pair. The index represents the point in the event processing chain at which the timestamp was recorded, for example “upstream entry to semantic mapper” and the value is a floating point number representing the time. The `TimestampSet` class defines a set of standard indexes but a custom plug-in can define additional indexes for even finer-grained measurements. When you add a custom index definition, be sure to preserve the correct order, for example, an index denoting an “entry” point should be less than an one denoting an “exit” point from that component.

Timestamps are relative measurements and are meant to be compared only to other timestamps in the same or similar processes on the same computer.

[Using the latency framework](#)

Java timestamp set

A timestamp set is the collection of timestamps that are associated with an event. The latency framework API provides functions that developers can use to add, inspect, and remove timestamps from an event’s timestamp set.

The timestamp set is represented as a `dictionary` of integer-float pairs, where the integer index refers to the location at which the timestamp was added and the floating-point time gives the time at which an event was there.

[Using the latency framework](#)

Java timestamp configuration object

The constructors and `updateProperties()` methods for transport and codec plug-ins take this additional argument: `TimestampConfig`.

A timestamp configuration object contains a set of fields that a plug-in can use to decide whether to record and/or log timestamp information. The fields in the object are:

- `recordUpstream` — If true, the plug-in should record timestamps for all upstream events it processes, and pass these along to the upstream component, if any.
- `recordDownstream` — If true, the plug-in should record timestamps for all downstream events it processes, and pass these along to the downstream component, if any.
- `logUpstream` — If true, the plug-in should log the latency for all upstream events it processes, at the logging level given by the `logLevel` member. A plug-in may implicitly enable upstream timestamp recording if upstream logging is enabled.
- `logDownstream` — If true, the plug-in should log the latency for all downstream events it processes, at the logging level given by the `logLevel` member. A plug-in may implicitly enable downstream timestamp recording if downstream logging is enabled.
- `logRoundtrip` — If true, the plug-in should log the “round trip” latency for all events it processes in either direction, if possible. At its simplest, the round trip latency can just be the difference between the largest and smallest timestamps passed to the plug-in, or an individual plug-in may choose to present some more plug-in-specific latency number. As with the other logging options, the logging level given by the `logLevel` member should be used.
- `logLevel` — The logging verbosity level to use if any of the timestamp logging options are enabled.

Java latency framework API

The Java interface for the latency framework is declared in the header file `com.apama.util.TimestampSet` class.

The functions of interest are the following.

Table 6. Java latency framework API

<code>void addNow(java.lang.Integer index)</code>	Add a new name-time pair to the event.
<code>void addTime(java.lang.Integer index, java.lang.Double theTime)</code>	Add a new index-time pair to the timestamp.
<code>void clear()</code>	Removes all mappings from this map.
<code>boolean containsKey(java.lang.Object index)</code>	Returns true if this map contains the specified key
<code>boolean containsValue(java.lang.Object time)</code>	Returns true if this map maps one or more names to the specified time.
<code>java.util.Set<java.util.Map.Entry<java.lang.Integer, java.lang.Double>> entrySet()</code>	
<code>java.lang.Double findTime(java.lang.Integer index)</code>	Search for a named element in the timestamp set and return its time.
<code>java.lang.Double findTimeAndRemove(java.lang.Integer index)</code>	Search for a named element in the timestamp set and return its time.
<code>java.lang.Double get(java.lang.Object index)</code>	Returns the time to which the specified name is mapped, or null if the map contains no mapping for this index.
<code>static double getMicroTime()</code>	Get the current microsecond-accurate relative timestamp.
<code>boolean isEmpty()</code>	Returns true if this map contains no index-time mappings.
<code>java.util.Iterator<java.util.Map.Entry <java.lang.Integer,java.lang.Double>> iterator()</code>	Returns a standard Java Iterator over the contents of the TimestampSet using Map.Entry objects.

<code>java.util.Set<java.lang.Integer> keySet()</code>	
<code>java.lang.Double put(java.lang.Integer index, java.lang.Double time)</code>	Adds or replaces the specified (index,time) pair in the underlying map
<code>void putAll(java.util.Map<? extends java.lang.Integer, ? extends java.lang.Double> m)</code>	Copies all of the mappings from the specified map to this map These mappings will replace any mappings that this map had for any of the indices currently in the specified map.
<code>java.lang.Double remove(java.lang.Object index)</code>	Removes the mapping for this index from this map if present.
<code>void replace(java.lang.Integer index, java.lang.Double newTime)</code>	Change the time of a named element in the timestamp set.
<code>void replaceWithNow(java.lang.Integer index)</code>	Change the time of a named element in the timestamp set.
<code>int size()</code>	Get the number of elements (name-time pairs) currently stored by the event.
<code>java.lang.String toString()</code>	Return a printable string representation of the timestamp set.
<code>java.util.Collection<java.lang.Double> values()</code>	Returns a collection view of the times contained in this map.

Using the latency framework

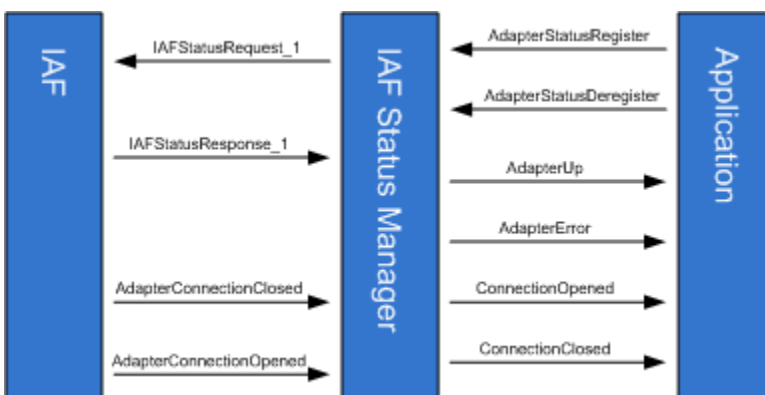
Chapter 15: Monitoring Adapter Status

■ IAFStatusManager	345
■ Application interface	345
■ Returning information from the getStatus method	348
■ Connections and other custom properties	351
■ Asynchronously notifying IAFStatusManager of connection changes	352
■ StatusSupport	355

Status information is available between the correlator and an adapter. When developing an IAF adapter, the adapter author can provide the ability to make use of this status information. Basic information, such as whether an adapter is up or down, is available using a standard Apama monitor. Other information, such as the number of connections an adapter has, can be provided by using the `getStatus()` method in an adapter's transport and codec. Optionally, adapter authors can also add code to the adapter's service monitors to send and receive specific status information that application developers can then use when they write Apama applications that connect to the adapters.

