

Connecting Apama Applications to External Components

Version 10.11.3

April 2022

This document applies to Apama 10.11.3 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 © 2013-2022 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 <https://softwareag.com/licenses/>.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at <https://softwareag.com/licenses/> and/or in the root installation directory of the licensed product(s).

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 <https://softwareag.com/licenses/> and/or in the root installation directory of the licensed product(s).

Document ID: PAM-EXT-10113-20220411

Table of Contents

About this Guide.....	9
Documentation roadmap.....	10
Online Information and Support.....	11
Data Protection.....	12
 I Working with Connectivity Plug-ins.....	 13
1 Getting Started with Connectivity Plug-ins.....	15
Concepts.....	16
Adding the connectivity bundles.....	19
Specifying the main settings in the properties file.....	20
Specifying the settings for the connectivity chains in the YAML file.....	20
Controlling how the correlator interacts with a chain.....	21
Using codecs.....	21
Writing EPL.....	21
2 Using Connectivity Plug-ins.....	23
Overview of using connectivity plug-ins.....	24
Static and dynamic connectivity chains.....	26
Configuration file for connectivity plug-ins.....	26
Host plug-ins and configuration.....	30
Translating EPL events using the apama.eventMap host plug-in.....	32
Using reliable transports.....	33
Creating dynamic chains from EPL.....	37
Sending and receiving events with connectivity plug-ins.....	38
Deploying plug-in libraries.....	40
3 Developing Connectivity Plug-ins.....	41
Chain components and messages.....	42
Requirements of a plug-in class.....	43
Requirements of a transport chain manager plug-in class.....	47
Building plug-ins.....	49
C++ data types.....	51
Map contents used by the apama.eventMap host plug-in.....	54
Metadata values.....	58
Lifetime of connectivity plug-ins.....	60
Creating dynamic chains from a chain manager plug-in.....	62
User-defined status reporting from connectivity plug-ins.....	63
Logging and configuration.....	65
Threading.....	66
Developing reliable transports.....	67
General notes for developing transports.....	70
 II Standard Connectivity Plug-ins.....	 71
4 The Universal Messaging Transport Connectivity Plug-in.....	73
About the Universal Messaging transport.....	74
Overview of using Universal Messaging in Apama applications.....	74

Setting up Universal Messaging for use by Apama.....	81
Configuring the Universal Messaging connectivity plug-in.....	82
EPL and Universal Messaging channels.....	91
Using Universal Messaging connectivity from EPL.....	91
Monitoring Apama application use of Universal Messaging.....	92
5 The MQTT Transport Connectivity Plug-in.....	93
About the MQTT transport.....	94
Using MQTT connectivity from EPL.....	94
Loading the MQTT transport.....	95
Configuring the connection to MQTT.....	95
Mapping events between MQTT messages and EPL.....	97
Payload for the MQTT message.....	98
Wildcard topic subscriptions.....	98
Metadata for the MQTT message.....	98
Restrictions.....	99
6 The Digital Event Services Transport Connectivity Plug-in.....	101
About the Digital Event Services transport.....	102
Using Digital Event Services connectivity from EPL.....	103
Reliable messaging with Digital Event Services.....	104
7 The HTTP Server Transport Connectivity Plug-in.....	107
About the HTTP server transport.....	108
Loading the HTTP server transport.....	110
Configuring the HTTP server transport.....	110
Handling responses in EPL.....	114
Serving static files.....	116
Mapping events between EPL and HTTP server requests.....	116
HTTP server security.....	126
Monitoring status for the HTTP server.....	128
8 The HTTP Client Transport Connectivity Plug-in.....	131
About the HTTP client transport.....	132
Loading the HTTP client transport.....	132
Configuring the HTTP client transport.....	133
Mapping events between EPL and HTTP client requests.....	136
Monitoring status for the HTTP client.....	153
Configuring dynamic connections to services.....	154
Using predefined generic event definitions to invoke HTTP services with JSON and string payloads.....	154
9 The Kafka Transport Connectivity Plug-in.....	157
About the Kafka transport.....	158
Loading the Kafka transport.....	158
Configuring the connection to Kafka (dynamicChainManagers).....	158
Configuring message transformations (dynamicChains).....	160
Payload for the Kafka message.....	161
Metadata for the Kafka message.....	161
10 The Cumulocity IoT Transport Connectivity Plug-in.....	163
About the Cumulocity IoT transport.....	164
Configuring the Cumulocity IoT transport.....	165
Loading the Cumulocity IoT transport.....	170
Using managed objects.....	171
Using alarms.....	175

Using events.....	179
Using measurements.....	183
Using measurement fragments.....	188
Using operations.....	191
Receiving update notifications.....	195
Paging Cumulocity IoT queries.....	197
Invoking other parts of the Cumulocity IoT REST API.....	199
Invoking microservices.....	200
Monitoring status for Cumulocity IoT.....	201
Finding tenant options.....	202
Getting user details.....	203
Sample EPL.....	204
11 Codec Connectivity Plug-ins.....	209
The String codec connectivity plug-in.....	210
The Base64 codec connectivity plug-in.....	211
The JSON codec connectivity plug-in.....	212
The Classifier codec connectivity plug-in.....	216
The Mapper codec connectivity plug-in.....	217
The Batch Accumulator codec connectivity plug-in.....	221
The Message List codec connectivity plug-in.....	222
The Unit Test Harness codec connectivity plug-in.....	225
The Diagnostic codec connectivity plug-in.....	228
III Correlator-Integrated Support for the Java Message Service (JMS).....	231
12 Using the Java Message Service (JMS).....	233
Overview of correlator-integrated messaging for JMS.....	234
Getting started with simple correlator-integrated messaging for JMS.....	236
Getting started with reliable correlator-integrated messaging for JMS.....	246
Mapping Apama events and JMS messages.....	248
Dynamic senders and receivers.....	275
Durable topics.....	276
Receiver flow control.....	276
Monitoring correlator-integrated messaging for JMS status.....	277
Logging correlator-integrated messaging for JMS status.....	278
JMS configuration reference.....	285
Designing and implementing applications for correlator-integrated messaging for JMS.....	296
Diagnosing problems when using JMS.....	311
JMS failures modes and how to cope with them.....	313
IV Working with IAF Plug-ins.....	317
13 The Integration Adapter Framework.....	319
Overview.....	320
Architecture.....	321
The transport layer.....	323
The codec layer.....	323
The Semantic Mapper layer.....	324
Contents of the IAF.....	325
14 Using the IAF.....	327

The IAF runtime.....	328
IAF Management – Managing a running adapter I.....	335
IAF Client – Managing a running adapter II.....	336
IAF Watch – Monitoring running adapter status.....	337
The IAF configuration file.....	339
IAF samples.....	364
15 C/C++ Transport Plug-in Development.....	369
The C/C++ transport plug-in development specification.....	370
Transport example.....	373
Getting started with transport layer plug-in development.....	373
16 C/C++ Codec Plug-in Development.....	375
The C/C++ codec plug-in development specification.....	376
Transport example.....	384
Getting started with codec layer plug-in development.....	384
17 C/C++ Plug-in Support APIs.....	387
Logging from IAF plug-ins in C/C++.....	388
Using the latency framework.....	388
18 Transport Plug-in Development in Java.....	391
The transport plug-in development specification for Java.....	392
Example.....	395
Getting started with Java transport layer plug-in development.....	395
19 Java Codec Plug-in Development.....	397
The codec plug-in development specification for Java.....	398
Java codec example.....	403
Getting started with Java codec layer plug-in development.....	403
20 Plug-in Support APIs for Java.....	405
Logging from IAF plug-ins in Java.....	406
Using the latency framework.....	407
21 Monitoring Adapter Status.....	409
IAFStatusManager.....	411
Application interface.....	411
Returning information from the getStatus method.....	412
Connections and other custom properties.....	413
Asynchronously notifying IAFStatusManager of connection changes.....	414
StatusSupport.....	417
DataView support.....	420
22 Out of Band Connection Notifications.....	423
Mapping example.....	424
Ordering of out of band notifications.....	425
23 The Event Payload.....	429
V Standard IAF Plug-ins.....	431
24 The Database Connector IAF Adapter (ADBC).....	433
Overview of using ADBC.....	434
Registering your ODBC database DSN on Windows.....	435
Adding an ADBC adapter to an Apama project.....	436
Configuring the Apama database connector.....	437
The ADBCHelper application programming interface.....	444
The ADBC Event application programming interface.....	456

The Visual Event Mapper.....	478
Playback.....	480
Sample applications.....	481
Format of events in .sim files.....	481
25 The File IAF Adapter (JMultiFileTransport).....	483
File adapter plug-ins.....	484
File adapter service monitor files.....	485
Adding the File adapter to an Apama project.....	485
Configuring the File adapter.....	486
Overview of event protocol for communication with the File adapter.....	487
Opening files for reading.....	488
Specifying file names in OpenFileForReading events.....	490
Opening comma separated values (CSV) files.....	491
Opening fixed width files.....	492
Sending the read request.....	493
Requesting data from the file.....	493
Receiving data.....	493
Opening files for writing.....	494
LineWritten event.....	495
Monitoring the File adapter.....	496
26 The Basic File IAF Adapter (FileTransport/JFileTransport).....	497
27 Codec IAF Plug-ins.....	499
The String codec IAF plug-in.....	500
The Null codec IAF plug-in.....	501
The Filter codec IAF plug-in.....	503
The XML codec IAF plug-in.....	507
The CSV codec IAF plug-in.....	521
The Fixed Width codec IAF plug-in.....	524
VI Developing Custom Clients.....	529
28 The Client Software Development Kits.....	531
The client libraries.....	532
Working with event objects.....	534
Logging.....	534
Exception handling and thread safety.....	534
29 Engine Management API.....	537
30 Engine Client API.....	541
31 Event Service API.....	545
32 Scenario Service API.....	547

About this Guide

■ Documentation roadmap	10
■ Online Information and Support	11
■ Data Protection	12

Connecting Apama Applications to External Components describes how to connect Apama applications to any event data source, database, messaging infrastructure, or application.

Documentation roadmap

Apama provides documentation in the following formats:

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

You can access the HTML documentation on your machine after Apama has been installed:

- **Windows.** Select **Start > All Programs > Software AG > Tools > Apama *n.n* > Apama Documentation *n.n***. Note that **Software AG** is the default group name that can be changed during the installation.
- **UNIX.** Display the `index.html` file, which is in the `doc/apama-onlinehelp` directory of your Apama installation directory.

The following guides are available:

Title	Description
<i>Release Notes</i>	Describes new features and changes introduced with the current Apama release as well as earlier releases.
<i>Installing Apama</i>	Summarizes all important installation information and is intended for use with other Software AG installation guides such as <i>Using Software AG Installer</i> .
<i>Introduction to Apama</i>	Provides a high-level overview of Apama, describes the Apama architecture, discusses Apama concepts and introduces Software AG Designer, which is the main development tool for Apama.
<i>Using Apama with Software AG Designer</i>	Explains how to develop Apama applications in Software AG Designer, which is an Eclipse-based integrated development environment.
<i>Developing Apama Applications</i>	Describes the different technologies for developing Apama applications: EPL monitors, Apama queries, and Java. You can use one or several of these technologies to implement a single Apama application. In addition, there are C++ and Java APIs for developing components that plug in to a correlator. You can use these components from EPL.
<i>Connecting Apama Applications to External Components</i>	Describes how to connect Apama applications to any event data source, database, messaging infrastructure, or application.

Title	Description
<i>Building and Using Apama Dashboards</i>	Describes how to build and use an Apama dashboard, which provides the ability to view and interact with DataViews. An Apama project typically uses one or more dashboards, which are created in the Dashboard Builder. The Dashboard Viewer provides the ability to use dashboards created in the Dashboard Builder. Dashboards can also be deployed as simple web pages. 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>	Describes how to deploy components with Software AG Command Central, how to deploy and manage queries, and how to deploy Apama applications using Docker and Kubernetes. It also provides information for improving Apama application performance by using multiple correlators, for managing and monitoring Apama components over REST (Representational State Transfer), and for using correlator utilities and configuration files.

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

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

Online Information and Support

Product Documentation

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

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

Product Training

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

Tech Community

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

- Browse through our vast knowledge base.
- Ask questions and find answers in our discussion forums.
- Get the latest Software AG news and announcements.
- Explore our communities.
- Go to our public GitHub and Docker repositories at <https://github.com/softwareag> and <https://hub.docker.com/publishers/softwareag> and discover additional Software AG resources.

Product Support

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

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

Data Protection

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

I Working with Connectivity Plug-ins

1	Getting Started with Connectivity Plug-ins	15
2	Using Connectivity Plug-ins	23
3	Developing Connectivity Plug-ins	41

1 Getting Started with Connectivity Plug-ins

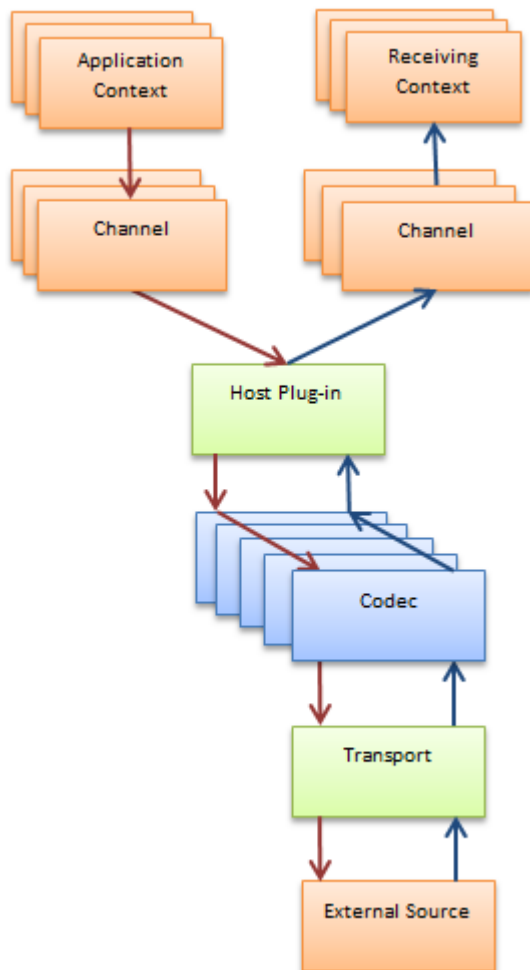
■ Concepts	16
■ Adding the connectivity bundles	19
■ Specifying the main settings in the properties file	20
■ Specifying the settings for the connectivity chains in the YAML file	20
■ Controlling how the correlator interacts with a chain	21
■ Using codecs	21
■ Writing EPL	21

Concepts

A connectivity chain is used to get events of a certain shape from a particular external source and convert them from a given format into Apama events and back. There will be at least one chain for each event source. There could also be one chain for each format of events you want to receive from that source. There can be multiple instances of a given chain definition in the configuration file, corresponding to multiple connections to that source. The choice of which chain to use for a given event may vary between different sources.

All chains begin with a host plug-in, which defines how events are converted into Apama events. All chains end with a transport, which determines the external source we are connecting to. Between these you have an ordered sequence of any number of codecs. These are used to convert messages from the format produced by the transport to that which the host plug-in will consume and vice versa. Messages are passed between the elements in a connectivity chain using an abstract message format made up of maps, lists, byte arrays and various primitive formats. As well as the (arbitrary) payload, there is also a (nested) map of metadata with string keys and any of the supported types as values.

The following diagram shows a typical connectivity chain:



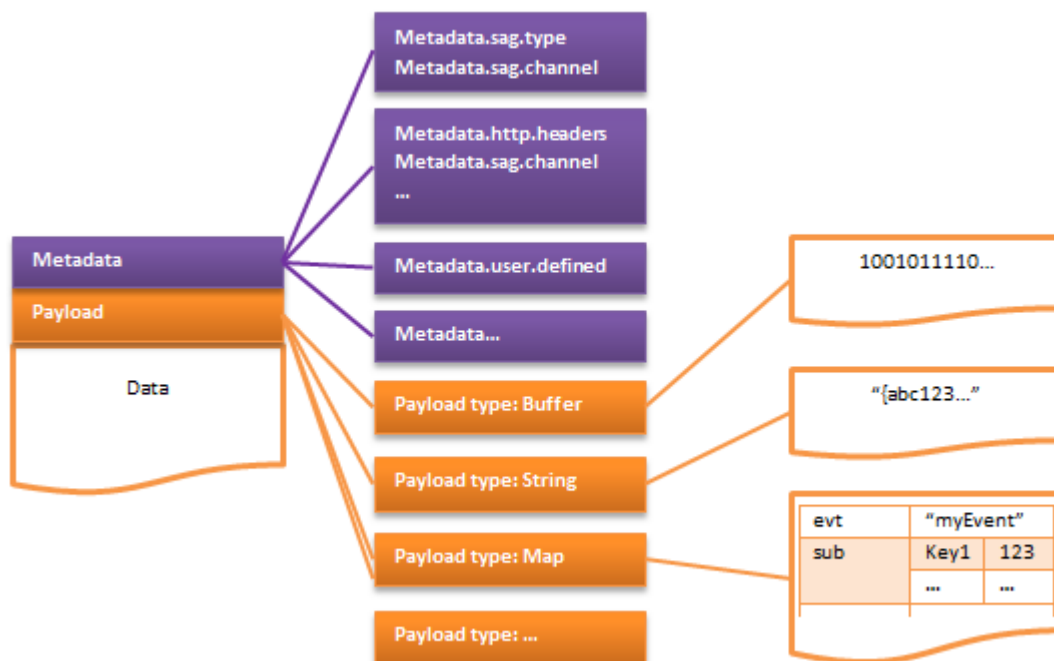
The connectivity chain in the above diagram uses several plug-ins and codecs:

- The host plug-in is the connection between the correlator and the chain of codecs that manipulate and transform the messages as they pass through the chain. There are two host plug-ins to choose from:
 - `apama.eventMap`. This is the most common choice for a host plug-in. It produces and consumes Apama events as a map with the keys being the names of the fields in the event and the `metadata.sag.type` element set to the name of the event.
 - `apama.eventString`. This host plug-in consumes events in Apama's proprietary string format. It is usually used for interaction with legacy systems such as the IAF which use Apama's string format.

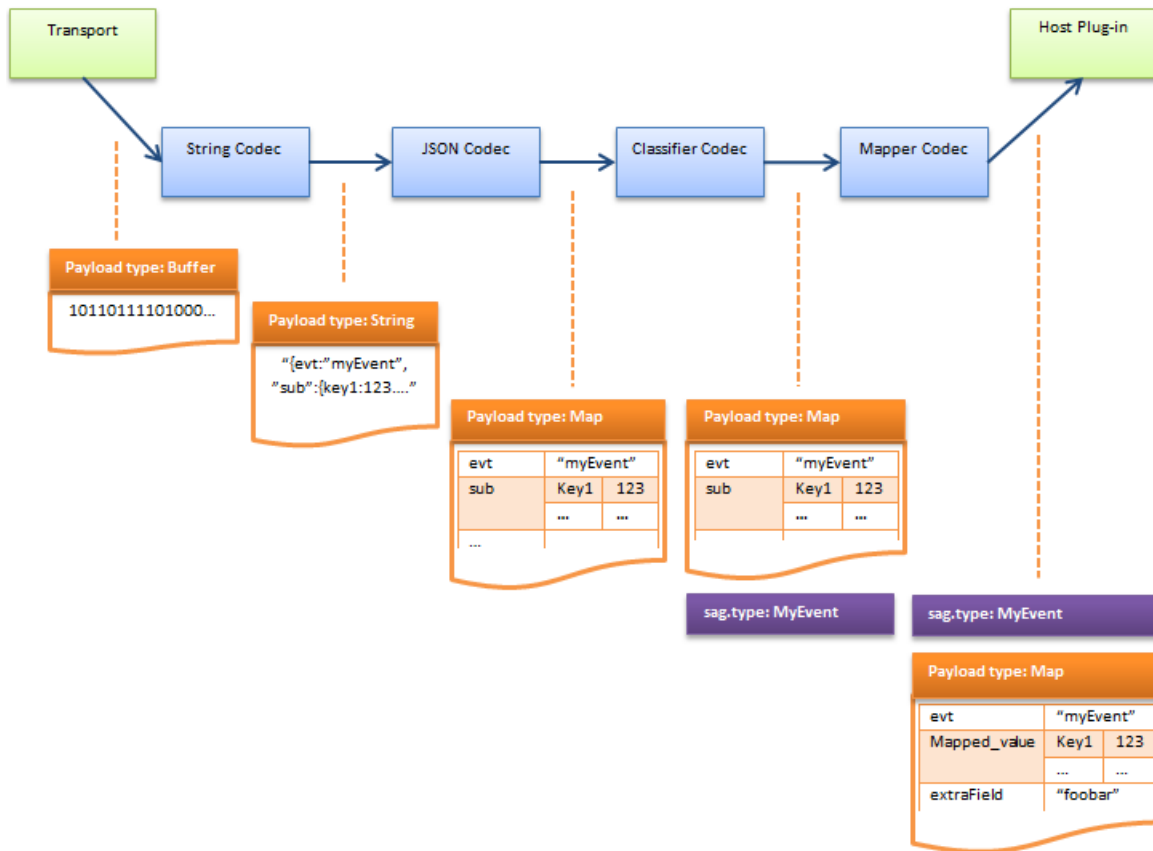
For detailed information, see [“Host plug-ins and configuration” on page 30](#).

- Any message is transformed into a compatible form for either the host or the transport as it flows through the codec pipeline toward its destination.
- The transport is the plug-in that can actually retrieve or send the message to the external resource. This can be one of the standard plug-ins such MQTT or HTTP (see for detailed information) or a plug-in written by the user.

Any message passing through from a transport plug-in to the host has common features that will be similar in most implementations. The following diagram describes the common features (metadata and payload) and indicates some of the common values they can take.



Codecs may be of various types. The following diagram describes an incoming message transformation:



Apama provides a selection of codecs by default for converting from common formats and for use with customer-provided codecs to provide standard transformations. The above diagram uses the following standard codecs:

- The String codec transforms an inbound message to an Apama string. It converts a buffer (transport side) to a string (host side) by decoding it as UTF-8.

The outbound flow performs the reverse mapping in the other direction (see the diagram below) and places a string representation of the message into an Apama string.

- The JSON codec transforms the inbound message to a compatible representation. It converts a string (transport side) to a structured type (usually map) by parsing it as a string JSON object.

The outbound flow performs the reverse mapping in the other direction (see the diagram below) and transforms a JSON-compatible representation to a JSON string in the payload.

- Messages to the host are identified as a particular event type by matching patterns on the metadata or message contents. The Classifier codec defines what the `metadata.sag.type` will be and uses rules that are triggered by values held by a field or fields in the inbound message.

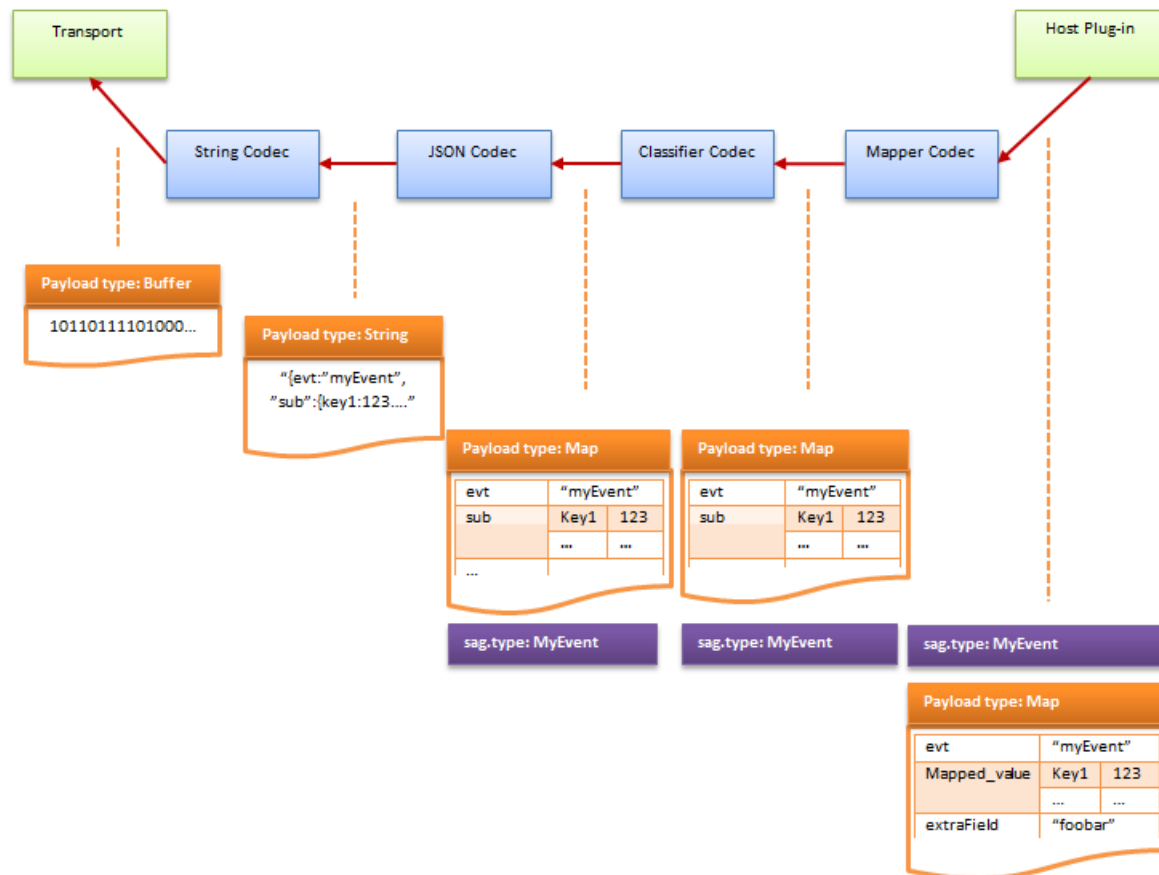
This does not apply to outbound messages (see the diagram below).

- The Mapper codec sets up the map fields ready for either the correlator or transport depending upon the direction of the flow. It moves fields in the event around and adds default values to missing fields. It can have different rules for different event types, and it can have symmetric

rules or different rules depending on direction. `towardsHost` is the inbound direction, and `towardTransport` is the outbound direction. The metadata and payload keys provide the source or destination for the values.

For detailed information (and more codecs), see [“Codec Connectivity Plug-ins” on page 209](#).

The following diagram describes an outgoing message transformation:



Adding the connectivity bundles

You can easily add connectivity bundles, either to a new Apama project or to an existing Apama project. You can do this in Software AG Designer or using the `apama_project` tool. For more information, see "Adding adapters to projects" in *Using Apama with Software AG Designer* and "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications*.

To get started quickly, we recommend that you import the `genericSendReceive` sample as a project into Software AG Designer. This is a simple generic EPL application for sending and receiving test messages, for use with any connectivity plug-in. You can find it in the `samples/connectivity_plugin/application` directory of your Apama installation. See the `README.txt` file in that directory for detailed information on how to import and run this sample.

With the above sample, you can add one or more connectivity bundles to the project.

You configure a connectivity plug-in by editing the properties and YAML files that come with the connectivity bundle.

Specifying the main settings in the properties file

Edit the properties file (with the file extension `.properties`) which comes with the connectivity bundle to specify the main settings such as hosts, ports, and credentials.

The properties defined in this file are used to replace specific substitution values in the YAML file. In many cases, all the configuration that is required is in the `.properties` file.

See also "Using properties files" in *Deploying and Managing Apama Applications*.

Specifying the settings for the connectivity chains in the YAML file

The configuration file for a connectivity plug-in describes both the chains and the plug-ins making up each chain. It is written in the YAML markup language. See also "Using YAML configuration files" in *Deploying and Managing Apama Applications*.

Edit the YAML configuration file (with the file extension `.yaml`) which comes with the connectivity bundle and specify all required information for creating the connectivity chains.

Find out from the documentation for the standard connectivity plug-ins how connectivity chains are created for the transport you are using. See .

- If the transport provides a dynamic chain manager, then the chains are created by the transport. In this case, you have to provide one or more `dynamicChains` definitions in the YAML configuration file so that the chains are then created from these definitions.
- If the transport does not provide a dynamic chain manager, you have to choose between
 - defining the chains statically in your YAML configuration file using `startChains`, or
 - creating the chains dynamically from EPL using `dynamicChains` definitions in the YAML configuration file.

Your decision on which transport to use affects which sections of the YAML configuration file you have to edit. See also [“Static and dynamic connectivity chains” on page 26](#).

Sometimes it is necessary to make more changes in the YAML configuration file. For example:

- You can add custom connectivity transport plug-ins or codec plug-ins that you have built yourself in Java or C++, or have downloaded from the Apama community. See also [“Configuration file for connectivity plug-ins” on page 26](#).
- You can change the configuration settings for the transport. This depends on whether your transport has a dynamic chain manager that will be either in the `dynamicChainManager` section

and/or under the transport's name in the `startChains` or `dynamicChains` sections of the YAML configuration file. For an example, see [“Configuring the HTTP server transport” on page 110](#).

Note:

The Digital Event Services connectivity plug-in is an exception. In this case, the YAML configuration file should not be modified by the user. All of the required configuration is to be done in the properties file.

Controlling how the correlator interacts with a chain

A host plug-in controls how the correlator interacts with a chain. In the YAML configuration file, the host plug-in is the first element in the configuration of a chain. You can configure the `apama.eventMap` host plug-in, for example, to specify the following:

- A default channel to send to.
- A channel to subscribe to for events that are sent towards the transport.
- A default event type for events that are sent towards the correlator (host).

See also [“Host plug-ins and configuration” on page 30](#) and [“Translating EPL events using the `apama.eventMap` host plug-in” on page 32](#).

Using codecs

You can add one or more codec plug-ins to each chain which is defined in a YAML configuration file.

For example, you can add or configure the Classifier codec to have rules that determine which Apama event types to use for each message that comes from the transport and is sent towards the correlator (host). Alternatively you can use the Mapper codec for more advanced cases, or for simple cases where there is only one incoming event type for which you can set a `defaultEventType` in the `apama.eventMap`.

Or you can add or configure the Mapper codec rules to have rules that customize the mapping between the fields in your Apama event definitions and the payload and metadata of the transport messages, and that set default values in case any fields are missing. In some cases, you may wish to write rules to get or set the `metadata.sag.type` which specifies the Apama event type, or the `metadata.sag.channel` which specifies the correlator (EPL) channel name.

You can add, remove or configure any other standard codecs you wish to use, such as the JSON codec. For more information, see [“Codec Connectivity Plug-ins” on page 209](#).

Writing EPL

After you have edited the properties file for the connectivity plug-in (and maybe also the YAML configuration file), you have to write some EPL to cover the following main steps:

- Define the Apama event types for the messages you wish to send or receive.

- Use `monitor.subscribe` to subscribe to the correlator channel(s) from which you wish to receive messages from the transport. Add an event listener for these events, perhaps logging the incoming events to check that everything is working.
- Send events to any correlator channels to which the connectivity chain is subscribed. Keep in mind that the channel names depend on the transport and how it is configured.
- Call `ConnectivityPlugins.onApplicationInitialized` once your EPL is ready to receive incoming messages.

For simple applications, this can be done in the `onload()` action.

For real applications, we recommend the following:

1. Define an event to indicate when the application is fully injected.
 2. Send that event by providing an event (`.evt`) file, which is always sent by default after all EPL has been injected.
 3. Call `ConnectivityPlugins.onApplicationInitialized` once that event has been received.
- If your transport does not have a dynamic chain manager and you wish to create chains dynamically from EPL (rather than statically in YAML), you also have to create those chains using `com.softwareag.connectivity.ConnectivityPlugins.createDynamicChain`.

Note:

If you use the `genericsendreceive` sample as recommended in “[Adding the connectivity bundles](#)” on page 19, all required EPL code is already available in the `SendReceiveSample.mon` file.

2 Using Connectivity Plug-ins

■ Overview of using connectivity plug-ins	24
■ Static and dynamic connectivity chains	26
■ Configuration file for connectivity plug-ins	26
■ Host plug-ins and configuration	30
■ Translating EPL events using the <code>apama.eventMap</code> host plug-in	32
■ Using reliable transports	33
■ Creating dynamic chains from EPL	37
■ Sending and receiving events with connectivity plug-ins	38
■ Deploying plug-in libraries	40

Connectivity plug-ins perform a similar role to IAF adapters: both allow plug-ins to transform and handle delivery of events. In most cases, we recommend using connectivity plug-ins instead of the IAF for new adapters. See also "How Apama integrates with external data sources" in *Introduction to Apama*, which gives the reasons for using connectivity plug-ins.

The `samples/connectivity_plugin/application/genericsendreceive` directory of your Apama installation includes a simple sample which provides an easy way to get started with sending and receiving messages to or from any connectivity plug-in. For more information, see the `README.txt` file in the above directory and ["Sending and receiving events with connectivity plug-ins" on page 38](#).

Overview of using connectivity plug-ins

Connectivity plug-ins can be written in Java or C++, and run inside the correlator process to allow messages to be sent and received to/from external systems. Individual plug-ins are combined together to form *chains* that define the path of a message, with the correlator host process at one end and an external system or library at the other, and with an optional sequence of message mapping transformations between them.

You can configure connectivity plug-ins and also develop applications that use them with Software AG Designer. To do so, you have to add an instance of the **User Connectivity** connectivity bundle to your project. See "Adding adapters to projects" in *Using Apama with Software AG Designer* or "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications* for more information.

A configuration file describes both the chains and the plug-ins making up each chain. The configuration file is written using the YAML markup language, and can express structured configuration (maps, lists and simple values) for plug-ins. The default text encoding of the configuration file is UTF-8.

An example configuration may look like the following:

```
connectivityPlugins:
  stringCodec:
    libraryName: connectivity-string-codec
    class: StringCodec
  mapperCodec:
    libraryName: MapperCodec
    class: MapperCodec
  jsonCodec:
    libraryName: connectivity-json-codec
    class: JSONCodec
  HTTPClientTransport:
    libraryName: connectivity-http-client
    class: HTTPClient
startChains:
  weatherService:
    - apama.eventMap:
        defaultEventType: com.apamax.Weather
    - mapperCodec:
        "*":
          towardsTransport:
            mapFrom:
              - metadata.requestId: payload.id
```



```

    defaultValue:
      - metadata.http.path: /data/2.5/weather?q=Cambridge,uk
      - metadata.http.method: GET
  - jsonCodec
  - stringCodec
  - HTTPClientTransport:
    host: api.openweathermap.org

```

A chain is a combination of plug-ins with configuration. Every chain consists of the following:

- **Codec plug-in.** Optionally, one or more codec plug-ins are responsible for applying transformations to the messages (for example, the JSON codec in the above example) to prepare them for the next plug-in in the chain.
- **Transport plug-in.** One transport plug-in is responsible for sending/receiving messages to/from an external system (for example, HTTPClientTransport in the above example).
- **Host plug-in.** One built-in host plug-in is responsible for sending/receiving messages to/from the correlator process that is hosting the chain. These are built-in plug-ins (which do not need to be specified in the `connectivityPlugins` stanza) which the correlator supports. Host plug-ins determine in which format events are passed in and out of the correlator. Thus, a chain should specify a host plug-in that is compatible with the next codec or transport element in the chain. Host plug-ins can also specify on which channel the chain receives events from the correlator, and can specify a default channel to send events in to the correlator (for example, `apama.eventMap` in the above example).

Each transport plug-in and codec plug-in used in the chain must also be described in the `connectivityPlugins` stanza. All of the plug-ins in a chain can optionally take configuration that is specified in the configuration file nested below them.

Plug-ins can pass messages in a number of different forms (strings, maps, plug-in-specific objects). Codecs can be used to translate from one form into another. For example, the JSON codec in the above example would convert the map objects from the host plug-in to strings in JSON format. Transport plug-ins and codec plug-ins written in Java and C++ may be used together in the same chain regardless of language, using strings or maps of values to represent messages passed across the language boundary.

Plug-in chains support sending events in both directions, to and from the external system:

- An Apama application can send events *to* a connectivity chain in the same way as it would send them to any other receiver connected to the correlator, that is, using the `send` or `emit` keywords. Events from the EPL application are translated into the form specified by the host plug-in (the first in the chain configuration). They are then passed through each codec in turn, and then delivered to the transport. The host plug-in (`apama.eventMap` in the above example) by default listens for events from the application using the chain's name as a channel name. The host plug-in can be configured to listen on a specific set of channels with the `subscribeChannels` configuration property.
- Events *from* a connectivity chain's transport are passed through the codecs in the reverse order and are translated by the host plug-in to Apama events which are enqueued to the Apama application on the desired channel. The channel can be specified per event, or a default channel can be configured in the host plug-in using the `defaultChannel` configuration property.

See [“Host plug-ins and configuration” on page 30](#) for more information on the above mentioned configuration properties.

Static and dynamic connectivity chains

Chains can be created statically using a YAML configuration file or dynamically at runtime. How the chains are created depends on the type of transport:

- Some transports have a dynamic chain manager which manages chain creation in a transport-specific way. New chains are created, for example:
 - in response to external requests (for example, for each connection made to the HTTP server connectivity plug-in), or
 - when an Apama channel with a particular prefix is first used from EPL (for example, the Universal Messaging connectivity plug-in creates a chain by default when a channel beginning with “um:” is used from EPL, in a `monitor.subscribe(...)` method or a `send ... to` statement).

These transports always have a `dynamicChainManagers` section in their YAML configuration file. The connectivity chains are created dynamically by a transport chain manager plug-in, using chain definitions specified in `dynamicChains`. See [“Configuration file for connectivity plug-ins” on page 26](#) for information on how to configure dynamic chain managers and dynamic chains in a YAML configuration file.

For more information on dynamic chain managers, see [“Requirements of a transport chain manager plug-in class” on page 47](#).