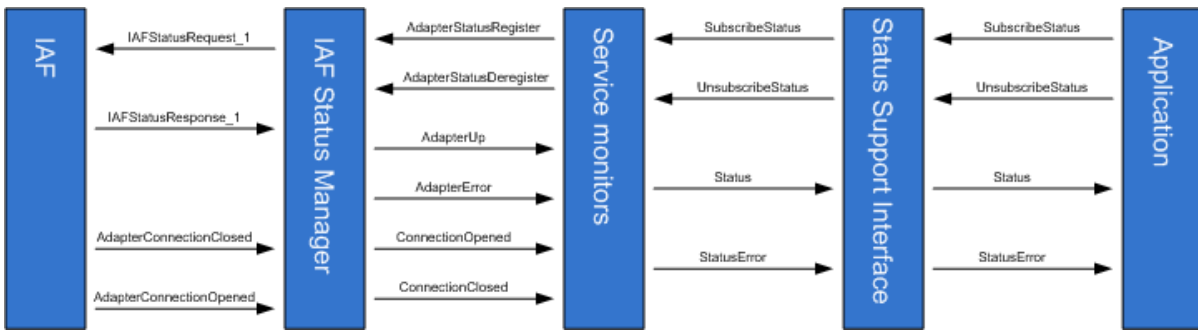
Apama provides the following two mechanisms for handling IAF Adapter status information:

- **IAFStatusManager** — The IAFStatusManager manages the connection status and other status information from the adapter to the correlator. In order to retrieve adapter status information, the IAFStatusManager needs to be injected into the correlator and the adapter author needs to add a small amount of code to the adapter. Application authors can then make use of status information available from the IAFStatusManager.



For information on using the IAFStatusManager, see ["IAFStatusManager" on page 345](#)

- **StatusSupport** — StatusSupport is a generic interface (or contract) between an Apama application and an adapter's service monitors. This interface provides a way to provide an application with a similar view of all the status information available from multiple adapters. In order to use the StatusSupport interface, an adapter author writes code in the adapter's service monitors that send or receive specific StatusSupport events. In turn, the application author writes code to implement the desired behavior for handling the StatusSupport events.



Using the StatusSupport interface is optional. For more information on using this interface, see ["StatusSupport" on page 355](#).

Developing Custom Adapters

IAFStatusManager

The IAFStatusManager translates events from the adapter into simple status events for applications to consume. The monitor, `IAFStatusManager.mon` is found in the Apama installation's `adapters\monitors` directory. In order to use the monitor:

- The adapter author is required to return information about the adapter's open connections in the adapter's `getStatus` method, which is called every few seconds when the IAFStatusManager service monitor polls the IAF for status. Adapters written in Java must return an `ExtendedTransportStatus` or `ExtendedCodecStatus` object from `getStatus()`; adapters written in C++ must return `AP_EventTransportStatus` or `AP_ExtendedCodecStatus`.
- The adapter may optionally also send notifications about a connection as soon as it is opened or closed, by sending a normalized event representation of the `AdapterConnectionOpened` or `AdapterConnectionClosed` events to the correlator. This simply allows the correlator to find out about connectivity change more quickly than is the case if it needs to wait for the next status poll.

The IAFStatusManager has the following interfaces:

- An *application* interface to communicate with the consumers of the adapter status information — usually adapter service monitors.
- An IAF *adapter* interface is optional and can be used by adapter authors to issue connection notifications.

Application interface

The `IAFStatusManager.mon` file defines the event interface between it and a consumer of an adapter's status information, which is usually an adapter's service monitor. The application interface can be used to communicate status information to both the adapter's service monitors as well as Apama applications. The application interface events are either input events or output events. Input events are sent from a consumer of adapter status information to the IAFStatusManager. Output events are sent from the IAFStatusManager to a consumer of adapter status information.

Input events

The IAFStatusManager is a subscription based interface. This means that a consumer of adapter status information, such as an application service monitor, needs to send the input events `AdapterStatusRegister` and `AdapterStatusDeregister` events to register or deregister as a consumer for adapter status information. Once a subscription is made to the IAFStatusManager, the IAFStatusManager periodically receives information from the adapter and begins sending status information to the registered consumer in the form of output events — see ["Output events" on page 347](#).

The IAFStatusManager defines the following input events:

- `AdapterStatusRegister` — An event sent by a client that is interested in receiving status events from the specified codec and transports. The fields of this event uniquely identify a subscription.

```
event AdapterStatusRegister {
    string adapterName;
    string codec;
    string transport;
    string codecVersion;
    string transportVersion;
    string configVersion;
    string channel;
}
```

The fields of the `AdapterStatusRegister` event are:

- `adapterName` — An identifier that will be used to refer to this transport and codec pair when registering, deregistering or monitoring adapter status.
- `codec` — The name of the adapter's codec as it is specified in the adapter configuration file.
- `transport` — The name of the adapter's transport as it is specified in the adapter configuration file.
- `codecVersion` — The codec version that the client depends on, or an empty string ("") if the client can use any version of the codec.
- `transportVersion` — The transport version that the client depends on, or an empty string ("") if the client can use any version of the transport.
- `configVersion` — The value of this field can be empty, but if a value is specified it must agree with the `CONFIG_VERSION` key returned in the `ExtendedTransportStatus` OR `ExtendedCodecStatus` object (for Java) or in the `statusDictionary` of an `AP_EventTransportStatus` OR `AP_EventCodecStatus` (for C++).
- `channel` — The name of the channel the IAF adapter is receiving events from.
- `AdapterStatusDeregister` — An event sent by a client to unregister its subscription for status events.

```
event AdapterStatusDeregister {
    string adapterName;
}
```

The `adapterName` field is the identifier used to refer to this transport and codec pair when deregistering

Output events

Once a consumer of status information (such as an application service monitor) is registered with the `IAFStatusManager`, it begins to receive status information in the form of `IAFStatusManager` output events. Output events include connection information, adapter availability, and any custom information put into the dictionary by the transport or codec. For more information about adding custom information, see ["Connections and other custom properties" on page 351](#).