- For transports that do not provide a dynamic chain manager, chains are created either
 - statically using the `startChains` section of the YAML file, or
 - dynamically from EPL using `ConnectivityPlugins.createDynamicChain`. See [“Creating dynamic chains from EPL” on page 37](#).

A transport that permits user-controlled chain creation never has a `dynamicChainManagers` section in its YAML configuration file.

See the documentation for each transport in on how chains are created. See [“Configuration file for connectivity plug-ins” on page 26](#) for more details on `startChains`, `dynamicChains` and `dynamicChainManagers`.

Configuration file for connectivity plug-ins

A configuration file for the connectivity plug-ins is specified using the `--config` option when starting the correlator with the correlator executable. It is possible to specify multiple configuration files. See the description of the `--config` option in “Starting the correlator” in *Deploying and Managing Apama Applications*.

A configuration file for the connectivity plug-ins is written in YAML. See also “Using YAML configuration files” in *Deploying and Managing Apama Applications*.

A configuration file should contain a map at the top level which has keys for `connectivityPlugins` and some of `startChains`, `dynamicChains` and `dynamicChainManagers`.

- The value of `connectivityPlugins` is a map which specifies how each plug-in is to be loaded. The keys name the plug-ins, and the values specify how the host loads the plug-ins. Plug-ins may be written in:

- **Java.** In this case, a `class` key and a `classpath` key should exist.

The `class` is name of the plug-in class, which must include the package. The plug-in's documentation specifies the class name to be used. The class is a transport, a codec or dynamic transport chain manager. See also [“Requirements of a plug-in class” on page 43](#).

The `classpath` can be either a single string or a list of strings. For example:

- **Single string:**

```
classpath: one.jar
```

- **List of strings** where each string is written on a new line with a preceding dash and space:

```
classpath:
- one.jar
- two.jar
- three.jar
```

- **List of strings** where the strings are delimited by semicolons (;). For example:

```
classpath: one.jar;two.jar;three.jar
```

Each string can name an absolute or relative jar file or directory.

An optional `directory` key specifies a directory which is where the jar files will be found (unless an absolute path is specified for a `classpath` element).

- **C++.** In this case, a `class` key and a `libraryName` key should exist.

The `class` is the base name of the class, without a package. The plug-in's documentation specifies the class name to be used. The class is a transport, a codec or dynamic transport chain manager. See also [“Requirements of a plug-in class” on page 43](#).

The `libraryName` is the base filename name of the library, excluding the operating system-specific prefixes and suffixes (that is, excluding the “lib” prefix and “.so” suffix for UNIX, and the “.dll” suffix for Windows).

An optional `directory` key specifies a directory which is where the library will be found (see [“Deploying plug-in libraries” on page 40](#)).

`globalConfig` is an optional map providing default configuration options for this plug-in, which are used by all chains/chain definitions using this plug-in that do not provide their own value for them. The `globalConfig` configuration can be overridden by configuration per chain.

- **Chains** under `startChains` are created at startup. The value of `startChains` is a map where each key is a string that names a chain. Each value should be a list, naming the plug-ins that

make up the chain. Each chain must contain one host plug-in, which is one of the built-in supported host plug-ins, optionally followed by a number of codecs, and end with a plug-in that is the transport. Configuration can optionally be specified for each plug-in, by following the plug-in name with a colon and space, and providing the configuration below it. Note that in YAML terms, the chain entry is a map rather than a string.

- The `dynamicChains` map is used to provide chain *definitions* that are used by chain manager plug-ins or EPL code that dynamically create chain instances after the correlator has started. Each key in the map is a chain definition identifier, which is the string that will be used by the chain manager or from EPL to identify what kind of chain it wants to create. Each value should be a list, naming the plug-ins that make up the chain, similar to what you would specify in `startChains`. Each chain must contain one host plug-in, which is one of the built-in supported host plug-ins, optionally followed by a number of codecs, and end with a plug-in that is the transport. Configuration can optionally be specified for each plug-in, by following the plug-in name with a colon and space, and providing the configuration below it as a map value. One difference between `dynamicChains` and `startChains` is that the plug-in configurations used in `dynamicChains` can specify `@{varname}` variable placeholders which get replaced when a chain instance is created from the chain definition, with values provided dynamically by the chain manager plug-in or the EPL `createDynamicChain` call. If you are using a chain manager plug-in, see the plug-in's documentation for information about any `@{varname}` substitutions that it supports. Note that this is unrelated to the `${varname}` replacements that are performed statically when YAML files are loaded at startup.
- The value of `dynamicChainManagers` is a map where each key is a manager name, that is, a string naming an instance of a dynamic chain manager plug-in class. Each value is a map providing the configuration for the chain manager instance (such as details for connecting to a specific external system) and the name of the transport plug-in it is associated with. The manager should have the following keys:
 - `transport`: Specifies the transport plug-in associated with this dynamic chain manager. This must match the key used in `connectivityPlugins` to load this chain manager, and also the name used in the `dynamicChains` definition to identify the transport plug-in at the end of each chain. This is the name used for the plug-in in the configuration file, not the name of the class that implements the plug-in.
 - `managerConfig`: Specifies the configuration map that will be passed to the chain manager constructor when it is created at startup. The available configuration options are defined by the plug-in author, therefore, see the plug-in's documentation for details. If the `managerConfig` is invalid and the chain manager throws an exception, the correlator logs an error message and fails to start. The `managerConfig` usually includes details for connecting to a specific external server or system. Some chain managers may also provide some options that are set in the transport plug-in's configuration section under `dynamicChains`, for example, options specific to the protocol or message format described by that chain definition.

Note that there can be more than one manager instance configured for a given transport, for example, if you need to connect to several different servers of the same type. Each manager can make use of more than one chain definition, for example, if different message formats (such as XML and JSON) are being used with the same server or chain manager. In simple configurations where a transport only ever had a single manager instance and a single chain definition, it is common to use the same string for the transport name, dynamic chain definition identifier and manager name. However, there is no requirement for them to be the same.

You can use `.properties` files to specify values for `${varname}` substitution variables in configuration files. See "Using properties files" in *Deploying and Managing Apama Applications* for further information.

There are a few values which can be written into the configuration file which will be substituted at runtime. This is to aid portability of configuration files between different deployments. Specifically the following variables may be used:

Variable	Description
<code>\${PARENT_DIR}</code>	The absolute normalized path of the directory containing the properties file or YAML file currently being processed.
<code>\${APAMA_HOME}</code>	The path to the Apama installation.
<code>\${APAMA_WORK}</code>	The path to the Apama work directory.
<code>\${\$}</code>	The literal \$ sign.

Example:

```
connectivityPlugins:
  myTransport:
    directory: ${APAMA_WORK}/build
    classpath:
      - myTransport.jar
    class: org.my.Transport
startChains:
  service:
    - apama.eventMap
    - myTransport:
        apamaInstall: ${APAMA_HOME}
```

The following example shows a more complex configuration for a transport plug-in that uses two dynamic chain manager instances and also two different chain definitions:

```
connectivityPlugins:
  # a transport plug-in that uses a chain manager to instantiate
  # its chains and transport instances dynamically
  myTransport:
    directory: ${APAMA_WORK}/build
    classpath:
      - myTransport.jar
    class: org.my.MyTransportChainManager

  # ... codecs defined here too

dynamicChainManagers:
  # example of multiple chain managers for the same transport;
  # an instance is created during correlator startup for each
  # manager listed here
  myTransportManager1:
    # must match transport plug-in name specified under connectivityPlugins
    transport: myTransport

    managerConfig:
      myManagerConfigOption: ${myTransport.foo}
```

```
myTransportManager2:
  # must match transport plug-in name specified under connectivityPlugins
  transport: myTransport

  managerConfig:
    # some managers specify which chain definition
    # to use in their configuration (others decide it at runtime)
    myStaticallyConfiguredChainId: myJSONChainDefinition

dynamicChains:
  myJSONChainDefinition:
    - apama.eventMap
    - jsonCodec
    # must match transport plug-in name specified under connectivityPlugins
    - myTransport:
        myTransportChainDefOption1: @{{bar}}
        myTransportChainDefOption2: ${{myTransport.baz}}

  myXMLChainDefinition:
    - apama.eventMap
    - myXMLCodec
    - myTransport
```

To see a fully working example of using a dynamic chain manager plug-in, try adding the Universal Messaging connectivity plug-in to your project (see "Adding the Universal Messaging connectivity plug-in to a project" in *Using Apama with Software AG Designer*).

In a configuration file, you can also specify the following:

- Additional YAML configuration files that are to be processed. For more details, see "Including YAML configuration files inside another YAML configuration file" in *Deploying and Managing Apama Applications*.
- JVM options which the correlator is to pass to the embedded JVM. For more details, see "Specifying JVM options" in *Deploying and Managing Apama Applications*.

Host plug-ins and configuration

The first element in the configuration of a chain must be a host plug-in. This is a special type of plug-in that controls how the correlator interacts with the chain. The type of plug-in will determine in which form events are passed to and accepted from the chain. The host plug-in must use a compatible type with the first codec (or, if no codecs specified, the transport), otherwise errors will be reported and events will not be delivered.

Overview of host plug-ins

The following host plug-ins are supported:

- `apama.eventMap`

The `eventMap` plug-in translates EPL events to and from nested maps, which allows chains to convert arbitrary structured data into forms that can be automatically translated into EPL events without having to know the exact definition of the EPL event, provided the field names

of the event definition match the keys in the map. See [“Translating EPL events using the apama.eventMap host plug-in” on page 32](#) for further information.

■ `apama.eventString`

The `eventString` plug-in transfers events in Apama string event format, as used by the `engine_send` and `engine_receive` tools and the Apama client library.

Common configuration properties

All Apama host plug-ins take the following configuration properties:

Configuration Property	Description
<code>subscribeChannels</code>	<p>Optional. Defines the channel or channels to which the chain subscribes in order to receive events from the correlator.</p> <p><code>subscribeChannels</code> is only recommended for chains defined as <code>startChains</code>. Dynamic chains will have their channel subscription specified either from EPL or by the chain manager which will override any setting in the configuration file.</p> <p>To send an event to a chain from EPL, use the EPL <code>send</code> statement with the name of a channel to which the chain has subscribed.</p> <p>Type of configuration: string or list of strings.</p> <p>Default: the chain name.</p>
<code>defaultChannel</code>	<p>Optional. Defines the default channel to deliver events from the transport to the correlator from the chain.</p> <p>Some chain managers provide a value for <code>defaultChannel</code>. It is not permitted to specify <code>defaultChannel</code> for chains used by such chain managers. It is recommended only for chains defined as <code>startChains</code> or which you intend to create from EPL.</p> <p>Chains can also specify a channel name in the metadata of each message. A channel provided in metadata takes precedence over this configuration value.</p> <p>Type of configuration: string.</p> <p>Default: the default channel is an empty string (""), thus delivering the event to all public contexts.</p>
<code>suppressLoopback</code>	<p>Optional. If set to <code>true</code>, any events which will be received by this chain will not also be sent to contexts subscribed to the same channel. It is assumed that this receiver may send the events back into the correlator to be received by subscribed</p>

Configuration Property	Description
	contexts. This is typically used when this chain is connected to a message bus. Any other external receivers are unaffected. Type of configuration: boolean. Default: false.
description	Optional. A textual description which will appear in the Management interface (see also "Using the Management interface" in <i>Developing Apama Applications</i>).
remoteAddress	Optional. A textual address for the remote component (if any) to which this chain connects. This address will also appear in the Management interface.

Translating EPL events using the `apama.eventMap` host plug-in

The `eventMap` plug-in translates events to or from map objects, reflecting the structure of the event. Each map entry has a key which is the same as an EPL event field. The values of the map can be simple values (strings, numbers) or further maps or lists which correspond to dictionaries, nested events or sequences.

If the message's payload does not contain all EPL fields, then by default the message is dropped and a warning is logged. The `allowMissing` configuration property can be set to `true`, in which case missing fields or fields with empty values are set to their default values.

If the message's payload has fields that do not have corresponding EPL fields (or which are perhaps optional), then the map entries are ignored by default. An event definition can specify a `com.softwareag.connectivity.ExtraFieldsDict` annotation that names a dictionary field; extra values are placed in the dictionary (see [“Map contents used by the `apama.eventMap` host plug-in” on page 54](#) for more information). If needed, this can be disabled by setting the `extraFields` configuration property to `false`. The dictionary must be one of the following types:

- `dictionary<string,string>` - Keys and values are coerced into strings. Lists generate the string form of `sequence<string>`. Maps generate the string form of `dictionary<string,string>`.
- `dictionary<any,any>` - Values are mapped to the corresponding EPL type, or `sequence<any>` for lists and `dictionary<any,any>` for maps without names.
- `dictionary<string,any>` - Keys are coerced into strings.

When events are sent from a chain to the correlator, the correlator needs to know what event type they are. This can be set by a chain plug-in (in the metadata of a message) or by setting the `defaultEventType` configuration property. The metadata will take precedence to specify a message's type. Some chains will set the event type on every message, so the default event type does not need to be set in the configuration. Other chains may not be aware of event types, so the event type must be set.

Configuration Property	Description
<code>defaultEventType</code>	<p>Optional. The name of the EPL type to which events that are going into the correlator are converted if no event type is specified on a message.</p> <p>Type of configuration: string.</p> <p>Default: none - requires that the chain send messages with the event type set.</p>
<code>allowMissing</code>	<p>Optional. Defines whether missing fields or fields with empty values (<code>null</code> values in Java) are permitted on inbound events. If they are permitted, they are set to the EPL default for that type. Similarly, empty values in nested events, elements in sequences and key/value pairs in dictionaries are also set to their default values.</p> <p>There is an exception: an empty value that maps to an <code>optional<type></code> or <code>any</code> in EPL is permitted even if <code>allowMissing</code> is false. See also the descriptions of the <code>optional</code> and <code>any</code> types in the <i>API Reference for EPL (ApamaDoc)</i>.</p> <p>Type of configuration: boolean.</p> <p>Default: false - this results in a <code>WARN</code>, and the events are dropped if there are any missing or empty fields.</p>
<code>extraFields</code>	<p>Optional. Defines whether to place map keys that do not name fields in an <code>extraFields</code> dictionary member that is identified with the <code>@ExtraFieldsDict</code> annotation (see above).</p> <p>Type of configuration: boolean.</p> <p>Default: true.</p>

Using reliable transports

Reliable messaging gives you the tools to write event-processing applications that are resilient against message loss (for example, due to crashes or network outages).

To make use of reliable messaging in your connectivity plug-ins, you must:

- Configure the transports for reliable messaging. The nature of this configuration is transport-specific. Reliable messaging is only supported in chains that use the `apama.eventMap` host plug-in. See also [“Translating EPL events using the apama.eventMap host plug-in” on page 32](#).
- Write EPL to send and receive events via these transports and to handle acknowledgments. For detailed information, see the event descriptions for `Chain`, `AckRequired` and `FlushAck` in

the `com.softwareag.connectivity` and `com.softwareag.connectivity.control` packages in the *API Reference for EPL (ApamaDoc)*.

Note:

Not all transports support reliable messaging.

Reliable-messaging-aware transports only support at-least-once delivery, which admits the possibility of duplicate messages, especially after recovery from downtime. Your applications should be written to handle this.

In this version, reliable messaging with connectivity plug-ins is controlled exclusively from EPL. At this time, reliable messaging cannot be automatically tied-in to correlator persistence.

If a license file cannot be found, reliable messaging with connectivity plug-ins is disabled. See "Running Apama without a license file" in *Introduction to Apama*.

This transport connectivity plug-in	supports reliable messaging
Universal Messaging	No
MQTT	No
Digital Event Services	Yes
HTTP server	No
HTTP client	No
Kafka	No
Cumulocity IoT	No

Message identifiers

Messages going from the transport to the host contain unique message identifiers. Each identifier is stored as `sag.messageId` in the metadata. See [“Metadata values” on page 58](#). You only need access to the message identifiers if you want to acknowledge individual events.

Where a message maps to an event type that has the `MessageId` annotation, the message identifier in the metadata is copied into a field on that event. You should not name a field that you expect to have a real value. See "Adding predefined annotations" in *Developing Apama Applications*.

The following EPL example shows how to use the `MessageId` annotation:

```
using com.softwareag.connectivity.MessageId;

@MessageId("messageIdentifier")
event MyEvent {
    string s;
    integer i;
    string messageIdentifier; // Contains the sag.messageId from the
                             // message that mapped to this event
}
```

Chains

In general, when receiving or sending reliably, you need to know which connectivity chain is receiving (from transport to host) or sending (from host to transport) the events. To identify that connectivity chain, you use the EPL Chain event, which provides a wrapper with helpful actions pertaining to reliability. There are two actions on the `ConnectivityPlugins` event that can be used to get the Chain event for the chain you want:

- `ConnectivityPlugins.getChainByChannel`

This action looks up a chain instance by a channel it is subscribed to or sending to.

- `ConnectivityPlugins.getChainById`

This action looks up a chain instance by its identifier.

See the *API Reference for EPL (ApamaDoc)* for more information on these actions.

Reliable receiving

A transport can be configured for reliable receiving. This means the events are going from the outside world into the correlator, and you make sure that they are not lost in the case of a failure.

The EPL that receives the events is obliged to acknowledge when the events have been *fully processed* by the application. That is because the remote system to which a reliable transport connects typically keeps track of what messages have been acknowledged and what messages have not been acknowledged. In the event of a failure, any messages that have not been acknowledged are resent to you after reconnection/restart.

Keep in mind that “fully processed” is different from just receiving an event. It means that you have preserved the *effect* of that event, and done so safely enough that you will no longer need the event to be resent in the event of a failure. As an example, that might mean committing the contents of the event to a database, writing it to a file, or having sent an output event and received an acknowledgment for it.

There are often performance implications for an application that is late with acknowledgments. You should therefore acknowledge all events as soon as possible after receiving. There is no guarantee, however, that an acknowledgment will be processed immediately. For example, if you acknowledge some events in EPL and then the system goes down quite soon afterwards, the events may not have been fully acknowledged to the remote system and will therefore get redelivered.

Once the events have been fully processed by the EPL application, they can be acknowledged in either of the following ways:

- Listen for `AckRequired` events from the `com.softwareag.connectivity.control` package, and call the `ackUpTo` action on them. Doing it this way means you are acknowledging potentially large batches of events at a time.
- Use the `ackUpTo` action on the `Chain` event type to acknowledge all previously received events, up to and including a specific event of your choice. In this case, you identify the specific event with its message identifier. If the event definition has the `MessageId` annotation, you can obtain the message identifier from the named field.

The detailed technical reasons for choosing between the above mechanisms are given in the descriptions of the Chain and AckRequired events in the *API Reference for EPL (ApamaDoc)*.

The following EPL example shows how an application can write reliably-received events to a file. It uses a fictional plug-in named `filePlugin` for this purpose.

```
using com.softwareag.connectivity.ConnectivityPlugins;
using com.softwareag.connectivity.Direction;
using com.softwareag.connectivity.Chain;
using com.softwareag.connectivity.control.AckRequired;
...

monitor.subscribe("incomingEvents"); // The chain is sending us events
                                     // on this channel
Chain chn := ConnectivityPlugins.getChainByChannel("incomingEvents",
    Direction.TOWARDS_HOST); // Get the chain itself
on all MyEvent() as e { // The events the application is interested in
    evtSequence.append(e);
}

on all AckRequired(chainId=chn.getId()) as ackRequired {
    // Periodically acknowledge all previously received events,
    // but only after safely writing their contents to a file
    filePlugin.writeAndSync(evtSequence.toString());
    evtSequence.clear();
    ackRequired.ackUpTo();
}
```

In all reliable receiving, you have to consider the possibility that *some* events that have already been acknowledged might be resent to your application, especially when recovering after a failure. Your application should be written to either eliminate duplicates or tolerate them.

Any towards-host messages that get dropped by the chain due to errors (for example, when a codec cannot translate from one form to another due to an invalid format) are treated as if they have already been acknowledged.

Reliable sending

Reliable sending is symmetrical with reliable receiving for most purposes. With reliable sending, your EPL application can ask to be notified when a remote system has safely processed your events. As before, you have to know what chain is being used for reliable sending of these events, and so you have to get the relevant Chain instance. After sending some events, you may call the `flush()` action of the Chain.

Once all events sent to the chain before this call have been safely stored on (or have been processed by) the remote system, your application will see an acknowledgment in the form of a `FlushAck` event. Your application might then respond to this, for example, by acknowledging reliably-received events that caused these events to be sent, or by recording the fact that the event has been sent in a way that means that the application will not send it again. In the event of a restart, your application should be written such that it is able to resend any events that were sent but not acknowledged in its previous incarnation.

The following is an example of EPL application using the `flush()` action:

```
using com.softwareag.connectivity.ConnectivityPlugins;
```

```

using com.softwareag.connectivity.Direction;
using com.softwareag.connectivity.Chain;
using com.softwareag.connectivity.control.FlushAck;
...

// Get the chain for the channel we are sending events to
Chain chn := ConnectivityPlugins.getChainByChannel("chanWeSendTo",
    Direction.TOWARDS_TRANSPORT);

on all wait(0.1) {
    globalInteger := globalInteger + 1;
    MyEvent e := MyEvent(globalInteger);
    send e to "chanWeSendTo";

    // All previously sent events have now been safely processed by
    // the remote system
    on FlushAck(requestId=chn.flush()) {
        log "Fully sent " + e.toString() at INFO;
    }
}

```

Your application should be written with the idea that it might send duplicate events in the case of a problem (for example, if your application sends out some events which are then processed by some remote system, but there is a crash before your application can see the `FlushAck` event).

Creating dynamic chains from EPL

You can define chains under the `dynamicChains` section of the configuration file which are not tied to a specific `dynamicChainManager`. These chains are not created on startup. Instead, multiple instances of these chains can be created on demand from EPL using the `ConnectivityPlugins` EPL API. There is a static method on the `com.softwareag.connectivity.ConnectivityPlugins` event type:

```

createDynamicChain(string chainInstanceId, sequence<string> channels, string
chainDefnName, dictionary<string, string> substitutions, string defaultChannelTowardsHost)
returns Chain;

```

Calling this method creates and starts a new instance of a chain defined under `dynamicChains` and returns a `com.softwareag.connectivity.Chain` object which can later be used to destroy the chain via the `destroy()` method.

The arguments to `createDynamicChain` are:

Argument	Description
<i>chainInstanceId</i>	The identifier to use for the new chain instance. This identifier is used for logging, and it must be unique.
<i>channels</i>	A sequence of channels to which this chain is to be subscribed in the correlator. Events sent to these channels from EPL are delivered to this chain.

Argument	Description
<i>chainDefnName</i>	The name of a chain defined under <code>dynamicChains</code> in the configuration file. This must not be for a transport which has a <code>dynamicChainManager</code> .
<i>substitutions</i>	A map of key-value pairs to be substituted into the chain definition using <code>@{key}</code> syntax.
<i>defaultChannelTowardsHost</i>	The default channel to use for sending a message towards the host if no channel is specified on the message. Specify an empty string if you do not want to use a default channel.

Note:

You must not specify a non-empty value for `defaultChannelTowardsHost` if the configuration property `defaultChannel` is also specified for the host plug-in (see also [“Host plug-ins and configuration” on page 30](#)). An error will occur in this case.

At the point when `createDynamicChain` returns, the chain is created and able to receive events. The chain exists until either the correlator is shut down or the `.destroy()` method is called on the `Chain` object returned by `createDynamicChain`.

Sending and receiving events with connectivity plug-ins

When the correlator starts up, any connectivity chains listed in the configuration file are loaded and started. At this point, events may be sent from EPL to the chains, through all of the codecs towards the transport.

onApplicationInitialized

While the transport is able to send events towards the host (the correlator), the correlator does not process those events immediately. This prevents problems with events that are sent from the transport to the correlator before the correlator has had event definitions injected, or the EPL to handle those events has been injected or is ready to process the events. Instead, these events are queued in the correlator.

An EPL application that sends or receives events to a transport should call the `onApplicationInitialized` method on the `com.softwareag.connectivity.ConnectivityPlugins` EPL object. This notifies the correlator that the application is ready to process events. Any events that the transport sends towards the host (correlator) before this is called are then delivered to the correlator. Events from a transport are maintained in the correct order.

Calling `onApplicationInitialized` notifies all codecs and transports that the host is ready by calling the `hostReady` method. The transport may choose not to receive events (for example, from a JMS topic) until the application is ready if doing so may have adverse effects.

Initialization:

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.

Software AG Designer, engine_deploy and other tools all ensure that .evt files are sent in after all EPL has been injected.

2. The monitor that handles the application-defined start event (from step 1) should use this event object to notify the correlator that the application is initialized and ready to receive messages, for example:

```
on com.mycompany.myapp.Start() {
    com.softwareag.connectivity.ConnectivityPlugins.onApplicationInitialized();
    // Any other post-injection startup logic goes here too.
}
```

Note:

For simple applications, you can add the EPL bundle **Automatic onApplicationInitialized** to your project (see also "Adding bundles to projects" in *Using Apama with Software AG Designer*). This bundle will ensure that `onApplicationInitialized` is called as soon as the entire application has been injected into the correlator. However, in cases where you need to wait for a `MemoryStore`, database or another resource to be prepared before your application is able to begin to process incoming messages, you should not use the bundle. In these cases, you should write your own start event and application logic.

To aid diagnosing problems when part of the system is not ready in a timely manner, the correlator logs this on every `Status: log` line (by default, every 5 seconds). For example:

```
Application has not called onApplicationInitialized yet - 500 events from connectivity transports will not be processed yet
```

If EPL has not yet called `onApplicationInitialized` or if a plug-in in `myChain` has not returned from `hostReady` yet, the following is logged:

```
Chain myChain is handling hostReady call
```

Calling `ConnectivityPlugins.onApplicationInitialized` also notifies the correlator-integrated messaging for JMS, if enabled, that it is ready to receive events. It will then implicitly perform the `JMS.onApplicationInitialized()` call (see [“Using EPL to send and receive JMS messages” on page 245](#)). You should only call `ConnectivityPlugins.onApplicationInitialized()` once the application is ready to receive all incoming events, either from connectivity chains or JMS.

Diagnostic codec

You can use the Diagnostic codec to view the messages being sent at any point in the connectivity chain. This is very useful for diagnosing problems, and for configuring message transformations using codecs such as the Mapper and Classifier codecs. For more information, see [“The Diagnostic codec connectivity plug-in” on page 228](#) for further information.

Simple sample

The `samples/connectivity_plugin/application/genericsendreceive` directory of your Apama installation includes a simple sample which provides an easy way to get started with sending and receiving messages to or from any connectivity plug-in.

The sample contains some simple event definitions called `mypackage.MySampleInput` and `mypackage.MySampleOutput` that can be used for input and output messages. These event definitions can be customized with additional fields as you wish.

In addition, the sample contains a monitor that subscribes to a specific transport channel and logs all events received from it, and also sends test events to a specific transport channel.

To use the sample, simply copy it into your Apama work directory or import it into Software AG Designer as an existing project. Then customize the channel (or channels) it is sending to/from to match the channel naming scheme specified in the documentation or configuration of the transport you are using (for example, `um:MyChannelName` for Universal Messaging). Finally, add and configure the connectivity plug-in you wish to use (see also). Depending on the transport and chain configuration you are using, you may also need to configure Mapper and/or Classifier codec rules in the YAML file (see also [“Codec Connectivity Plug-ins” on page 209](#)) to use the `mypackage.MySampleInput` and `mypackage.MySampleOutput` event definitions used by the sample.

See the `README.txt` file in the `genericsendreceive` sample directory for more details.

Deploying plug-in libraries

Every Java plug-in is loaded in a separate class loader. Static variables cannot be shared between different plug-ins, unless a class is placed on the system's classpath. This allows different plug-ins to use different versions of Java libraries without interference.

For C++ plug-ins, note that while a library can be loaded from any directory by specifying a `directory` key in the `connectivityPlugins` section of the configuration file (see [“Configuration file for connectivity plug-ins” on page 26](#)), any libraries that the plug-in depends on will only be loaded from the system library path (using the `LD_LIBRARY_PATH` environment variable and standard locations on UNIX, or using the `PATH` on Windows). The Apama Command Prompt (see “Setting up the environment using the Apama Command Prompt” in *Deploying and Managing Apama Applications*) adds `$APAMA_WORK/lib` to the system library path, which is the recommended place to put any libraries that plug-ins require. Any libraries that the plug-in loads will be shared across the entire process, therefore different plug-ins will need to use the same version of third-party libraries where applicable. On Windows, ensure that the class and package names of plug-ins are different across plug-ins.

In general, C++ plug-ins will give better performance than Java libraries, as there is a cost in converting objects between Java and the C++ types. In particular, avoid mixing many interleaved Java and C++ plug-ins in the same chain. If possible, put the C++ plug-ins on the host side of the chain, as the correlator host plug-ins are C++ and adjacent plug-ins of the same type are cheaper than conversions. However, the language bindings of any libraries required by a plug-in and familiarity with the programming environment should be the primary factors when deciding in which language to write a new plug-in.

3 Developing Connectivity Plug-ins

■ Chain components and messages	42
■ Requirements of a plug-in class	43
■ Requirements of a transport chain manager plug-in class	47
■ Building plug-ins	49
■ C++ data types	51
■ Map contents used by the apama.eventMap host plug-in	54
■ Metadata values	58
■ Lifetime of connectivity plug-ins	60
■ Creating dynamic chains from a chain manager plug-in	62
■ User-defined status reporting from connectivity plug-ins	63
■ Logging and configuration	65
■ Threading	66
■ Developing reliable transports	67
■ General notes for developing transports	70

Chain components and messages

Connectivity chains consist of zero or more codecs and one transport.

Codecs perform some transformation of events, processing events going to and from the correlator, and passing them on to the next component in the chain. Examples of codecs include:

- Translating events from structured form into a serialized form such as JSON or XML.
- Changing the name of a field to translate between different naming conventions.
- Removing unnecessary fields.
- Filtering, merging or splitting events.

Transports are the end of a chain. They may deliver the events to an external system that is not part of the chain and may be out of process, or may perform some operation and send the result to back to the correlator. Examples of transports include:

- Sending HTTP requests and receiving responses.
- Receiving events from a message bus.
- Performing operations on a file system.

Codecs and transports may be written in Java or C++, and chains can contain a mixture of C++ and Java plug-ins. Messages will be automatically converted between C++ and Java forms. The language bindings of any libraries required by a plug-in and familiarity with the programming environment should be the primary factors when deciding on the language in which to write a new plug-in. Note that conversions between C++ and Java forms are copies and there are overheads in performing these conversions. As the Apama host plug-ins are implemented in C++ (as is the core of the correlator), a chain consisting of only C++ plug-ins will perform better. In particular, avoid mixing many interleaved Java and C++ plug-ins in the same chain. If possible, put the C++ plug-ins on the host side of the chain. Where adjacent plug-ins in a chain are of the same type (C++/Java), messages are passed by reference/pointer (they are not copied).

Plug-ins communicate with each other and the correlator (also referred to as the host) by sending batches of messages. Messages are converted to/from the events within a correlator. When a chain sends a message to the host, the host plug-in converts it to an event and sends it into the correlator. When the correlator emits an event to a channel on which a chain is listening, then the host plug-in converts it from an event to a message, and delivers it to the chain. The plug-ins in a chain may do conversions, for example, a codec may convert a map to a single string (for example, a JSON codec), but that can be passed within a message. When it gets to a transport, it may be taken from the message and delivered in some other form (for example, an HTTP request). A message consists of a payload (which can be of different types according to the needs of the plug-in) and a metadata map of strings. For more information, see the *Message* class in the *API Reference for Java (Javadoc)* or *API Reference for C++ (Doxygen)*.

Metadata holds data about the event, such as the channel on which the event was delivered or the type of the event. Plug-ins can use metadata to pass extra information about events that are not part of the event (for example, latency through a chain could be measured by adding timestamps to metadata and comparing that with the time needed for processing an event).

The message payload can be null. This means that the message is not a real event. This can be useful for passing non-event information between plug-ins; many plug-ins will ignore this. For example, a request to terminate a connection could be sent from one codec to signal the transport to disconnect, and intermediate codecs that perform transformations such as JSON encoding would ignore the event.

Messages are passed in batches so that transports (and codecs) can take advantage of amortizing costs if operating at high throughput. Most plug-ins can be written by subclassing `AbstractSimpleCodec` or `AbstractSimpleTransport` classes (see [“Requirements of a plug-in class” on page 43](#)) and only need to process a single `Message` at a time. The delineation of messages into batches does not carry any significance beyond the events are all available at the same time. This is only an opportunity for optimizations, not for passing extra semantic information. Codecs are not required to maintain the same batches and can re-batch messages if desired.

Messages are not copied between plug-ins and do not perform any locking or synchronization. If a codec wants to keep hold of a pristine copy of a message, it should store a copy of the message.

Every chain will need to work with one of the supplied host plug-ins. Most chains will use the `apama.eventMap` plug-in which allows events to be sent without needing to know the exact event definition. See also [“Map contents used by the `apama.eventMap` host plug-in” on page 54](#).

Requirements of a plug-in class

Java and C++ plug-ins are identified in the `connectivityPlugins` section of the configuration file for the connectivity plug-ins. See [“Configuration file for connectivity plug-ins” on page 26](#) for detailed information.

The named class must be a descendent of either `AbstractCodec` or `AbstractTransport`, unless it is a transport with a dynamic chain manager in which case the class must subclass `AbstractChainManager` (see [“Requirements of a transport chain manager plug-in class” on page 47](#) for more information about developing chain managers).

In most cases, the easiest way to write codec and transport classes is by subclassing `AbstractSimpleCodec` or `AbstractSimpleTransport`. However, in some cases, a plug-in can achieve better performance by directly subclassing the base class `AbstractCodec` or `AbstractTransport`; these classes support handling a batch of multiple messages in a single call.

The classes are summarized in the following table. They are all in the `com.softwareag.connectivity` package (Java) or in the `com::softwareag::connectivity` namespace (C++). See the *API Reference for Java (Javadoc)* and *API Reference for C++ (Doxygen)* for more information.

Base class	Subclasses deal in	Minimum methods subclasses need to implement
<code>AbstractCodec</code>	Batches of messages	<code>sendBatchTowardsTransport</code> , <code>sendBatchTowardsHost</code>
<code>AbstractTransport</code>	Batches of messages	<code>sendBatchTowardsTransport</code>

Base class	Subclasses deal in	Minimum methods subclasses need to implement
AbstractSimpleCodec	Individual messages	transformMessageTowardsHost, transformMessageTowardsTransport Note that every message results in at most one message out.
AbstractSimpleTransport	Individual messages	deliverMessageTowardsTransport

All of the above classes also provide members or default implementations of:

- Member chainId
- Member config
- Member logger for logging (see below)
- Member hostSide - the next component in the chain towards the host
- Member transportSide - the next component in the chain towards the transport (for codecs only)
- start
- hostReady
- shutdown

The start, hostReady and shutdown methods can be overridden if required. See also [“Lifetime of connectivity plug-ins” on page 60](#).

Plug-in class constructor

Subclasses should provide a constructor like the following, for Java:

```
public <ClassName>(org.slf4j.Logger logger,  
    TransportConstructorParameters params) throws Exception,  
    IllegalArgumentException {  
    super(logger, params);  
    ...  
}
```

or for C++:

```
public:  
    <ClassName>(const TransportConstructorParameters &params)  
        : AbstractSimpleTransport(params)  
    {  
        ...  
    }
```

The constructors for codecs follow the same pattern as for transports. The `TransportConstructorParameters` or `CodecConstructorParameters` object provides access to the plug-in's configuration and other useful information and capabilities, some of which are also made available as member fields for convenience.

Note that transports with an associated dynamic chain manager are created by the chain manager's `createTransport` method (for Java) or must have a public constructor with signature `(const ManagedTransportConstructorParameters &, ...)` where `...` are any extra parameters passed in the `createChain` call (for C++).

In both Java and C++, there is a logger available using the `logger` field (for Java, this is also passed into the constructor). Note that all messages logged using the logger will include the `chainId` and `pluginName`, so there is no need to explicitly add that information to each message. See the *API Reference for Java (Javadoc)* and *API Reference for C++ (Doxygen)* for detailed information.

Using AbstractSimpleCodec and AbstractSimpleTransport

The `AbstractSimpleCodec` and `AbstractSimpleTransport` classes handle batches by iterating through each message within a batch and calling one of the methods listed above for each message. For Java codecs, the result of the `transform` method replaces that message in the batch. For C++ codecs, the `transform` method passes a reference to a message which can be mutated or the message discarded if the method returns `false`. By default, messages with a null payload are ignored by the `AbstractSimpleCodec` and `AbstractSimpleTransport` classes, but subclasses may override methods to handle them (see the *API Reference for Java (Javadoc)* and *API Reference for C++ (Doxygen)* for details).

Exceptions from processing one message are logged by default (this behavior can be overridden by implementing `handleException`) and the next message is then processed.

To deliver events to the correlator, transports call the `sendBatchTowardsHost` method on the `hostSide` member of `AbstractSimpleTransport`, passing a batch of messages as a `List<Message>` (they can use `Collections.singletonList()` if needed).

Using AbstractCodec and AbstractTransport

Chains are bidirectional, passing events to and from the correlator. The order of plug-ins within a chain is defined by the configuration file: first the host plug-in, then codecs, and finally a transport. Plug-ins are connected such that the `hostSide` and `transportSide` members of `AbstractCodec` point to the previous and next plug-in in the chain; and for `AbstractTransport`, `hostSide` points to the last codec (or the host plug-in if there are no codecs).

Events from the correlator are sent to the first codec (or transport if there are no codecs). Each codec should pass the message through to the next component, invoking the `sendBatchTowardsTransport` method on the `transportSide` member.

Events to the correlator originate from the transport and are delivered by invoking the `sendBatchTowardsHost` method on the `hostSide` member which delivers the events to the last codec. The last codec should invoke the `sendBatchTowardsHost` method on its `hostSide` object, thus traversing plug-ins in the reverse order. For Java, transports must always provide `hostSide` a batch of messages as a `List<Message>` (they can use `Collections.singletonList()` if needed). For C++ plug-ins, the batches are passed as a pair of start and end pointers to `Message`. The batch is defined

as starting from the message pointed to by `start` and up to just before the message pointed to by `end` - similar to `begin()` and `end()` iterators on C++ containers. Thus, the messages in a batch can be iterated with a loop such as:

```
for (Message *it = start; it != end; ++it) {
    handleMessage(*it);
}
```

Plug-ins are provided with a `HostSide` and (for codecs only) `TransportSide` interface view of the next component in the chain (as members of `AbstractTransport` or `AbstractCodec`).

Codecs are not required to maintain a one-to-one mapping of events going in and out. They may choose to discard or merge multiple messages or split a message into separate messages.

Exporting the required symbols from C++ plug-ins

C++ plug-ins also require a macro which exports the symbols that the correlator needs to create and manage the plug-in object. The macro has one of the following names:

- `SAG_DECLARE_CONNECTIVITY_TRANSPORT_CLASS(class-name)`

This macro should not be used for transports with a chain manager.

- `SAG_DECLARE_CONNECTIVITY_CODEC_CLASS(class-name)`

- `SAG_DECLARE_CONNECTIVITY_TRANSPORT_CHAIN_MANAGER_CLASS(class-name)`

This macro is used for exporting a chain manager class.

The macro takes the base name of the class - the class's name excluding any package. Software AG recommends declaring codecs and transports in a package to avoid name collisions, and using the macro within the namespace declaration, or where a `using` statement applies. For example:

```
#include <sag_connectivity_plugins.hpp>
using namespace com::softwareag::connectivity;

namespace my_namespace {

class MyTransport: public AbstractSimpleTransport
{
public:
    MyTransport(const TransportConstructorParameters &params)
        : AbstractSimpleTransport(params)
    {
        ...
    }
    virtual void deliverMessageTowardsTransport(Message &m)
    {
        logger.info("deliverMessageTowardsTransport()");
    }
    ...
};

SAG_DECLARE_CONNECTIVITY_TRANSPORT_CLASS(MyTransport)
} // end my_namespace
```

Note:

For a chain manager, you should include the header file `sag_connectivity_chain_managers.hpp` instead of `sag_connectivity_plugins.hpp` which is used in the above example.

Requirements of a transport chain manager plug-in class

A transport plug-in can control the lifetime of chains involving that transport, by providing a dynamic chain manager. The chain manager can decide when to create or destroy chains, and is typically controlled by either listening to channel subscriptions from the correlator host, or by listening to external connections.

For example, any topic or queue on a message bus can be exposed dynamically without having to provide a list of the topics/queues to connect to. On a channel-created notification, the chain manager would check if there is a topic/queue to which it can connect, and create a chain instance to connect to that topic/queue on demand.

Alternatively, the chain manager may listen to accept new connections, and each new connection can create a new chain instance. For example, new incoming connections could each create a new chain instance, with the chain manager holding a server socket, and on accepting connections creating a suitable chain instance to handle messages on that connection. In both cases, the chain manager will typically hold some connection object, which it then needs to pass to transport instances when they are created. Thus, the chain manager and transport are usually tightly coupled, and a chain manager can only create chains using its own transport class.

A transport that uses a dynamic chain manager to create its instances consists of a subclass of `AbstractTransport` (or `AbstractSimpleTransport`), and a subclass of `AbstractChainManager` which is the class that must be specified in the configuration file's `connectivityPlugins` section. See [“Configuration file for connectivity plug-ins” on page 26](#) and [“Requirements of a plug-in class” on page 43](#).

The chain manager is responsible for:

- Creating and destroying chains as needed, often in response to notifications about the channels that the EPL application is using or to handle new connections initiated from another process. Some managers will create a single chain for sending messages in both directions on a given channel (towards host and towards transport), others may create separate chains for each direction, or may only support one direction. For detailed information about this, see [“Creating dynamic chains from a chain manager plug-in” on page 62](#). In summary, there are two main aspects to chain creation:
 - Selecting which chain definition to use when creating a new chain instance, if there is more than one chain definition available for this transport. For more information, see [“Creating dynamic chains from a chain manager plug-in” on page 62](#).
 - Instantiating the transport plug-in during creation of the chain, by calling the transport's constructor.

With Java, the chain manager can simply pass through the `logger` and `params` arguments to a transport constructor with the same signature as the `createTransport` method, or can

pass additional information that the transport needs - such as a reference to the chain manager or connection, or information about the host channel(s) the chain is sending to/from.

With C++, the transport's constructor is invoked directly, with a signature of `(const ManagedTransportConstructorParameters &, ...)` where ... are any extra parameters passed in the `createChain` call.

- Instantiating and managing the lifetime of any connection to an external server or other resources that should be shared by all associated transports. Usually, it is undesirable for each transport or chain to have its own separate connection to any external server that the transport is using, as the number of chains may be large. In many protocols, connections are heavyweight entities that you would not want to have lots of. The chain manager can create its connections at any time, but it is recommended to create the initial connection in the chain manager's `start()` method if it is desirable for the correlator to delay coming up until the connection is established, and for the correlator to fail to start if an exception is thrown while making the initial connection. If not, it should happen on a background thread created by the `start()` method.
- Optionally, reporting status information that applies to the chain manager rather than to individual transports. For example, status about a connection shared across all transports could be reported by the chain manager, as could aggregated KPI statistics from all transport chains.

The transport class is responsible only for sending and/or receiving messages, often making use of a connection owned by the chain manager. Transports can also report their own status values if desired, though if it is likely there will be a large number of transports, individual status for each may be less useful and more expensive to report than aggregated status for the whole chain manager.

Every chain manager is required to implement the following:

- A public constructor that will be called during correlator startup for each configured `dynamicChainManager`, with the same signature as the `AbstractChainManager` constructor.
- `createTransport` (Java only; for C++, the transport's constructor is invoked directly as described above)
- `start`
- `shutdown`

The `AbstractChainManager` base class has a number of member fields that provide access to logging, the configuration for all dynamic chain definitions associated with its transport, and a `ChainManagerHost` interface which supports creating chains and registering channel listeners.

A typical chain manager would use its `start()` method to create any required connection(s) to external servers, and to add a `ChannelLifecycleListener` providing notifications when channels with a specific prefix are created or destroyed.

It is possible to listen for all channels regardless of prefix, but using a prefix to limit the subset of channels monitored by each chain manager is recommended to improve performance. The `ChannelLifecycleListener` will fire to indicate that a channel has been created when the channel name is used for the first time, typically as a result of the Apama application calling

`monitor.subscribe(channel)` or `send event` to `channel`. When this happens, the manager must first decide whether it needs to have a chain for the specified channel, as some managers may only wish to take action if a channel with the specified name exists on the external system they are connected to. The manager must also check if it already has a chain for this channel in the specified direction, since in some situations the listener will notify about creation of the same channel more than once (see `flushChannelCache` in "Shutting down and managing components" in *Deploying and Managing Apama Applications*). If the manager has established that a chain is needed for this channel and none already exists, it should create one before returning from the listener callback. Or if a chain already exists for this channel, but is no longer needed, it should destroy it. In other cases, it should do nothing.

The first EPL `monitor.subscribe(channel)` or `send event` to `channel` call to use a channel with a registered listener will block until the listener returns to ensure that no messages are missed if the manager does decide to create a chain for that channel. If an error occurs in the chain manager's implementation of the listener callback, it will be logged but no exception is thrown in the EPL application. See [“Creating dynamic chains from a chain manager plug-in” on page 62](#) for more details about how to create chains from a dynamic chain manager.

When the `onChannelDestroyed` method of the `ChannelLifecycleListener` is called to indicate that a channel has been destroyed (that is, implies that there are no remaining EPL monitors using the channel for the specified direction), the chain manager should call `destroy` on the chain to shut down and disconnect all associated transport and codec plug-ins. Chain managers should not implement reference counting, as the `destroy` notification will not be fired until all uses of the channel have finished.

Note that at present, channel destroy notifications are only sent for the `TOWARDS_HOST` direction (`monitor.subscribe()`) since in the `TOWARDS_TRANSPORT` direction (`send event` to `channel`) there is no unambiguous way of determining when a channel is no longer needed.

If using correlator persistence, the required channel lifecycle notifications for channels in use by any persistent monitors will be replayed to chain managers during recovery, so there is no need for chain managers to persist any state across restarts to support correct operation of persistence.

For detailed information about the classes and interfaces involved in creating a chain manager, including more detailed information about how to use the listener API correctly and safely, see the *API Reference for Java (Javadoc)* on the `com.softwareag.connectivity.chainmanagers` package, or see the *API Reference for C++ (Doxygen)* on the `com::softwareag::connectivity::chainmanagers` namespace.

For a complete example of a working Java chain manager and transport, see the Kafka sample in the `samples/connectivity_plugin/java/KafkaTransport` directory of your Apama installation.

A skeleton sample for C++ is provided in the `samples/connectivity_plugin/cpp/skeleton_chainmanager` directory of your Apama installation. You can use this sample as a starting point to write your own C++ chain manager and transport.

Building plug-ins

See the `samples/connectivity_plugin` directory of your Apama installation for working samples of connectivity plug-in source code, Ant build files, makefiles, or Microsoft Visual Studio projects

for building C++ plug-ins (note that the build instructions in the `samples/connectivity_plugin` directory assume that you are using a recent version of Microsoft Visual Studio).

Building Java plug-ins

Java plug-ins require the `connectivity-plugins-api.jar` file in the `lib` directory of your Apama installation to be on the compiler's classpath as it defines `Message`, `AbstractCodec`, `AbstractTransport`, `AbstractChainManager` and associated classes. The classes are in the `com.softwareag.connectivity.*` packages.

All code samples shown in this connectivity plug-ins documentation assume either that the following lines of code are present in the source file, or that the classes are imported individually.

```
import com.softwareag.connectivity.*;
import java.util.*; // Map, HashMap, List, ArrayList
// are commonly used classes in these samples
```

You can develop Java-based connectivity plug-ins in Software AG Designer. To do so, you have to add the Apama Java support to your Apama project. See "Creating Apama projects" in *Using Apama with Software AG Designer* for more information. This will automatically take care of the classpath for you.

Building C++ plug-ins

C++ plug-ins require the header files in the `include` directory of your Apama installation to be on the compiler's include path. The plug-in should be linked as a shared library and it should link against the `apclient` library in the `lib` directory of your Apama installation. The resultant library will thus depend on the `apclient` library.

All code samples shown in this connectivity plug-ins documentation assume either that the following lines of code are present in the source file, or that individual `using` statements are used for each class.

```
#include <sag_connectivity_plugins.hpp>

using namespace com::softwareag::connectivity;
```

For chain manager classes, the following is also needed:

```
#include <sag_connectivity_chain_managers.hpp>
using namespace com::softwareag::connectivity::chainmanagers;
```

For information on the compilers that have been tested and are supported, refer to the *Supported Platforms* document for the current Apama version. This document is available from <http://documentation.softwareag.com/apama/index.htm>.

Connectivity plug-in headers are a wrapper around a C ABI. Unlike other plug-ins, the C++ plug-ins are therefore not sensitive to which C++ compiler product, compiler version and compiler configuration (for example, a debug or release build) is used. The C++ compiler used does need to correctly support parts of the C++11 standard, and exact settings required for each compiler will vary.

If you are building a shared library to be used by multiple plug-ins and using the plug-in-specific data structures as part of your API between the library and the plug-ins, then you must ensure that the library and all of the plug-ins are compiled using the same version of the Apama header files. This means that if you upgrade Apama and want to recompile one of them, you must recompile all of them. You can choose not to recompile anything and they will still work.

If you compile with headers from multiple service packs of Apama, then you may see errors similar to the following when you try to link them.

- Linux :

```
undefined reference to `Foo::test(com::softwareag::connectivity10_5_3::data_t const&)'
```

- Windows:

```
testlib2.obj : error LNK2019: unresolved external symbol "public: void __cdecl
Foo::test(class com::softwareag::connectivity10_5_3::data_t const &)"
(?test@Foo@@QEAXAEBVdata_t@connectivity10_5_3@softwareag@com@@@Z) referenced in
function "public: void __cdecl Bar::test(class
com::softwareag::connectivity10_5_3::data_t const &)"
(?test@Bar@@QEAXAEBVdata_t@connectivity10_5_3@softwareag@com@@@Z)

testlib2.dll : fatal error LNK1120: 1 unresolved externals
```

If you encounter a similar error, try recompiling all your components with the same version of the headers.

If you are compiling a single plug-in, or multiple completely independent plug-ins, you can recompile them in any combination at any time.

If you want to develop plug-ins in C++, you have to use your own C++ compiler/development environment.

C++ data types

C++ plug-ins handle messages, which have a payload which can be any of the following:

- string (null terminated UTF-8) (const char*)
- integer (64 bit) (int64_t)
- float (64 bit) (double)
- decimal (64 bit) (decimal_t)
- boolean (bool)
- map of any of these types to any of these types (map_t)
- list of any of these types (list_t)
- byte buffer (buffer_t)
- a custom object - a pointer to a user-defined class (custom_t)

- an empty or “null” value

To facilitate this, the payload member of a message is of the `com::softwareag::connectivity::data_t` class type. The `data_t` type is a “smart union” that holds one of the above types, and knows which type it holds. It has a similar API to a boost variant. The `data_t` class has constructors from each of the above types, and a no-argument constructor which creates an empty value. Accessing the data contained in a `data_t` instance can be performed as described below.

- Use the `get` free template function. For example:

```
data_t data("some text");
const char *text = get<const char*>(data);
map_t map;
data_t mapdata(map);
map &mapref = get<map_t>(mapdata);
```

For compound types `map_t`, `list_t`, `custom_t` and `buffer_t`, this returns a reference to the object.

- You can attempt to convert integer, boolean, string or float values inside a `data_t` to each other, regardless of the underlying type. The following is an example for turning a string into its numerical representation:

```
data_t data("10");
int64_t i = convert_to<int64_t>(data);
double f = convert_to<double>(data);
```

- Use a visitor and the `apply_visitor` free template function. A visitor is a class with `operator()` methods for each of the types (and no arguments for empty `data_t`). If you wish to use a visitor that only handles a few types and throws an error on all other types, then sub-class the provided visitor or `const_visitor` template and override one or more of the following methods:

```
visitEmpty
visitInteger
visitDouble
visitBoolean
visitDecimal
visitBuffer
visitList
visitMap
visitCustom
```

The result of `apply_visitor` is of type `visitor::result_type` (typically a typedef), or the second template argument of `visitor/const_visitor`. For example:

```
struct print_data: public const_visitor<print_config, void>
{
```

```

void visitString(const char *s) const { std::cout << s; }
void visitList(const list_t &l) const
{
    std::cout << "[";
    for (list_t::const_iterator it = l.begin(); it != l.end(); ++it) {
        apply_visitor(print_data(), *it);
        if (it+1 != l.end()) std::cout << ", ";
    }
    std::cout << "]";
}
void visitMap(const map_t &m) const
{
    std::cout << "{";
    std::vector<std::string> keys;
    for (map_t::const_iterator it = m.begin(); it != m.end(); ++it) {
        keys.push_back(get<const char*>(it.key()));
    }
    std::sort(keys.begin(), keys.end());
    for (std::vector<std::string>::iterator it = keys.begin();
        it != keys.end(); ++it) {
        std::cout << *it << "=";
        apply_visitor(print_data(),
            m.find(data_t(it->c_str()))>().value());
        if (it+1 != keys.end()) std::cout << ", ";
    }
    std::cout << "}";
}
};

data_t data;
apply_visitor(print_data(), data);

```

Containers and custom values

The `list_t` and `map_t` classes are containers that hold a list or unordered (hash) map of `data_t` elements (and for `map_t`, keys). These are similar in behavior to the C++ standard library classes `std::vector` and `std::unordered_map` - with a subset of the methods available. `list_t` maintains order of the elements in them, and allows access with the operator `[]` overload or iterators returned from `begin` and `end` (or `rbegin` and `rend`, or `cbegin` and `cend`). `map_t` does not maintain ordering, and should give average $O(1)$ cost for insertions and lookups. `map_t` does not permit a `data_t` holding a `custom_t` value to be used as a key.

When using iterators over the `map_t` and `list_t` types, or references to entries within the container, you must not modify the parent container while iterating over it, or before accessing those references. Trying to use an iterator after modifying the parent container will assert, or throw an exception if asserts are disabled. There is no such protection for references. Note that if you have a non-const `map_t`, then the operator `[]` can count as a mutation - it will add an entry if the entry does not already exist.

The `buffer_t` is similar to `list_t`, but its element type is `byte` (`uint8_t`). `buffer_t` can be translated to and from a Java `byte[]`, but not to host plug-ins as there is no correlator type that maps to or from them.

The `custom_t` type behaves like a `std::unique_ptr` to a user-specified class, with an explicit copy method. The class must have a copy constructor and destructor that do not throw exceptions. It

is up to you to ensure that the correct type is used; but if all classes wrapped in `custom_t` are virtual, then it is possible to use `dynamic_cast` or `typeid` to distinguish the types of the objects held by `custom_t`. Note that visitors are called with a `sag_underlying_custom_t` reference; this needs to be cast with `static_cast` to the expected `custom_t<Type>` reference. `custom_t` values can only be passed between C++ plug-ins; they cannot be passed to host plug-ins or Java plug-ins (and you need to ensure that the plug-ins share the same definition of the class).

Decimals

The Apama decimal type is converted to/from a `decimal_t` struct. This has a single `int64_t` which is the bit pattern of the IEEE754 64-bit floating point decimal. This can be serialized, copied or moved, but no facilities are provided for arithmetic or conversion to string or normal floating point types; a third-party decimal library is required if such functionality is required.

Copying, moving and swapping

The `data_t` and compound types `list_t`, `map_t`, `buffer_t` and `custom_t` deliberately hide access to the copy constructor and assignment operator to avoid accidental copies. Explicit copies are possible with the `copy()` method, which performs a deep copy (that is, for a map or list value, it copies each element, and each element of those if they are compound types). Rather than copying values, consider if the move constructor or move assignment operator can be used (these leave the object moved from as empty). To call these, the argument needs to be enclosed in the `std::move()`.

Map contents used by the `apama.eventMap` host plug-in

The payloads that the `apama.eventMap` generates for transportward messages and that it requires for hostward messages are maps. For Java chains, this is `java.util.Map<Object, Object>`. For C++ chains, this is a `map_t`.

Each key in the map is the name of a field in the EPL event definition and the value the corresponding EPL value. Each event containing other events is represented as a Map value within the top-level field, allowing nesting of events, dictionaries and sequences. For events sent from chains into the correlator, all fields must have non-empty values and must be present as keys in the map, unless the configuration setting `allowMissing` is set to `true`. Keys that do not correspond to fields are ignored by default. There is an exception: an empty value that maps to an `optional<type>` or `any` in EPL is permitted even if `allowMissing` is `false` (see also the descriptions of the `optional` and `any` types in the *API Reference for EPL (ApamaDoc)*).

Events can be annotated with the `com.softwareag.connectivity.ExtraFieldsDict` annotation (see "Adding predefined annotations" in *Developing Apama Applications*) which names a dictionary field, in which case any unmapped keys are placed into this dictionary field. This can be disabled with the `extraFields` configuration property. The dictionary must be one of:

- `dictionary<string,string>` - Keys and values are coerced into strings. Lists generate the string form of `sequence<string>`. Maps generate the string form of `dictionary<string,string>`.

- `dictionary<any,any>` - Values are mapped to the corresponding EPL type, or `sequence<any>` for lists and `dictionary<any,any>` for maps without names.
- `dictionary<string,any>` - Keys are coerced into strings. Values are mapped as described above.

The types are converted as described below:

EPL type	Transportward events will contain Java or C++ type	Hostward events can also convert from types
Event	<code>java.util.Map</code> or <code>map_t</code>	
dictionary	<code>java.util.Map</code> or <code>map_t</code>	
sequence	<code>java.util.List</code> or <code>list_t</code>	
location	<code>java.util.Map</code> or <code>map_t</code> (keys are <code>x1</code> , <code>y1</code> , <code>x2</code> , <code>y2</code>)	
<code>com.apama.Channel</code>	<code>java.lang.String</code> or <code>const char*</code> (if it is a string channel)	
string	<code>java.lang.String</code> or <code>const char*</code>	all numeric types, boolean
integer	<code>java.lang.Long</code> or <code>int64_t</code>	all numeric types (except NaN float values), strings if they can be parsed as an integer
float	<code>java.lang.Double</code> or <code>double</code>	all numeric types, strings if they can be parsed as a float
decimal	<code>java.math.BigDecimal</code> (but see the notes below) or <code>decimal_t</code>	all numeric types, strings if they can be parsed as a float
boolean	<code>java.lang.Boolean</code> or <code>bool</code>	string, if true or false
optional	<p>EPL values of type <code>optional<T></code> translate into one of the following:</p> <ul style="list-style-type: none"> ■ null or corresponding Java type (see the above conversions), or ■ a <code>data_t</code> that is either empty or of type <code>T</code> (see the above conversions). 	<p>A Java object or <code>data_t</code> that corresponds to an EPL value of type <code>optional<T></code> is translated if</p> <ul style="list-style-type: none"> ■ the Java object is null or of type <code>T</code>, ■ <code>data_t</code> is empty or of type <code>T</code>.

EPL type	Transportward events will contain Java or C++ type	Hostward events can also convert from types
any	<p>EPL values of type any translate into one of the following:</p> <ul style="list-style-type: none"> ■ null or corresponding Java type (see the above conversions), or ■ a data_t that is either empty or of the underlying concrete type (see the above conversions). <p>See also the note below for Event mappings.</p>	<p>A Java object or data_t that corresponds to an EPL value of type any is translated if</p> <ul style="list-style-type: none"> ■ the Java object is null or of a concrete type (see above), ■ data_t is empty or of a concrete type (see above). <p>See also the note below for Event mappings.</p>

Note:

An any type containing an Event is represented as either `com.softwareag.connectivity.NamedMap` or `map_t` and the name field is set to the event type.

Non-native conversions (a floating point to integer conversion or vice versa) may lose precision, and conversions to/from strings or decimals are more expensive than float or integer conversions. If anything other than an exact match is found, a debug-level log message is generated; you may wish to verify that there are none if the conversion is performance-sensitive.

The following applies to Java only: an EPL decimal value which is NaN (not a number) or an infinity is mapped to/from a `Double` representation of NaN or infinity, as the `BigDecimal` Java type does not support them.

Events containing the following types cannot be sent into the correlator, as they cannot be serialized:

- chunk
- listener
- action variables

Events containing the following can be sent in, provided `allowMissing` is set to true in the host plug-in configuration and no value is provided for that field:

- context
- `com.apama.exceptions.Exception`
- `com.apama.exceptions.StackTraceElement`

Events containing cycles cannot be sent into or out of the correlator, but arbitrary nesting is permitted. Aliases will be flattened.

For Java plug-ins, handling messages from the `apama.eventMap` plug-in thus involves casting the payload of the message from `Object` to `Map`, and then accessing members of that, casting as necessary

(or, for flexibility, introspecting their types by using the `instanceof` operator). For example, for the following event definition, the `CustomerOrder` is translated to a map with `deliveryAddress`, `userId` and `items` keys, and `items` will be a list of maps containing `itemId`, `price` and `qty`.

```
event LineItem {
    string itemId;
    float price;
    integer qty;
}
event CustomerOrder {
    string deliveryAddress;
    string userId;
    sequence<LineItem> items;
}
```

To print the total cost of an order (sum of product of `qty` and `price` for each item), the Java code would be as follows:

```
public void deliverMessageTowardsTransport(Message message) {
    MapExtractor mList = new MapExtractor((Map)message.getPayload(),
        "CustomerOrder");
    List<MapExtractor> items = mList.getListOfMaps("items", false);
    double total = 0.0;
    for(MapExtractor item : items) {
        double price = item.get("price", Double.class);
        long qty = item.get("qty", Long.class);
        total = total + price * qty;
    }
    LOGGER.info("Order value is "+total);
}
```

Note that due to type erasure, the type parameters on `Map` or `List` are not checked or guaranteed. In the above example, it is convenient to cast the list representing EPL field `sequence<LineItems>` to `List<Map>` to avoid having to cast the entries within it. The `Map`, however, is still treated as a map of objects as it has different types (`String`, `Double`, `Long`) in it.

For C++ plug-ins, handling messages from the `apama.eventMap` plug-in involves using the `get<map_t>` function and accessing the members of that, using `get<>` as necessary. If code needs to be flexible as to which type it accepts, then use the visitor pattern (see [“C++ data types” on page 51](#)). For example, using the event definition above, the following C++ code will print the total cost of the order:

```
virtual void deliverMessageTowardsTransport(Message &message) {
    map_t &payload = get<map_t>(message.getPayload());
    list_t &items = get<list_t>(payload[data_t("items")]);
    double total = 0.0;
    for(list_t::iterator it = items.begin(); it != items.end(); it++) {
        MapExtractor m( get<map_t>( *it ), "LineItem" );
        double price = m.get<double>("price");
        long qty = m.get<int64_t>("qty");
        total = total + price * qty;
    }
    logger.info("Order value is %f", total);
}
```

The following constructs and sends an order with one line item into the correlator:

```
Map<String, Object> payload = new HashMap<>();
payload.put("deliveryAddress", "1 Roadsworth Avenue");
payload.put("userId", "jbloggs");
List<Map> items = new ArrayList<>();
Map<String, Object> item = new HashMap<String, Object>();
item.put("itemId", "item1");
item.put("price", 3.14);
item.put("qty", 10);
items.add(item);
payload.put("items", items);

Map<String, String> metadata = new HashMap<String, String>();
metadata.put(Message.HOST_MESSAGE_TYPE, "CustomerOrder");
Message msg = new Message(payload, metadata);
hostSide.sendBatchTowardsHost(Collections.singletonList(msg));
```

The above can also be written more compactly:

```
hostSide.sendBatchTowardsHost(Collections.singletonList(new
Message(payload).putMetadataValue(Message.HOST_MESSAGE_TYPE, "CustomerOrder")));
```

This would typically be done in a more automated fashion, translating data from some other form, rather than laboriously setting each field as needed - though some combination will often be needed.

The equivalent C++ code is:

```
map_t payload;
payload.insert(data_t("deliveryAddress"), data_t("1 Roadsworth Avenue"));
payload.insert(data_t("userId"), data_t("jbloggs"));
list_t items;
map_t item;
item.insert(data_t("itemId"), data_t("item1"));
item.insert(data_t("price"), data_t(3.14));
item.insert(data_t("qty"), data_t((int64_t) 10));
items.push_back(data_t(std::move(item)));
payload[data_t("items")] = data_t(std::move(items));

Message msg(data_t(std::move(payload)));
msg.putMetadataValue(Message::HOST_MESSAGE_TYPE(), "CustomerOrder");
hostSide->sendBatchTowardsHost(&msg, (&msg)+1);
```

Metadata values

Every message has a metadata member, which for Java is a Map object containing String keys and Object values. For C++, it is a map_t which by convention only contains const char * keys, but any type as values.

The metadata holds information about the event:

Value	Description
sag.type	The name of the event type. This is required when sending events into the apama.eventMap plug-in, unless the defaultEventType configuration

Value	Description
	property is set. For the <code>apama.eventString</code> plug-in, the event type name comes from the string form of the event itself.
<code>sag.channel</code>	<p>The name of the channel from which the event originated or to which it is to be delivered. This is optional for hostwards messages.</p> <p>Note: If the transport uses the same channel through its lifetime, it is recommended that you set the <code>defaultChannel</code> property in the configuration file, rather than setting the <code>sag.channel</code> for every message. See also “Host plug-ins and configuration” on page 30.</p>
<code>sag.messageId</code>	<p>The message identifier. This is used for reliable receiving (that is, in reliable messages going towards the host). The message identifier should be unique within the scope of the chain and deployment and during the lifetime of the application. Typically it will be generated by the message bus to which the connectivity plug-in transport is connected. See also “Using reliable transports” on page 33.</p> <p>CAUTION: If you are using a codec to make the message identifier visible as an event field in EPL, it is important to <i>copy</i> the value from <code>sag.messageId</code>. Moving the value (and thus removing it from <code>sag.messageId</code>) will disrupt reliable receiving.</p>

In Java, these values are available as constants on the `Message` class:

- `Message.HOST_MESSAGE_TYPE`
- `Message.CHANNEL`
- `Message.MESSAGE_ID`

In C++, these are the following methods:

- `HOST_MESSAGE_TYPE()`
- `CHANNEL()`
- `MESSAGE_ID()`

Plug-in components can use the metadata to pass other auxiliary data about a message between chain components. These could be headers from an HTTP connection, authentication tokens, or signalling for transaction boundaries. It is recommended that all metadata keys are namespaced. The `sag` namespace is reserved for Software AG use. Host plug-ins currently only use the metadata keys above.

The metadata contents can be manipulated directly by calling methods on the map returned by `Message.getMetadataMap()`. A `Message.getMetadata()` is also available in order to manipulate the

stringified version of the metadata values. Values can be inserted into the metadata by using `Message.putMetadataValue(...)`.

Lifetime of connectivity plug-ins

Instances of connectivity chains can be created in different ways. See [“Static and dynamic connectivity chains” on page 26](#) for detailed information.

At correlator startup:

- Each codec, transport and dynamic chain manager class is loaded using the `classpath` or `libraryName`.
- Each dynamic chain manager listed in `dynamicChainManagers` is instantiated using its public constructor and passing the `managerConfig` from the configuration file.
- The `start()` method is called on any dynamic chain managers. Chain managers can create dynamic chains at any point after this, though in practice, dynamic chains are usually created after correlator startup, once the Apama application is injected and running.
- Each chain listed in `startChains` is created and started (see below).

The correlator is only pingable and available for external access after all of the above operations have completed.

Whenever a new chain instance is created (either during correlator startup if listed in `startChains`, or at any time dynamically by EPL or a chain manager):

- The correlator determines the list of codec and transport plug-ins in the chain and the configuration for each as follows:
 - If the chain is statically configured, the plug-ins and plug-in configurations listed under `startChains` are used.
 - If the chain is being created dynamically, the chain manager implementation or EPL `createDynamicChain` call specifies which of the chain definitions listed under `dynamicChains` is to be used, and the configuration for this chain instance is prepared by replacing any `@{varname}` runtime substitution variables in the chain definition using the map passed in to `createChain` or supplied by the chain manager.
- A new instance of each transport and codec class in the chain is constructed using the public constructor, as described in [“Requirements of a plug-in class” on page 43](#). If the transport has a dynamic chain manager, the manager's `createTransport` method is used instead of calling the transport constructor directly (for Java) or extra parameters to the `createChain` call are passed through to the constructor (C++), which gives the chain manager the opportunity to pass extra information required by the managed transport (such a reference to itself).
- `hostSide` and `transportSide` members are set on all transport and codec plug-ins in the chain.
- Static and EPL-created chains are started automatically once created. Chain managers must explicitly call `start()` on the newly created chain when they are ready.
- The `start()` method is first called on all codecs in the chain.

- Then the `start()` method is called on the transport.
- Messages may begin flowing.

If any of the constructors or `start()` methods invoked during correlator startup throw an exception, that will be logged as an error and the correlator will fail to start. These methods should complete quickly; delays here will delay the correlator starting up. Blocking or long running operations should be handled by a separate thread.

After `start()` is called on all members of the chain, events may flow through the chain in either direction (if an EPL application is emitting events to the chain, they will be delivered as messages and delivered through the codecs towards the transport). The transport is permitted to send events hostwards, but they will be queued by the correlator until the application is ready for them.

Soon after the EPL application has been injected (and, if necessary, it has performed initialization), the EPL application should call `ConnectivityPlugins.onApplicationInitialized()`. At this point:

- `hostReady()` is called on every codec.
- `hostReady()` is called on the transport.

Dynamic chains that are created after `onApplicationInitialized` has been called will have `hostReady` called as soon as the chain is created.

If an exception is thrown by a plug-in's `hostReady()` method or by the `start()` method of a dynamically instantiated plug-in, that will be logged as an error and the chain will be disconnected. These methods should complete quickly; delays here will delay the EPL application. Blocking or long running operations should be handled by a separate thread. Any events previously sent to the host will now be delivered, but the order of all events from a chain will be maintained.

When the correlator is shut down (for example, via `engine_management -s`) or when the dynamic chain is destroyed by EPL or a dynamic chain manager, chains will be stopped:

- `shutdown()` is called on all chain managers (if any exist)
- `shutdown()` is called on every codec.
- `shutdown()` is called on the transport.

The `shutdown()` method gives chain managers an opportunity to destroy any chains they are managing in an orderly fashion.

The `shutdown` method on transports should make the transport discard any further messages sent to the transport, and unblock if any threads are currently delivering messages to the transport and are blocked. If possible, the `sendEventsTowardsTransport` method should be written to allow any blocking behavior to be unblocked when a shutdown occurs. For example, if a socket is being used by a transport, it should be shut down or closed so that any threads reading or writing on the socket's streams terminate.

Any messages delivered to a plug-in once the `shutdown` method has been called may be discarded by the plug-in. Messages may be delivered to a plug-in even after the `shutdown` call has completed, and the plug-in should not crash if that occurs.

If threads are required by a transport to deliver events to the transport or read from a connection, they would normally be started by the `hostReady` method and stopped and joined in the `shutdown` method.

Note:

For C++ plug-ins only: the plug-in object of each plug-in is destroyed, so the plug-in class's destructor (if defined) is called. No events should be flowing through a chain at this point.

Exceptions thrown from any of `sendBatchTowards`, `transformEvent` or `deliverEvent` will be logged and not propagated to their callers. Exceptions are not a suitable means to provide information between plug-ins as they are ambiguous if a large batch of events are being processed, and some codecs may choose to send events on a separate thread. Use messages to send such events; these can be null payload with information stored in the metadata, in which case most codecs will ignore the messages and pass them through.

Creating dynamic chains from a chain manager plug-in

If a transport has an associated chain manager, the chain manager is responsible for creating all chains involving that transport. Note that this is the only way to create chains involving such a transport, they cannot be created using `startChains` or from EPL's `ConnectivityPlugin.createDynamicChain` action.

Chain managers may create chains at any time after `start()` has been called and before `shutdown()`, and for any reason. However, most managers create chains in response to a notification that a channel has been created, which means it is in use for the first time. See [“Requirements of a transport chain manager plug-in class” on page 47](#) for more information about how to do this.

When a chain manager is ready to create a new chain, it does so by calling `ChainManagerHost.createChain()`, usually making use of the `host` field on `AbstractChainManager`. The following information must be supplied when creating a chain:

- `chainInstanceIdSuffix` - A string identifier which will be suffixed onto `managerName` “-” to uniquely identify the new chain instance.

CAUTION:

A small amount of memory is allocated for each unique chain instance identifier. This memory is not freed when the chain is destroyed. Therefore, if you are creating many chains, consider reusing old chain instance identifiers. If you create more than 1000 unique identifiers, a warning is written to the correlator log file to notify you of this. You cannot have two active chains with the same chain instance identifier, so only reuse identifiers which are no longer in use.

- `dynamicChainDefinition` - Specifies which of the chain definitions that contain this transport should be used. The `AbstractChainManager` provides `getChainDefinition()` helper methods to select a chain definition based on its identifier or by assuming that only one definition will be configured. For more complex cases, a collection of all the chain definitions for this transport is provided in the `chainDefinitions` field which a manager can iterate over to find the one with

the desired transport plug-in configuration. There are various possible approaches to selecting which chain definition to use to create a chain:

- For some managers, it may not make sense to support multiple chain definitions and can be written to just use a singleton chain definition.
- Some managers may allow the user to specify a chain definition by providing a chain definition identifier as a configuration option for the manager in `managerConfig`.
- Another approach is for the manager to search through the available chain definitions and use the transport plug-in's configuration of each one to decide which to use, for example, by providing a channel prefix or regular expression pattern as part of the transport configuration.
- substitutions - The chain manager can provide zero or more `@{varname}` variable replacement values. This provides a way to use information from the manager or transport to configure the host or codec plug-ins, for example, by having the Mapper codec set a field with details about the manager's connection.
- `defaultChannelTowardsHost` and `subscribeChannels` - Used to specify the channel or channels that this chain will send to (unless overridden in individual messages) or subscribe to. You can either use a single chain to send messages in both directions, or have a separate chain for each direction, that is, each transport instance will only be responsible for sending or receiving, but not both.

The transport from the chain definition should match the transport that contains the chain manager making the call. To create the transport object, the chain host will call `createTransport` rather than the transport's constructor (for Java), or the transports constructor, passing through any extra parameters passed to `createChain` (for C++). Once the chain has been created, it needs to be started by calling the `start()` method on the returned Chain object (a Chain pointer for C++).

For more detailed information about the classes and interfaces involved in creating a chain manager, see the *API Reference for Java (Javadoc)* on the `com.softwareag.connectivity.chainmanagers` package, or see the *API Reference for C++ (Doxygen)* on the `com::softwareag::connectivity::chainmanagers` namespace.

User-defined status reporting from connectivity plug-ins

Connectivity plug-ins can add any number of user-defined status values which are reported as part of the correlator's status information from the REST API, the `engine_watch` tool, the Engine Client API, and from the EPL Management interface. Status values can be reported by transports, codecs, or dynamic chain managers.

If the status keys follow the conventions listed in "Monitoring KPIs for EPL Applications and Connectivity Plug-ins" in *Deploying and Managing Apama Applications*, the status and KPIs of your application's connectivity plug-ins can be displayed by Command Central.