The `IAFStatusManager` defines the following output events:

- **AdapterUp** — An event routed from the IAF Signaling Service to notify that the adapter is up.

```
event AdapterUp {
    string adapterName;
    float latency;
    dictionary<string, string> codecStatus;
    dictionary<string, string> transportStatus;
}
```

The fields of the `AdapterUp` event are:

- **adapterName** — This is the identifier used to refer to this transport and codec pair when monitoring adapter status.
 - **latency** — This is the difference (in seconds) between the time a request for status is sent from the `IAFStatusManager` to the IAF and the time the correlator receives the response.
 - **codecStatus** — Contains information about the adapter's codec. Standard keys are `VERSION`, `CONFIG_VERSION`, and `CONNECTION` (or in the case of multiple connections, keys of the form `CONNECTION_connectionName`).
 - **transportStatus** — Contains information about the adapter's transport. Standard keys are `VERSION`, `CONFIG_VERSION`, and `CONNECTION` (or in the case of multiple connections, keys of the form `CONNECTION_connectionName`).
- **AdapterError** — An event routed from the IAF Signaling Service to tell that there was an error getting the status of the adapter.

```
event AdapterError {
    string adapterName;
    string description;
}
```

The fields of the `AdapterError` event are:

- **adapterName** — This is the identifier used to refer to this transport and codec pair when monitoring adapter status.
 - **description** — A free form string describing the problem.
- **ConnectionClosed** — An event routed when the IAF Signaling Service discovers that the adapter's connection to an external service is closed.

```
event ConnectionClosed {
    string adapterName;
    string connectionName;
    string connectionGeneration;
}
```

The fields of the `ConnectionClosed` event are:

- **adapterName** — This is the identifier used to refer to this transport and codec pair when monitoring adapter status.

- `connectionName` — This is a unique identifier for the connection. If the adapter manages more than one connection, this will be the connection name returned by the adapter as a `CONNECTION_connectionNamegetStatus` key (but without the `CONNECTION_` prefix), or "" if the adapter only manages one connection. The `connectionName` is often a number but could be a string. One event will be sent for each connection the adapter manages.
- `connectionGeneration` — This identifies a successful connection attempt. If the connection fails, and then is successfully connected again, this will change. The `connectionGeneration` is often a number that is initialized with a timestamp when the adapter is created, then incremented every time it reconnects.
- `ConnectionOpened` — An event routed when the IAF Signaling Service discovers that the adapter's connection to an external service is established.

```
event ConnectionOpened {
    string adapterName;
    string connectionName;
    string connectionGeneration;
}
```

The fields of the `ConnectionOpened` event are:

- `adapterName` — This is the identifier used to refer to this transport and codec pair when monitoring adapter status.
- `connectionName` — This is a unique identifier for the connection. If the adapter manages more than one connection, this will be the connection name returned by the adapter as a `CONNECTION_connectionNamegetStatus` key (but without the `CONNECTION_` prefix), or "" if the adapter only manages one connection. The `connectionName` is often a number but could be a string. One event will be sent for each connection the adapter manages.
- `connectionGeneration` — This identifies a successful connection attempt. If the connection fails, and then is successfully connected again, this will change. The `connectionGeneration` is often a number that is initialized with a timestamp when the adapter is created, then incremented every time it reconnects.

Returning information from the `getStatus` method

The adapter's transport and codec `getStatus` methods periodically update status information. The transport and codec send this information to the `IAFStatusManager`. To take advantage of the `IAFStatusManager` for an adapter written in Java, the adapter author should implement the `getStatus` method so that it returns an `ExtendedTransportStatus` or `ExtendedCodecStatus` object. These objects include a `Properties` parameter, `statusInfo`, which contains custom information about the adapter.

For adapters written in C or C++, the adapter author should implement the `getStatus` function to include the `statusDictionary` in an `AP_EventTransportStatus` or `AP_EventCodecStatus` structure.

The `IAFStatusManager` then forwards the information to registered consumers of that transport or codec's status information in the form of a dictionary added to the `AdapterUp` event.

ExtendedTransportStatus

The `ExtendedTransportStatus` object is defined as follows:

```
public ExtendedTransportStatus(java.lang.String status,
    long totalReceived,
    long totalSent),
```

```
java.util.Properties statusInfo)
```

The object's parameters are:

- `status` - A string containing a transport-specific status message. Strings longer than 1024 characters will be truncated.
- `totalReceived` - The total number of downstream events received since the IAF was run.
- `totalSent` - The total number of upstream events sent since the IAF was run.
- `statusInfo` - Any additional status information about this transport. The standard `statusInfo` keys are:

- `VERSION=transport_version_string`

- `CONFIG_VERSION=config_version_string`

- `CONNECTION=connectionGeneration` (if the adapter manages only one connection)

or

- `CONNECTION_connectionId=connectionGeneration` (if the adapter manages multiple connections)

For more information on specifying the `CONNECTION` or `CONNECTION_connectionId` key, see ["Asynchronously notifying IAFStatusManager of connection changes" on page 352](#).

ExtendedCodecStatus

The `ExtendedCodecStatus` object is defined as follows:

```
public CodecStatus(java.lang.String status,
                  long totalDecoded,
                  long totalEncoded),
                  java.util.Properties statusInfo
```

The object's parameters are:

- `status` - A string containing a codec-specific status message. Strings longer than 1024 characters will be truncated.
- `totalDecoded` - The number of events decoded
- `totalEncoded` - The number of events encoded
- `statusInfo` - Any additional status information about this codec. Standard `statusInfo` keys are:

- `VERSION=codec_version_string`

- `CONFIG_VERSION=config_version_string`

- `CONNECTION=connectionGeneration` (if the adapter manages only one connection),

or

- `CONNECTION_connectionId=connectionGeneration` (if the adapter manages multiple connections)

AP_EventTransportStatus

The `AP_EventTransportStatus` object is defined as:

```
typedef struct {
    AP_char8* status;
    AP_uint64 totalReceived;
    AP_uint64 totalSent;
    AP_NormalisedEvent* statusDictionary;
```

```
} AP_EventTransportStatus;
```

The object's parameters are:

- `status` - A free-form text string containing a transport-specific status message. Strings longer than 1024 characters will be truncated.
- `totalReceived` - The total number of downstream events received since the IAF was run.
- `totalSent` - The total number of upstream events sent since the IAF was run.
- `statusDictionary` - Any additional status information about this transport. The standard `statusDictionary` keys are:
 - `VERSION=transport_version_string`
 - `CONFIG_VERSION=config_version_string`
 - `CONNECTION=connectionGeneration` (if the adapter manages only one connection)or
 - `CONNECTION_connectionId=connectionGeneration` (if the adapter manages multiple connections)

AP_EventCodecStatus

The `AP_EventCodecStatus` object is defined as:

```
typedef struct {
    AP_char8* status;
    AP_uint64 totalDecoded;
    AP_uint64 totalEncoded;
    AP_NormalisedEvent* statusDictionary;
} AP_EventCodecStatus;
```

The object's parameters are:

- `status` - A free-form text string containing a codec-specific status message. Strings longer than 1024 characters will be truncated.
- `totalDecoded` - The number of events decoded.
- `totalEncoded` - The number of events encoded.
- `statusDictionary` - Any additional status information about this codec. Standard `statusDictionary` keys are:
 - `VERSION=codec_version_string`
 - `CONFIG_VERSION=config_version_string`
 - `CONNECTION=connectionGeneration` (if the adapter manages only one connection),or
 - `CONNECTION_connectionId=connectionGeneration` (if the adapter manages multiple connections)

Example

In the following example, the custom status information for `VERSION` and `CONNECTION` is included in the information returned by the `getStatus` method:

```
public static final String TRANSPORT_VERSION="1";
protected long connGeneration;
...
public TransportStatus getStatus()
```



```

{
    Properties properties=new Properties();
    properties.setProperty("VERSION", TRANSPORT_VERSION);
    if(market!=null)
    {
        properties.setProperty("CONNECTION",
            String.valueOf(connGeneration));
    }
    return new ExtendedTransportStatus("OK", numReceived, numSent, properties);
}

```

For more information on specifying the `CONNECTION` property, see ["Asynchronously notifying IAFStatusManager of connection changes" on page 352](#).

Monitoring Adapter Status

Connections and other custom properties

An adapter may deal with no connections, a single connection, or an arbitrary number of connections (for example, if it is a server socket that accepts clients connecting to it); an adapter may also deal with a set number of connections. In any case, an identifier needs to be assigned to each connection. A connection may be broken and then reconnected, with either the same or different identifier. It is useful to be able to detect a connection that has been dropped and then reconnected even if it has the same identifier. To facilitate this, a “generation” identifier can be associated with each connection identifier. While typically this generation identifier will be a number that is incremented, extra information may be contained in it.

Monitors can therefore detect when a connection has been reconnected; at this point any logon procedure needs to be repeated as the generation identifier has changed.

The state of all connections should be supplied in the `statusDictionary` field of the status struct in C/C++, or the `statusInfo` field of the `ExtendedCodecStatus` or `ExtendedTransportStatus` in Java.

Along with any other custom information, the adapter author can include connection information here. This will be passed to the correlator in event form and the `IAFStatusManager` will automatically attempt to pull out connection information from this data structure. If there is a single connection, a key should be supplied called `CONNECTION`. The value will be the generation identifier, typically a number. If the generation identifier changes, the `IAFStatusManager` will assume the connection has been dropped and reestablished, and will send appropriate events to the consumer of the status events.

If there are multiple connections, a key for each one should be supplied in the form `CONNECTION_<id>` to distinguish the different connections. Each one will also have a generation identifier associated with it. The same rules apply with the generation identifier as with a single connection.

In either case, if the connection is up, the property should be included, and if the connection is down, the property should not be included. This allows monitors to recover the state of what connections are made after losing connection to the IAF, and to determine when connections are opened or closed by polling.

The following Java example shows a simple adapter that reports the status of a single connection.

```

private long connectionGeneration = System.currentTimeMillis();
public TransportStatus getStatus()
{
    Properties properties = new Properties();
    properties.setProperty("VERSION", "MyTransport_v1.0");
    properties.put("CONFIG_VERSION", "1");
}

```

```

if (connected)
{
    properties.setProperty("CONNECTION",
        String.valueOf(connectionGeneration));
}
return new ExtendedTransportStatus("OK", totalReceived,
    totalSent, properties);
}

```

The following Java example demonstrates usage with multiple connections, iterating through a collection of `MyConnection` objects.

```

public TransportStatus getStatus()
{
    Properties properties = new Properties();
    properties.put("VERSION", "MyTransport_v1.0");
    properties.put("CONFIG_VERSION", "1");
    for (MyConnection con : connections.values())
    {
        if (!con.isClosed())
        {
            properties.put("CONNECTION_" + con.getId(), con.getGeneration());
        }
    }
    return new ExtendedTransportStatus(statusMessage, totalReceived,
        totalSent, properties);
}

```

Asynchronously notifying IAFStatusManager of connection changes

In addition to returning status information in response to a poll from the `IAFStatusManager`, an adapter may also send out events asynchronously when a connection is opened or closed.

This is done by creating and sending a `NormalisedEvent` object from the transport or codec to the semantic mapper. The `NormalisedEvent` object has special fields that allow for automatic mapping to an Apama event type— either `AdapterConnectionOpened` or `AdapterConnectionClosed`. The `AdapterConnectionOpened` and `AdapterConnectionClosed` events are then sent through the correlator to the `IAFStatusManager`.

The `NormalisedEvent` must have the following fields:

Table 7. NormalizedEvent fields

Field name	Field value
<code>AdapterConnectionOpenEvent</code> or <code>AdapterConnectionClosedEvent</code>	No value (empty string). This will either represent a connection opened or connection closed and be translated into <code>AdapterConnectionOpened</code> or <code>AdapterConnectionClosed</code> events respectively for the <code>IAFStatusManager</code> to consume.
<code>codecName</code>	Name of codec
<code>transportName</code>	Name of transport
<code>connectionName</code>	No value (empty string) if there is only one connection. If there is more than one connection, this

Field name	Field value
	should contain <code>CONNECTION_<id></code> and one event should be sent for every connection the adapter is concerned with.
<code>connectionGeneration</code>	Connection generation identifier. This identifies a successful connection attempt with a <code>connectionName</code> . If the connection fails, then is successfully connected again, this should change. This is usually a number that is incremented.

This connection information should have a direct correlation to the connection information sent in the `getStatus` implementation. Note that if the transport deals with only a single connection at a time, the `connectionName` will be "" (the empty string) instead of `CONNECTION`, as it is in the `getStatus` implementation.

The following is an example in Java of sending a `NormalisedEvent` that provides status information.

```
protected void sendAdapterConnectionStatusChangeNotification(boolean open,
    String reason, TimestampSet tss)
{
    if(decoder==null) return;
    NormalisedEvent ne=new NormalisedEvent();
    ne.add("codecName", codecName);
    ne.add("transportName", transportName);
    if(reason==null)
    {
        reason="";
    }
    if(open)
    {
        ne.add("AdapterConnectionOpenEvent", reason);
    }
    else
    {
        ne.add("AdapterConnectionClosedEvent", reason);
    }
    ne.add("connectionGeneration", String.valueOf(connGeneration));
    ne.add("connectionName", "");
    try
    {
        decoder.sendTransportEvent(ne, tss);
    }
    catch (CodecException e)
    {
        logger.error("Could not send message due to Codec error: ", e);
    }
    catch (SemanticMapperException e)
    {
        logger.error("Could not send message due to Semantic Mapper
            error: ", e);
    }
}
```

Note: When using these events, the `(J)NullCodec` must be used, unless you write a codec that handles these and passes them on to the correlator. For example, the `XMLCodec` by default will not forward these events to the semantic mapper. If you want to use the `XMLCodec`, you need to use the `(J)NullCodec` as the codec to send these particular events.