For example, a transport plug-in might report a status value to indicate whether it is currently online and working correctly, or failed. Or it can report numeric KPIs indicating the number of

messages sent towards the host (correlator) and towards the transport. A dynamic chain manager might report information about a connection it maintains, and perhaps provide some KPI statistics aggregated across all the transport instances it is managing.

To report status information, create a *status item* by calling the `getStatusReporter().createStatusItem(...)` method on your plug-in class, specifying the key for this status item and its initial value, and store the resulting `StatusItem` object in a field so its value can be updated as necessary. Status items are automatically removed when a transport or codec plug-in is shut down or when the chain is destroyed (in C++, this assumes the `StatusItem` is held by a `std::unique_ptr` in a member of the plug-in class, as we recommend). Be sure to use a unique name to identify the plug-in in each status key; we recommend using the `chainId` and `pluginName` as a prefix for transport and codec plug-ins, or the `managerName` for chain managers. Status keys will have leading and trailing whitespace stripped. Keys cannot be empty. For example, in Java:

```
final StatusItem transportStatus =
    getStatusReporter().createStatusItem(chainId+"."+pluginName
    + ".status", StatusReporter.STATUS_STARTING);

final StatusItem messagesTowardsHost =
    getStatusReporter().createStatusItem(chainId+"."+pluginName
    +".messagesTowardsHost", 0);

...

transportStatus.setStatus(StatusReporter.STATUS_ONLINE);
messagesTowardsHost.increment();
```

Or in C++:

```
std::unique_ptr<StatusReporter::StatusItem> transportStatus;
std::unique_ptr<StatusReporter::StatusItem> messagesTowardsHost;

MyPluginConstructor(...):
    : ...,

    transportStatus(getStatusReporter().createStatusItem(
        chainId+"."+pluginName+".status",
        StatusReporter::STATUS_STARTING())),
    messagesTowardsHost(getStatusReporter().createStatusItem(
        chainId+"."+pluginName+".messagesTowardsHost", 0))
{ ...
}

...
transportStatus->setStatus(StatusReporter::STATUS_ONLINE());
messagesTowardsHost->increment();
```

We recommend using the `STATUS_*` constants provided on `StatusReporter` for values of `".status"` items, to provide consistency and allow the status to be represented correctly if viewed using Command Central.

In addition to the `StatusItem` interface, there is a separate method for atomically setting multiple related items in a single call (for example, a status and an error message). But as the `StatusItem` method is more efficient, it should be used in most cases, especially for items that might be updated frequently such as message counters.

All user-defined status values are currently represented as strings, but for convenience when reporting KPI numbers, an overload of `setStatus` exists that accepts an integer argument for the value, which is automatically converted to a string by the method. There is also an `increment()` method.

For transports and codecs, status reporting is only permitted when your plug-in provides the `TransportConstructorParameters` and `CodecConstructorParameters` constructors. It is not supported when using the older deprecated constructors.

For examples of how to report status information from a connectivity plug-in, see the `samples\connectivity_plugin\cpp\httpclient` and `samples\connectivity_plugin\java\HTTPServer` directories of your Apama installation.

See the `StatusReporter` interface in the *API Reference for Java (Javadoc)* and *API Reference for C++ (Doxygen)* for more information about how to report status.

See also "Using the Management interface" in *Developing Apama Applications* for information about how status values can be set and retrieved by EPL code.

For other ways to view the correlator's status, see "Managing and Monitoring over REST" and "Watching correlator runtime status", both in *Deploying and Managing Apama Applications*

Logging and configuration

For Java plug-ins, the plug-in's constructor is passed a configuration object, the chain name and a logger object. The Abstract classes supplied store these as members (the logger object is named `logger`).

For C++ plug-ins, the Abstract classes have a `logger` member with methods to log at log levels from `TRACE` to `CRIT`, with a `printf` format and variadic arguments. Expensive computations can be predicated on a check of `is<Level>Enabled()`.

Plug-ins should use the SLF4J logger object provided for logging. You should avoid using `System.out` or `System.err` for logging. For both plug-ins written in C++ and Java, log messages are prefixed with `connectivity.PluginName.ChainName`, which is also the category to configure log levels using the `correlatorLogging` section in the YAML configuration file (see "Setting correlator and plug-in log files and log levels in a YAML configuration file" in *Deploying and Managing Apama Applications*). This means, it is not required to identify the plug-in or chain in every log statement.

If your plug-in uses a third-party library that logs with SLF4J or Log4j 2, then the log output goes to the main correlator log file automatically. You can customize log levels as needed using `correlatorLogging` in the YAML configuration file (see "Setting correlator and plug-in log files and log levels in a YAML configuration file" in *Deploying and Managing Apama Applications*). When using a library which uses some other logging implementation, such as Log4j 1, the JDK logger, or Apache Java commons logging (JCL), then add a bridging jar to convert it to SLF4J where possible. Several bridges are available in the `common/lib/ext` and `common/lib/ext/log4j` directories of your Software AG installation.

The configuration contains the definitions from the configuration file for connectivity plug-ins (any `globalConfig` is merged with the per-chain configuration so that the per-chain configuration

takes precedence). The configuration is a map with `String` keys. The values will be one of the following classes:

- `List<Object>` (for C++, `data_t` of type `list_t`) for list YAML entries. Each element will be one of these types.
- `Map<String, Object>` (for C++, `data_t` of type `map_t`) for map YAML entries. Each value will be one of these types.
- `String` (for C++, `data_t` of type `const char *`). Even if the entry in the configuration file is numeric or a boolean, it will be provided as a string.

Plug-ins should use the `MapExtractor` class to extract values from the configuration map which makes it easy to check for invalid configuration options and produce helpful error messages if a required value is missing or of the wrong type.

You can also use the Diagnostic codec to diagnose issues with connectivity plug-ins. See [“The Diagnostic codec connectivity plug-in” on page 228](#) for further information.

Threading

For events being delivered from the correlator to a chain towards the transport, the correlator will only ever call `sendBatchTowardsTransport` from a single thread at a time. Most codecs will call the next component in the chain in the thread that invoked them, but are not required to. A codec can queue events and drain the queue from a separate thread if desired.

Transports and codecs should only make a single call at a time to the `hostSide` plug-in (and thus only one thread at a time passes events towards the host) as the next plug-in may not be thread-safe. Similarly, codecs should only make one call at a time to the `transportSide` plug-in, though one codec may have threads invoking both `hostSide` and `transportSide` concurrently. Plug-ins should not assume that they are called on the same thread each time (in particular, the correlator will use different threads for sending batches of events), but they can assume that no more than one thread at a time sends events to the transport.

Transports and codecs will typically be processing events towards the transport and towards the host in different threads concurrently. The `start`, `hostReady` and `shutdown` methods will be called from different threads to any other operation and while other calls are in progress.

When a chain is disconnected or when the correlator is shut down, the `shutdown` method on the plug-in is called. This should ensure the following:

- Any threads calling into the plug-in which are blocked in the plug-in (particularly for transports) should unblock and return.
- Any threads that the plug-in has started have been stopped and joined.
- The plug-in should ensure any in-progress calls out to other plug-ins have completed.
- The plug-in must ensure no more calls are made out of a plug-in to send messages to other plug-ins.

This is particularly important for C++ plug-ins. Methods calling out after returning from shutdown, or in progress at the point the shutdown method completes, could cause a crash. In practice, a plug-in that starts a thread to read from a socket or other connection and send messages towards the host should close the socket and join the thread (waiting for it to terminate) to meet these requirements.

For C++ plug-ins, we recommend use of the standard libraries such as `std::thread` and `std::mutex` for managing threads and locking in plug-ins. If not available, we provide some simple macros in the `sag_connectivity_threading.h` header file. See the *API Reference for C++ (Doxygen)* for using it.

Developing reliable transports

This section explains how to develop transports that support reliable messaging. For information on how to use them, see [“Using reliable transports” on page 33](#).

Reliable messaging uses control messages, which are special messages that are sent between the host and the transport. They are used to signal actions that the host or transport should take as well as the acknowledgments from these actions. The control messages have `null` (Java) or empty (C++) payloads, and instead store all their information in the metadata.

The type of a control message is stored in a metadata field that can be accessed with the `CONTROL_TYPE` constant of the `com.softwareag.connectivity.Message` (Java) or `com::softwareag::connectivity::Message` (C++) class. The value of this field should be one of the type names listed below. These names are also accessed by constants. For more information, see the `Message` class in the *API Reference for Java (Javadoc)* or *API Reference for C++ (Doxygen)*.

Type	Constant	Description
AckRequired	CONTROL_TYPE_ACK_REQUIRED	This control message is sent from the transport to the host. It is used to ask the host to acknowledge all events that have been sent towards the host before this AckRequired.
AckUpTo	CONTROL_TYPE_ACK_UPTO	This control message is sent from the host to the transport, and it is the acknowledgment for the AckRequired control message. It is used to inform the transport that a particular AckRequired request has been fulfilled.
Flush	CONTROL_TYPE_FLUSH	This control message is sent from the host to the transport. It is used to ask the transport to acknowledge all events that have been sent towards the transport before this Flush.
FlushAck	CONTROL_TYPE_FLUSH_ACK	This control message is sent from the transport to the host, and it is the

Type	Constant	Description
		acknowledgment for the <code>Flush</code> control message. It is used to inform the host that a particular <code>Flush</code> request has been fulfilled.

The control message metadata also contains fields that can be accessed with the following constants:

■ `MESSAGE_ID`

This constant names a metadata field used for uniquely identifying non-control messages (that is, *real* events with payloads) that are being sent towards the host. This constant also names a metadata field on the `AckRequired` and `AckUpTo` control messages that are used for reliable receiving. In `AckRequired`, it contains the message identifier of the immediately preceding non-control message. In `AckUpTo`, it contains the message identifier of the `AckRequired` that is being responded to.

■ `REQUEST_ID`

This constant names a metadata field on the `Flush` and `FlushAck` messages that are used for reliable sending. The field denotes a unique identifier for matching up a `Flush` with its corresponding `FlushAck`.

Transports receive and send the above-mentioned control messages. The exact logic of how they should be processed depends on the exact nature of the external system that the transport connects to. More information and examples are provided below.

Note:

The Java examples below are not intended to be used as a starting point. They only illustrate the core concept of handling control messages.

Writing a transport for reliable receiving

This section describes the obligations of a transport that wishes to see acknowledgments of messages that it is sending towards the host, in order that it can pass those acknowledgments to the reliable messaging system that it is connected to. Such a transport must declare its reliability before any messaging can take place, before the plug-in is fully started. This is achieved by calling the `enableReliability` function on the `PluginHost` member of the transport, either from the constructor or `start()` method.

```
public MyReliableTransport(Logger logger, TransportConstructorParameters params)
    throws IllegalArgumentException, Exception
{
    super(logger, params);
    host.enableReliability(Direction.TOWARDS_HOST);
}
```

A transport must place unique identifiers on any non-control messages (that is, *real* events) that it is sending towards the host. Ideally, these correspond to identifiers provided by the remote messaging system that your code is receiving from. While not 100% necessary, it makes tracing a message through the wider system much easier.

```
MyExternalMessage externalMessage = fictionalRemoteSystem.get();
Message msg = transformToMessage(externalMessage);
msg.putMetadataValue(Message.MESSAGE_ID,
    externalMessage.getUniqueIdentifier());
hostside.sendBatchTowardsHost(Collections.singletonList(msg));
```

A transport must decide how regularly it wishes to receive acknowledgments (AckUpTo) from the host application, by deciding when it sends AckRequired control messages towards the host. In general, you should attempt to space these messages as widely as possible, so as not to put too much burden on the EPL application. The steps taken to “commit” the effects of received events may be quite expensive. However, the frequency of acknowledgments will probably also be constrained by the nature of the remote messaging system your transport is connected to. For example, it may only permit 1,000 unacknowledged messages to be outstanding before blocking receipt of further messages. In this case, you will want to be sending out AckRequired control messages after every n real messages where n is a large fraction of 1,000.

Time is another factor to consider. In the worst case, for example, if acknowledgments are too sparse, a reconnecting application may face 10 minutes of redelivered messages that did not get acknowledged in a previous session. So in general, a transport should make sure to issue AckRequired control messages at least every few seconds, assuming that any non-control messages have been sent towards the host since the last AckRequired.

An AckRequired control message must also contain the message identifier of the preceding non-control message, in order to identify which tranche of previous messages is covered by a corresponding acknowledgment.

```
Message ackRequired = new Message(null);
ackRequired.putMetadataValue(Message.CONTROL_TYPE,
    Message.CONTROL_TYPE_ACK_REQUIRED);
ackRequired.putMetadataValue(Message.MESSAGE_ID, lastId);
```

Finally, a transport should be prepared to act on acknowledgments received from the EPL application, that is, AckUpTo control messages from the host. Each AckUpTo corresponds exactly to a previously issued AckRequired, with both containing the same MESSAGE_ID. AckUpTo messages are seen in the exact same order that the AckRequired messages were issued.

```
public void deliverNullPayloadTowardsTransport(Message message)
    throws Exception {
    Map<String, Object> metadata = message.getMetadataMap();
    if (metadata.containsKey(Message.CONTROL_TYPE))
    {
        String controlType = (String)metadata.get(Message.CONTROL_TYPE);
        if (Message.CONTROL_TYPE_ACK_UPTO.equals(controlType))
        {
            String messageId = metadata.get(Message.MESSAGE_ID);
            fictionalRemoteSystem.ackUpToAndIncluding(messageId);
        }
    }
}
```

Writing a transport for reliable sending

This section describes the obligations of a transport that wishes to reliably acknowledge messages that are being sent to it from an EPL application, that is, from the host. As before, the transport should declare its reliable nature and direction.

```
public MyReliableTransport(Logger logger, TransportConstructorParameters params)
    throws IllegalArgumentException, Exception
{
    super(logger, params);
    host.enableReliability(Direction.TOWARDS_TRANSPORT);
}
```

The transport should be prepared to act on Flush control messages, ensuring that all preceding non-control messages are reliably delivered to a remote reliable messaging system. Once done, the transport should respond with a FlushAck control message towards the host, with a REQUEST_ID set to match it with the corresponding Flush.

Frequent Flush messages are automatically coalesced into individual messages that are more widely spaced. So a transport need not be concerned with the performance impact of responding to every Flush request. Also, Flush messages are subsumed by subsequent Flush messages and their acknowledgments. For example, if a transport receives three Flush messages, a FlushAck corresponding to the final Flush is interpreted as being a response to all three.

```
@Override
public void deliverNullPayloadTowardsTransport(Message message)
    throws Exception {
    Map<String, Object> metadata = message.getMetadataMap();
    if (metadata.containsKey(Message.CONTROL_TYPE))
    {
        String controlType = (String)metadata.get(Message.CONTROL_TYPE);
        if (Message.CONTROL_TYPE_FLUSH.equals(controlType))
        {
            fictionalRemoteSystem.commitEverythingSoFar();
            Message response = new Message(null);
            response.putMetadataValue(Message.CONTROL_TYPE,
                Message.CONTROL_TYPE_FLUSH_ACK);
            response.putMetadataValue(Message.REQUEST_ID,
                Long.parseLong(metadata.get(Message.REQUEST)));
            hostSide.sendBatchTowardsHost(Collections.singletonList(response));
        }
    }
}
```

General notes for developing transports

OpenSSL

OpenSSL initialization and cleanup is handled internally by the correlator process itself. User-developed transports must not perform these tasks.

II Standard Connectivity Plug-ins

4	The Universal Messaging Transport Connectivity Plug-in	73
5	The MQTT Transport Connectivity Plug-in	93
6	The Digital Event Services Transport Connectivity Plug-in	101
7	The HTTP Server Transport Connectivity Plug-in	107
8	The HTTP Client Transport Connectivity Plug-in	131
9	The Kafka Transport Connectivity Plug-in	157
10	The Cumulocity IoT Transport Connectivity Plug-in	163
11	Codec Connectivity Plug-ins	209

4 The Universal Messaging Transport Connectivity Plug-in

■ About the Universal Messaging transport	74
■ Overview of using Universal Messaging in Apama applications	74
■ Setting up Universal Messaging for use by Apama	81
■ Configuring the Universal Messaging connectivity plug-in	82
■ EPL and Universal Messaging channels	91
■ Using Universal Messaging connectivity from EPL	91
■ Monitoring Apama application use of Universal Messaging	92

About the Universal Messaging transport

Universal Messaging 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 Universal Messaging.

You can use the Apama integration to Universal Messaging as a simpler and more deeply integrated alternative to connecting to a Universal Messaging realm via the Java Message Service (JMS). This can be used both to connect different Apama components together using the internal Apama message format and for integrating with non-Apama systems over Universal Messaging. The Universal Messaging connectivity plug-in supports configurable mapping between Apama events and whatever formats the non-Apama components are using.

Only Universal Messaging channels can be used with Apama.

This transport provides a dynamic chain manager which creates chains automatically when EPL subscribes or sends to a correlator channel with the configured prefix, typically `um:`. The selection of which `dynamicChains` definition to use is based on the `channelPattern` regular expression configured in the Universal Messaging transport of each dynamic chain. See [“Configuring the Universal Messaging connectivity plug-in” on page 82](#) for more details on the mapping between correlator channel names and the associated Universal Messaging channel or topic names.

Note:

Universal Messaging queues are not supported in this Apama release.

The Universal Messaging connectivity plug-in does not support reliable messaging.

Overview of using Universal Messaging in Apama applications

In an Apama application, correlators can connect to Universal Messaging realms or clusters. A correlator connected to a Universal Messaging realm or cluster uses Universal Messaging as a message bus for sending Apama events between Apama components. Connecting a correlator to Universal Messaging is an alternative to specifying a connection between two correlators by executing the `engine_connect` correlator tool.

Using Universal Messaging can simplify an Apama application configuration. Instead of specifying many point-to-point connections, you specify only the address (or addresses) of the Universal Messaging realm or cluster. Apama components connected to the same Universal Messaging realm can use Universal Messaging channels to send and receive events. (Universal Messaging channels are equivalent to JMS topics.) Connections to Universal Messaging are automatically made as needed, giving extra flexibility in how the application is designed.

When an Apama application uses Universal Messaging, a correlator automatically connects to the required Universal Messaging channels. There is no need to explicitly connect Universal Messaging channels to individual correlators. A correlator automatically receives events on Universal

Messaging channels that monitors subscribe to and automatically sends events to Universal Messaging channels.

When using the connectivity plug-in for Universal Messaging, you can also talk to non-Apama applications which are connected to Universal Messaging and configure a chain definition to translate those message payloads into ones suitable for Apama.

Comparison of Apama channels and Universal Messaging channels

In an Apama application configured to use Universal Messaging, when an event is sent and a channel name is specified, the default behavior is that Apama determines whether there is a Universal Messaging channel with that name. If there is, then Apama uses the Universal Messaging message bus and the specified Universal Messaging channel to deliver the event to any subscribers. Subscribed contexts can be in either the originating correlator or other correlators connected to the Universal Messaging broker.

If a Universal Messaging 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 Universal Messaging channel or an Apama channel, events are delivered directly to receivers that are connected directly to the correlator.

The following table compares the behavior of Apama channels and Universal Messaging channels.

Apama channels	Universal Messaging channels
Configuration of multiple point-to-point connections.	Specification of the same Universal Messaging realm address or addresses.
Each execution of <code>engine_connect</code> specifies the correlator to connect to. Each IAF adapter configuration specifies each correlator that adapter connects to.	Startup options for connected correlators specify the same Universal Messaging realm to connect to. Each IAF adapter configuration specifies the same address for connecting to Universal Messaging.
Correlators and IAF adapters require explicitly set connections to communicate with each other.	Correlators and IAF adapters automatically connect to Universal Messaging to communicate with each other.
Configuration changes are required when an Apama component is moved to a different host.	No configuration change is needed when an Apama component is moved to a different host if both hosts are connected to the same Universal Messaging 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 Universal Messaging realm.

Apama channels	Universal Messaging channels
Events sent on an Apama channel go directly to subscribers.	Events sent on a Universal Messaging channel go to the Universal Messaging broker and then to subscribers.
Connection configurations must be synchronized with application code.	Connection to a Universal Messaging 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.
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 Universal Messaging realm, and then returns to the correlator.
Default channel, the empty string, is allowed.	No default channel.

Choosing when to use Universal Messaging channels and when to use Apama channels

Typically, you want to

- Use Universal Messaging 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 Universal Messaging channels when your application needs the flexibility for a monitor or context to be moved to another correlator. With Universal Messaging, you can re-deploy monitors sending or subscribing to Universal Messaging channels among the correlators connected to the same Universal Messaging realm without having to change any of the configuration for the Universal Messaging connectivity.
- 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 Universal Messaging realm. Specification of a Universal Messaging 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, or
- in different correlators on the same host, or
- in different correlators on different hosts.

The first time a channel is used, the default behavior is that Apama determines whether it is a Universal Messaging channel or an Apama channel, and the designation is cached. After the first use, the presence or not of the channel in the Universal Messaging broker is cached, so further use of the channel is not impacted.

Using Universal Messaging channels lets you take advantage of some Universal Messaging features:

- Using a Universal Messaging cluster can guard against failure of an individual Universal Messaging realm server. See the [Universal Messaging documentation](#) for more information on clusters.
- Universal Messaging 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 Universal Messaging channel rather than an Apama channel can have a lower throughput and higher latency. If there is a Universal Messaging 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 Universal Messaging channel are delivered by means of the Universal Messaging 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.

General steps for using Universal Messaging in Apama applications

Before you perform the steps required to use Universal Messaging 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 Universal Messaging channels and which, if any, should be Apama channels.

For an Apama application to use Universal Messaging, the tasks you must accomplish are:

1. Use Software AG Installer to install both Apama and the Universal Messaging client libraries in the same Software AG installation directory.
2. Plan and implement the configuration of the Universal Messaging cluster that Apama will use. See the [Universal Messaging documentation](#) and “[Setting up Universal Messaging for use by Apama](#)” on page 81.
3. Use Software AG Designer to add one of the Universal Messaging connectivity bundles to your Apama project. For detailed information, see “Adding the Universal Messaging connectivity plug-in to a project” in *Using Apama with Software AG Designer*.

Note:

In addition to using Software AG Designer to add bundles, you can also do this using the `apama_project` command-line tool. See “Creating and managing an Apama project from the command line” in *Deploying and Managing Apama Applications* for more information.

4. Open the `um.properties` file in your Apama project and specify the location of the Universal Messaging realm server(s) you wish to connect to. You can optionally edit the `um.yaml` file if you need to perform more advanced configuration tasks, such as enabling authentication or customizing the way Universal Messaging messages are mapped to Apama events. See “[Configuring the Universal Messaging connectivity plug-in](#)” on page 82 for detailed information.

5. In your EPL code, subscribe to receive events delivered on Universal Messaging channels. See "Subscribing to channels" in *Developing Apama Applications*.

As with all connectivity plug-ins, the EPL application is responsible for telling the system when it is ready to start receiving events with `onApplicationInitialized`. See also ["Sending and receiving events with connectivity plug-ins" on page 38](#).

6. In your EPL code, specify Universal Messaging channels when sending events. See "Generating events with the send statement" in *Developing Apama Applications*.
7. Monitor the Apama application's use of Universal Messaging. See ["Monitoring Apama application use of Universal Messaging" on page 92](#).

Using Universal Messaging channels instead of `engine_connect`

When you are using Universal Messaging channels in an Apama application, you connect multiple correlators by specifying the same Universal Messaging realm when you start each correlator. By using Universal Messaging channels, you probably do not need to use the `engine_connect` tool at all.

While it is possible to configure an Apama application to use both Universal Messaging channels and the `engine_connect` tool, this is not recommended.

Using Universal Messaging channels instead of configuring IAF adapter connections

Note:

Use of Universal Messaging from the IAF is deprecated and will be removed in a future release. It is recommended that you now change any IAF-based adapter configurations using Universal Messaging with a `<universal-messaging>` element in the configuration file to use an `<apama>` element to talk directly to the correlator. See ["Apama correlator configuration" on page 354](#).

In an Apama application, you can use Universal Messaging as the communication mechanism between an IAF adapter and one or more correlators. If you do, then keep in mind the following:

- IAF adapters must send events on named channels. IAF adapters cannot use the default (empty string) channel.
- A service monitor that communicates with an IAF adapter should either be run on only one correlator, or be correctly designed to use multiple correlators. See ["Considerations for using Universal Messaging channels" on page 79](#).

When an IAF 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 Universal Messaging channel. However, even in this situation, it is possible and might be preferable to use a Universal Messaging channel. See ["Comparison of Apama channels and Universal Messaging channels" on page 75](#).

See also: ["Configuring IAF adapters to use Universal Messaging" on page 356](#).

Considerations for using Universal Messaging channels

When using Universal Messaging 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 Universal Messaging message bus can be configured to throttle or otherwise limit events, in which case not all events sent to a channel will be processed.
- Only events can be sent or received by means of Universal Messaging. You cannot use Universal Messaging 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 Universal Messaging 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 tool might be a better choice than using a Universal Messaging channel.
- Universal Messaging channels can be configured for fixed capacity, and that is the default configuration used if the correlator creates a Universal Messaging 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 Universal Messaging channels creating a deadlock.
- When the Universal Messaging channel names are not escaped, it is possible to create or use nested channels. In this case, the slash (/) and backslash (\) characters are treated as path separators on both Windows and Linux.

CAUTION:

Apama treats slash (/) and backslash (\) as different characters while Universal Messaging treats them as identical characters (Universal Messaging generally changes a slash to a backslash). You must choose to use one of these characters in your application and standardize on this. Use of both characters as path separators will result in undefined behavior. In some circumstances, an error message indicating that the user is already subscribed to a channel may be logged when both slashes and backslashes are used.

- It is possible to use the Universal Messaging client libraries (available for Java, C#, C++ and other languages) to send events to or receive events from Apama correlators and adapters.
- Universal Messaging is not used by the following:
 - Apama client library connections.
 - Correlator tools 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 Universal Messaging channel when you use these Apama interfaces. Even though you specify the name of a Universal Messaging channel, Universal Messaging 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 Universal Messaging channels to send events to and receive events from Apama components.

Third-party applications sending and receiving is supported subject to having a suitable chain definition to handle the third-party message format. This is recommended over the JMS integration.

- The name of an Apama channel can contain any UTF-8 character. However, the name of a Universal Messaging channel is limited to the following character set:

0-9

a-z

A-Z

/ (slash, used as path separator when escaping is disabled; do not use both slash and backslash characters within the same application as this will result in undefined behavior - see also the above information)

\ (backslash, used as path separator when escaping is disabled)

#

_ (underscore)

- (hyphen)

Consequently, some escaping is required if Universal Messaging needs to work with an Apama channel name that contains characters that are not supported in Universal Messaging 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 Universal Messaging, for example in a Universal Messaging client application for Java, you will need to consider escaping.

When creating Universal Messaging 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 Universal Messaging channel names. To use those same channels with Universal Messaging, you need to create the channels in Universal Messaging, and when you do, you must escape the unsupported characters.

The escape sequence is the hash (#) symbol, followed by the UTF-8 character number in hexadecimal (lowercase) which again is followed by the hash (#) symbol. For example, the following sequence would be used to escape a period in a channel name:

#2e#

Suppose that in Universal Messaging you want to create a channel whose name in Apama is `My.Channel`. In Universal Messaging, you need to create a channel with the following name:

`My#2e#Channel`

- Universal Messaging supports different protocols. Lower latency can be achieved by using the shm (Shared Memory) protocol if both the correlator and the broker are running on the same host. See the [Universal Messaging documentation](#) for information on how to configure the SHM driver.

Note that the SHM driver keeps a CPU core busy for each end of a connection as it uses spin loops rather than network I/O, which means that two CPU cores are used for each session. As a result, it is recommended to carefully consider and experiment with how many sessions should be used. The default number of sessions is 8, which will typically reduce throughput as it will use too much CPU for Universal Messaging connections.

Setting up Universal Messaging for use by Apama

For Apama to use the Universal Messaging message bus, there are some required Universal Messaging tasks. These steps will be familiar to experienced Universal Messaging users.

Plan and implement the configuration of the Universal Messaging cluster that Apama will use. The recommendation is to have at least three Universal Messaging realms in a cluster because this supports Universal Messaging quorum rules for ensuring that there is never more than one master in a cluster. However, if you can have only two Universal Messaging realms, you can use the `isPrime` flag to correctly configure a two-realm cluster. For details about configuring a Universal Messaging cluster, see the topics in the [Universal Messaging documentation](#) that describe the following:

- Quorum
- Clusters with Sites, which describes an exception to the quorum rule.

To set up Universal Messaging for use by Apama, do the following for each Universal Messaging realm to be used by Apama:

1. Use the Software AG Installer to install Universal Messaging. Make sure to select the option to install the Universal Messaging client libraries.

The Universal Messaging server can be installed on any machine (not necessarily on the same machine as Apama), but you must ensure that the Universal Messaging client libraries are present in the same Software AG installation directory as Apama, as these libraries are required by the Universal Messaging transport connectivity plug-in.

Note:

If you are using JMS or Digital Event Services to access Universal Messaging, installing the client libraries is not required.

2. Start a Universal Messaging server.

3. Use Universal Messaging's Enterprise Manager or Universal Messaging client APIs to set the access control lists of the Universal Messaging server to allow the user that the correlator is running on. See the Universal Messaging documentation for details.
4. By default, Apama automatically creates channels on the Universal Messaging server (that is, the configuration option `missingChannelMode` is set to `create`; see [“Configuring the connection to Universal Messaging \(dynamicChainManagers\)” on page 83](#)).

In production, it is usually better to change `missingChannelMode` to `error` and then add the channels that Apama will use explicitly using Universal Messaging's Enterprise Manager (or the client APIs). In this case, configure the following Universal Messaging channel attributes. Together, these attributes provide behavior similar to that provided by using the Apama correlator tool `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 also the information on 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.

Other channel attributes are allowed. However, it is possible to set Universal Messaging channel attributes in a way that might prevent all events from being delivered to all intended receivers, which includes correlators. For example, Universal Messaging 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 Universal Messaging channel. Consult the Universal Messaging documentation for more details before you set channel attributes that are different from the recommended attributes.

Configuring the Universal Messaging connectivity plug-in

When you have added the Universal Messaging connectivity plug-in to your Apama project in Software AG Designer (see also [“General steps for using Universal Messaging in Apama applications” on page 77](#)), you can edit the YAML configuration file that comes with the plug-in.

The YAML file is configured to do the following:

1. Load the Universal Messaging transport from the `UMConnectivity` library.
2. Under `dynamicChainManagers`, configure one Universal Messaging chain manager to connect to different Universal Messaging realms. Keep in mind that you can have only one chain manager here.
3. Configure one or more `dynamicChains` to handle transforming messages from Universal Messaging into the correlator.

Detailed information for this and much more is given in the topics below.

You can have different `dynamicChains` processing messages on different channels in different formats.

Connection-related configuration is specified in the `managerConfig` stanza on the `dynamicChainManagers` instance, including the `rnames` connection string for Universal Messaging.

Per-channel configuration of how to parse received messages is configured via the individual `dynamicChains`.

Selection of which chain a manager is to use for a given channel name is done via a `channelPattern` stanza on the `UMTransport` for each chain.

By default, the `UM` chain manager listens for subscribe and send-to requests in your EPL. It subscribes to channels with a specific prefix in Apama (by default, this is `um:`) and connect that to the corresponding Universal Messaging channel without the prefix. Therefore, EPL subscribing or sending to `um:channelName` will subscribe or send to the Universal Messaging channel `channelName`. The prefix and whether it is included on Universal Messaging can be configured via the `managerConfig`.

For more information on YAML configuration files, see [“Using Connectivity Plug-ins” on page 23](#) and “Configuring the correlator” in *Deploying and Managing Apama Applications*.

Loading the Universal Messaging transport

The Universal Messaging transport is loaded with the following `connectivityPlugins` stanza:

```
connectivityPlugins:
  UMTransport:
    libraryName: UMConnectivity
    class: UMTransport
```

Configuring the connection to Universal Messaging (dynamicChainManagers)

The Universal Messaging dynamic chain manager monitors channels that are subscribed to or sent-to within the correlator and, as per its configuration, connects these to Universal Messaging channels as needed. This is where you specify how to connect to Universal Messaging. When the dynamic chain manager identifies a channel that should be connected to a Universal Messaging channel, it will create a chain using one of the templates in the `dynamicChains` section of the YAML configuration (see also [“Configuring message transformations \(dynamicChains\)” on page 87](#)).

One `UM` dynamic chain manager is created under `dynamicChainManagers` which specifies the Universal Messaging realm or cluster of realms you are connecting to. The `managerConfig` stanza contains all of the configuration for connecting to that Universal Messaging realm.

```
dynamicChainManagers:
  UM:
    transport: UMTransport
    managerConfig:
      rnames: nsp://127.0.0.1:9000
```

```
session:
  poolSize: 8
certificateAuthorityFile: /mypath/my_UM_certificate
authentication:
  username: me
  certificateFile: /mypath/my_client_certificate
  certificatePassword: mycertificatepassword
channel:
  prefix: "um:"
  missingChannelMode: create
  escapeNamesOnUM: true
  includePrefixOnUM: false
createChannelPermissions:
  user@host:
    - Manage ACL
    - Full
  "*@*":
    - Publish
    - Subscribe
    - Last EID
  Everyone:
    - Purge
```

The following table describes the options that can be used in the `managerConfig` section:

Option	Description
<code>rnames</code>	<p>Required. The Universal Messaging connection string of realm names to connect to.</p> <p>You can specify one or more Universal Messaging realm names (RNAME) separated by commas or semicolons.</p> <p>Commas indicate that you want the correlator to try to connect to the Universal Messaging realms in the order in which you specify them here. For example, if you have a preferred local server you could specify its associated realm name first and then use commas as separators between specifications of other realm names, which would be connected to if the local server is down.</p> <p>Semicolons indicate that the correlator can try to connect to the specified Universal Messaging 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.</p> <p>You can specify multiple Universal Messaging realms only when they are connected in a single Universal Messaging cluster. That is, all realm names you specify must belong to the same Universal Messaging cluster. Since channels are shared across a cluster, connecting to more than one Universal Messaging realm lets you take advantage of Universal Messaging's failover capability.</p>

Option	Description
	<p>For additional information on communication protocols and realm names, and on clusters, see the Universal Messaging documentation.</p> <p>Default value: none.</p>
<code>session/poolSize</code>	<p>The number of sessions (connections) to create to the Universal Messaging realm. Either a number of sessions or a string of the form <code>/N</code> which indicates a divisor applied to the number of CPUs to obtain the number of sessions. Channels are allocated to sessions in a round-robin style.</p> <p>Default value: 8.</p>
<code>certificateAuthorityFile</code>	<p>The path to a CA certificate for the Universal Messaging realm. This is used by the Apama to confirm the identity of the Universal Messaging realm server.</p> <p>Default value: none.</p>
<code>authentication/username</code>	<p>The username that is to be used to connect to the Universal Messaging realm. The default is the user which the correlator is running as.</p> <p>Default value: none.</p>
<code>authentication/certificateFile</code>	<p>The path to a certificate that is used by the Universal Messaging realm server to authenticate this client.</p> <p>Default value: none.</p>
<code>authentication/certificatePassword</code>	<p>The password for the certificate.</p> <p>Default value: none.</p>
<code>channel/prefix</code>	<p>A prefix for the channel. Only channels with this prefix will be considered as Universal Messaging channels. If the prefix ends with a colon (:), it needs to be enclosed in quotation marks (see also "Using YAML configuration files" in <i>Deploying and Managing Apama Applications</i>).</p> <p>Default value: <code>"um:"</code>.</p>
<code>channel/includePrefixOnUM</code>	<p>If set to <code>false</code>, the channel prefix is stripped from the Apama channel name before it is looked up on Universal Messaging.</p> <p>Default value: <code>false</code>.</p>
<code>channel/escapeNamesOnUM</code>	<p>If set to <code>true</code>, non-alphanumeric characters in the Apama channel name are escaped on Universal Messaging. Set this</p>

Option	Description
	<p>to false if you want to use the slash (/) for hierarchical channels on Universal Messaging.</p> <p>Default value: true.</p>
channel/missingChannelMode	<p>Defines the behavior when subscribing to channels which do not exist. You can define one of the following options:</p> <ul style="list-style-type: none">■ error - Print an error to the correlator log file. The channel remains only accessible within the correlator.■ ignore - Silently ignore the failure and therefore do not print an error to the correlator log. The channel remains only accessible within the correlator.■ create - Create the channel on the Universal Messaging realm, then subscribe/send to it. See also “Subscribing to automatically created Universal Messaging channels” on page 89. <p>Default value: create.</p>
createChannelPermissions	<p>Defines the ACL (access control list) permissions for automatically created channels in the following format:</p> <pre>client: - permission1 - permission2 - ...</pre> <p>The <i>client</i> can be either a subject (of the format <i>user@host</i>) or a group. Universal Messaging supports the * wildcard for representing all users/hosts. You can specify the following permissions:</p> <ul style="list-style-type: none">■ Manage ACL■ Full■ Publish■ Subscribe■ Purge■ Last EID <p>For additional information on subject, group, wildcards and permission ACLs, see the Universal Messaging documentation.</p> <p>Default:</p>

Option	Description
	<pre>creating-user@*: - Subscribe - Publish - Last EID</pre>
	<p>Note:</p> <p>Apama clients require Last EID permission in addition to Subscribe permission for a client to subscribe to a channel.</p> <p>Setting channel permissions correctly is important to protect the security of your application, and also to protect any personal data included in the messages. For more information, see "Protecting Personal Data in Apama Applications" in <i>Developing Apama Applications</i>.</p>

Configuring message transformations (dynamicChains)

dynamicChains contains templates of chains. The Universal Messaging manager will pick a chain template that ends with the UMTransport plug-in, and use that configuration. The manager also provides configuration from the channel which the chain can use with `@{varname}` substitutions (see [“Using dynamic replacement variables” on page 88](#)). The manager uses the `channelPattern` property of the UMTransport configuration to decide which chain template should be used for a given channel.