For more information on the implicit rules that the semantic mapper uses to automatically map the objects to `AdapterConnectionOpened` and `AdapterConnectionClosed` events, see ["Mapping AdapterConnectionClosed and AdapterConnectionOpened events" on page 354](#).

Mapping AdapterConnectionClosed and AdapterConnectionOpened events

As described in ["Asynchronously notifying IAFStateManager of connection changes" on page 352](#), the semantic mapper contains implicit rules to map `NormalisedEvent` objects that contain special fields to `AdapterConnectionClosed` and `AdapterConnectionOpened` events. This means you do not need to add mapping rules to your adapter's configuration file. These implicit rules are:

```
<event name="AdapterConnectionClosed"
  package="com.apama.adapters"
  direction="downstream"
  breakDownstream="false">
  <id-rules>
    <downstream>
      <id fields="codecName,
        transportName,
        connectionName,
        connectionGeneration"
        test="exists"/>
      <id fields="AdapterConnectionClosedEvent"
        test="exists"/>
    </downstream>
  </id-rules>
  <mapping-rules>
    <map apama="codecName"
      transport="codecName"
      type="string" default=""/>
    <map apama="transportName"
      transport="transportName"
      type="string" default=""/>
    <map apama="connectionName"
      transport="connectionName"
      type="string" default=""/>
    <map apama="connectionGeneration"
      transport="connectionGeneration"
      type="string" default=""/>
  </mapping-rules>
</event>
<event name="AdapterConnectionOpened"
  package="com.apama.adapters"
  direction="downstream"
  breakDownstream="false">
  <id-rules>
    <downstream>
      <id fields="codecName,
        transportName,
        connectionName,
        connectionGeneration"
        test="exists"/>
      <id fields="AdapterConnectionOpenEvent"
        test="exists"/>
    </downstream>
  </id-rules>
  <mapping-rules>
    <map apama="codecName"
      transport="codecName"
      type="string" default=""/>
    <map apama="transportName"
      transport="transportName"
      type="string" default=""/>
    <map apama="connectionName"
      transport="connectionName"
      type="string" default=""/>
    <map apama="connectionGeneration"
      transport="connectionGeneration"
      type="string" default=""/>
  </mapping-rules>
</event>
```

```
        type="string" default=""/>
    </mapping-rules>
</event>
```

StatusSupport

Consumers of the IAFStatusManager events are typically the adapter service monitors. In some cases it may be desirable for an Apama application to have a more generic view of components and their status information so that getting status information will look the same across all components in a system, regardless of component type. For example, in addition to the information provided by the IAFStatusManager such as whether the adapter is up or connected, it may be useful to provide confirmation that the adapter has successfully logged in to an external system or a message that the external system is down.

Apama provides an interface called the StatusSupport event interface to help define this. It allows applications (EPL code or scenarios and blocks) to see state from service monitors such as the adapter service monitors. In order to implement this behavior, adapter authors add code to the adapter service monitors to handle the various StatusSupport events. Developers of Apama applications can then add code to take appropriate actions for the StatusSupport events to their applications that use the adapters. In this way, an application can act as a “health monitor” and be notified when a component is down or what its status is at any given time.

The StatusSupport events are described ["StatusSupport events" on page 355](#).

The StatusSupport event interface is a subscription based interface, so consumers of this information will need to subscribe before receiving status information. The adapter service monitors need to reference count the status subscribers, so they do not stop sending status information if there are any interested consumers left. A subscription will only be removed when the call to remove the last one is made.

StatusSupport events

The StatusSupport event interface is defined in the `StatusSupport.mon` file, which is found in the `monitors` directory of the Apama installation (note, this is not the same directory as `adapters\monitors`).

All of the StatusSupport events contain the following fields:

- `serviceID` — The serviceID to subscribe to, a blank in this field targets all services
- `object` — The object to request status of - this may include:
 - “Connection” - whether connected or not
 - “MarketState” - a market may be “Open”, “Closed”, or other states
- `subServiceID` — The subService ID to subscribe to. Some services may expose several services. The interpretation of this string is adapter-specific.
- `connection` — The connection to subscribe to. Some services may expose several services. The interpretation of this string is adapter-specific.

The StatusSupport interface defines the following events:

- `SubscribeStatus` — This event is sent to the service monitor to subscribe to status.

```
event SubscribeStatus {
    string serviceID;
```

```

    string object;
    string subServiceID;
    string connection;
}

```

- **UnsubscribeStatus** —

```

event UnsubscribeStatus {
    string serviceID;
    string object;
    string subServiceID;
    string connection;
}

```

- **Status** —

```

event Status {
    string serviceID;
    string object;
    string subServiceID;
    string connection;
    string description;
    sequence<string> summaries;
    boolean available;
    wildcard dictionary <string, string> extraParams;
}

```

The additional fields for the `Status` event type are:

- `description` — A free-form text string giving a description of the status.
- `summaries` — The status of the object requested. This will be a well recognized sequence of words - for example, a financial market's "MarketState" may be "Open", "Closed", "PreOpen", etc. A Connection may be "Connected", "Disconnected", "Disconnected LoginFailed", "Disconnected TimedOut", etc. There should be at least one entry in the sequence.
- `available` — True if the object is "available" - the exact meaning is adapter specific; for example, connected, open for general orders, etc.
- `extraParams` — Extra parameters that do not map into any of the above. Convention is that keys are in TitleCase. e.g. "Username", "CloseTime", etc.

A `Status` event does not denote a change of state, merely what the current state is — in particular, one will be sent out after every `SubscribeStatus` request.

Any adapter specific information that the application needs to supply or be supplied can be passed in the `extraParams` dictionary — these are free-form (though there are conventions on the keys, see below).

- **StatusError** —

```

event StatusError {
    string serviceID;
    string object;
    string subServiceID;
    string connection;
    string description;
    boolean failed;
}

```

The additional field for this event type is:

- `failed` — Whether the subscription has been terminated. Any subscribers will need to send a new `SubscribeStatus` request after this.

Note that the purpose of the `StatusError` event is to report a problem in the delivery of status information, not to report an "error" status. A `StatusError` should be sent when the service is unable

to deliver status for some reason. For example, reports on the status of an adapter transport's connection to a downstream server cannot be sent if the correlator has lost its connection to the adapter — in this case the service would be justified in sending a `StatusError` event for the downstream connection status. However, in the same situation the service should continue to send normal `Status` events for the correlator-adapter connection status, as this status is known. The `available` flag in these `Status` events would of course be set to `false` to indicate that the connection is down.

If the `failed` flag in a `StatusError` event is `true`, this indicates that the failure in status reporting is permanent and any active status subscriptions will have been cancelled and receivers will need to re-subscribe if they wish to receive further status updates from the service. If the `failed` flag is `false`, the failure is temporary and receivers should assume that the flow of `Status` events will resume automatically at some point.

Chapter 16: Out of Band Connection Notifications

■ Mapping example	358
■ Ordering of out of band notifications	360

When a sender and receiver component, such as a correlator, connects to or disconnects from the Integration Adapter Framework (IAF), the IAF automatically sends *out of band* notification events to adapter transports. Out of band notifications are events that are automatically sent to all public contexts in a correlator whenever any component (an IAF adapter, dashboard, another correlator, or a client built using the Apama SDKs) connects or disconnects from the correlator. These out of band events, which are defined in the `com.apama.oob` package, are:

- `ReceiverConnected`
- `SenderConnected`
- `ReceiverDisconnected`
- `SenderDisconnected`

The `ReceiverConnected` and `SenderConnected` events contain the name of the component that is connecting. When correlators and IAF adapters send a notification event, the format of the string that contains the component name is as follows:

```
"name (on port port_number) "
```

The `name` is the name that was specified when the component was started. For correlators and IAF adapters, you can specify a name with the `--name` option when you start the component. The name defaults to `correlator` or `iaf` according to the type of component. The `port_number` is the port that the connecting receiver or sender is running on.

Out of band events make it possible for a developer of an adapter to add appropriate actions for the adapter to take when it receives notice that a component has connected or disconnected. For example, an adapter can cancel outstanding orders or send a notification to an external system. In order to make use of the out of band events, adapters need to provide suitable mapping in the adapter configuration file. Adapters are also free to ignore these events.

For general information about using out of band notifications, see "Out of band connection notifications" in *Developing Apama Applications*.

Developing Custom Adapters

Mapping example

Out of band events will only be received by codecs and transports if the semantic mapper is configured to allow them through. The semantic mapper should be configured as for any other set of events which it may wish to pass down. For more information on creating semantic mapping rules, see "[The <event> mapping rules](#)" on page 265.

For example:

```

<event package="com.apama.oob" name="ReceiverDisconnected"
  direction="upstream" encoder="$CODEC$" inject="false">
  <id-rules>
    <upstream />
  </id-rules>
  <mapping-rules>
    <map type="string" default="OutOfBandReceiverDisconnected" transport="_name" />
    <map apama="physicalId" transport="physicalId" default="" type="string" />
    <map apama="logicalId" transport="logicalId" default="" type="string" />
  </mapping-rules>
</event>
<event package="com.apama.oob" name="ReceiverConnected"
  direction="upstream" encoder="$CODEC$" inject="false">
  <id-rules>
    <upstream />
  </id-rules>
  <mapping-rules>
    <map type="string" default="OutOfBandReceiverConnected" transport="_name" />
    <map apama="name" transport="appName" default="" type="string" />
    <map apama="host" transport="address" default="" type="string" />
    <map apama="physicalId" transport="physicalId" default="" type="string" />
    <map apama="logicalId" transport="logicalId" default="" type="string" />
  </mapping-rules>
</event>
<event package="com.apama.oob" name="SenderDisconnected"
  direction="upstream" encoder="$CODEC$" inject="false">
  <id-rules>
    <upstream />
  </id-rules>
  <mapping-rules>
    <map type="string" default="OutOfBandSenderDisconnected" transport="_name" />
    <map apama="physicalId" transport="physicalId" default="" type="string" />
    <map apama="logicalId" transport="logicalId" default="" type="string" />
  </mapping-rules>
</event>
<event package="com.apama.oob" name="SenderConnected" direction="upstream"
  encoder="$CODEC$" inject="false">
  <id-rules>
    <upstream />
  </id-rules>
  <mapping-rules>
    <map type="string" default="OutOfBandSenderConnected" transport="_name" />
    <map apama="name" transport="appName" default="" type="string" />
    <map apama="host" transport="address" default="" type="string" />
    <map apama="physicalId" transport="physicalId" default="" type="string" />
    <map apama="logicalId" transport="logicalId" default="" type="string" />
  </mapping-rules>
</event>

```

The events are transmitted to signify the following events:

- **ReceiverConnected** — an external receiver has connected; the IAF can now send events to it.
- **ReceiverDisconnected** — an external receiver has disconnected; events will not be sent to this external receiver until it reconnects.
- **SenderConnected** — an external sender has connected. This external sender may send events following this event.
- **SenderDisconnected** — an external sender has disconnected. No more events will be received from this sender until a new **SenderConnected** message event is received.

However, adapters can make use of a disconnect message to not transmit events until such time as a connect occurs. For example, an adapter can coalesce events or tell external system to stop sending. Note that if multiple senders and receivers are connected and disconnected, the adapter will need to keep track of which one is connected.

Ordering of out of band notifications

The following guidelines describe when out of band connection and disconnection messages are received, and how this interacts with the framework provided to IAF adapters:

Transports and codecs will not be sent events until after their start function has been called and completed. Transports should not start generating events until their start function has been called. The first event that is delivered after the start function is called will be a `SenderConnected` or `ReceiverConnected` event, if the semantic mapper is configured to pass them through. An adapter will always receive the `SenderConnected` before it begins to receive any other events, but the ordering of the `ReceiverConnected` and `SenderConnected` events is not guaranteed.

If a correlator (or other component) disconnects or terminates while the adapter is running, the adapter will receive both `ReceiverDisconnected` and `SenderDisconnected` events. Again, the ordering of these events is not guaranteed. Once a `SenderDisconnected` event is received, no further events from that correlator will be received until a `SenderConnected` event is received. When a `ReceiverDisconnected` event is received, no more events will be sent to that correlator until a `ReceiverConnected` event is received. Note that in this situation, some previously sent events may not yet have reached that correlator. The events will be discarded (or sent to other receivers, if other receivers are connected).

On a reload of an adapter, the adapter will be stopped, new configuration loaded, and the adapter restarted. During this period, the IAF will not drop its connection unless the configuration of which components to connect to has changed. As such, if prior to stopping for a reload the correlator was connected, it is safe to assume that it remains connected unless, on reload, the adapters receive `SenderDisconnected` or `ReceiverDisconnected` events.