With either the `apama.eventMap` or `apama.eventString` host plug-ins, we recommend use of the `suppressLoopback` configuration property to avoid messages which are sent to that channel being delivered internally as well as being sent to and received from Universal Messaging. We also recommend setting the `description` and `remoteAddress` properties in order to improve logging and debugging. See [“Host plug-ins and configuration” on page 30](#) for more information.

The following chain sends and receives events with the members of the event being set in the `nEventProperties` of the Universal Messaging events and an empty payload.

```
dynamicChains:
  UMChain:
    - apama.eventMap:
        suppressLoopback: true
        description: "@{um.rnames}"
        remoteAddress: "@{um.rnames}"
    - UMTransport:
        setTypeFromTag: true
        channelPattern: ".*"
```

The following options can be used with UMTransport:

Option	Description
setTypeFromTag	<p>If set to <code>true</code>, the Universal Messaging message tag is translated to the <code>sag.type</code> metadata field, if the Universal Messaging tag is present.</p> <p>If desired, the Mapper codec (see “The Mapper codec connectivity plug-in” on page 217) can be used to set a different value for <code>metadata.sag.type</code> when sending messages towards the Universal Messaging transport in order to use a string other than the Apama event type for the Universal Messaging tag. When receiving messages from the Universal Messaging transport, the Classifier codec (see “The Classifier codec connectivity plug-in” on page 216) can be used to set the Apama event type to be used if the incoming Universal Messaging messages do not specify an Apama event type in their “tag”.</p> <p>For performance-critical applications where the event type is known or can be set in the chain, we recommend setting this option to <code>false</code>.</p> <p>Default value: <code>true</code>.</p>
channelPattern	<p>Required. A regular expression that is used to select which chain is used for which channel.</p> <p>Only one chain definition may match any channel, except for the “fallback” definition with the channel pattern <code>".*"</code>, which will be used if no other patterns match.</p> <p>Default value: <code>none</code>.</p>

Using dynamic replacement variables

The `UMTransport` provides the following dynamic replacement variables which can be used with the `@{varname}` replacement syntax:

Variable	Description
<code>um.channelName</code>	The name of the Universal Messaging channel.
<code>um.rnames</code>	The address of the Universal Messaging realm.
<code>um.channelCapacity</code>	The maximum capacity of the Universal Messaging channel.

For example, for using all channels as mapping directly to `nEventProperties`:

```
UMMessagePropertiesChain:
- apama.eventMap:
  suppressLoopback: true
  description: "@{um.rnames}"
```



```
remoteAddress: "@{um.rnames}"
- UMTransport:
  channelPattern: ".*"
```

Subscribing to automatically created Universal Messaging channels

By default, the configuration option `missingChannelMode` is set to `create` so that a Universal Messaging channel can be automatically created if it does not already exist when an Apama application needs to use it. See also [“Configuring the connection to Universal Messaging \(dynamicChainManagers\)” on page 83](#).

When a Universal Messaging channel is automatically created, which is helpful for getting started, it has the attributes described in [“Setting up Universal Messaging for use by Apama” on page 81](#). If you want a Universal Messaging channel to have any other attributes, then you must create the channel in Universal Messaging before any Apama component sends to or subscribes to the channel.

In production, it is usually better to change `missingChannelMode` to `error` and to configure the channels explicitly as described in [“Setting up Universal Messaging for use by Apama” on page 81](#).

You can specify the ACL (access control list) permissions for the channel being created in a YAML configuration file using the `createChannelPermissions` option. See [“Configuring the connection to Universal Messaging \(dynamicChainManagers\)” on page 83](#).

Channel lookup caching

After Apama looks up a channel name to determine whether it is a Universal Messaging channel, Apama caches the result and does not look it up again. Consequently, the following situation is possible:

1. You use Universal Messaging interfaces to create channels.
2. You start a correlator with `missingChannelMode` set to `ignore`.
3. Apama looks up, for example, `channelA` and determines that it is not a Universal Messaging channel.
4. You use Universal Messaging interfaces to create, for example, `channelA`.

For Apama to recognize `channelA` as a Universal Messaging channel, you must either restart the correlator or issue a `flushChannelCache` request using the `engine_management` tool (see also [“Shutting down and managing components” in *Deploying and Managing Apama Applications*](#)):

```
engine_management -r flushChannelCache
```

This operation may take a long time since it verifies the existence of every channel subscribed to in the correlator on Universal Messaging. Therefore, we recommend that you ensure all your channels have been created on Universal Messaging before starting your Apama application.

Supported payloads

The Universal Messaging transport supports different types of Apama message payload:

- **Binary payloads (Java `byte[]` or C++ `buffer_t`)**. Apama messages with binary payloads are mapped to the data payload of a Universal Messaging message. Optionally, Universal Messaging message properties/headers (`nEventProperties`) may be mapped using metadata values in string form with the prefix `um.properties`. You can also map all of the Universal Messaging message properties to an EPL dictionary with a Mapper codec rule that moves `metadata.um.properties` in its entirety; see also [“The Mapper codec connectivity plug-in” on page 217](#). If you wish to put a string into the Universal Messaging message data payload, use the String codec to convert it into binary form (as UTF-8); see also [“The String codec connectivity plug-in” on page 210](#).
- **Map payloads (Java `Map` or C++ `map_t`)**. Apama messages with a map payload are mapped to the Universal Messaging message properties/headers (`nEventProperties`), and the Universal Messaging message data payload is empty.

If you are sending and receiving using the `eventMap` host plug-in (see also [“Translating EPL events using the `apama.eventMap` host plug-in” on page 32](#)), you probably want to make use of the Mapper and Classifier codecs (see [“Codec Connectivity Plug-ins” on page 209](#)), unless the Apama event type name is stored in the “tag” of the Universal Messaging messages. Typically, the Apama event format does not match exactly to the format you want in the `nEventProperties`, and the Mapper codec allows you to fix that.

When setting `nEventProperties` either from the map payload or via `um.properties` metadata values, the following EPL types are unsupported and sending events to Universal Messaging will therefore fail:

- `dictionary<>` types with keys which are not Apama primitives (that is, anything except integer, boolean, decimal, float, string). For example, `dictionary<Sequence<some type>, String>` is not supported, but `dictionary<decimal, <sometype>>` is supported.
- Apama decimal type and `dictionary<>` keys are stringified when sending the events. That also means that `sequence <decimal>` is sent as a sequence of strings.
- Sequences within sequences. For example, `sequence<sequence<any type>>`. Note that `sequence <Apama Event>` or `sequence <dictionary<some primitive, any type>>` are supported.

The binary payload may represent a string. If this is a case, then the binary payload must be converted to a string payload before further processing can happen as a string. To do this, use the String codec. This converts binary payloads to string payloads for hostward messages and string payloads to binary payloads for transportward messages. The String codec should be the last codec in the chain. See [“The String codec connectivity plug-in” on page 210](#) for detailed information.

You can also use other codecs such as the JSON codec (see [“Codec Connectivity Plug-ins” on page 209](#) for more information). For example:

```
dynamicChains:
  UMJsonChain:
    - apama.eventMap:
        suppressLoopback: true
```

```

        description: "@{um.rnames}"
        remoteAddress: "@{um.rnames}"
    - jsonCodec
    - stringCodec
    - UMTTransport:
        channelPattern: ".*"

```

EPL and Universal Messaging channels

In an Apama application that is configured to use Universal Messaging, 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 Universal Messaging channel. You must specify a Universal Messaging channel name to use Universal Messaging.

As with all connectivity plug-ins, the EPL application is responsible for telling the system when it is ready to start receiving events with `onApplicationInitialized`. See also [“Sending and receiving events with connectivity plug-ins” on page 38](#).

A monitor that subscribes to a Universal Messaging 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 Universal Messaging channels makes it easier to scale an application across multiple correlators because Universal Messaging channels can automatically connect parts of the application as required. If you use the EPL `integer.incrementCounter("UM")` 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.incrementCounter("UM")` with the correlator's physical ID. Obtain the physical ID from Apama's Management interface with a call to the `getComponentPhysicalId()` method. For further information, see "Using the Management interface" in *Developing Apama Applications*.

Using Universal Messaging connectivity from EPL

In EPL, in order to receive events from a Universal Messaging channel, you just need to subscribe to a channel with the appropriate prefix:

```

on all EventTypeOnUM() { ... }
monitor.subscribe("um:UMChannelName");

```

This creates a chain with a channel pattern matching `um:UMChannelName` and subscribe to `UMChannelName` on the connected realm. Events from that channel are delivered to the context after being parsed by the chain.

To send to a Universal Messaging channel, you just need to use the `send...to` statement to deliver an event to that channel name:

```

send EventTypeOnUM() to "um:UMChannelName";

```

This will use the same chain definition as above to deliver the mapped event to the Universal Messaging channel `UMChannelName`.

The `samples/connectivity_plugin/application/genericsendreceive` directory of your Apama installation includes a simple sample which provides an easy way to get started with sending and receiving messages to or from any connectivity plug-in. For more information, see the `README.txt` file in the above directory and [“Sending and receiving events with connectivity plug-ins” on page 38](#).

Monitoring Apama application use of Universal Messaging

You can use the Universal Messaging Enterprise Manager or Universal Messaging APIs to find out about the following:

- Which correlators are subscribed to which Universal Messaging channels.
- The number of events flowing through a Universal Messaging channel.
- The contents of the events going through a Universal Messaging channel.

See the [Universal Messaging documentation](#) for more information on the Enterprise Manager.

To monitor and manage Apama components, you must use Apama tools and APIs.

5 The MQTT Transport Connectivity Plug-in

■ About the MQTT transport	94
■ Using MQTT connectivity from EPL	94
■ Loading the MQTT transport	95
■ Configuring the connection to MQTT	95
■ Mapping events between MQTT messages and EPL	97
■ Payload for the MQTT message	98
■ Wildcard topic subscriptions	98
■ Metadata for the MQTT message	98
■ Restrictions	99

About the MQTT transport

MQTT is a publish/subscribe-based "lightweight" message protocol designed for communication between constrained devices, for example, devices with limited network bandwidth or unreliable networks. See <http://mqtt.org/> for detailed information.

Note:

While it is possible to use MQTT to communicate between Apama and Cumulocity IoT, we recommend using the Cumulocity IoT transport connectivity plug-in provided with Apama. See [“The Cumulocity IoT Transport Connectivity Plug-in” on page 163](#) for detailed information.

Apama provides a connectivity plug-in, the MQTT transport, which can be used to communicate between the correlator and an MQTT broker, where the MQTT broker uses topics to filter the messages. MQTT messages can be transformed to and from Apama events by listening for and sending events to channels such as `prefix:topic` (where the prefix is configurable).

The MQTT transport automatically reconnects in case of a connection failure. The transport will retry sending any messages sent after the connection has been lost when reconnection has succeeded.

You configure the MQTT connectivity plug-in by editing the files that come with the **MQTT** bundle. The properties file defines the substitution variables that are used in the YAML configuration file which also comes with the bundle. See "Adding the MQTT connectivity plug-in to a project" in *Using Apama with Software AG Designer* for further information.

Note:

In addition to using Software AG Designer to add bundles, you can also do this using the `apama_project` command-line tool. See "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications* for more information.

This transport provides a dynamic chain manager which creates chains automatically when EPL subscribes or sends to a correlator channel with the configured prefix, typically `mqtt:.` For the MQTT transport, there must be exactly one chain definition provided in the `dynamicChains` section of the YAML configuration file.

For more information on YAML configuration files, see [“Using Connectivity Plug-ins” on page 23](#) and especially [“Configuration file for connectivity plug-ins” on page 26](#).

Note:

The MQTT connectivity plug-in does not support reliable messaging.

Using MQTT connectivity from EPL

The MQTT transport can either subscribe to or send to a particular topic, depending on whether your EPL is subscribing to or sending to a particular channel.

In EPL, in order to receive an MQTT message, you just need to subscribe to an MQTT topic with the appropriate prefix. For example:

```
monitor.subscribe("mqtt:topic_a");
```

```
on all A() as a {
    print a.toString();
}
```

To send an Apama event to the MQTT broker, you just need to use the `send...to` statement to deliver the event to the MQTT topic. For example:

```
send A("hello world") to "mqtt:topic_a";
```

As with all connectivity plug-ins, the EPL application is responsible for telling the system when it is ready to start receiving events with `onApplicationInitialized`. See also [“Sending and receiving events with connectivity plug-ins” on page 38](#).

The `samples/connectivity_plugin/application/genericsendreceive` directory of your Apama installation includes a simple sample which provides an easy way to get started with sending and receiving messages to or from any connectivity plug-in. For more information, see the `README.txt` file in the above directory and [“Sending and receiving events with connectivity plug-ins” on page 38](#).

Loading the MQTT transport

The MQTT transport is loaded with the following `connectivityPlugins` stanza:

```
mqttTransport:
  libraryName: connectivity-mqtt
  class: MQTTTransport
```

Configuring the connection to MQTT

You configure one or more `dynamicChainManagers` to connect to different MQTT brokers. For example:

```
dynamicChainManagers:
  mqttManager:
    transport: mqttTransport
    managerConfig:
      brokerURL: tcp://localhost:1883
```

Connection-related configuration is specified in the `managerConfig` stanza on the `dynamicChainManagers` instance. The following configuration options are available for `managerConfig`:

Configuration option	Description
<code>brokerURL</code>	<p>URL for the MQTT broker.</p> <p>For example, you can use the following URL for non-TLS connections:</p> <pre>tcp://localhost:1883</pre> <p>To enable SSL/TLS, you simply indicate this in the broker URL. For example:</p>

Configuration option	Description
	ssl://localhost:8883
	Type: string.
channelPrefix	<p>Prefix for dynamic mapping. If the prefix ends with a colon (:), it needs to be enclosed in quotation marks (see also "Using YAML configuration files" in <i>Deploying and Managing Apama Applications</i>).</p> <p>When the channel is mapped to an MQTT topic, the prefix is not used. For example, if the prefix is "mqtt:", then the channel <code>mqtt:test/a</code> maps to the MQTT topic <code>test/a</code>.</p> <p>Type: string.</p> <p>Default: "mqtt:".</p>
mqttClientId	<p>Optional. By default, a random client identifier is generated during startup. If you do not want to use this random identifier, you can set this option to configure your own client identifier. This can be any alphanumeric value.</p> <p>Type: string.</p>
cleanSession	<p>Starts a clean session with the MQTT broker. Set this to <code>false</code> if the previous session is to be resumed. You should only do this in conjunction with setting the <code>mqttClientId</code>.</p> <p>Type: bool.</p> <p>Default: <code>true</code>.</p>
acceptUnrecognizedCertificates	<p>Used with TLS. By default, connections to unrecognized certificates are terminated. Set this to <code>true</code> if non-validated server certificates are to be accepted.</p> <p>Type: bool.</p> <p>Default: <code>false</code>.</p>
certificateAuthorityFile	<p>Used with TLS. By default, server certifications signed by all standard Certificate Authorities are validated. Optionally, you can set this option to provide a path to a CA certificate file in PEM format to authenticate the host with.</p> <p>Type: string.</p>

Configuration option	Description
authentication/username	User name for authentication. Type: string.
authentication/password	Password for authentication. Type: string.
authentication/certificateFile	Used by TLS. Optionally, you can set this option to provide a path to a CA certificate file in PEM format to authenticate the client with. Type: string.
authentication/certificatePassword	Used by TLS. Optional password used to decrypt the client private key file, if encrypted. Type: string.
authentication/privateKeyFile	Used by TLS. Optional path to a PEM file containing the private key, if not already included in the certificate file. Type: string.

Important:

If you provide a password for authentication via the configuration file, you must ensure to protect the configuration file against any unauthorized access, since the password will be readable in plain text.

Mapping events between MQTT messages and EPL

You can use the `apama.eventMap` host plug-in in a dynamic chain to translate events to or from nested messages like JSON data. You configure exactly one `dynamicChains` section to handle transforming messages from the MQTT broker into the correlator, and vice versa.

The following description shows how to configure the `genericSendreceive` sample to send/receive data to/from an MQTT broker. The advantage of this sample is that all required EPL code is already available (see also [“Writing EPL” on page 21](#)). The sample is located in the `samples/connectivity_plugin/application/genericSendreceive` directory of your Apama installation. To use the sample, do the following:

1. Import the sample into Software AG Designer as an existing project (make sure to create a copy).
2. Add the MQTT connectivity plug-in to that project (see also "Adding the MQTT connectivity plug-in to a project" in *Using Apama with Software AG Designer*) and edit the `MQTT.yaml` file as per your application.

3. Configure both the input and output channels to apamax by sending the `ConfigureSample` event (in order to send continuous data, `keepSending` must be set to `true`).
4. Send the `AppReady` event to start the application.

We recommend use of the `suppressLoopback` configuration property to prevent undesirable behavior. See [“Host plug-ins and configuration” on page 30](#) for further information.

Payload for the MQTT message

As with all other transports, the translation between EPL events and MQTT payloads is based on the choice of host plug-in and codecs. See [“Host plug-ins and configuration” on page 30](#) and [“Codec Connectivity Plug-ins” on page 209](#) for further information.

The payload for the MQTT message is a byte array. Therefore, the String codec should usually be used to convert a `byte[]` (Java) or `buffer_t` (C++) type payload into a hostward string event. The same String codec can also be used to convert a string event to a transportward message with a `byte[]` or `buffer_t` type.

Wildcard topic subscriptions

MQTT supports a hierarchical topic namespace and allows you to subscribe to every topic in a namespace using a wildcard symbol such as `#`. Any MQTT messages that are sent to the broker and that satisfy the topic namespace are sent to the correlator.

A potential result of this may be that a single MQTT message that is sent to the broker is received more than once by the correlator. For example, assume that Apama subscribes to both of the following channels:

```
"mqtt:SENSOR/#"  
"mqtt:SENSOR/1"
```

If a single MQTT message is sent to the broker using the topic name `SENSOR/1`, then this MQTT message will be received *twice* by the correlator. You should be aware of such situations and write your EPL accordingly to handle this.

Metadata for the MQTT message

Messages coming from the transport have useful pieces of information inserted into their metadata. This information is stored as a map associated with the `mqtt` key. This map contains the following information:

Field	Description
<code>metadata.mqtt.topic</code>	Contains the full name of the topic from which the message originated. This allows you to differentiate between messages coming from different sources in the case of a transport subscribed with a wildcard.

Restrictions

Not all MQTT features are supported by the MQTT transport. The following features are not supported:

- Reliable messaging, that is, session persistency and QoS (Quality of Service) level greater than 0.
- Retained messages.
- Last will and testament options.

6 The Digital Event Services Transport Connectivity Plug-in

■ About the Digital Event Services transport	102
■ Using Digital Event Services connectivity from EPL	103
■ Reliable messaging with Digital Event Services	104

About the Digital Event Services transport

Software AG Digital Event Services is a messaging system for communicating between different Software AG products using events. Digital Event Services allows event definitions to be converted between a product's internal event or document definition to digital event types and vice versa, so participating products can share a set of event definitions. When you develop Apama applications that make use of Digital Event Services, the translation between digital event type definitions and Apama event types is done automatically. When digital events are sent to or received from Digital Event Services, they are converted to or from Apama events.

For details of the event mapping, see the `.mon` source file that is generated into the **autogenerated** node of your Apama project in Software AG Designer. Note that digital event types that contain nested events (or sequences of nested events) are converted to Apama event definitions that have an optional member (or sequence of optionals) for that event type. See the description of the `optional` type in the *API Reference for EPL (ApamaDoc)* for more information. When digital events are converted to Apama events, fields of other types with no value set are set to the default value for that type (see also "Default values for types" in *Developing Apama Applications*).

To use digital event types in your Apama application, proceed as follows:

1. Use Software AG Installer to install Digital Event Services. See *Installing Software AG Products* for more information.
2. Configure Digital Event Services as described in *Using Digital Event Services to Communicate between Software AG Products*. This guide also explains how to use SSL with Digital Event Services.
3. Use Software AG Designer to add the **Digital Event Services** connectivity bundle to your Apama project. For detailed information, see "Using the Digital Event Services connectivity bundle" in *Using Apama with Software AG Designer*.

Note:

In addition to using Software AG Designer to add bundles, you can also do this using the `apama_project` command-line tool. See "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications* for more information.

4. Use the Digital Event Types editor to turn digital event types into Apama event types and existing Apama event types into digital event types. Apama event types that correspond to digital event types can be used just like ordinary Apama events. You can create digital events, create event expressions for them, set and get their contents, pass them around internally between monitors and contexts, and much more. See the above-mentioned topic for more information.
5. This transport provides a dynamic chain manager which creates chains automatically when EPL subscribes or sends to a correlator channel with a name corresponding to the `.CHANNEL` constant on a Digital Event Services event. For this transport, there is no need to customize the chain configuration in the YAML configuration file in any way.

For more information on YAML configuration files, see ["Using Connectivity Plug-ins" on page 23](#) and especially ["Configuration file for connectivity plug-ins" on page 26](#).

6. Edit the `DigitalEventServices.properties` file to configure the Digital Event Services connectivity plug-in which defines the Apama connection to Digital Event Services. See the above-mentioned topic for more information.

Using Digital Event Services connectivity from EPL

As with all connectivity plug-ins, the EPL application is responsible for telling the system when it is ready to start receiving events with `onApplicationInitialized`. See also [“Sending and receiving events with connectivity plug-ins” on page 38](#).

Each Digital Event Services event type maps to its own dedicated Apama channel. The channel name is accessed via the static `CHANNEL` constant on the EPL type. With `CHANNEL`, you can send, subscribe and unsubscribe in the same way as any other Apama channel.

For example, you have a digital event type called `pkg1.pkg2.MyEvent`, with fields `anInteger` and `aString` on it. If you select this type in Software AG Designer, you are able to write EPL such as the following:

```
using com.softwareag.connectivity.ConnectivityPlugins;
using pkg1.pkg2.MyEvent;
...

ConnectivityPlugins.onApplicationInitialized();
monitor.subscribe(MyEvent.CHANNEL); // This context will now receive digital
                                     // events of type 'pkg1.pkg2.MyEvent'

on all MyEvent() as e {
    print "Got an event from DES: " + e.toString();
}

MyEvent e2 := new MyEvent;
e2.anInteger := 100;
e2.aString := "Hello world!";
send e2 to MyEvent.CHANNEL; // Sends this event out to Digital Event Services
```

There is one thing about digital events that is totally different from Apama events. Digital events of different types are not guaranteed to be received by Digital Event Services in the same order as they were sent. When you are sending Apama events as shown in the following example, then it is guaranteed that the destination context (`ctx`) can see the A and B events in the same order as they were sent.

```
send A(1) to ctx;
send B(1) to ctx;
send A(2) to ctx;
send B(2) to ctx;
```

If you are sending digital events as shown in the example below, any other product (or even Apama) that is receiving these events from Digital Event Services is guaranteed to see A(2) after A(1), and B(2) after B(1). These A and B events, however, might (or might not) be interleaved differently.

```
send A(1) to A.CHANNEL;
send B(1) to B.CHANNEL;
send A(2) to A.CHANNEL;
```

```
send B(2) to B.CHANNEL;
```

If you want to deploy or export an Apama application which uses digital events (for example, using Command Central or an Ant script) to another machine (for example, from development to production), keep in mind that you also have to deploy the digital event type repository from one machine to the other. The same type repository on which you have developed your application needs to be available in all the places in which you run your application. For more information, see *Using Digital Event Services to Communicate between Software AG Products*.

Reliable messaging with Digital Event Services

Digital Event Services offers reliability with only a couple of small requirements:

- **Reliable sending.** The delivery mode of the Digital Event Services event type must be *persistent*. This allows you to perform flush operations for that event type.
- **Reliable receiving.** The delivery mode of the Digital Event Services event type must be *persistent*. In addition, a `subscriberId` must be set in the configuration file. This requires that you perform acknowledgments for events of that type.

For detailed information on how to configure the delivery mode, see *Using Digital Event Services to Communicate between Software AG Products*.

See the properties file `DigitalEventServices.properties` for information on the `subscriberId` and other configuration options. For more detailed information on using reliable messaging in general, see [“Using reliable transports” on page 33](#).

Shared durable subscribers

The Digital Event Services transport makes use of shared durable subscribers for reliable receiving. When a single correlator is connected with a particular `subscriberId`, the correlator will receive and acknowledge events. Events are resent after a failure once the failed component has been restarted/reconnected. With multiple correlators sharing the same `subscriberId`, events are delivered in a round-robin fashion to each available receiver.

If all subscribed monitor instances explicitly unsubscribe from a type, or if those monitor instances terminate, then that does not count as a failure. Any events of this type that are sent afterwards will not be received, and will not be resent upon resubscription.

CAUTION:

In a system with multiple correlators sharing the same `subscriberId`, an explicit unsubscribe from one correlator will unsubscribe the other correlators from that type.

Reliable receiving

The `CHANNEL` constant on the auto-generated EPL type allows you to find the connectivity chain (`Chain`) used for receiving events of this type, so that the `Chain` can be used for reliable messaging operations.

Example:


```

using com.softwareag.connectivity.Chain;
using com.softwareag.connectivity.ConnectivityPlugins;
using com.softwareag.connectivity.Direction;
using com.softwareag.connectivity.control.AckRequired;
using a.pkg.myEvent; // our auto-generated EPL type from DES

monitor receiver
{
    action onload()
    {
        monitor.subscribe(myEvent.CHANNEL);
        Chain c := ConnectivityPlugins.getChainByChannel(myEvent.CHANNEL,
            Direction.TOWARDS_HOST);
        on all myEvent() as ev
        {
            // process ev
        }
        on all AckRequired(chainId=c.getId()) as ar
        {
            // make sure all events before the AckRequired have been fully processed
            ...
            // and only acknowledge them once that is done
            ar.ackUpTo();
        }
    }
}

```

If you need the message identifier of an event for doing per-event acknowledgment (`Chain.ackUpTo`), then it will be available as a field on the event. The specific field will be called out in the `MessageId` annotation of the auto-generated EPL representation of the Digital Event Services type.

Reliable sending

Again, the `CHANNEL` constant allows you to find the connectivity chain (`Chain`) used for sending events of this type, so that the `Chain` can be used for reliable messaging operations.

Important:

When using reliable sending, the Digital Event Services storage location must be in a safe place, as events are acknowledged after they have been persisted to disk, but before they are sent to the remote system. This does not apply if the store-and-forward queue has been disabled, in which case events are acknowledged only once they have been committed to the remote system.

Example:

```

using com.softwareag.connectivity.Chain;
using com.softwareag.connectivity.ConnectivityPlugins;
using com.softwareag.connectivity.Direction;
using com.softwareag.connectivity.control.FlushAck;
using a.pkg.myEvent; // our auto-generated EPL type from DES

monitor sender
{
    action onload()
    {
        Chain c := ConnectivityPlugins.getChainByChannel(myEvent.CHANNEL,
            Direction.TOWARDS_TRANSPORT);
        on all wait (0.1)
    }
}

```

```
{
  send myEvent("hello") to myEvent.CHANNEL;
  // flush after each send and listen for the acknowledgment
  on FlushAck(requestId = c.flush()) as fa
  {
    // event acknowledged, we no longer need to hold on to it
  }
}
}
```

7 The HTTP Server Transport Connectivity Plug-in

■ About the HTTP server transport	108
■ Loading the HTTP server transport	110
■ Configuring the HTTP server transport	110
■ Handling responses in EPL	114
■ Serving static files	116
■ Mapping events between EPL and HTTP server requests	116
■ HTTP server security	126
■ Monitoring status for the HTTP server	128

About the HTTP server transport

The HTTP server is a transport for use in connectivity plug-ins which external services can connect to over HTTP/REST. It can handle both an HTTP submission-only API which delivers events to the correlator and a REST request/response API where the responses are controlled from EPL. In addition to this, it can serve static files. It also allows support for TLS alongside HTTP basic authentication.

The HTTP server transport can decode HTTP requests and encode EPL responses or static files with gzip or deflate compression format. It also supports HTML form decoding and can decode multipart/form-data or application/x-www-form-urlencoded media types to a dictionary payload.

This transport provides a dynamic chain manager (the chain manager for each host/port is configured by an entry under `dynamicChainManagers` in the YAML configuration file) which creates chains automatically whenever an HTTP client connects to that host/port. For the HTTP server transport, there must be exactly one chain definition provided in the `dynamicChains` section of the YAML configuration file. The EPL channel that incoming requests are sent to is specified in the configuration of the `dynamicChains`, by rules in the `mapperCodec` section that set the `metadata.sag.channel`.

HTTP requests are received by the transport and sent to the chain where they are mapped to EPL events as described in [“Mapping events between EPL and HTTP server requests” on page 116](#). Whether the response to the HTTP request is generated automatically or by the EPL application is controlled as described in [“Handling responses in EPL” on page 114](#).

For more information on YAML configuration files, see [“Using Connectivity Plug-ins” on page 23](#) and especially [“Configuration file for connectivity plug-ins” on page 26](#).

Persistent connections to the server are supported for multiple requests. Details of the individual requests are configured through the events sent to the chain. The HTTP server supports HTTP version 1.1 and TLS version 1.2 and above.

The HTTP server is designed to listen for REST services and supports all GET, POST, PUT and DELETE operations which have been specified in the configuration file. Other than GET requests served by static files, all requests are treated identically.

The `samples/connectivity_plugin/application/httpserver` directory of your Apama installation includes a sample which demonstrate how to use the HTTP server connectivity plug-in to send and receive HTTP requests containing events into the correlator through various configurations. See the `README.txt` file included with the sample for complete instructions on how to run the sample application.

Note:

The HTTP server connectivity plug-in does not support reliable messaging.

OpenAPI definitions

OpenAPI is an open description format for REST APIs. The OpenAPI Specification (OAS), and the related tools available from Swagger (<https://swagger.io>), can be used to design, document,

deploy and test the REST API for an application. The specification allows for API definitions to be written in either YAML or JSON.

Apama API definitions are supplied in JSON format to the OpenAPI/Swagger 3.0 specification.

HTTP response codes

The transport returns a response to the client. If responses are automatically generated, we return a “202 Accepted” response after HTTP parsing, but before processing by the correlator, to indicate that a failure may still occur later in processing the event. If the response is handled by EPL, the response code is defined by the EPL application and configuration. If there is a failure in parsing the HTTP part of the request, an error code is returned instead.

The various response codes that we currently support are described below.

Code	Reason
202 Accepted	Success response code for automatic responses. On a successful submission, this indicates that while we have accepted it, processing will occur later and we cannot guarantee completion.
400 Bad Request	Any other error we can conclusively say is due to a malformed request.
401 Unauthorized	We have enabled HTTP basic authentication and the user either does not supply an <code>Authorization</code> header or it is incorrect.
405 Method Not Allowed	The request has a method we do not support (depending on what is configured in the configuration file).
413 Request Entity Too Large	The uncompressed payload is larger than defined with the <code>maxRequestBytes</code> configuration option. See “Configuring the HTTP server transport” on page 110 for more information on this configuration option.
415 Unsupported Media Type	The client has sent an unsupported <code>Content-Encoding</code> header.
429 Too Many Requests	If too many authentication failures occur (<code>maxAttempts</code>), then requests are throttled for the defined cool-down period (<code>coolDownSecs</code>) to protect the running correlator. See “Configuring the HTTP server transport” on page 110 for more information on the configuration options <code>maxAttempts</code> and <code>coolDownSecs</code> .
500 Internal Server Error	Any other error which occurs before we send the event into the correlator.
503 Host Not Ready	The HTTP server received a request before the application called <code>onApplicationInitialized()</code> in the correlator. See

Code	Reason
	“Sending and receiving events with connectivity plug-ins” on page 38 for more information on this method.
504 Gateway Timeout	The EPL application did not respond within the configured timeout.
Other HTTP response codes	As defined by the EPL application and configuration.

Loading the HTTP server transport

You can load the HTTP server transport by adding the **HTTP Server** connectivity bundle to your project in Software AG Designer (see “Adding the HTTP server connectivity plug-in to a project” in *Using Apama with Software AG Designer*) or using the `apama_project` tool (see “Creating and managing an Apama project from the command line” in *Deploying and Managing Apama Applications*). Alternatively, you can load the transport with the following `connectivityPlugins` stanza in your YAML configuration file:

```
connectivityPlugins:
  HTTPServerTransport:
    libraryName: connectivity-http-server
    class: HTTPServer
```

Configuring the HTTP server transport

The HTTP server has a manager that deals with connections and a chain that deals with mapping events into the correlator. There must be exactly one chain definition which will be used by all managers. If you require multiple ports (that is, with different options), then you need multiple managers. The HTTP server should be added to a manager and chain containing the appropriate mapping rules (see [“Mapping events between EPL and HTTP server requests” on page 116](#) for detailed information).

Manager

Example:

```
dynamicChainManagers:
  HTTPServerManager:
    transport: HTTPServerTransport
    managerConfig:
      port: 15910
      bindAddress: 10.13.23.125
      tls: false
      tlsKeyFile: ${PARENT_DIR}/servername.key.pem
      tlsCertificateFile: ${PARENT_DIR}/servername.cert.pem
      connectionTimeoutSecs: 60
      maxConnections: 16
      keepAliveTimeSecs: 120
      concurrentChains: true
      staticFiles:
```

```
/swagger.json:
  file: ${PARENT_DIR}/swaggerDefault.json
  contentType: application/json
  charset: utf-8
```

The following configuration options are available for the manager on the HTTP server:

Configuration option	Description
port	<p>Required. The user-defined port on which the server is accessible.</p> <p>Type: integer.</p>
bindAddress	<p>Optional. Binds to specific interfaces, potentially on multiple ports. Each entry is either a host, or a <i>host:port</i> combination. If a port is provided, it is used. Otherwise, the port option applies. The default is to bind to all interfaces on the configured port.</p> <p>Type: string or list<string>.</p> <p>Default: <i>blank</i>.</p>
tls	<p>Optional. Set this to true to enable TLS (https).</p> <p>Type: bool.</p> <p>Default: false.</p>
tlsKeyFile	<p>The private key for the certificate in PEM format. Required if TLS is enabled.</p> <p>Type: path.</p>
tlsCertificateFile	<p>The server certificate file in PEM format. Required if TLS is enabled.</p> <p>Type: path.</p>
connectionTimeoutSecs	<p>Maximum time to handle a single request before returning a timeout (in seconds).</p> <p>Type: integer.</p> <p>Default: 60.</p>
maxConnections	<p>Maximum number of simultaneous connections which can be handled.</p> <p>Type: integer.</p> <p>Default: 16.</p>

Configuration option	Description
keepAliveTimeSecs	<p>Optional. Set this to the maximum idle time in seconds between requests on a persistent connection before it is closed. If not set, the default value is used.</p> <p>Type: integer.</p> <p>Default: 15.</p>
concurrentChains	<p>Optional. Set this to true to enable concurrent chains where each connection uses a different chain into the HTTP server to process requests and responses, up to a maximum of <code>maxConnections</code>. Requests on the same connection are processed in order.</p> <p>If set to false (default), concurrent chains are disabled. A single chain is used for all connections, and it only processes a single request at a time.</p> <p>Type: bool.</p> <p>Default: false.</p>
staticFiles	<p>Optional. Map of static files. Elements are of the form:</p> <pre>/url: file: \${PARENT_DIR}/source_file.txt contentType: text/plain charset: utf-8</pre> <p>file and contentType are required, charset is optional.</p> <p>Type: Map.</p> <p>Default: <i>undefined</i>.</p>

Chain

Example:

```
dynamicChains:
  HTTPServerChain:
    - apama.eventMap
      mapping rules...
    - HTTPServerTransport:
      authentication:
        authenticationType: none
        allowedUsersFile: ${PARENT_DIR}/userfile.txt
        maxAttempts: 5
        coolDownSecs: 30
      automaticResponses: false
      responseCompression: "ifRequested"
      responseTimeoutMs: 5000
      allowedMethods: [PUT]
```


The following configuration options are available for the chain on the HTTP server:

Configuration option	Description
<code>authentication/authenticationType</code>	<p>Set this to <code>HTTP_BASIC</code> if you require HTTP basic authentication.</p> <p>Type: <code>HTTP_BASIC</code> or <code>none</code>.</p> <p>Default: <code>none</code>.</p>
<code>authentication/allowedUsersFile</code>	<p>Path to the password file (see “Authentication” on page 127). Required if the authentication type is <code>HTTP_BASIC</code>.</p> <p>Type: <code>path</code>.</p>
<code>authentication/maxAttempts</code>	<p>Maximum number of failed login attempts before throttling the requests for that user. See “Authentication” on page 127 for more information.</p> <p>Type: <code>integer</code>.</p> <p>Default: <code>3</code>.</p>
<code>authentication/coolDownSecs</code>	<p>The number of seconds after the maximum number of failed login attempts before the HTTP server attempts authentication of the user again. See “Authentication” on page 127 for more information.</p> <p>Type: <code>integer</code>.</p> <p>Default: <code>20</code>.</p>
<code>automaticResponses</code>	<p>Set this to <code>true</code> if you want a submission-only API where the responses are generated automatically by the transport. If set to <code>false</code>, the transport will wait for a response from the EPL application, subject to a timeout.</p> <p>Type: <code>bool</code>.</p>
<code>responseCompression</code>	<p>The <code>Accept-Encoding</code> header is used for negotiating content encoding. Set this to <code>ifRequested</code> if you want to encode an EPL response or a static file. If set to <code>never</code>, no encoding is applied to the entity-body.</p> <p>Type: <code>string</code>.</p> <p>Default: <code>never</code>.</p>

Configuration option	Description
<code>responseTimeoutMs</code>	<p>The number of milliseconds we wait for a response from the EPL application before returning to the client.</p> <p>Type: <code>integer</code>.</p> <p>Default: 5000 (5s).</p>
<code>allowedMethods</code>	<p>Required. List of permitted HTTP verbs (for example, <code>PUT</code> or <code>GET</code>).</p> <p>Type: <code>string</code> or <code>list<string></code>.</p>
<code>maxRequestBytes</code>	<p>Maximum permitted HTTP payload size in bytes.</p> <p>Type: <code>integer</code>.</p> <p>Default: 1048576 (1MB).</p>

Handling responses in EPL

In order to have the response to an HTTP request handled by your EPL application, you need to configure the HTTP server chain correctly and then respond to the event delivered to your application. The transport must have `automaticResponses` set to `false` in the configuration (see also [“Configuring the HTTP server transport” on page 110](#)), and it must map the following variables into the message to be able to send responses.