During a reload, the IAF can also load new adapters. In this event, as the IAF may already have a connection open, no `ReceiverConnected` or `ReceiverDisconnected` event may be received by the new adapters. It is thus recommended to not change transports and codecs when reconfiguring the IAF if the adapters depend on receiving the out of band events. In practice, it is unusual to change the loaded transports or codecs.

Once an adapter has entered a stopped state, it will not receive any further events (unless it later re-enters a started state). Since the shutdown order of the IAF is to move all adapters to their 'stopped' state, then disconnect from downstream processes, adapters will not receive a final 'disconnected' event. Therefore, the adapter may need to notify external systems on the `stop` function being called, as well as on disconnected events.

The following topics describe the ordering the transport will see of calls to `start`, `stop` and the transport receiving out of band and normal events.

When starting the IAF

- IAF Begins Initialization
- Adapters Initialize
- IAF connects to Correlator
- [IAF Receives `SenderConnected` and `ReceiverConnected` - these are queued]
- Adapter changes state to Started

-
- Prior to receiving any other events, the semantic mapper (and then codec and adapter) receive the now unqueued out of band `SenderConnected` and `ReceiverConnected` events.
 - The `SenderConnected` event will arrive before any other events from said sender are delivered

IAF shutdown requested

- Adapters state changes from started to stopped
- IAF disconnects from correlator
- Because transport is in state 'stopped', no events are received
- IAF terminates

IAF Configuration Reload

- Transport is in state 'started'
- IAF transitions transport to state 'stopped'
- IAF keeps its connection to the correlator up
- IAF transitions transport to state started
- Transport checks state, notices that it believes a connection is up, and continues to work without any changes

IAF Configuration reload changes correlator connection

- Transport is in state 'started'
- IAF transitions transport to state 'stopped'
- IAF breaks its connection to the correlator
- IAF receives `ReceiverDisconnected` and `SenderDisconnected`
- Since the transports are stopped, these events are queued
- IAF opens a new connection to a new correlator
- IAF receives `ReceiverConnected` and `SenderConnected`
- Since the transports are stopped, these events are queued
- IAF transitions transport to state started
- Transport checks state, notices that it believes a connection is up, and continues to work without any changes
- Prior to receiving any other events, the `ReceiverDisconnected` and `SenderDisconnected` events are received
- Following these, but prior to receiving any other events, the `ReceiverConnected` and `SenderConnected` events are received
- The transport can then behave as if a new connection has been made

Correlator dies (and a new one is started) while the IAF is running

- Transport is in state 'started'

-
- Correlator breaks its connection to the IAF
 - IAF receives `ReceiverDisconnected` and `SenderDisconnected`
 - Transport receives `ReceiverDisconnected` and `SenderDisconnected`
 - Following `SenderDisconnected` no more events should arrive from the correlator
 - Time Passes
 - A new correlator makes a connection to the IAF
 - IAF receives `ReceiverConnected` and `SenderConnected`
 - Transport receives `ReceiverConnected` and `SenderConnected`
 - The transport can now behave as if a new connection has been made

Out of Band Connection Notifications

Chapter 17: The Event Payload

■ Creating a payload field	363
■ Accessing the payload in the correlator	364

As already described, Apama events are rigidly structured and need to comply with a precise event type definition. This describes the structure of a particular event: in particular its name, as well as the order, name, type and number of its constituent fields.

By contrast, external events, even when they are of the same ‘type’ or nature (e.g. all `Trade` events or `News` headlines) might vary in format and structure, even when originating from the same source or feed.

In order to accommodate this, Apama provides an optional *payload* field in Apama events. The payload field, typically the last field in an event type definition, can embed any number of additional optional fields in addition to the always-present primary fields.

For example, consider an external message that can appear in several guises, but where each always consists of a particular subset of critical fields together with a variable number of additional optional fields.

If it is desired that these varying guises are mapped to a single Apama event type, then this needs to be defined so that its fields correspond to the subset of critical (and always present) fields, followed by a payload field into which the additional (and optional) fields are embedded.

Developing Custom Adapters

Creating a payload field

When so configured, the Semantic Mapper will transparently create a payload field in an event.

As described in "[Event mappings configuration](#)" on page 259, one of the attributes of the event-mapping element `<event>` is `copyUnmappedToDictionaryPayload`.

Note: As of Release 4.1, the `copyUnmappedToPayload` attribute is deprecated.

The `copyUnmappedToDictionaryPayload` attribute defines what the Semantic Mapper should do with any fields in the incoming messages that do not match with any field mapping, i.e. if there are no rules that specifically copy their contents into a field within the Apama event being generated.

If both these attributes are set to `false` (or if one is `false` and the other is not present), any unmapped fields are discarded. If the value of either `copyUnmappedToDictionaryPayload` or `copyUnmappedToPayload` is set to `true`, unmapped fields will be packaged into a payload field, called `__payload`, set to be the last field of the Apama event type generated by the Semantic Mapper. If the values of both `copyUnmappedToDictionaryPayload` and `copyUnmappedToPayload` are set to `true`, `copyUnmappedToPayload` is ignored.

The Event Payload

Accessing the payload in the correlator

Apama recommends that you use `copyUnmappedToDictionaryPayload` instead of the deprecated `copyUnmappedToPayload`. Using `copyUnmappedToDictionaryPayload` puts all the payload fields in a standard EPL dictionary, it is more easily accessed, more efficient, and easier to use. In contrast, `copyUnmappedToPayload` uses a custom format that requires the use of the `PayloadPlugin` to decode and access.

The contents of the payload field used by `copyUnmappedToPayload` are structured as a list of field name and value pairs. Although this may be directly accessed as a string within EPL code, Apama provides the Payload Correlator plug-in to make extraction and manipulation of specific fields more straightforward. This plug-in is included with Apama.

The Payload Extraction plug-in is available as `libPayloadPlugin.so` in the Apama installation's `lib` directory on UNIX. On Microsoft Windows it is available as `PayloadPlugin.dll` in the Apama installation's `bin` folder.

For information on how to use the Payload Extraction plug-in, and on the suite of EPL functions it provides, see "Using the payload extraction plug-in" in *Developing Apama Applications*.

[The Event Payload](#)

IV Developing Custom Clients

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](#)" on page 241.

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 Java EngineClient API, the EventService API, or the ScenarioService API.

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

Chapter 18: The Client Software Development Kits

■ Basic operations	366
■ The client software development kits	367

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](#)" on page 241.

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.
- Java EngineClient 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.