■ `metadata.requestId`

This variable is set by the transport for every message. Responses must also have the same `metadata.requestId` set. This is normally done by mapping it to a payload field in your request for the message sent to the host and then back into the metadata for the response.

■ `@{httpServer.responseChannel}`

This variable is set when creating the chain. This should be set in your request messages. It tells the EPL application to which channel responses should be sent back. Responding messages should also set `metadata.http.statusCode` correctly.

Note:

You must send the response to the channel specified in the corresponding request event. The channel name is not guaranteed to be constant even within a single manager.

For example:

```
dynamicChains:
  HTTPServerChain:
    - apama.eventMap:
        defaultEventType: RequestEvent
```

```

    defaultChannel: requests
  - mapperCodec:
    "★":
      towardsHost:
        mapFrom:
          - payload.requestId: metadata.requestId
        defaultValue:
          - payload.responseChannel: "@{httpServer.responseChannel}"
      towardsTransport:
        mapFrom:
          - metadata.requestId: payload.requestId
        defaultValue:
          - metadata.http.statusCode: 200
  - jsonCodec
  - stringCodec
  - HTTPServerTransport:
    automaticResponses: false
    allowedMethods: [ PUT ]

```

Your EPL application must then respond to messages, preserving the `requestId` and responding on the correct channel. For example:

```

on all RequestEvent() as re {
  any data := // do something to get the response data
  send ResponseEvent(re.requestId, data) to re.responseChannel;
}

```

Note:

The request and response events given here are examples. You must define your own events appropriate to your application. For more examples, see [“Examples” on page 122](#).

If a response is not received by the transport within the configured timeout, then the transport returns a “504 Gateway Timeout” response. This timeout can be configured with the `responseTimeoutMs` configuration option (see also [“Configuring the HTTP server transport” on page 110](#)).

The response messages must be converted and mapped using the chain configuration to meet the following requirements:

- The response payload is a binary message. This will probably be created using the String codec from the event.
- The `metadata.http.statusCode` variable is set. This will usually be set to 200 by the Mapper codec.
- The `metadata.contentType` and `metadata.charset` variables are set. These will usually be set by the JSON codec and String codecs when in use, but can also be set by the Mapper codec.

In addition, you can set other HTTP headers. For more details, see [“Mapping events between EPL and HTTP server requests” on page 116](#).

Serving static files

The HTTP server allows you to serve static files from disk. You can list the static file URI which will be available using a GET request, and it will be served by that file. GET requests that match a static file do not get passed into the correlator.

Static file requests do not go through the checks that all other requests go through, which are:

- Transport status (host ready)
- HTTP basic authentication
- Allowed methods
- Maximum request size

You must list static files individually in the configuration file, and you must provide the MIME type of the file being served. Optionally, you can also provide the charset type.

```
staticFiles:
  /swagger.json:
    file: ${PARENT_DIR}/swagger.json
    contentType: application/json
    charset: utf-8
```

Mapping events between EPL and HTTP server requests

The HTTP server can either be used as a general event submission API or as a general request/response API. A request to the HTTP server contains either a binary payload or a dictionary payload if the request had either an `application/x-www-form-urlencoded` or `multipart/form-data` content type. In the latter case, there will also be additional metadata fields. For the requests to be useful to EPL, they must be converted into the format expected by Apama. This is done using the Classifier codec, Mapper codec and other codecs (see [“Codec Connectivity Plug-ins” on page 209](#)). For request/response APIs, the same process is used in reverse to turn EPL events into the responses.

The event types used in EPL should be specific to your application and then mapped in the chain from the fields produced by the HTTP server. The following fields are created in each event by the HTTP server. Field names containing periods (.) indicate nested map structures within the metadata. This nesting is automatically handled by the Mapper codec, and fields can be referred to there just using these names (see also [“The Mapper codec connectivity plug-in” on page 217](#)).

The fields for requests from the transport to EPL are:

Field	Description
payload	The binary payload of the request.
metadata.requestId	A unique integer identifier which must be preserved in the response when using EPL-supplied responses.

Field	Description
<code>metadata.contentType</code>	The MIME type of the payload (string), taken from the first parameter of the HTTP Content-Type header, converted to lower case and with spaces trimmed off. See also “Handling HTTP headers” on page 119 .
<code>metadata.charset</code>	The charset parameter of the Content-Type header (string), converted to lower case, with spaces trimmed off. See also “Handling HTTP headers” on page 119 .
<code>metadata.http.path</code>	The path component (string) of the URI.
<code>metadata.http.method</code>	The HTTP method of the request: PUT, POST, GET, or DELETE.
<code>metadata.http.user</code>	When HTTP basic authentication is enabled, the authenticated user name (string).
<code>metadata.http.cookies</code>	A key-value map of cookies from the request (map). See also “Dealing with cookies” on page 121 .
<code>metadata.http.queryString</code>	A key-value map of the options in the query-string component of the request URI (map). See also “Providing HTTP query parameters” on page 121 .
<code>metadata.http.headers</code>	A key-value map of the HTTP headers sent by the request (map). Key names are converted to lower case regardless of original capitalization. See also “Handling HTTP headers” on page 119 .
<code>metadata.http.source</code>	The address and port of the client connection which generated this request.

The fields for EPL-supplied responses are:

Field	Description
<code>payload</code>	The binary or dictionary payload of the response.
<code>metadata.requestId</code>	The <code>requestId</code> of the corresponding request. Must be correctly set in responses.
<code>metadata.contentType</code>	The MIME type of the payload (string). This is used to construct the HTTP Content-Type header. See also “Handling HTTP headers” on page 119 .
<code>metadata.charset</code>	The charset of the payload, for text-format payloads. This is used to construct the HTTP

Field	Description
	Content-Type header. See also “Handling HTTP headers” on page 119 .
<code>metadata.http.statusCode</code>	The HTTP status code (integer). Must be set in responses.
<code>metadata.http.cookies</code>	A key-value map of cookies to set on the client. See also “Dealing with cookies” on page 121 .
<code>metadata.http.headers</code>	A key-value map of additional HTTP headers to send in the response. See also “Handling HTTP headers” on page 119 .
<code>metadata.http.form.name.contentType</code>	The media type of the form data. See also “Handling HTTP form decoding” on page 120 .
<code>metadata.http.form.name.charset</code>	The encoding of the form data. See also “Handling HTTP form decoding” on page 120 .
<code>metadata.http.form.name.filename</code>	The filename of the form data. See also “Handling HTTP form decoding” on page 120 .

Distinguishing request types

A single chain will often deal with multiple event types received within requests. For messages towards the host, the event type will not yet have been set. The Mapper and Classifier codecs can use fields in the message (payload or metadata) to set the event type.

You can write the configuration to behave in whatever way you like. There are several ways of determining to which event type the request corresponds. In the default configuration that we supply, the event type is provided as part of the request, but it is also possible to infer the event type from the content of the request.

Below are some examples of what is possible.

You can use the Mapper codec to set the type and channel from the payload as shown below. The type is part of the request. The Mapper code assigns it to `metadata.sag.type`.

```
- mapperCodec:
  "*":
    towardsHost:
      mapFrom:
        - metadata.sag.type: payload.type
        - metadata.sag.channel: payload.channel
        - payload: payload.data
```

You can use the Classifier codec to determine the event type based on incidental fields in the event, such as the method and path:

```
- classifierCodec:
```

```
rules:
  - KickEvent:
    - metadata.http.method: GET
    - metadata.http.path: /kick
  - DocumentSubmissionEvent:
    - metadata.http.method: PUT
    - metadata.http.path: /submit
  - DocumentUpdateEvent:
    - metadata.http.method: PUT
    - metadata.http.path: /update
```

The default event type is generally used if all events received in requests are the same:

```
- apama.eventMap:
  defaultEventType: TestEvent
```

You can use regular expressions with the Classifier codec to match more than one REST URL to a single event type. The following example shows a rule that matches two different REST URLs such as `/database/emptable/78451339` and `/database/managertable/50044897`:

```
- classifierCodec:
  rules:
    - com.apama.swagger.ISSPositionResponse:
      - regex:metadata.http.path: /database/[a-zA-Z0-9]*/[0-9]*
```

For detailed information on these codecs, see [“The Mapper codec connectivity plug-in” on page 217](#) and [“The Classifier codec connectivity plug-in” on page 216](#).

Handling HTTP headers

The HTTP server reads any number of headers from the received request and puts them into `metadata.http.headers`. Similarly, when using EPL-supplied responses, headers are read from `metadata.http.headers` and written into the response as individual HTTP header lines. Some special handling is applied as described below.

All HTTP headers are converted from ISO-8859-1 (the character set for HTTP headers as defined in the RFC publications) to UTF-8 in the metadata and vice-versa.

All HTTP header keys are converted to lowercase (since HTTP header keys are defined to be case-insensitive). You should use lowercase in all of your mapping and classification rules.

Any HTTP headers for which multiple values have been provided for a single key (after normalization of case) are dropped.

The content type and charset in requests, which are parsed from the Content-Type header, are provided in `metadata.contentType` and `metadata.charset` respectively. For responses, the two metadata fields are combined into the Content-Type header.

If HTTP basic authentication is enabled, then the authorization header is removed from `metadata.http.headers`, but in this case the user name is still available in `metadata.http.user`. If authorization is none, then the authorization type is passed through verbatim.

All cookies in requests are put into the `metadata.http.cookies` field and that field is used to generate Set-Cookie headers in responses. See also [“Dealing with cookies” on page 121](#).

To protect the security of personal data, see "Protecting Personal Data in Apama Applications" in *Developing Apama Applications*.

Handling HTTP form decoding

The HTTP server transport decodes `multipart/form-data` or `application/x-www-form-urlencoded` media types to a dictionary payload.

If the `Content-Type` header field contains the `application/x-www-form-urlencoded` media type, the request payload is decoded to a dictionary payload with string keys and string values.

If the `Content-type` header field contains the `multipart/form-data` media type, the request payload is decoded to a dictionary payload with string keys and either string or binary values.

For the parts that have binary data, additional metadata is created. This metadata contains the `contentType`, `charset` and `filename` information for each binary part.

You can get the metadata as follows:

```
metadata.http.form.name.contentType  
metadata.http.form.name.charset  
metadata.http.form.name.filename
```

where *name* corresponds to the data in `payload.name`.

Simple example

In this example, a client sends HTTP POST requests to the HTTP server transport and the `Content-Type` header is set to `multipart/form-data`. The request payload contains two form fields, one field has both a string key and string value, and the other field has a string key and binary value.

Simple raw HTTP POST request:

```
POST http://localhost:80/  
Content-Length: 737  
Content-Type: multipart/form-data; boundary=--123456789  
--123456789  
Content-Disposition: form-data; name="foo"  
bar  
--123456789  
Content-Disposition: form-data; name="file"; filename="file.txt"  
Content-Type: text/plain; charset=utf-8  
File data  
--123456789--
```

For the above request, the HTTP server transport sends a dictionary payload(`{"foo": "bar", "file": File data}`) to EPL.

Metadata created for the `file` parts have `text/plain` as the content type, `utf-8` as the character set, and `file.txt` as the filename. You can map the metadata using the Mapper codec:

```
- mapperCodec:  
  "*":
```



```
towardsHost:
  mapFrom:
    - payload.contentType: metadata.http.form.file.contentType
    - payload.charset: metadata.http.form.file.charset
    - payload.filename: metadata.http.form.file.filename
```

Parts metadata is only created for binary or file-upload form-data.

Mapping the body

The HTTP server accepts the payload as a binary object. What the payload consists of depends on the service you wish to provide. Many services use string-based protocols (such as JSON). For these types of payload, you can use the String codec (see [“The String codec connectivity plug-in” on page 210](#)). For messages towards the host, the String codec takes a byte array and decodes it to a string using the UTF-8 encoding. If you are using the String codec, you should put it as the last codec before the HTTP server.

The resulting string can then be mapped directly into a field in an EPL event, or it can be further processed by other codecs (such as the JSON codec, as used in our default configuration) before the resulting fields are mapped into the Apama event.

If you need to vary your processing depending on the type of the data received, you may need to write a custom codec in order to handle this. To help with distinguishing different payload types, the HTTP server sets top-level fields to indicate the type of the payload. The HTTP header indicates the MIME type populated into `metadata.http.contentType`. If present, then the character set from the same HTTP header is copied into `metadata.http.charset`.

When using EPL-supplied responses, the mapping rules must be bidirectional to map both the request and the response.

Dealing with cookies

The HTTP server stores cookies in `metadata.http.cookies.keyname` entries.

In requests, the HTTP server takes any number of HTTP Cookie headers and turns them into corresponding `metadata.http.cookies` entries. You can either map the entire set of cookies to a dictionary field in an event, or you can map a specific cookie key to a field in an event.

In responses, the HTTP server adds Set-Cookie headers for each entry in `metadata.http.cookies`. You must use the Mapper codec to map things from your response events into the metadata entries.

Providing HTTP query parameters

HTTP requests can be set to contain request parameters, which are encoded at the end of the URL in the following form:

```
/path?key=value&key=value
```

The request parameters are decoded and added to the `metadata.http.queryString` map as key-value pairs. The parameters can either be mapped to a dictionary field in an event, or a specific named parameter can be mapped to a single field. For example:

```
- mapperCodec:
  Request:
    towardsHost:
      mapFrom:
        # set one query parameter individually
        - payload.paramValue: metadata.http.queryString.param
        # alternatively set all query parameters in an EPL dictionary
        - payload.parameters: metadata.http.queryString
```

Examples

Generic engine_send HTTP service

This example is the same as the default configuration supplied with Apama.

YAML dynamic chain:

```
dynamicChains:
  HTTPServerChain:
    - apama.eventMap
    - mapperCodec:
        "*":
          towardsHost:
            mapFrom:
              - metadata.sag.type: payload.type
              - metadata.sag.channel: payload.channel
              - payload: payload.data
    - jsonCodec
    - stringCodec
    - HTTPServerTransport:
        authentication:
          authenticationType: none
          allowedUsersFile: ${PARENT_DIR}/userfile.txt
        automaticResponses: true
        allowedMethods: [PUT]
```

EPL:

```
event Temperature
{
  integer sensorId;
  string sensorName;
  float temperature;
  dictionary<string,any> extra;
}

monitor.subscribe("myChannel");
on all Temperature() as e {
  // ...
}
```

Curl example:

```
curl -X PUT http://localhost:8080/ -d '{"type":"Temperature",
  "channel":"myChannel", data:{"sensorId":666, "sensorName":"FooBar",
  "temperature":3.14,{"A":"alpha"}}}' -H "Content-Type:application/json"
```

Event type and channel information is specified in headers

YAML dynamic chain:

```
dynamicChains:
  HTTPServerChain:
    - apama.eventMap
    - mapperCodec:
        "*":
          towardsHost:
            mapFrom:
              - metadata.sag.type : metadata.http.headers.x-apamaeventtype
              - metadata.sag.channel : metadata.http.headers.x-apamachannel
    - jsonCodec
    - stringCodec
    - HTTPServerTransport:
        authenticationType: none
        allowedUsersFile: ${PARENT_DIR}/userfile.txt
        automaticResponses: true
        allowedMethods: [PUT]
```

EPL:

```
event Temperature
{
  integer sensorId;
  string sensorName;
  float temperature;
  dictionary<string,any> extra;
}

monitor.subscribe("myChannel");
on all Temperature() as e {
  // ...
}
```

Curl example:

```
curl -X PUT -H "X-ApamaEventType:Temperature" -H "X-ApamaChannel:myChannel"
http://localhost:8080/ -d '{"sensorId":666, "sensorName":"FooBar",
"temperature":3.14,{"A":"alpha"}}' -H "Content-Type:application/json"
```

Event type and channel information is specified in the query string

YAML dynamic chain:

```
dynamicChains:
  HTTPServerChain:
    - apama.eventMap
    - mapperCodec:
        "*":
          towardsHost:
            mapFrom:
              - metadata.sag.type : metadata.http.queryString.eventType
              - metadata.sag.channel : metadata.http.queryString.channel
    - jsonCodec
    - stringCodec
    - HTTPServerTransport:
```

```
authenticationType: none
allowedUsersFile: ${PARENT_DIR}/userfile.txt
automaticResponses: true
allowedMethods: [PUT]
```

EPL events:

```
event Temperature
{
    integer sensorId;
    string sensorName;
    float temperature;
    dictionary<string,any> extra;
}

monitor.subscribe("myChannel");
on all Temperature() as e {
    // ...
}
```

Curl example:

```
curl -X PUT 'http://host:port/submit?eventType=Temperature&channel=myChannel'
-d '{"sensorId":666, "sensorName":"FooBar", "temperature":3.14,{"A":"alpha"}} '
```

Event types are tied to the method and path and the channel is defaulted

YAML dynamic chain:

```
dynamicChains:
  HTTPServerChain:
    - apama.eventMap
    - mapperCodec:
        KickEvent:
          towardsHost:
            - metadata.sag.channel: kickEvents
        DocumentSubmissionEvent:
          towardsHost:
            mapFrom:
              - payload.data: payload
            defaultValue:
              - metadata.sag.channel: submissionEvents
        DocumentUpdateEvent:
          towardsHost:
            mapFrom:
              - payload.data: payload
            defaultValue:
              - metadata.sag.channel: updateEvents
    - classifierCodec:
        rules:
          - KickEvent:
              - metadata.http.method: GET
              - metadata.http.path: /kick
          - DocumentSubmissionEvent:
              - metadata.http.method: PUT
              - metadata.http.path: /submit
          - DocumentUpdateEvent:
              - metadata.http.method: PUT
              - metadata.http.path: /update
```

```

- stringCodec
- HTTPServerTransport:
  authenticationType: none
  allowedUsersFile: ${PARENT_DIR}/userfile.txt
  automaticResponses: true
  allowedMethods: [PUT, GET]

```

EPL events:

```

event KickEvent { }
event DocumentSubmissionEvent { string data; }
event DocumentUpdateEvent { string data; }

```

Delivering Apama event strings

This example is using the string form of the event native to Apama. You should only use this example if you have a system that encodes events in that format.

YAML dynamic chain:

```

dynamicChains:
  HTTPServerChain:
    - apama.eventString
    - mapperCodec:
      "x":
        towardsHost:
          mapFrom:
            - metadata.sag.channel: metadata.http.path
    - stringCodec
    - HTTPServerTransport:
      authenticationType: none
      allowedUsersFile: ${PARENT_DIR}/userfile.txt
      automaticResponses: true
      allowedMethods: [PUT]

```

EPL:

```

monitor.subscribe("/channel/ChannelName");
on all Temperature() as e { ... }

```

Curl example:

```

curl -X PUT http://host:port/channel/ChannelName -d 'Temperature(10, "Baz",
6.022e23)'

```

EPL-controlled responses

This example generates responses to the HTTP requests in EPL. Requests should be JSON objects containing `objectId` and `requestType`. Responses are arbitrary JSON objects. See also [“Handling responses in EPL” on page 114](#).

YAML dynamic chain:

```

dynamicChains:
  HTTPServerChain:
    - apama.eventMap:
      defaultChannel: requests

```

```
    defaultEventType: HTTPRequest
  - mapperCodec:
    "*":
      towardsHost:
        mapFrom:
          - payload.requestId: metadata.requestId
        defaultValue:
          - payload.channel: "@{httpServer.responseChannel}"
      towardsTransport:
        mapFrom:
          - metadata.requestId: payload.requestId
          - payload: payload.responseData
        defaultValue:
          - metadata.http.statusCode: 200
  - jsonCodec
  - stringCodec
  - HTTPServerTransport:
    automaticResponses: false
    allowedMethods: [ PUT ]
```

EPL:

```
monitor.subscribe("requests");
on all HTTPRequest() as r {
    send HTTPResponse(r.requestId, getResponseData(r.requestType, r.objectId))
    to r.channel;
}
```

EPL events:

```
event HTTPRequest {
    integer requestId;
    integer objectId;
    string requestType;
    string channel;
}
event HTTPResponse {
    integer requestId;
    any responseData;
}
```

HTTP server security

TLS

We provide TLS-based security with the HTTP server and we recommend that you use this in production. In order to be compatible with our system, you must use TLS version 1.2 or above.

We also recommend that your internet deployment is behind a reverse proxy for optimum security.

In order to use this, you must enable TLS in the YAML configuration file and supply a TLS server certificate file and corresponding key file, as shown in the following example:

```
dynamicChainManagers:
  HTTPServerManager:
    transport: HTTPServerTransport
```

```
managerConfig:
  port: 443
  tls: true
  tlsKeyFile: ${PARENT_DIR}/servername.key.pem
  tlsCertificateFile: ${PARENT_DIR}/servername.cert.pem
```

Authentication

Note:

HTTP basic authentication is not applied to static file requests. See [“Serving static files” on page 116](#).

HTTP basic authentication support is provided by comparing the request authentication contents against an authentication password file supplied during configuration. We recommend that you only use this if you also have TLS enabled. For more complex use cases, webMethods Integration Server should be used.

If you are using HTTP basic authentication, you must provide a valid authentication password file using the `allowedUsersFile` configuration option.

This password file expected by the HTTP server for HTTP basic authentication is compatible with the output of Apache's `htpasswd -B`. There is also a bundled application called `httpserver_passman` which can create and update password files. You can find the executable for this tool in the `bin` folder of your Apama installation. The syntax for this is:

```
httpserver_passman password_file [options] username [password]
```

If you only provide a username and no password, then the password is prompted for interactively. This adds the specified user with the given password, or replaces the password if the user already exists in the password file.

The options are:

Option	Description
<code>-h</code> <code>--help</code>	Displays usage information.
<code>-c</code> <code>--createNew</code>	Creates a new file and overwrites anything currently there.
<code>-D</code> <code>--delete</code>	Deletes the given user, rather than updating or adding the user.
<code>--</code>	Does not treat subsequent arguments as options. Thus, it is possible to enter a username that starts with one or two minus signs.

If HTTP basic authentication is enabled, then the authorization header is removed from `metadata.http.headers`, but in this case the user name is still available in `metadata.http.user`. If authorization is none, then the authorization type is passed through verbatim.

Note:

Enabling authentication significantly reduces the maximum achievable throughput on a single connection since HTTP_BASIC requires verifying credentials on every request. It is not suitable for high-throughput applications.

Using the configuration options `maxAttempts` and `coolDownSecs`, you can protect against brute force attacks on users and passwords (see also [“Configuring the HTTP server transport” on page 110](#)). The initial response to a failed authentication attempt is a “401 Unauthorized” response. This response occurs until the defined number of failed login attempts (`maxAttempts`) has been reached. After this, the HTTP server ignores authentication attempts for the defined cool-down period (`coolDownSecs`). During that period, the HTTP server returns “429 Too Many Requests” with a reason of “Too many failed authentication requests, please try again later.”. When the cool-down period has expired, the HTTP server attempts to authenticate any further request. If it fails that attempt, the user is immediately placed back into a cool-down period without retries.

Note:

Requests from unknown users are treated in the same way as requests from allowed users to avoid user information leakage.

To protect the security of personal data, see “Protecting Personal Data in Apama Applications” in *Developing Apama Applications*.

Monitoring status for the HTTP server

The HTTP server component provides status values via the user status mechanism. It provides the following metrics (where *prefix* is the name of the dynamic chain manager, typically `HTTPServerManager`):

Key	Description
<code>prefix.status</code>	Moves from STARTING to ONLINE when <code>hostReady</code> is called.
<code>prefix.eventsTowardsHost</code>	Number of requests resulting in events being sent to the correlator. This is the primary KPI for this component.
<code>prefix.failedRequests</code>	Number of non-2xx responses sent to clients, including errors generated from EPL. This is expected to be 0 and is a KPI with a warning threshold at 1.
<code>prefix.staticFileRequests</code>	Number of static files served to clients. This is a KPI.
<code>prefix.authenticationFailures</code>	Number of requests with invalid credentials. This KPI is not shown in Command Central by default. It has a warning threshold at 1.
<code>prefix.numChains</code>	Number of active chains for connections into this HTTP server instance. The chains can be reused between connections, but a single connection only uses one chain. This is expected to be between 0 and the maximum number of simultaneous connections which

Key	Description
	can be handled (see also <code>maxConnections</code> in “Configuring the HTTP server transport” on page 110).
<code>prefix.requestSizeEWMAShortBytes</code>	A quickly-evolving exponentially-weighted moving average of request sizes, in bytes.
<code>prefix.requestSizeEWMLongBytes</code>	A longer-term exponentially-weighted moving average of request sizes, in bytes.
<code>prefix.requestSizeMaxInLastHourBytes</code>	The maximum request size in bytes since the start of the last 1 hour measurement period.
<code>prefix.responseSizeEWMAShortBytes</code>	A quickly-evolving exponentially-weighted moving average of response sizes, in bytes.
<code>prefix.responseSizeEWMLongBytes</code>	A longer-term exponentially-weighted moving average of response sizes, in bytes.
<code>prefix.responseSizeMaxInLastHourBytes</code>	The maximum response size in bytes since the start of the last 1 hour measurement period.

For each request/response that is processed, the above `MaxInLastHour` values are updated if either of the following conditions is true:

- The size of the current message is greater than the existing maximum.
- The existing maximum value was set more than 1 hour ago.

Automatic responses are not included in the response size metrics.

Error responses are not included in the response size metrics. The request size metrics are calculated before compression and the response size metrics are calculated after decompression.

For more information about monitor status information published by the correlator, see "Managing and Monitoring over REST" and "Watching correlator runtime status", both in *Deploying and Managing Apama Applications*.

When using Software AG Command Central to manage your correlator, see also "Monitoring the KPIs for EPL applications and connectivity plug-ins" in *Deploying and Managing Apama Applications*.

8 The HTTP Client Transport Connectivity Plug-in

■ About the HTTP client transport	132
■ Loading the HTTP client transport	132
■ Configuring the HTTP client transport	133
■ Mapping events between EPL and HTTP client requests	136
■ Monitoring status for the HTTP client	153
■ Configuring dynamic connections to services	154
■ Using predefined generic event definitions to invoke HTTP services with JSON and string payloads	154

About the HTTP client transport

The HTTP client is a transport for use in connectivity plug-ins which can connect to external services over HTTP/REST, perform requests on them and return the response as an event. It can be used by either customizing what codec to use (for example, the JSON codec) and what events to map to, or it can be used using “generic” events and a predefined chain using a JSON codec, where instances are managed via an EPL API and JSON payloads are sent and received. Mapping to events requires more preparation, but gives a powerful type-safe interface for accessing the results and can support more complex mappings and codecs other than JSON, while the generic events allow quick access to simple services over JSON.

The HTTP client transport can encode HTTP requests and decode HTTP responses with gzip or deflate compression format. It also supports HTML form encoding and can encode a dictionary payload to either `multipart/form-data` or `application/x-www-form-urlencoded` media types.

When using the event mappings, for each service (host and port combination) that you want to connect to, you must create a new instance of a connectivity chain in your configuration file. To use the service, you send events to that chain, where the events are correctly mapped as described in [“Mapping events between EPL and HTTP client requests” on page 136](#). The response is sent back by the same chain instance, with the configured mapping rules.

This transport does not provide a dynamic chain manager. So chains are created either dynamically from EPL using `ConnectivityPlugins.createDynamicChain` and a named chain definition specified in the `dynamicChains` section of the YAML configuration file, or statically using the `startChains` section of the YAML configuration file. For more information on YAML configuration files, see [“Using Connectivity Plug-ins” on page 23](#) and especially [“Configuration file for connectivity plug-ins” on page 26](#).

Note:

When you are using the “generic” event definitions, dynamic chains are always used. See [“Using predefined generic event definitions to invoke HTTP services with JSON and string payloads” on page 154](#) for further information.

Persistent connections to the server are used for multiple requests if this is supported by the service. Connection details to the service are part of the configuration of the transport in the configuration file. Details of the individual requests are configured through the events sent to the chain. The HTTP client supports HTTP version 1.1 and TLS version 1.2 and above.

The HTTP client is designed to talk to REST services and supports GET, POST, PUT and DELETE operations.

Note:

The HTTP client connectivity plug-in does not support reliable messaging.

Loading the HTTP client transport

You can load the HTTP client transport by adding the **HTTP Client** connectivity bundle to your project in Software AG Designer (see [“Adding the HTTP client connectivity plug-in to a project” in *Using Apama with Software AG Designer*](#)) or using the `apama_project` tool (see [“Creating and](#)

managing an Apama project from the command line" in *Deploying and Managing Apama Applications*). Alternatively, you can load the transport with the following `connectivityPlugins` stanza in your YAML configuration file:

```
connectivityPlugins:
  HTTPClientTransport:
    libraryName: connectivity-http-client
    class: HTTPClient
```

Configuring the HTTP client transport

The HTTP client should be added to a chain containing the appropriate mapping rules (see [“Mapping events between EPL and HTTP client requests” on page 136](#) for detailed information). Connection information is configured through the `HTTPClientTransport` element in each chain. For example:

```
startChains:
  HTTPClientChain:
    - apama.eventMap
    codecs...
    - HTTPClientTransport:
      host: www.google.com
      basePath: "/myapi/v123"
      port: 80
      timeoutSecs: 120
      tls: false
      tlsAcceptUnrecognizedCertificates: false
      tlsCertificateAuthorityFile: ""
      followRedirects: true
      cookieJar: true
      authentication:
        authenticationType: none
        username: ""
        password: ""
      proxy:
        host: ""
        port: ""
        authentication:
          authenticationType: none
          username: ""
          password: ""
```

The configuration options below can either be configured statically in the configuration file, or via replacement variables. Variables of the form `${varname}` are replaced at correlator startup time either from a provided `.properties` file or from the correlator command line. Variables of the form `@{varname}` are replaced at chain creation time if using dynamic connections to services (see also [“Configuring dynamic connections to services” on page 154](#)).

Note:

When you have selected the “generic” option when adding the **HTTP Client** connectivity bundle in Software AG Designer or using the `apama_project` tool (see “Creating and managing an Apama project from the command line” in *Deploying and Managing Apama Applications*), variables of the form `@{varname}` are passed from EPL. See [“Using predefined generic event definitions to invoke HTTP services with JSON and string payloads” on page 154](#) for further information.

The following configuration options are available for the HTTP client:

Configuration option	Description
host	Required. The name of the host to connect to. Type: string.
basePath	Optional path to be prefixed to the <code>metadata.http.path</code> for all messages sent to this transport. If you have multiple remote applications on a single host but with different base paths, you will need to create multiple transport instances with different <code>basePath</code> values. The <code>metadata.http.path</code> in responses will include the prefix, if any. Type: string.
port	The port number to connect to. Type: integer. Default: 443 if the <code>tls</code> configuration option is <code>true</code> , otherwise 80.
timeoutSecs	Client TCP timeout in seconds. Type: integer. Default: 120.
tls	If <code>true</code> , TLS is used for the connection to the host. Type: bool. Default: <code>false</code> .
tlsAcceptUnrecognizedCertificates	By default, connections to unrecognized certificates are terminated. Set this to <code>true</code> if non-validated server certificates are to be accepted. Type: bool. Default: <code>false</code> .
tlsCertificateAuthorityFile	By default, server certifications signed by all standard Certificate Authorities are validated. Optionally, you can set this option to provide a path to a CA certificate file in PEM format to authenticate the host with.

Configuration option	Description
<code>followRedirects</code>	<p>Type: string.</p> <p>If set to <code>true</code>, HTTP redirects are to be followed transparently to the new URL. This pertains to responses with status codes for permanent redirections (301 and 308) and temporary redirections (302, 303 and 307). If set to <code>false</code>, the responses with the above status codes are delivered to EPL and must be handled there.</p> <p>In some cases, following a redirect will result in the server responding with one or more further redirects. To prevent redirect loops, the total number of automatic redirects is limited. An error status code (400) will be sent to the EPL application when the limit has been reached.</p> <p>For security reasons, redirects to a different host or to a different protocol (for example, from HTTP to HTTPS) are not followed.</p> <p>Type: <code>bool</code>.</p> <p>Default: <code>true</code>.</p>
<code>cookieJar</code>	<p>If set to <code>true</code>, cookies are to be stored in memory and added to subsequent outgoing requests. If set to <code>false</code>, cookies are placed in the metadata and must be handled by EPL. For more information, see “Dealing with cookies” on page 146.</p> <p>Type: <code>bool</code>.</p> <p>Default: <code>true</code>.</p>
<code>authentication/authenticationType</code>	<p>Set this to <code>HTTP_BASIC</code> if you want to authenticate using HTTP basic authentication.</p> <p>Type: <code>HTTP_BASIC</code> or <code>none</code>.</p> <p>Default: <code>none</code>.</p>
<code>authentication/username</code>	<p>Optional user name for HTTP basic authentication.</p> <p>Type: string.</p>
<code>authentication/password</code>	<p>Optional password for HTTP basic authentication.</p>

Configuration option	Description
	Type: string. <div>Important: If you provide the password for HTTP_BASIC authentication via the configuration file, you must ensure to protect the configuration file against any unauthorized access, since the password will be readable in plain text. To avoid this, you can provide the password via a replacement variable from EPL (see also “Configuring dynamic connections to services” on page 154).</div>
proxy/host	The name of the proxy server to connect to. Type: string.
proxy/port	The port number of the proxy server to connect to. Required if proxy/host is configured. Type: integer.
proxy/authentication/authenticationType	Set this to HTTP_BASIC if you want to authenticate the proxy server using HTTP basic authentication. Type: HTTP_BASIC or none. Default: none.
proxy/authentication/username	Optional proxy user name for HTTP basic authentication. Type: string.
proxy/authentication/password	Optional proxy password for HTTP basic authentication. Type: string.

Mapping events between EPL and HTTP client requests

The information in this section applies when you have added the **HTTP Client** connectivity bundle with the **JSON with application-specific event definitions** option in Software AG Designer.

Note:

The **JSON with generic request/response event definitions** option provides predefined configurations and events for the HTTP client transport which already define the mapping between EPL and the HTTP client requests, and you need not do anything. See [“Using predefined generic event definitions to invoke HTTP services with JSON and string payloads” on page 154](#) for further information.

The HTTP client accepts requests with metadata fields indicating how to make the request and a binary or dictionary payload to be submitted as the body of the request. Each entry in the dictionary payload should have a string key and either a string or a binary value. If the payload is a dictionary, then `metadata.contentType` must be set to either `multipart/form-data` or `application/x-www-form-urlencoded`. A response contains a binary payload which is the body of the response and further metadata fields describing the response. For the responses to be useful to EPL, they must be converted into the format expected by Apama. This is done using the Classifier codec, Mapper codec and other codecs (see [“Codec Connectivity Plug-ins” on page 209](#)).

In order for EPL to connect a response event to the correct request event, each request contains a top-level `requestId` field in the metadata. This is returned verbatim in the corresponding response event along with the path and method copied from the request. If these are mapped to or from EPL, then they can be used for a request-response protocol in EPL. For example:

```
integer id := integer.incrementCounter("HTTPClient.requestId"); // get a
// unique ID to differentiate different responses
// listen for success and failure responses
on Response(id=id) as response and not Error(id=id) {
    // handle successful requests
}
on Error(id=id) as error and not Response(id=id) {
    // handle unsuccessful requests
}
send Request(id, .../* more request data here */) to "httpchannel";
// send the request
```

The event types used in EPL should be specific to your application and then mapped in the chain to the fields expected by the HTTP client.

The following fields in each event are read by the HTTP client. Field names containing periods (.) indicate nested map structures within the metadata. This nesting is automatically handled by the Mapper codec, and fields can be referred to there just using these names (see also [“The Mapper codec connectivity plug-in” on page 217](#)).

Field	Description
<code>payload</code>	Binary or dictionary payload to submit with the request.
<code>metadata.requestId</code>	Required. A request ID (string) to include in the response.
<code>metadata.http.method</code>	Required. The HTTP method to use: GET, POST, PUT or DELETE.

Field	Description
<code>metadata.http.path</code>	Required. URI (string) on the host to submit the request to.
<code>metadata.http.headers.content-encoding</code>	The Content-Encoding to be applied to the entity-body. This can be one of the following: <code>gzip</code> , <code>deflate</code> or <code>identity</code> . When an unsupported content encoding is specified, the HTTP request is ignored and an error message is logged.
<code>metadata.http.headers.keyname</code>	An HTTP header (string) to set in the request. See also “Handling HTTP headers” on page 141 .
<code>metadata.http.cookies.keyname</code>	An HTTP cookie (string) to set in the request. See also “Dealing with cookies” on page 146 .
<code>metadata.http.queryString.keyname</code>	An HTTP query parameter (string) to be encoded as part of the path in the URI. See also “Providing HTTP query parameters” on page 147 .
<code>metadata.charset</code>	Describes the format of the payload (string). See also “Handling HTTP headers” on page 141 .
<code>metadata.contentType</code>	Describes the format of the payload (string). See also “Handling HTTP headers” on page 141 .
<code>metadata.http.form.name.contentType</code>	The media type of the form data. See also “Handling HTML form encoding” on page 142 .
<code>metadata.http.form.name.charset</code>	The encoding of the form data. See also “Handling HTML form encoding” on page 142 .
<code>metadata.http.form.name.filename</code>	The file name of the form data. See also “Handling HTML form encoding” on page 142 .