[Developing Custom Clients](#)

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, and Java. 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\index.html`

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.5.3` (on Solaris or Linux), or `engine_client5.3.lib` (on Windows). The equivalent Java classes are provided within `engine_client5.3.jar` and are documented in `doc\javadoc`. The .NET engine client libraries, `engine_client5.3.dll` and `engine_client_dotnet5.3.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.
```

To develop, build, and test an Apama application, the recommendation is that you use Oracle JDK 8. The minimum you can use is JDK 7.

To deploy an Apama application, the recommendation is that you use Oracle JRE 8, which is the version that Apama provides. Use of any JRE other than the one that Apama ships with is discouraged.

The Client Software Development Kits

Chapter 19: The Client Software Development Kits for C++ and Java

■ The library classes	369
■ The complete definitions	375

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 395](#) 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 Java EngineClient API" on page 411](#)
- ["The EventService API" on page 425](#)
- ["The ScenarioService API" on page 432](#)

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 371](#).

Developing Custom Clients

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:

- `sendEvents` — 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 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 explanation 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.EngineManagementFactory` – 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 375](#).

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 through 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 monitor in the Engine.
*
* @param name The name of the monitor 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 20: Using the SDKs – C++ and Java Examples

■ A simple echo server in C++	395
■ The full example in C++	397
■ The Java example	401

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.

Developing Custom Clients

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, see "Getting Started with Apama EPL" in *Developing Apama 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
            msg = "Failed to disconnect from engine";
            com::apama::engine::disconnectFromEngine(engine);
        }
        if (initDone) {
            // Shutdown the Engine library
            msg = "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 << msg << ": " << 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\LowLevelEngineManagement` 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" +
    "}" + "\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 21: The C Client Software Development Kit

■ Using the C SDK	404
■ The complete C example	407

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.

Developing Custom Clients

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 374](#) 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 375](#) 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>
#else
#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 22: The EngineClient API

■ The Java EngineClient API	411
■ The .NET Engine Client library	419

Apama provides the following APIs for programmatic interfacing to the event correlator:

- "The Java EngineClient API" on page 411
- "The .NET Engine Client library" on page 419

Developing Custom Clients

The Java EngineClient API

We recommend that where possible Java developers should use the Java EngineClient API instead of the more basic low-level management SDK. Also consider using the higher-level EventService API for clients that primarily involve sending and receiving events.

The full reference to the Java EngineClient 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 EngineClient API

The key elements

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

The Java EngineClient 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 Java EngineClient 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 418](#).

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

Overview of the `EngineClientBean`

Recommended usage

The recommended way of using the `EngineClientBean` 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 369 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

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 injectCDF(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 injectCDF(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 Java EngineClient API

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 distinctions between *delete*, *forced delete*, and *kill*:

- *Delete* works only if the EPL element being deleted (an event type or monitor) is not referenced by any other element. For example, you cannot delete an event type that is used by any monitors.
- *Forced delete* deletes 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 also deleted.
- For monitors only, *kill* terminates all instances of the specified monitor regardless of whether an instance is performing any processing. For example, killing a monitor that is stuck in an infinite loop, would remove the monitor at the next loop iteration. Any `ondie()` and `onunload()` actions defined in killed monitors are not executed.

The delete operations are:

- `void deleteAll()` - Deletes everything from the remote correlator (or *engine*). This is equivalent to a *deleteName* 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 monitor in the remote correlator.
- `void killNames(java.util.List<java.lang.String> names)` - *Kill* a number of EPL monitors.
- `void killNamesFromFile(java.util.List<java.lang.String> filenames)` - *Kill* a number of EPL monitors 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 Java EngineClient 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 Java `EngineClient` 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 Java EngineClient 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 Java EngineClient 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 Java EngineClient 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 Java EngineClient API

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\index.html`

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

- ["The Java EngineClient API" on page 411](#)
- ["The EventService API" on page 425](#)
- ["The ScenarioService API" on page 432](#)

This section provides the following information:

- ["Using the .NET client library" on page 420](#)
- ["Java and .NET namespace/class mapping" on page 421](#)

The EngineClient API

Using the .NET client library

To make use of the .NET wrapper, add the `engine_client_dotnet5.3.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_client5.3.dll`
- `engine_client_dotnet5.3.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\dotnet` 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_dotnet5.3.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.

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 8. Java and .NET namespace/class mapping

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

Java	.NET
com.apama.engine.beans.interfaces	Apama.Engine.Client
ConnectOperationsInterface	ICConnectOperations
ConsumerOperationsInterface	IConsumerOperations
DeleteOperationsInterface	(in IMessagingClient)
EngineClientInterface	IClientEngine
InjectOperationsInterface	(in IMessagingClient)
InspectOperationsInterface	(in ICorrelatorManagement)
ReceiveConsumerOperationsInterface	(in IMessagingClient)
ReceiveOperationsInterface	(in IMessagingClient)
SendOperationsInterface	(in IMessagingClient)
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

Java	.NET
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
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

Java	.NET
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

Chapter 23: The EventService API

■ The key elements	425
■ The IEventService interface	425
■ The IEventServiceChannel interface	426
■ The EventServiceFactory class	428
■ Examples of use	428

The Apama EventService application programming interface (API) is layered on top of the Java EngineClient API described in ["The Java EngineClient API" on page 411](#). The EventService API interface allows client applications to focus on events and channels.

Developing Custom Clients

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\java\EventService` 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).

- `IServiceChannel 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 `IServiceChannel` 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(IServiceListener eventListener)`

Add an `IServiceListener` 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(  
    IServiceListener eventListener,  
    EventType eventType)
```

Add an `IServiceListener` 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\java\EventService` 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 occurred)
        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 `asyncRequestResponse()` 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 24: The ScenarioService API

■ The key elements	432
■ The IScenarioService interface	433
■ The ScenarioDefinition interface	434
■ The IScenarioInstance interface	436
■ The ScenarioServiceFactory class	438
■ The ScenarioServiceConfig class	439
■ Examples of use	440

The Apama ScenarioService application programming interface (API) is layered on top of the Apama EventService API described in "[The EventService API](#)" on [page 425](#). 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.

Developing Custom Clients

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\java\ScenarioService` 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 `PropertyChangeEvents` 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 `PropertyChangeEvents`, 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<ParameterTypeEnum> 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 `PropertyChangeEvents`, 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 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 `ScenarioIDS` 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 `ScenarioIDS` 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 (`EventServiceChannels`), 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 (`EventServiceChannels`), 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\java\ScenarioService` 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 capabilities 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 `ScenarioIdS`.

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