The responses returned from the HTTP client contain the following fields:

Field	Description
<code>payload</code>	Binary payload received in the response. May be an empty buffer if no response, or <code>null</code> in some error cases.

Field	Description
<code>metadata.requestId</code>	The request ID (string) from the request. Always present in the response.
<code>metadata.http.method</code>	The HTTP method from the request: GET, POST, PUT or DELETE. Always present in the response.
<code>metadata.http.path</code>	The HTTP path (string) from the request. Always present in the response.
<code>metadata.http.statusCode</code>	HTTP status code (integer). Code 200 indicates success. All other codes indicate errors. Always present in the response. See also “Distinguishing response types” on page 140 .
<code>metadata.http.statusReason</code>	HTTP status reason (string). Always present in the response.
<code>metadata.http.headers.keyname</code>	The HTTP header (string) returned by the response. See also “Handling HTTP headers” on page 141 .
<code>metadata.http.cookies.keyname</code>	An HTTP cookie (string) being set by the response. Only present if this is in the response headers. See also “Dealing with cookies” on page 146 .
<code>metadata.charset</code>	Describes the format of the payload (string). Only present if this is in the response headers. See also “Handling HTTP headers” on page 141 .
<code>metadata.contentType</code>	Describes the format of the payload (string). Only present if this is in the response headers. See also “Handling HTTP headers” on page 141 .

You can use the Mapper codec to move things between the payload and the metadata, and vice versa. For example:

```
startChains:
  HTTPClientChain:
    - apama.eventMap
    - mapperCodec:
        MyRequest:
          towardsTransport:
            mapFrom:
              - metadata.http.path: payload.path
              - metadata.requestId: payload.id
              - payload: payload.body
            defaultValue:
              - metadata.http.method: GET
              - metadata.http.headers.accept: application/json
        MyResponse:
          towardsHost:
```

```

      mapFrom:
        - payload.body: payload
        - payload.path: metadata.http.path
        - payload.id: metadata.requestId
    Error:
      towardsHost:
        mapFrom:
          - payload.message: metadata.http.statusReason
          - payload.id: metadata.requestId
          - payload.path: metadata.http.path
          - payload.code: metadata.http.statusCode
- classifierCodec:
  rules:
    - MyResponse:
      - metadata.http.statusCode: 200
    - Error:
      - metadata.http.statusCode:
- stringCodec
- HTTPClientTransport

```

The above example also demonstrates how to use the Classifier codec to split responses into normal responses and error responses based on the status code (see also [“Distinguishing response types” on page 140](#)).

Examples of using the Mapper and Classifier codecs to set these fields can be found in [“Example mapping rules” on page 147](#).

Distinguishing response types

A single chain will often deal with multiple event types in either direction. In the direction towards the transport, the type is already known and can be used to create multiple stanzas in the Mapper codec. For messages towards the host, the event type will not yet have been set. The Classifier codec can use fields in the message (payload or metadata) to set the event type.

For the HTTP client, one of the major distinctions is between success replies and various types of failure. The HTTP status code (`metadata.http.statusCode`) is used to determine whether or not the response is a success. Typically, a response code of 200 indicates that the request was a success, and anything else would be some kind of error. Both errors returned by the remote host and issues which occur within the client itself are returned as messages with a status code other than 200.

For example, a Classifier codec which wants to just distinguish errors and success would look as follows:

```

- classifierCodec:
  rules:
    - MyResponse:
      - metadata.http.statusCode: 200
    - Error:
      - metadata.http.statusCode:

```

There may also be multiple types of success response, possibly from requests to different URLs in the same host. You can use other fields from the metadata or the payload to set the event type. For example:

```

- classifierCodec:

```

```

rules:
- LoginSuccess: # OK response with a session cookie set
  - metadata.http.statusCode: 200
  - metadata.http.cookies.session:
- DataResponse1:
  - metadata.http.statusCode: 200
  - payload.datatype: foo
- DataResponse2:
  - metadata.http.statusCode: 200
  - metadata.http.path: /data2
- Error:
  - metadata.http.statusCode:

```

Handling HTTP headers

The HTTP client reads any number of `metadata.http.headers.keyname` variables from your event and puts them into the HTTP request. Similarly, any headers returned in the response are mapped to the same variables in the response. Some special handling is applied as described below.

All HTTP headers are converted from ISO-8859-1 (the character set for HTTP headers as defined in the RFC publications) to UTF-8 in the metadata (and vice versa for requests).

All HTTP header keys are converted to lowercase in both directions (since HTTP header keys are defined to be case-insensitive). You should use lowercase in all of your mapping and classification rules.

Any HTTP headers for which multiple values have been provided for a single key (after normalization of case) are dropped in either direction.

The following HTTP headers are handled specially in requests:

Field	Value	Description
accept	from contentType	If not provided in the request, but contentType is set, this is set to the contents of metadata.contentType.
accept-charset	utf-8	Set to utf-8 if not set in the request.
accept-encoding	identity	Set to identity if not set in the request.
authorization	from configuration	Always overridden if the authentication type HTTP_BASIC is defined in the configuration. Otherwise, the value from the request metadata is used.
connection	keep-alive	Always overridden.
content-length	length of the payload	Always overridden.
content-type	from contentType and charset	Set from contentType and charset if not set in the request. Content types starting with text/ will have a charset parameter

Field	Value	Description
		appended from the <code>charset</code> field. Other content types will only have the type from the <code>contentType</code> with no parameters.
		This field will not be added if the body is empty and the <code>content-type</code> header is not set explicitly in the request.
<code>date</code>	current date and time	Set to the current date and time if not set in the request.
<code>host</code>	from configuration	Always overridden.
<code>user-agent</code>	Apama/\$VERSION (\$PLATFORM \$ARCH)	Set if not set in the request.

The following HTTP header is handled specially in responses:

Field	Value	Description
<code>Content-Length</code>	length of the payload	Always overridden.

In addition, the top-level fields `metadata.charset` and `metadata.contentType` are set in the response from the HTTP `content-type` header.

Cookie and Set-Cookie headers are handled specially. See [“Dealing with cookies” on page 146](#).

Handling HTML form encoding

If the body of the request is a dictionary payload having a string key and either a string or binary value, the request body is then encoded to either `multipart/form-data` or `application/x-www-form-urlencoded` media types, depending on `metadata.contentType`.

If `metadata.contentType` is set to `application/x-www-form-urlencoded`, then the dictionary payload must have string keys and string values and is transmitted as URL-encoded form data.

If `metadata.contentType` is set to `multipart/form-data`, then the dictionary payload is encoded to multi-part form data. This method must be used to send non-ASCII text or binary data. The binary data form fields should have the following additional metadata: `filename`, `contentType` and `charset`. `filename` is a required parameter.

You can put these metadata items in a form dictionary as follows:

```
metadata.http.form.name.contentType
metadata.http.form.name.charset
metadata.http.form.name.filename
```

where *name* corresponds to the data in `payload.name`.

Simple example

Send a dictionary payload request body which has both key and value strings using the `application/x-www-form-urlencoded` method:

```
event HTTPRequestURLEncoding {
  integer id;
  string method;
  string path;
  string contentType;
  dictionary<string, string> data;
}
```

Send a dictionary payload request body which has a string key and either a string or binary value using the `multipart/form-data` method; provide the metadata for binary form data using `formMetadata`:

```
event HTTPRequestMultiPartForm {
  integer id;
  string method;
  string path;
  string contentType;
  dictionary<string, string> data;
  dictionary<string, dictionary<string,string>> formMetadata;
}
```

Send a request:

```
monitor TestFormEncoding {
  action onload() {
    dictionary<string, string> dataURL :=
      {"string":"Hello World", "foo":"bar"};
    dictionary<string, string> dataMultiPart :=
      {"string":"Hello World", "binary": Binary Data};

    //Metadata for form data filed
    dictionary<string,dictionary<string,string>> formMetadata := {
      "binary":{
        "filename":"file.txt",
        "charset":"utf-8",
        "contentType":"text/plain"
      }
    };
    integer id := integer.incrementCounter("HTTPClient.requestId");

    //Using application/x-www-form-urlencoded media type
    send HTTPRequestURLEncoding(id, "POST", "/",
      "application/x-www-form-urlencoded", dataURL) to "http";

    id := integer.incrementCounter("HTTPClient.requestId");
    //Using multipart/form-data media type
    send HTTPRequestMultiPartForm(id, "POST", "/", "multipart/form-data",
      dataMultiPart, formMetadata) to "http";
  }
}
```

Map the metadata of binary form data using the Mapper codec:

```

- mapperCodec:
  HTTPRequestMultiPartForm:
    towardsTransport:
      mapFrom:
        - metadata.requestId: payload.id
        - metadata.http.method: payload.method
        - metadata.http.path: payload.path
        - metadata.contentType: payload.contentType
        - metadata.http.form.binary.contentType:
            payload.formMetadata.binary.contentType
        - metadata.http.form.binary.filename:
            payload.formMetadata.binary.filename
        - metadata.http.form.binary.charset:
            payload.formMetadata.binary.charset
        - payload: payload.data

```

Handling HTML form encoding using a predefined generic event definition

You can invoke an HTTP service with a payload encoded to either `multipart/form-data` or `application/x-www-form-urlencoded` media types using the predefined `FormRequest` event definition. For detailed information about this event definition, see the *API Reference for EPL (ApamaDoc)*.

The `FormRequest` event definition must be used if `metadata.contentType` is set to either `multipart/form-data` or `application/x-www-form-urlencoded`. The request payload must be a dictionary having a string key and string value.

If `metadata.contentType` is set to `application/x-www-form-urlencoded`, then the dictionary payload is transmitted as URL-encoded form data.

If `metadata.contentType` is set to `multipart/form-data`, then the dictionary payload is encoded to multi-part form data.

Note:

Binary data cannot be read in Apama EPL. Hence it is only possible to send non-ASCII text data form fields with a standard `HTTPClientGenericJSONChain`.

Simple example

Use `multipart/form-data` and `application/x-www-form-urlencoded` media types with non-ASCII text data form fields:

```

monitor TestHtmlEncoding
{
  action onload()
  {
    dictionary<string, string> payload := {"foo":"bar", "abc":"def"};
    dictionary<string, dictionary<string, string>> formMetadata :=
      new dictionary<string, dictionary<string, string>>;
    HttpTransport transport :=
      HttpTransport.getOrCreateWithConfigurations("my_host",
        8080, new dictionary<string, string>);
    HttpOptions httpOptions := new HttpOptions;
    // Using application/x-www-form-urlencoded media type

```



```

httpOptions.headers["content-type"] := "application/x-www-form-urlencoded";
FormRequest(
    transport.createRequest(RequestType.POST, "/", payload, httpOptions),
    formMetadata).execute(handleResponse);
// Using multipart/form-data media type
httpOptions.headers["content-type"] := "multipart/form-data";
FormRequest(
    transport.createRequest(RequestType.POST, "/", payload, httpOptions),
    formMetadata).execute(handleResponse);
}
action handleResponse(Response resp)
{
    log "Got response: " + resp.toString() at INFO;
}
}

```

For multipart/form-data, you can still encode binary data form fields. But to do that, you need to develop a custom plug-in which introduces binary data in your customized chain. In that case, the binary data form fields must have the following additional metadata:

- filename
- contentType
- charset

filename is a required parameter. You can provide this metadata as follows:

```

monitor TestHtmlEncodingBinaryFields
{
    action onload()
    {
        dictionary<string, string> payload :=
            {"foo":"bar", "binary_field": ...binary_data};
        dictionary<string, dictionary<string, string>> formMetadata := {
            "binary_field":{
                "filename":"file1.txt",
                "charset":"utf-8",
                "contentType":"text/plain"
            }
        };
        HttpTransport transport :=
            HttpTransport.getOrCreateWithConfigurations("my_host",
                8080, new dictionary<string, string>);
        HttpOptions httpOptions := new HttpOptions;
        // Using multipart/form-data media type
        httpOptions.headers["content-type"] := "multipart/form-data";
        FormRequest(transport.createRequest(RequestType.POST, "/", payload,
            httpOptions), formMetadata).execute(handleResponse);
    }
    action handleResponse(Response resp)
    { log "Got response: " + resp.toString() at INFO; }
}

```

Mapping the body

The HTTP client accepts and returns the payload as a binary object. What the payload consists of depends on the service to which you are connecting. Many services use string-based protocols (such as JSON). For these types of payload, you can use the String codec (see [“The String codec connectivity plug-in” on page 210](#)). On messages towards the transport, the String codec takes a string and encodes it in UTF-8 bytes. For messages towards the host, the String codec takes a byte array and decodes it to a string using the UTF-8 encoding. If you are using the String codec, you should put it as the last codec before the HTTP client.

To create a request with no payload (such as a GET request), you should pass an empty string to the String codec, which it will convert to a zero-byte payload. If you are using the “generic” JSON option (see also [“Using predefined generic event definitions to invoke HTTP services with JSON and string payloads” on page 154](#)), then you can do the same by sending a new any as the payload. For example:

```
transport.createRequest(RequestType.GET, "/path", new any, new HttpOptions);
```

The `createGETRequest` action will do this for you. In order to recreate this with your own custom chain using the JSON codec, then you need to have an empty payload (which will skip the JSON codec) and then use a second Mapper codec to add an empty string to the payload before the String codec:

```
- jsonCodec
- mapperCodec:
  "*":
    towardsTransport:
      defaultValue:
        payload: ""
- stringCodec
```

The resulting string can then be mapped directly into a field in an EPL event, or it can be further processed by other codecs (such as the JSON codec) before the resulting fields are mapped into the Apama event.

If you need to vary your processing depending on the type of the returned data, you may need to write a custom codec in order to handle this. To help with distinguishing different payload types, the HTTP client sets top-level fields to indicate the type of the payload. `metadata.contentType` contains the MIME type indicated in the Content-Type HTTP header. If present, then `metadata.charset` indicates the character set from the same HTTP header.

Dealing with cookies

Some HTTP services set cookies and require them to be set in further requests.

When the configuration option `cookieJar` is `true` (default), cookies received from the server are stored in memory and added to subsequent outgoing requests. Cookies forwarded using `metadata.http.cookies` are honored and not overwritten. The HTTP client also honors additional cookie attributes such as `path`, `expiry` and `max-age`. Expired cookies are automatically removed from the internal cache. See also [“Configuring the HTTP client transport” on page 133](#).

When the configuration option `cookieJar` is `false` and if you need to take a specific cookie in a response and return it in future requests, you need to map it out into a field in the response event, and then map it back from future request events. The HTTP client stores cookies in `metadata.http.cookies.keyname` entries. In requests, the HTTP client reads all of the `metadata.http.cookies` entries and combines them into a single HTTP Cookies header to send to the server. In responses, the HTTP client takes any number of HTTP Set-Cookie headers and turns them into corresponding `metadata.http.cookies` entries.

Providing HTTP query parameters

HTTP requests can contain request parameters, which are encoded at the end of the URL in the following form:

```
/path?key=value&key=value
```

The request parameters can be provided as part of the `metadata.http.path` element in a request. In this case, however, they must be correctly encoded within the request.

A better solution is to provide the request parameters as part of the `metadata.http.queryString` element. This is a map of key/value pairs which will be correctly HTTP encoded and appended to the end of the `metadata.http.path` in the request. The parameters can either be set as a map directly out of the payload, or they can be set individually via the Mapper codec. For example:

```
- mapperCodec:
  Request:
    towardsTransport:
      mapFrom:
        # set one query parameter individually
        - metadata.http.queryString.param: payload.paramValue
        # alternatively set all query parameters from an EPL dictionary
        - metadata.http.queryString: payload.parameters
```

Example mapping rules

A full example configuration can be found in the `samples` directory of your Apama installation. The monitoring sample, found in `samples/connectivity_plugin/app/monitoring`, can be run both with this pre-compiled HTTP client or with the simple HTTP client sample under `samples/connectivity_plugin/cpp/httpclient`.

Simple example

The following is a simple REST service with a single URL that is not interested in dealing with error cases:

```
event PutData {
  integer requestId;
  string requestString;
}
event PutDataResponse {
  integer requestId;
  string responseString;
}
```

Each PUT request contains a request string which performs an action on the server and returns another string in the response.

```
startChains:
  simpleRestService:
    - apama.eventMap:
      # Channel that responses are delivered on
      defaultChannel: SRS-response
    - mapperCodec:
      PutData: # requests
      towardsTransport:
        mapFrom:
          - metadata.requestId: payload.requestId
          - payload: payload.requestString
        defaultValue:
          - metadata.http.method: PUT
          - metadata.http.path: /path/to/service
      PutDataResponse:
      towardsHost:
        mapFrom:
          - payload.responseString: payload
          - payload.requestId: metadata.requestId
    - classifierCodec:
      rules:
        - PutDataResponse:
    - stringCodec
    - HTTPClientTransport:
      host: foo.com
```

CRUD service example

The following is a more complex service that implements a full CRUD (create, read, update, delete) service, with different types of request on different objects. There are several different request types with individual mapping rules. The create request is implemented with these events and mapping rules:

```
event CreateResource {
  integer id;
  string value;
}
event ResourceCreated {
  integer id;
  string resource;
}
```

There is one URL for adding new resources which returns the resource identifier which can be used to manipulate it in future via a redirection header.

```
- mapperCodec:
  CreateResource: # requests
  towardsTransport:
    mapFrom:
      - metadata.requestId: payload.id
      - payload: payload.value
    defaultValue:
      - metadata.http.path: /newResource
      - metadata.http.method: PUT
  ResourceCreated:
```

```

towardsHost:
  mapFrom:
    # redirects us to the new resource
    - payload.resource: metadata.http.headers.location

    - payload.id: metadata.requestId

```

The full example is provided below:

```

event GetValue {
  integer id;
  string resource;
}
event CurrentValue {
  integer id;
  string value;
}
event UpdateValue {
  integer id;
  string resource;
  string newValue;
}
event CreateResource {
  integer id;
  string value;
}
event ResourceCreated {
  integer id;
  string resource;
}
event DestroyResource {
  integer id;
  string resource;
}
event ResourceDestroyed {
  integer id;
}
event ResourceNotFound {
  integer id;
  string resource;
}
event InternalError {
  integer id;
  string error;
}

```

```

startChains:
  storageService:
    - apama.eventMap:
        # Channel that responses are delivered on
        defaultChannel: storageResponses
    - mapperCodec:
        CreateResource: # requests
        towardsTransport:
          mapFrom:
            - metadata.requestId: payload.id
            - payload: payload.value
          defaultValue:

```

```

        - metadata.http.path: /newResource
        - metadata.http.method: PUT
DestroyResource: # requests
  towardsTransport:
    mapFrom:
      - metadata.requestId: payload.id
      - metadata.path: payload.resource
    defaultValue:
      - metadata.http.method: DELETE
UpdateValue: # requests
  towardsTransport:
    mapFrom:
      - metadata.requestId: payload.id
      - metadata.path: payload.resource
      - payload: payload.newValue
    defaultValue:
      - metadata.http.method: PUT
GetValue: # requests
  towardsTransport:
    mapFrom:
      - metadata.requestId: payload.id
      - metadata.path: payload.resource
    defaultValue:
      - metadata.http.method: GET
ResourceCreated:
  towardsHost:
    mapFrom:
      # redirects us to the new resource
      - payload.resource: metadata.http.headers.location
      - payload.id: metadata.requestId
ResourceDestroyed:
  towardsHost:
    mapFrom:
      - payload.id: metadata.requestId
CurrentValue:
  towardsHost:
    mapFrom:
      - payload.value: payload
      - payload.id: metadata.requestId
ResourceNotFound:
  towardsHost:
    mapFrom:
      - payload.resource: metadata.http.path
      - payload.id: metadata.requestId
InternalError:
  towardsHost:
    mapFrom:
      - payload.error: metadata.statusReason
      - payload.id: metadata.requestId
- classifierCodec:
  rules:
    - ResourceCreated:
      - metadata.http.statusCode: 200
      - metadata.http.path: /newResource
    - CurrentValue:
      - metadata.http.statusCode: 200
      - metadata.http.method: GET
    - ResourceDestroyed:
      - metadata.http.statusCode: 200
      - metadata.http.method: DELETE

```

```

- ResourceNotFound:
  - metadata.http.statusCode: 404
- InternalError:
  - metadata.http.statusCode:
- stringCodec
- HTTPClientTransport:
  host: foo.com

```

Login example

An example with a login request that has to manage cookies might look like this when the service uses JSON:

```

event Command {
  string command;
  sequence<string> arguments;
}
event Login {
  string username;
  string password;
}
event LoginSuccess {
  dictionary<string, string> sessionCookies;
}
event ExecuteCommand {
  integer id;
  Command command;
  dictionary<string, string> sessionCookies;
}
event CommandResponse {
  integer id;
  string response;
}

```

The Login command sends a password and sets a cookie which must be set in all the following requests. In practice this may need to be repeated on startup, after some timeout period or certain errors.

```

startChains:
  remoteAccessService:
    - apama.eventMap:
      # Channel that you send requests to
      subscribeChannels: remoteAccess
      # Channel that responses are delivered on
      defaultChannel: remoteAccess
    - mapperCodec:
      Login: # requests
        towardsTransport:
          mapFrom:
            # payload.user and payload.password will be converted
            # into a JSON document
          defaultValue:
            - metadata.http.path: /login
            - metadata.http.method: PUT
            - metadata.requestId: "" # ignored
      ExecuteCommand: # requests
        towardsTransport:
          mapFrom:

```

```
- metadata.requestId: payload.id
# set the whole map of any cookies set by the server
- metadata.http.cookies: payload.sessionCookies
# a JSON object made from this event
- payload: payload.command
defaultValue:
- metadata.http.method: PUT
- metadata.path: /execute
LoginSuccess:
  towardsHost:
    mapFrom:
      # store all cookies set by the server,
      # no matter what they are
      - payload.sessionCookies: metadata.http.cookies
CommandResponse:
  towardsHost:
    mapFrom:
      # payload.response already parsed from the JSON response
      - payload.id: metadata.requestId
- classifierCodec:
  rules:
    - LoginSuccess:
      - metadata.http.statusCode: 200
      - metadata.http.cookies.session:
    - CommandResponse:
      - metadata.http.statusCode: 200
- jsonCodec
- stringCodec
- HTTPClientTransport:
  host: foo.com
  tls: true
```

Content-encoding example

The following example shows how to define content encoding for an HTTP request:

```
event HTTPRequest
{
  integer id;
  string path;
  string data;
  string method;
  string contentEncoding;
}
```

You can use the Mapper codec to map the encoding method as follows:

```
- mapperCodec:
  HTTPRequest:
    towardsTransport:
      mapFrom:
        - metadata.http.path: payload.path
        - metadata.requestId: payload.id
        - metadata.http.method: payload.method
        - metadata.http.headers.content-encoding: payload.contentEncoding
        - payload: payload.data
```


Monitoring status for the HTTP client

The HTTP client component provides status values via the user status mechanism. It provides the following metrics (where *prefix* consists of the chain identifier and plug-in name, typically `HTTPClientChain.HTTPClientTransport`):

Key	Description
<code>prefix.status</code>	FAILED if the most recent request has failed, otherwise ONLINE.
<code>prefix.errorsTowardsHost</code>	Number of error responses to requests which have been sent.
<code>prefix.responsesTowardsHost</code>	Number of success responses to requests which have been sent.
<code>prefix.requestLatencyEWMAShortMillis</code>	A quickly-evolving exponentially-weighted moving average of request latencies, in milliseconds.
<code>prefix.requestLatencyEWMALongMillis</code>	A longer-term exponentially-weighted moving average of request latencies, in milliseconds.
<code>prefix.requestSizeEWMAShortBytes</code>	A quickly-evolving exponentially-weighted moving average of request sizes, in bytes.
<code>prefix.requestSizeEWMALongBytes</code>	A longer-term exponentially-weighted moving average of request sizes, in bytes.
<code>prefix.requestSizeMaxInLastHourBytes</code>	The maximum request size in bytes since the start of the last 1 hour measurement period.
<code>prefix.responseSizeEWMAShortBytes</code>	A quickly-evolving exponentially-weighted moving average of response sizes, in bytes.
<code>prefix.responseSizeEWMALongBytes</code>	A longer-term exponentially-weighted moving average of response sizes, in bytes.
<code>prefix.responseSizeMaxInLastHourBytes</code>	The maximum response size in bytes since the start of the last 1 hour measurement period.

For each request/response that is processed, the above `MaxInLastHour` values are updated if either of the following conditions is true:

- The size of the current message is greater than the existing maximum.
- The existing maximum value was set more than 1 hour ago.

Error responses are not included in the response size metrics. The request size metrics are calculated before compression and the response size metrics are calculated after decompression.

For more information about monitor status information published by the correlator, see "Managing and Monitoring over REST" and "Watching correlator runtime status", both in *Deploying and Managing Apama Applications*.

When using Software AG Command Central to manage your correlator, see also "Monitoring the KPIs for EPL applications and connectivity plug-ins" in *Deploying and Managing Apama Applications*.

Configuring dynamic connections to services

Many applications have a single or small number of statically configured connections to services. For other applications, the connections can be configured dynamically at runtime. To configure the connections dynamically, define your chain under `dynamicChains` rather than `staticChains` with the configuration details using dynamic chain replacement variables (`@{varname}`):

```
dynamicChains:
  HTTPClientChain:
    - apama.eventMap
      mapping rules...
    - HTTPClientTransport:
      host: "@{HOST}"
      port: "@{PORT}"
```

Then you can create instances of that chain configured for specific hosts and ports using the `createDynamicChain` method on `ConnectivityPlugins`:

```
action connectToNewHost(string channelName, string host, integer port,
  string defaultChannelTowardsHost)
  returns Chain
{
  return ConnectivityPlugins.createDynamicChain(
    "http-"+host+": "+port.toString(), [channelName],
    "http", {"HOST":host,"PORT":port.toString()}, defaultChannelTowardsHost);
}
```

Events can be sent to the chain via the supplied `channelName`. When the connection is no longer needed, it can be destroyed via the returned `Chain` object.

Using predefined generic event definitions to invoke HTTP services with JSON and string payloads

JSON payloads

You can invoke an HTTP service with a JSON payload by using predefined generic Apama event definitions. To do so, you have to use the **JSON with generic request/response event definitions**

option when adding the HTTP client connectivity plug-in. See also "Adding the HTTP client connectivity plug-in to a project" in *Using Apama with Software AG Designer*.

This “generic” option uses a predefined chain definition with dynamic chain instances to invoke multiple HTTP services, and it uses event types in the `com.softwareag.connectivity.httpclient` package. For detailed information about the available event types, see the *API Reference for EPL (ApamaDoc)*.

The following example shows how to invoke an HTTP service using the generic events:

```
action performRequest() {
    // 1) Get the transport instance
    HttpTransport transport := HttpTransport.getOrCreate("www.example.com", 80);
    // 2) Create the request event
    Request req:= transport.createGETRequest("/geo/");
    // 3) Execute the request and pass the callback action
    req.execute(handleResponse);
}
action handleResponse(Response res) {
    // 4) Handle the response
    if res.isSuccess() {
        // 5) Extract data from the payload
        log res.payload.getString("location.city") at INFO;
    } else {
        log "Failed: " + res.statusMessage at ERROR;
    }
}
```

Overriding the content-type header of an HTTP request to allow non-JSON string payloads

You can override the content-type header of an HTTP request to allow for non-JSON string payloads.

Whenever the content-type header of a request is not overridden, the payloads are encoded as JSON (this is the default setting). When you override the content-type header, the JSON codec is skipped and the payload is not encoded as JSON, allowing string data to be passed through. The decoding of the response to the request depends on the content type provided by the server.

The following example demonstrates how to override an HTTP request's content-type header to send string data:

```
// 1) HTTP PUT request with string ("example string payload") payload
req := transport.createPUTRequest("/plain_string", "example string payload");
// 2) Override the request's content-type header
req.setHeader("content-type", "text/plain");
// 3) Execute the request, passing the callback action handleResponse
req.execute(handleResponse);
```


9 The Kafka Transport Connectivity Plug-in

■ About the Kafka transport	158
■ Loading the Kafka transport	158
■ Configuring the connection to Kafka (dynamicChainManagers)	158
■ Configuring message transformations (dynamicChains)	160
■ Payload for the Kafka message	161
■ Metadata for the Kafka message	161

About the Kafka transport

Kafka is a distributed streaming platform. See <https://kafka.apache.org/> for detailed information.

Apama provides a connectivity plug-in, the Kafka transport, which can be used to communicate with the Kafka distributed streaming platform. Kafka messages can be transformed to and from Apama events by listening for and sending events to channels such as *prefix:topic* (where the prefix is configurable).

You configure the Kafka connectivity plug-in by editing the files that come with the **Kafka** bundle. The properties file defines the substitution variables that are used in the YAML configuration file which also comes with the bundle. See "Adding the Kafka connectivity plug-in to a project" in *Using Apama with Software AG Designer* for further information.

Note:

In addition to using Software AG Designer to add bundles, you can also do this using the `apama_project` command-line tool. See "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications* for more information.

This transport provides a dynamic chain manager which creates chains automatically when EPL subscribes or sends to a correlator channel with the configured prefix, typically `kafka:`. For the Kafka transport, there must be exactly one chain definition provided in the `dynamicChains` section of the YAML configuration file.

For more information on YAML configuration files, see "Using Connectivity Plug-ins" on page 23 and especially "Configuration file for connectivity plug-ins" on page 26.

Note:

The Kafka connectivity plug-in does not support reliable messaging.

Loading the Kafka transport

You can load the Kafka transport by adding the **Kafka** connectivity bundle to your project in Software AG Designer (see "Adding the Kafka connectivity plug-in to a project" in *Using Apama with Software AG Designer*). Alternatively, you can load the transport with the following `connectivityPlugins` stanza in your YAML configuration file:

```
kafkaTransport:
  classpath: ${APAMA_HOME}/lib/connectivity-kafka.jar
  class: com.apama.kafka.ChainManager
```

Configuring the connection to Kafka (dynamicChainManagers)

You configure one or more `dynamicChainManagers` to connect to different Kafka brokers. For example:

```
dynamicChainManagers:
```

```
kafkaManager:
  transport: kafkaTransport
  managerConfig:
    channelPrefix: "kafka:"
    bootstrap.servers: "localhost:9092"
```

Connection-related configuration is specified in the `managerConfig` stanza on the `dynamicChainManagers` instance. The following configuration options are available for `managerConfig`:

Configuration option	Description
<code>bootstrap.servers</code>	<p>This is the only option in the Kafka configuration for which you must specify a value. You can either set it under <code>managerConfig</code> (to be used as the default for all consumers and producers) or in the configuration of a specific consumer or producer (which overrides any default given in the parent chain manager).</p> <p>When this option is set under <code>managerConfig</code>, it is used as the default. It needs to be enclosed in quotation marks. Example:</p> <pre>bootstrap.servers: "localhost:62618"</pre> <p>Type: <code>string</code>.</p>
<code>channelPrefix</code>	<p>Prefix for dynamic mapping. If the prefix ends with a colon (:), it needs to be enclosed in quotation marks (see also "Using YAML configuration files" in <i>Deploying and Managing Apama Applications</i>).</p> <p>When the channel is mapped to a Kafka topic, the prefix is not used. For example, if the prefix is <code>"kafka:"</code>, then the channel <code>kafka:test/a</code> maps to the Kafka topic <code>test/a</code>.</p> <p>Type: <code>string</code>.</p> <p>Default: <code>"kafka:"</code>.</p>
<code>consumerConfig</code>	<p>Keys and values of the consumer configuration options in Kafka. See the Kafka documentation at https://kafka.apache.org/documentation/ for detailed information on the <i>consumer configs</i>.</p> <p>Some default values are provided by the Kafka transport, but you can override them by specifying different values.</p> <p>The default values are:</p> <ul style="list-style-type: none"> ■ <code>group.id</code>: A unique identifier for every instance. ■ <code>session.timeout.ms</code>: <code>"30000"</code>

Configuration option	Description
producerConfig	<ul style="list-style-type: none">■ <code>key.deserializer:</code> "org.apache.kafka.common.serialization.StringDeserializer"■ <code>value.deserializer:</code> "org.apache.kafka.common.serialization.StringDeserializer"
	Type: map.
	Keys and values of the producer configuration options in Kafka. See the Kafka documentation at https://kafka.apache.org/documentation/ for detailed information on the <i>producer configs</i> .
	Some default values are provided by the Kafka transport, but you can override them by specifying different values.
	The default values are:
	<ul style="list-style-type: none">■ <code>linger.ms:</code> 0■ <code>key.serializer:</code> "org.apache.kafka.common.serialization.StringSerializer"■ <code>value.serializer:</code> "org.apache.kafka.common.serialization.StringSerializer"
	Type: map.

Kafka allows clients to connect over SSL. You use the `consumerConfig` and `producerConfig` configuration options of the Kafka transport to specify the SSL configuration. See the Kafka documentation at <https://kafka.apache.org/> for detailed information on how to configure Kafka clients to use SSL.

Configuring message transformations (dynamicChains)

You configure exactly one `dynamicChains` section to handle transforming messages from the Kafka broker into the correlator, and vice versa. For example:

```
dynamicChains:
  kafkaChain:
    - apama.eventMap:
        defaultEventType: Evt
        suppressLoopback: true
    - jsonCodec
```


– kafkaTransport

We recommend use of the `suppressLoopback` configuration property to prevent undesirable behavior. See [“Host plug-ins and configuration” on page 30](#) for further information.

Payload for the Kafka message

As with all other transports, the translation between EPL events and Kafka payloads is based on the choice of host plug-in and codecs. See [“Host plug-ins and configuration” on page 30](#) and [“Codec Connectivity Plug-ins” on page 209](#) for further information.

The default payload for the Kafka message is a string (with conversion from the underlying bytes using the classes `StringDeserializer` and `StringSerializer` from the `org.apache.kafka.common.serialization` package).

The following is a simple example of a YAML configuration file where the payload of the Kafka message will be the string form of the Apama event:

```
dynamicChainManagers:
  kafkaManager:
    transport: kafkaTransport
    managerConfig:
      bootstrap.servers: "localhost:9092"

dynamicChains:
  myChain:
    - apama.eventString:
    - kafkaTransport:
```

You can configure alternative serializers and deserializers using the `consumerConfig` and `producerConfig` options of the Kafka connectivity plug-in (see also [“Configuring the connection to Kafka \(dynamicChainManagers\)” on page 158](#)). You can use a third-party serializer/deserializer implementation or you can create your own. You just need to include the relevant classes in the same classpath of the Kafka plug-in itself so that it can locate them. See the Kafka documentation for more information about Kafka serializers and deserializers. Additional transformations (for example, from a string containing JSON to a map) can be performed after the Kafka transport using connectivity codec plug-ins.

Metadata for the Kafka message

Messages going from/to the transport have useful pieces of information inserted into their metadata. This information is stored as a map associated with the `kafka` key. This map contains the following information:

Field	Description
<code>key</code>	Contains the Kafka record key. This works in both directions. If a message from Kafka has a key, then the metadata will contain it. If a message that is being sent to Kafka has the key in the metadata, then the Kafka record key will be set with it.

10 The Cumulocity IoT Transport Connectivity Plug-in

■ About the Cumulocity IoT transport	164
■ Configuring the Cumulocity IoT transport	165
■ Loading the Cumulocity IoT transport	170
■ Using managed objects	171
■ Using alarms	175
■ Using events	179
■ Using measurements	183
■ Using measurement fragments	188
■ Using operations	191
■ Receiving update notifications	195
■ Paging Cumulocity IoT queries	197
■ Invoking other parts of the Cumulocity IoT REST API	199
■ Invoking microservices	200
■ Monitoring status for Cumulocity IoT	201
■ Finding tenant options	202
■ Getting user details	203
■ Sample EPL	204

About the Cumulocity IoT transport

Cumulocity IoT is used for communication with connected IoT devices. See <http://cumulocity.com/> for detailed information.

Apama provides a connectivity plug-in, the Cumulocity IoT transport, which allows you to communicate with the IoT devices connected to Cumulocity IoT. For example, you can receive events from the devices and send operations to the devices.

You configure the Cumulocity IoT connectivity plug-in by editing the `.properties` file that comes with the **Cumulocity Client** connectivity bundle. See "Adding the Cumulocity IoT connectivity plug-in to a project" in *Using Apama with Software AG Designer* for further information.

In addition to the **Cumulocity Client** connectivity bundle, the following EPL bundles are also available (see also "Adding EPL bundles to projects" in *Using Apama with Software AG Designer*):

- **Event Definitions for Cumulocity.** This EPL bundle defines all events that can be used for interacting with Cumulocity IoT. This includes definitions for events that you receive from Cumulocity IoT, events that you can send to Cumulocity IoT, and event APIs that you can use for requesting data from Cumulocity IoT. For more information, see the `com.apama.cumulocity` package in the *API Reference for EPL (ApamaDoc)*.
- **Utilities for Cumulocity.** This EPL bundle contains useful utilities for EPL code that is interacting with Cumulocity IoT. It also contains a geofence helper utility for determining whether a location is part of a geofence or not. For more information, see the `com.apama.cumulocity.Util` and the `com.apama.cumulocity.GeoFenceContainer` events in the *API Reference for EPL (ApamaDoc)*.

Note:

In addition to using Software AG Designer to add the above mentioned connectivity and EPL bundles, you can also do this using the `apama_project` command-line tool. See "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications* for more information.

As with other connectivity plug-ins, the EPL application should call `com.softwareag.connectivity.ConnectivityPlugins.onApplicationInitialized()`. For more information, see "[Sending and receiving events with connectivity plug-ins](#)" on page 38.

The `samples/cumulocity` directory of your Apama installation includes samples which show how to use the Cumulocity IoT transport. For more information, see the `README.txt` file in the corresponding samples folder.

Note:

The Cumulocity IoT connectivity plug-in does not support reliable messaging.

Configuring the Cumulocity IoT transport

When you add the **Cumulocity Client** connectivity bundle in Software AG Designer, a `.properties` configuration file is created. You have to provide all required information in that file in order to connect to Cumulocity IoT.

Note:

It is strongly recommended that you do not change the YAML configuration file which also comes with the bundle. You should always set the properties in the `.properties` configuration file, which defines the substitution variables to be used in the YAML configuration file.

The following is an example of a filled out `.properties` configuration file:

```
# Username and password must be provided for authentication
CUMULOCITY_USERNAME=MYNAME
CUMULOCITY_PASSWORD=MYPW
# Application key and the URL of the application
CUMULOCITY_APPKEY=MYAPP
CUMULOCITY_SERVER_URL=https://myserver.cumulocity.com
# TLS certificate authority file
CUMULOCITY_AUTHORITY_FILE=
# Allow connection to Cumulocity IoT instance with unknown certificate
CUMULOCITY_ALLOW_UNAUTHORIZED_SERVER=false
# Set this to the tenant ID if you don't have a per-tenant hostname
CUMULOCITY_TENANT=
# Set Cumulocity IoT measurement format
CUMULOCITY_MEASUREMENT_FORMAT=BOTH
CUMULOCITY_FORCE_INITIAL_HOST=true
# Proxy server host and port to start using HTTP proxy
CUMULOCITY_PROXY_HOST=proxy_host
CUMULOCITY_PROXY_PORT=

# Proxy username and password must be provided for basic authentication
CUMULOCITY_PROXY_USERNAME=ProxyUser
CUMULOCITY_PROXY_PASSWORD=ProxyPW
```

In order to connect to Cumulocity IoT, it is required that you set the following properties.

Property	Description
CUMULOCITY_USERNAME	Username for authentication. This can be specified either as a username alone or in the form of <i>tenantID/username</i> . In recent versions of Cumulocity IoT, the tenant ID is visible in the web applications in the user menu in the top-right. Type: string.
CUMULOCITY_PASSWORD	Password for authentication. Type: string.

Property	Description
CUMULOCITY_APPKEY	<p>Unique key for the application defined on the Cumulocity IoT instance.</p> <p>The application key is defined in Cumulocity IoT. Log in to your account in Cumulocity IoT, and use the Administration application to add an external application. You can then specify the application key and the URL of the application. See the Cumulocity IoT documentation at http://cumulocity.com/guides/ for more information.</p> <p>Type: string.</p>
CUMULOCITY_SERVER_URL	<p>URL of the Cumulocity IoT tenant.</p> <p>Type: string.</p>

Under normal conditions in the cloud, the above properties are all you need to set. The properties listed below may be useful for custom on-premises installations of Cumulocity IoT or for Cumulocity IoT Edge.

Property	Description
CUMULOCITY_AUTHORITY_FILE	<p>The TLS certificate authority file. If you are using your own server and it is not signed by a trusted Certificate Authority (CA), provide the certificate of your signing authority here.</p> <p>Type: string.</p>
CUMULOCITY_ALLOW_UNAUTHORIZED_SERVER	<p>Set this to true when the user is connecting to a Cumulocity IoT platform whose certificate is not signed by a trusted CA authority. This generally happens in the Cumulocity IoT Edge instance where the installation is using a self-signed certificate.</p> <p>Default: false.</p>
CUMULOCITY_TENANT	<p>Unique name of the application tenant. This configuration option is useful in the case of Cumulocity IoT Edge.</p> <p>Type: string.</p>
CUMULOCITY_MEASUREMENT_FORMAT	<p>The measurement format mode used by the tenant. Two modes are available: MEASUREMENT_ONLY and BOTH. For more information, see “Turning measurement fragments on/off” on page 190.</p> <p>Type: string.</p>

Property	Description
	Default: BOTH.
CUMULOCITY_FORCE_INITIAL_HOST	<p>If set to <code>false</code>, the endpoint details returned by the Cumulocity IoT platform are used. If set to <code>true</code>, the Cumulocity IoT SDK always uses the URL provided during session initialization instead of the endpoint details. This is helpful in deployment scenarios where the Cumulocity IoT instance is reachable only with an IP address.</p> <p>Type: <code>boolean</code>.</p> <p>Default: <code>true</code>.</p>
CUMULOCITY_PROXY_HOST	<p>The name of the proxy server to connect to.</p> <p>Type: <code>string</code>.</p>
CUMULOCITY_PROXY_PORT	<p>The port number of the proxy server to connect to.</p> <p>Both host and port are required to enable an HTTP proxy.</p> <p>Type: <code>integer</code>.</p>
CUMULOCITY_PROXY_USERNAME	<p>Optional proxy user name for HTTP basic authentication.</p> <p>Type: <code>string</code>.</p>
CUMULOCITY_PROXY_PASSWORD	<p>Optional proxy password for HTTP basic authentication.</p> <p>Provide both user name and password if the proxy server has basic authentication enabled.</p> <p>Type: <code>string</code>.</p>

The following properties are not provided by default in the `.properties` configuration file. If you add them, they will be used.

Property	Description
CUMULOCITY_INITIAL_DELAY_SECS	<p>The initial delay (in seconds) that can be set for querying tenant subscriptions.</p> <p>Type: <code>float</code>.</p> <p>Default: 0 seconds.</p>

Property	Description
CUMULOCITY_MAX_BATCH_SIZE	<p>The maximum number of Apama events that can be batched as a single request before sending to Cumulocity IoT. The only event type that supports batching is <code>com.apama.cumulocity.Measurement</code>.</p> <p>Type: integer.</p> <p>Default: 1000.</p>
CUMULOCITY_LATENCY_SLOW_THRESHOLD_SECS	<p>Update the <code>mostRecentSlowRequestDetails</code> status (see “Monitoring status for Cumulocity IoT” on page 201) if the time for fetching one page of response multiplied by the number of total pages is greater than this threshold.</p> <p>Set this to 0 to disable updates.</p> <p>Type: integer.</p> <p>Default: 1 second.</p>
CUMULOCITY_LATENCY_LOG_THRESHOLD_SECS	<p>Log a warning if a single-paged or multi-paged request takes more time to complete than defined by this threshold.</p> <p>Set this to 0 to disable logging.</p> <p>Type: integer.</p> <p>Default: 10 seconds.</p>
CUMULOCITY_LATENCY_BATCH_THRESHOLD_SECS	<p>Log a warning if a batch of requests takes more time to complete than defined by this threshold. If a warning for an individual request of the batch has already been logged with <code>CUMULOCITY_LATENCY_LOG_THRESHOLD_SECS</code>, then a warning for this batch is not logged.</p> <p>Set this to 0 to disable logging.</p> <p>Type: integer.</p> <p>Default: 50 seconds.</p>
CUMULOCITY_SMS_SENDER_NAME	<p>The sender name to be used as the default if it is not specified in the <code>SendsMS</code> event and not configured in the</p>

Property	Description
	<p><code>messaging/sms.senderName</code> tenant option of Cumulocity IoT.</p> <p>The tenant option is given preference over the value of <code>CUMULOCITY_SMS_SENDER_NAME</code>. The tenant option is checked for every <code>SendsSMS</code> event. If the check does not find it in Cumulocity IoT, then only the value of <code>CUMULOCITY_SMS_SENDER_NAME</code> is used as the default sender name.</p> <p>Type: string.</p> <p>Default: Apama.</p>
<code>CUMULOCITY_SMS_SENDER_ADDRESS</code>	<p>The sender address to be used as the default if it is not specified in the <code>SendsSMS</code> event and not configured in the <code>messaging/sms.senderAddress</code> tenant option of Cumulocity IoT.</p> <p>The tenant option is given preference over the value of <code>CUMULOCITY_SMS_SENDER_ADDRESS</code>. The tenant option is checked for every <code>SendsSMS</code> event. If the check does not find it in Cumulocity IoT, then only the value of <code>CUMULOCITY_SMS_SENDER_ADDRESS</code> is used as the default sender address.</p> <p>You can provide a sender address in the following formats: <code>PROTOCOL:NUMBER</code> or just <code>NUMBER</code>. Valid protocols include <code>tel</code>, <code>SHORT</code>, <code>ICCID</code> and <code>ACR</code>. If the protocol is missing or invalid, <code>tel</code> is used as the default protocol.</p> <p>Type: string.</p> <p>Default: apama.</p>

For advanced use cases, it is possible to edit the following configuration options directly in the YAML configuration file. There are no corresponding entries in the `.properties` file.

Configuration option	Description
<code>requestAllDevices</code>	<p>Deprecated. Request all assets at startup.</p> <p>Type: boolean.</p>

Configuration option	Description
	Default: true. Note: requestAllDevices is set to false in the YAML configuration file. You should explicitly request for all available devices on startup using the <code>com.apama.cumulocity.FindManagedObject</code> API. For more information, see “Sample EPL” on page 204 .
subscribeToAllMeasurements	Subscribe to measurements, events and alarms of all devices during startup. Type: boolean. Default: true.
subscribeToDevices	Subscribe to all device-related updates. Type: boolean. Default: true.
subscribeToOperations	Subscribe to all device operations. Type: boolean. Default: false. Note: subscribeToOperations is set to false by default when it is not explicitly specified in the YAML configuration file. However, for your convenience, when you add a new Cumulocity Client connectivity bundle in Software AG Designer or by using the <code>apama_project</code> tool, this option is already set to true in the resulting YAML configuration file.

See also [“Receiving update notifications” on page 195](#).

Loading the Cumulocity IoT transport

You can load the Cumulocity IoT transport by adding the **Cumulocity Client** bundle to your project in Software AG Designer (see “Adding the Cumulocity IoT connectivity plug-in to a project” in *Using Apama with Software AG Designer*). Alternatively, you can load the transport with the following `connectivityPlugins` stanza in your YAML configuration file:

```
cumulocityTransport:
  classpath: ${APAMA_HOME}/lib/cumulocity/connectivity-cumulocity.jar
  class: com.apama.cumulocity.Transport
```

```
cumulocityCodec:
  libraryName: connectivity-cumulocity-codec
  class: CumulocityCodec
```

Using managed objects

During application initialization (`onApplicationInitialized`), if the `requestAllDevices` configuration option is enabled, the adapter sends all device/asset related information using the `com.apama.cumulocity.ManagedObject` event on the `com.apama.cumulocity.ManagedObject.SUBSCRIBE_CHANNEL` (same as `cumulocity.devices`) channel. After all devices/assets have been sent, the adapter sends a `com.apama.cumulocity.RequestAllDevicesComplete(-1)` event.

Note:

Use of the above-mentioned `requestAllDevices` configuration option is deprecated. Instead you should use the `com.apama.cumulocity.FindManagedObject` API to cause the adapter to send the device events when the application is ready. This will also work for applications deployed in Cumulocity IoT directly.

Example of a device event:

```
com.apama.cumulocity.ManagedObject("44578836","", "Device_1",
  ["c8y_Restart","c8y_Message","c8y_Relay"],
  ["c8y_TemperatureMeasurement","c8y_LightMeasurement"],
  [],[],[],[],{}),
{"c8y_IsDevice":any(dictionary<any,any>,new dictionary<any,any>),
 "owner":any(string,"Cumulocity_User")})
```

If the `subscribeToDevices` configuration option is enabled (`true` by default), any devices added to Cumulocity IoT after application initialization will be sent to the `com.apama.cumulocity.ManagedObject.SUBSCRIBE_CHANNEL` channel.

To fetch a list of all existing managed objects, use the `FindManagedObjects` API. For more information, see [“Querying for managed objects” on page 173](#).

Example

The following is a simple example of how to receive, update and send managed objects:

```
// Subscribe to receive all the devices from Cumulocity IoT
monitor.subscribe(ManagedObject.SUBSCRIBE_CHANNEL);

// Consume all the devices from Cumulocity IoT
on all ManagedObject() as mo {
  log mo.toString() at INFO;

  // Update a managed object

  mo.params.add("CustomMetadata", {"metadata": "Adding custom data"});
  send mo to ManagedObject.SEND_CHANNEL;
}
```

Updating a managed object

To enable use cases where information related to a managed object can be persisted, you can update any metadata information (such as the state) as properties of a managed object.

```
managedObject.params.add("<CUSTOM_PROPERTY>", <PROPERTY_VALUE>);  
send managedObject to com.apama.cumulocity.ManagedObject.SEND_CHANNEL
```

Where

- <CUSTOM_PROPERTY> is the property that is to be added.
- <PROPERTY_VALUE> is the value for the newly added property.

Sending managed objects requesting response and setting headers

When creating a new object or updating an existing one, it is recommended that you use the `withChannelResponse` action. This allows your application to receive a response on completion on the `ManagedObject.SUBSCRIBE_CHANNEL` channel. You will need to subscribe to the `ManagedObject.SUBSCRIBE_CHANNEL` channel first. The response can be one of two possibilities:

- `ObjectCommitted`

This includes the `reqId` which is the identifier of the original request, the `Id` which is the identifier of the newly created or updated object, and the actual object in JSON form.

- `ObjectCommitFailed`

This includes the `reqId` which is the identifier of the original request, the `statusCode` which is the HTTP status code of the failure, and the `body` which is the content of the response from the API (this might be in HTML format).

When using `withChannelResponse`, it allows the ability to set headers. This can be used, for example, to determine what processing mode Cumulocity IoT will use as shown in the example below.

Example of creating a managed object:

```
using com.apama.cumulocity.ManagedObject;  
using com.apama.cumulocity.ObjectCommitFailed;  
using com.apama.cumulocity.ObjectCommitted;  
  
monitor Test_ManagedObjects {  
  
    action onload {  
        monitor.subscribe(ManagedObject.SUBSCRIBE_CHANNEL);  
        ManagedObject mo := new ManagedObject;  
        mo.params.add("c8y_IsDevice", new dictionary<any, any>);  
        mo.name := "MyManagedObject";  
  
        mo.type := "DeviceType";  
        integer reqId := com.apama.cumulocity.Util.generateReqId();  
  
        // Create a new ManagedObject in Cumulocity, ensuring that a  
        // response is returned.  
        send mo.withChannelResponse(reqId, new dictionary<string, string>) to
```

```

        ManagedObject.SEND_CHANNEL;

        // If the ManagedObject creation succeeded do something with the
        // returned object or id.
        on ObjectCommitted(reqId=reqId) as c and not
        ObjectCommitFailed(reqId=reqId){
            log "New managed object successfully created " + c.toString()
            at INFO;
        }

        // If the ManagedObject creation failed, log an error.
        on ObjectCommitFailed(reqId=reqId) as cfail and not
        ObjectCommitted(reqId=reqId){
            log "Creating a new event failed " + cfail.toString() at ERROR;
        }
    }
}

```

Note:

The following ManagedObject reference fields cannot be set using ManagedObject events and are useful for read-only purposes in these events: childDeviceIds, childAssetIds, deviceParentIds, and assetParentIds. However, these fields can be set using the Cumulocity IoT REST API which can be invoked in EPL by using GenericRequest events. For more information, see “[Invoking other parts of the Cumulocity IoT REST API](#)” on page 199 and the information on child operations in the Cumulocity IoT OpenAPI documentation (<https://cumulocity.com/api/#tag/Child-operations>).

Querying for managed objects

To search for a managed object or a collection of managed objects, send the `com.apama.cumulocity.FindManagedObject` event to Cumulocity IoT, with a unique `reqId` to the `com.apama.cumulocity.FindManagedObject.SEND_CHANNEL` channel.

The transport will then respond with zero or more `com.apama.cumulocity.FindManagedObjectResponse` events and then one `com.apama.cumulocity.FindManagedObjectResponseAck` event on the `com.apama.cumulocity.FindManagedObjectResponse.SUBSCRIBE_CHANNEL` channel.

Example:

```

integer reqId := com.apama.cumulocity.Util.generateReqId();

com.apama.cumulocity.FindManagedObject request :=
    new com.apama.cumulocity.FindManagedObject;
request.reqId := reqId;

// Optionally provide the 'id' of the managed object
//request.deviceId := "<DEVICE_ID>";

// Filter based on fragmentType
request.params.add("fragmentType", "c8y_IsDevice");

// Subscribe to com.apama.cumulocity.FindManagedObjectResponse.SUBSCRIBE_CHANNEL to
// listen for responses
monitor.subscribe(com.apama.cumulocity.FindManagedObjectResponse.SUBSCRIBE_CHANNEL);

```

```
// Listen for responses based on reqId
on all com.apama.cumulocity.FindManagedObjectResponse(reqId=reqId) as response
// Avoid listener leaks by terminating the listener on completion of the request
and not com.apama.cumulocity.FindManagedObjectResponseAck(reqId=reqId)
{
    log "Received ManagedObject " + response.toString() at INFO;
}

// Listen for com.apama.cumulocity.FindManagedObjectResponseAck,
// this indicates that all responses have been received
on com.apama.cumulocity.FindManagedObjectResponseAck(reqId=reqId)
as requestCompleted
{
    log "Received all ManagedObject(s) for request "
        + requestCompleted.reqId.toString() at INFO;

    // Request is completed and we can unsubscribe from this channel
monitor.unsubscribe(com.apama.cumulocity.FindManagedObjectResponse.SUBSCRIBE_CHANNEL);
}

// Send request to find available managed objects
send request to com.apama.cumulocity.FindManagedObject.SEND_CHANNEL;
```

Supported query parameters

You can provide the following query parameters with the request:

Parameter	Description
deviceId	Search for a managed object based on deviceId. When deviceId is populated in a FindManagedObject request, all the query parameters listed below are ignored.
fragmentType	Search for managed objects based on the fragment type.
type	Search for managed objects based on the type.
owner	Search for managed objects based on the owner.
text	Search for managed objects based on the text.
childAssetId	Search for managed objects based on the asset identifier of the child.
childDeviceId	Search for managed objects based on the device identifier of the child.
ids	Search for managed objects based a comma-separated list of device identifiers.
pageSize	Indicates how many entries of the collection are to be retrieved from Cumulocity IoT. This is a batching parameter for getting multiple responses from Cumulocity IoT. A larger pageSize does fewer requests to Cumulocity IoT to retrieve all the managed objects, but each request

Parameter	Description
	is larger. By default, 1000 managed objects are in each page and there is an upper limit of 2000.
currentPage	Retrieve a specific page of results for the given pageSize. If currentPage is set, then only a single page is requested. If currentPage is not set (default), all the pages are requested.

For a comprehensive list of allowed query parameters, see the Cumulocity IoT OpenAPI Specification at <https://cumulocity.com/api/#operation/getManagedObjectCollectionResource>.

Query result paging

Cumulocity IoT queries support paging of data for requesting a specific range of the responses received. For more information, see “Paging Cumulocity IoT queries” on page 197.

Using alarms

The `com.apama.cumulocity.Alarm` is sent on an alarm from a device. This event is sent to the `com.apama.cumulocity.Alarm.SUBSCRIBE_CHANNEL` (same as `cumulocity.alarms`) channel. This event contains the identifier of the source, a timestamp (same form as `currentTime`), message text, and optional parameters.

Example of an alarm:

```
com.apama.cumulocity.Alarm("44578840","c8y_UnavailabilityAlarm","44578839",
  1529496204.346,"No data received from device within required interval",
  "ACTIVE","MAJOR",1,{"creationTime":any(float,1529496204.067)})
```

Example

The following is a simple example of how to receive, update, create and send alarms:

```
// Subscribe to receive all the alarms published from Cumulocity IoT
monitor.subscribe(Alarm.SUBSCRIBE_CHANNEL);

// Consume all the alarms from Cumulocity IoT
on all Alarm() as alarm {
  log alarm.toString() at INFO;

  // Example for updating an alarm

  // Set alarm severity to MAJOR
  alarm.severity := "MAJOR";
  send alarm to Alarm.SEND_CHANNEL;
}

// Create a new alarm
Alarm alarm := new Alarm;
alarm.source := "<MANAGED_OBJECT_ID>";
alarm.type := "TestAlarm";
alarm.severity := "MINOR";
```

```
alarm.status := "ACTIVE";
alarm.time := currentTime;
alarm.text := "This is a sample alarm";
send alarm to Alarm.SEND_CHANNEL;
```

Creating a new alarm

```
send Alarm("", "c8y_SampleAlarm", "<SOURCE>", <TIME>,
  "Alarm text", "<STATUS>", "<SEVERITY>", 1, new dictionary<string, any>) to
Alarm.SEND_CHANNEL;
```

Where

- `<SOURCE>` is the source of the alarm (same as the ManagedObject identifier).
- `<TIME>` is the time at which the alarm was generated.
- `<STATUS>` is the status of the alarm. This can be ACTIVE, ACKNOWLEDGED or CLEARED.
- `<SEVERITY>` is the severity of the alarm. This can be CRITICAL, MAJOR, MINOR or WARNING.

Sending alarms requesting response and setting headers

When creating a new object or updating an existing one, it is recommended that you use the `withChannelResponse` action. This allows your application to receive a response on completion on the `Alarm.SUBSCRIBE_CHANNEL` channel. You will need to subscribe to the `Alarm.SUBSCRIBE_CHANNEL` channel first. The response can be one of two possibilities:

- `ObjectCommitted`

This includes the `reqId` which is the identifier of the original request, the `Id` which is the identifier of the newly created or updated object, and the actual object in JSON form.

- `ObjectCommitFailed`

This includes the `reqId` which is the identifier of the original request, the `statusCode` which is the HTTP status code of the failure, and the `body` which is the content of the response from the API (this might be in HTML format).

When using `withChannelResponse`, it allows the ability to set headers. This can be used, for example, to determine what processing mode Cumulocity IoT will use as shown in the example below.

Example of creating an alarm:

```
using com.apama.cumulocity.Alarm;
using com.apama.cumulocity.ObjectCommitFailed;
using com.apama.cumulocity.ObjectCommitted;

monitor TestCreatingAlarm {
  string deviceId; // Where this is populated from the actual device Id.
  string timestamp; // Where this is populated from the timestamp of
                  // the device.

  action onload {
    monitor.subscribe(Alarm.SUBSCRIBE_CHANNEL);
```



```

Alarm al := new Alarm;
string name := "MyTestAlarm";
al.status := "ACTIVE";
al.severity := "CRITICAL";
al.source := deviceId;
al.type := "c8y_TestAlarm";
al.text := "test alarm";
al.time := currentTime;

integer reqId := com.apama.cumulocity.Util.generateReqId();

// Create a new Alarm in Cumulocity, ensuring that a response is
// returned
// and the processing mode, indicating how to process the request,
// sent to Cumulocity is Transient.
send al.withChannelResponse(reqId, { "X-Cumulocity-Processing-Mode":
  "Transient" }) to Alarm.SEND_CHANNEL;

// If the Alarm creation succeeded do something with the returned
// object or id.
on ObjectCommitted(reqId=reqId) as c and not
  ObjectCommitFailed(reqId=reqId){
  log "New alarm successfully created " + c.toString() at INFO;
}

// If the Alarm creation failed, log an error.
on ObjectCommitFailed(reqId=reqId) as cfail and not
  ObjectCommitted(reqId=reqId){
  log "Creating a new event failed " + cfail.toString() at ERROR;
}
}
}

```

Alarm de-duplication

If an active or acknowledged alarm (does not work for the CLEARED status) with the same source and type exists, no new alarm is created. Instead, the existing alarm is updated by incrementing the count property, and the time property is also updated. Any other changes are ignored, and the alarm history is not updated. The first occurrence of the alarm is recorded in `firstOccurenceTime`.

Updating an existing alarm

You can update the text, status and severity fields.

```

send Alarm("<ALARM_ID>", "c8y_SampleAlarm", "<SOURCE>", "<TIME>",
  "Alarm Updated", "<STATUS>", "<SEVERITY>", 1, new dictionary<string, any>) to
Alarm.SEND_CHANNEL;

```

Where

- `<ALARM_ID>` is the identifier of the previously created alarm. The presence of `<ALARM_ID>` indicates that the request is for updating an existing alarm.

Querying for alarms

To search for an alarm or a collection of alarms, send the `com.apama.cumulocity.FindAlarm` event to Cumulocity IoT, with a unique `reqId` to the `com.apama.cumulocity.FindAlarm.SEND_CHANNEL` channel.

The transport will then respond with zero or more `com.apama.cumulocity.FindAlarmResponse` events and then one `com.apama.cumulocity.FindAlarmResponseAck` event on the `com.apama.cumulocity.FindAlarmResponse.SUBSCRIBE_CHANNEL` channel.

Example:

```
integer reqId := com.apama.cumulocity.Util.generateReqId();

com.apama.cumulocity.FindAlarm request := new com.apama.cumulocity.FindAlarm;
request.reqId := reqId;

// Filter based on alarms type
request.params.add("type", "c8y_UnavailabilityAlarm");

// Subscribe to com.apama.cumulocity.FindAlarmResponse.SUBSCRIBE_CHANNEL to listen
// for responses
monitor.subscribe(com.apama.cumulocity.FindAlarmResponse.SUBSCRIBE_CHANNEL);

// Listen for responses based on reqId
on all com.apama.cumulocity.FindAlarmResponse(reqId=reqId) as response
// Avoid listener leaks by terminating the listener on completion of the request
and not com.apama.cumulocity.FindAlarmResponseAck(reqId=reqId)
{
    log "Received Alarm " + response.toString() at INFO;
}

// Listen for com.apama.cumulocity.FindAlarmResponseAck,
// this indicates that all responses have been received
on com.apama.cumulocity.FindAlarmResponseAck(reqId=reqId) as requestCompleted
{
    log "Received all Alarm(s) for request " +
        requestCompleted.reqId.toString() at INFO;

    // Request is completed and we can unsubscribe from this channel
    monitor.unsubscribe(com.apama.cumulocity.FindAlarmResponse.SUBSCRIBE_CHANNEL);
}

// Send request to find available alarms
send request to com.apama.cumulocity.FindAlarm.SEND_CHANNEL;
```

Supported query parameters

You can provide the following query parameters with the request:

Parameter	Description
id	Search for an alarm based on <code>alarmId</code> . When searching for an alarm based on <code>Id</code> , all the query parameters listed below are ignored.

Parameter	Description
source	Search for alarms based on the device identifier or asset identifier of the source.
status	Search for alarms based on the status. The status can be any of ACTIVE, ACKNOWLEDGED or CLEARED.
severity	Search for alarms based on the severity. The severity can be any of CRITICAL, MAJOR, MINOR or WARNING.
type	Search for alarms based on the type.
fromDate	Search for alarms from a start date. The date and time is provided as seconds since the epoch.
toDate	Search for alarms within a time range. This is to be used in combination with fromDate. The date and time is provided as seconds since the epoch.
resolved	A boolean parameter used for filtering, based on the resolved state.
pageSize	Indicates how many entries of the collection are to be retrieved from Cumulocity IoT. This is a batching parameter for getting multiple responses from Cumulocity IoT. A larger pageSize does fewer requests to Cumulocity IoT to retrieve all the alarms, but each request is larger. By default, 1000 alarms are in each page and there is an upper limit of 2000.
currentPage	Retrieve a specific page of results for the given pageSize. If currentPage is set, then only a single page is requested. If currentPage is not set (default), all the pages are requested.

Query result paging

Cumulocity IoT queries support paging of data for requesting a specific range of the responses received. For more information, see [“Paging Cumulocity IoT queries” on page 197](#).

Using events

The `com.apama.cumulocity.Event` is sent on an event from a device. This event is sent to the `com.apama.cumulocity.Event.SUBSCRIBE_CHANNEL` (same as `cumulocity.events`) channel. This event contains the identifier of the source, a timestamp (same form as `currentTime`), message text, and optional parameters.

Example of an event:

```
com.apama.cumulocity.Event("48073557","c8y_EntranceEvent",
    "12346082",1519838833.6,
    "Entrance event triggered.",
    {"creationTime":any(float,1519838834.706)})
```

Example

The following is a simple example of how to receive, update, create and send events:

```
// Subscribe to receive all the events published from Cumulocity IoT
monitor.subscribe(Event.SUBSCRIBE_CHANNEL);

// Consume all the events from Cumulocity IoT
on all Event() as e {
    log e.toString() at INFO;

    // Example for updating an event

    // Update text
    e.text := "This is an updated text";
    send e to Event.SEND_CHANNEL;
}

// Create a new event
Event evt := new Event;
evt.source := "<MANAGED_OBJECT_ID>";
evt.type := "TestEvent";
evt.time := currentTime;
evt.text := "This is a sample event";
send evt to Event.SEND_CHANNEL;
```

Creating a new event

```
send Event("", "c8y_SampleEvent", "<SOURCE>", <TIME>,
    "Event text", new dictionary<string, any>) to Event.SEND_CHANNEL;
```

Where

- `<SOURCE>` is the source of the event (same as the ManagedObject identifier).
- `<TIME>` is the time at which the event was generated.

Sending events requesting response and setting headers

When creating a new object or updating an existing one, it is recommended that you use the `withChannelResponse` action. This allows your application to receive a response on completion on the `Event.SUBSCRIBE_CHANNEL` channel. You will need to subscribe to the `Event.SUBSCRIBE_CHANNEL` channel first. The response can be one of two possibilities:

- `ObjectCommitted`

This includes the `reqId` which is the identifier of the original request, the `Id` which is the identifier of the newly created or updated object, and the actual object in JSON form.

- `ObjectCommitFailed`

This includes the `reqId` which is the identifier of the original request, the `statusCode` which is the HTTP status code of the failure, and the `body` which is the content of the response from the API (this might be in HTML format).

When using `withChannelResponse`, it allows the ability to set headers. This can be used, for example, to determine what processing mode Cumulocity IoT will use as shown in the example below.

Example of creating an event:

```
using com.apama.cumulocity.Event;
using com.apama.cumulocity.ObjectCommitFailed;
using com.apama.cumulocity.ObjectCommitted;

monitor Test_CumulocityEvents {
    string timestamp; // Where this is populated from the timestamp of the
                    // device.

    action onload {
        monitor.subscribe(Event.SUBSCRIBE_CHANNEL);
        Event ev := new Event;
        string name := "MyEvent";
        ev.type := "DoorSensor";
        ev.source := "7104838";
        ev.text := "Door sensor was triggered";
        ev.time := currentTime;

        integer reqId := com.apama.cumulocity.Util.generateReqId();
        // Create a new Event in Cumulocity, ensuring that a response is
        // returned
        // and the processing mode, indicating how to process the request, sent
        // to Cumulocity is Transient.
        send ev.withChannelResponse(reqId, { "X-Cumulocity-Processing-Mode":
            "Transient" }) to Event.SEND_CHANNEL;

        // If the Event creation succeeded do something with the returned
        // object or id.
        on ObjectCommitted(reqId=reqId) as c and not
            ObjectCommitFailed(reqId=reqId){
            log "New event successfully created " + c.toString() at INFO;
        }

        // If the Event creation failed, log an error.
        on ObjectCommitFailed(reqId=reqId) as cfail and not
            ObjectCommitted(reqId=reqId){
            log "Creating a new event failed " + cfail.toString() at ERROR;
        }
    }
}
```

Updating an existing event

You can update the text field.

```
send Event("<EVENT_ID>", "c8y_SampleEvent", "<SOURCE>", "<TIME>",
    "Event Updated", new dictionary<string, any>) to Event.SEND_CHANNEL;
```

Where

- `<EVENT_ID>` is the identifier of the previously created event. The presence of `<EVENT_ID>` indicates that the request is for updating an existing event.

Querying for events

To search for an event or a collection of events, send the `com.apama.cumulocity.FindEvent` event to Cumulocity IoT, with a unique `reqId` to the `com.apama.cumulocity.FindEvent.SEND_CHANNEL` channel.

The transport will then respond with zero or more `com.apama.cumulocity.FindEventResponse` events and then one `com.apama.cumulocity.FindEventResponseAck` event on the `com.apama.cumulocity.FindEventResponse.SUBSCRIBE_CHANNEL` channel.

Example:

```
integer reqId := com.apama.cumulocity.Util.generateReqId();

com.apama.cumulocity.FindEvent request := new com.apama.cumulocity.FindEvent;
request.reqId := reqId;

// Filter based on event type
request.params.add("type", "c8y_DoorOpenedEvent");

// Subscribe to com.apama.cumulocity.FindEventResponse.SUBSCRIBE_CHANNEL to listen
// for responses
monitor.subscribe(com.apama.cumulocity.FindEventResponse.SUBSCRIBE_CHANNEL);

// Listen for responses based on reqId
on all com.apama.cumulocity.FindEventResponse(reqId=reqId) as response
// Avoid listener leaks by terminating the listener on completion of the request
and not com.apama.cumulocity.FindEventResponseAck(reqId=reqId)
{
    log "Received Event " + response.toString() at INFO;
}

// Listen for com.apama.cumulocity.FindEventResponseAck,
// this indicates that all responses have been received
on com.apama.cumulocity.FindEventResponseAck(reqId=reqId) as requestCompleted
{
    log "Received all Event(s) for request " +
        requestCompleted.reqId.toString() at INFO;

    // Request is completed and we can unsubscribe from this channel
    monitor.unsubscribe(com.apama.cumulocity.FindEventResponse.SUBSCRIBE_CHANNEL);
}

// Send request to find available events
send request to com.apama.cumulocity.FindEvent.SEND_CHANNEL;
```

Supported query parameters

You can provide the following query parameters with the request:

Parameter	Description
id	Search for an event based on <code>eventId</code> . When searching for an event based on <code>Id</code> , all the query parameters listed below are ignored.

Parameter	Description
source	Search for events based on the device identifier or asset identifier of the source.
type	Search for events based on the type.
fromDate	Search for events from a start date. The date and time is provided as seconds since the epoch.
toDate	Search for events within a time range. This is to be used in combination with <code>fromDate</code> . The date and time is provided as seconds since the epoch.
fromCreationDate	Similar to <code>fromDate</code> , but fetches the events based on the creation date. The date and time is provided as seconds since the epoch.
toCreationDate	Search for events that have been created within a date range. This is to be used in combination with <code>fromCreationDate</code> . The date and time is provided as seconds since the epoch.
pageSize	Indicates how many entries of the collection are to be retrieved from Cumulocity IoT. This is a batching parameter for getting multiple responses from Cumulocity IoT. A larger <code>pageSize</code> does fewer requests to Cumulocity IoT to retrieve all the events, but each request is larger. By default, 1000 events are in each page and there is an upper limit of 2000.
currentPage	Retrieve a specific page of results for the given <code>pageSize</code> . If <code>currentPage</code> is set, then only a single page is requested. If <code>currentPage</code> is not set (default), all the pages are requested.
revert	Boolean parameter. If a range query is used (that is, the query includes at least one of the <code>fromDate</code> or <code>toDate</code> parameters), you can reverse the order in which the matching events are returned by adding the query parameter <code>revert=true</code> . This returns the oldest events first. By default, Cumulocity IoT returns the latest events first.

Query result paging

Cumulocity IoT queries support paging of data for requesting a specific range of the responses received. For more information, see [“Paging Cumulocity IoT queries” on page 197](#).

Using measurements

During application initialization (`onApplicationInitialized`), if `subscribeToAllMeasurements` is enabled (`true` by default), the adapter sends all measurements using the `com.apama.cumulocity.Measurement` event on the `com.apama.cumulocity.Measurement.SUBSCRIBE_CHANNEL` (same as `cumulocity.measurements`) channel.

These events may be sent before all assets are sent. Measurement events contain the identifier of the source of the measurement, the type of measurement, timestamp, and a dictionary of values which contain the numeric value, units and optional type, quantity and state.

Examples of measurement events:

```
Measurement("1001","c8y_LightMeasurement","12346081",1464359004.89,
  {"c8y_LightMeasurement": {"e":com.apama.cumulocity.MeasurementValue(108.1,
    "lux", new dictionary<string, any>)}}},new dictionary<string, any>)

Measurement("1002","c8y_DistanceMeasurement","12346082",1464359005.396,
  {"c8y_DistanceMeasurement": {"distance":com.apama.cumulocity.MeasurementValue
    (344,"mm","", "", "", "", dictionary<string, any>)}}}, dictionary<string, any>)
```

Example

The following is a simple example of how to receive, create and send measurements:

```
// Subscribe to receive all the measurements published from Cumulocity IoT
monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);

// Consume all the measurements from Cumulocity IoT
on all Measurement() as m {
  log m.toString() at INFO;
}

// Create a new measurement

Measurement m := new Measurement;
m.source := "<MANAGED_OBJECT_ID>";
m.time := currentTime;
m.type := "TemperatureMeasurement";
MeasurementValue mv := new MeasurementValue;
mv.value := 100.0;
dictionary<string, MeasurementValue> fragment
  := new dictionary<string, MeasurementValue>;
fragment.add("temperature", mv);
m.measurements.add("TemperatureMeasurement", fragment);
send m to Measurement.SEND_CHANNEL;
```

Creating a new measurement

```
Measurement m := new Measurement;
m.type := <MEASUREMENT_TYPE>;
m.source := <SOURCE>;
m.time := currentTime;
MeasurementValue mv := new MeasurementValue;
mv.value := 1.0;
mv.unit := "V";
dictionary<string, MeasurementValue> dict := {"voltage": mv};
m.measurements.add(m.type, dict);
send m to Measurement.SEND_CHANNEL;
```

Where

- `<SOURCE>` is the source of the measurement (same as the ManagedObject identifier).

- `<MEASUREMENT_TYPE>` is the type of the measurement. For example, `c8y_VoltageMeasurement`.

Sending measurements requesting response and setting headers

When creating a new object, it is recommended that you use the `withChannelResponse` action. This allows your application to receive a response on completion on the `Measurement.SUBSCRIBE_CHANNEL` channel. You will need to subscribe to the `Measurement.SUBSCRIBE_CHANNEL` channel first. The response can be one of two possibilities:

- `ObjectCommitted`

This includes the `reqId` which is the identifier of the original request, the `Id` which is the identifier of the newly created or updated object, and the actual object in JSON form.

- `ObjectCommitFailed`

This includes the `reqId` which is the identifier of the original request, the `statusCode` which is the HTTP status code of the failure, and the `body` which is the content of the response from the API (this might be in HTML format).

When using `withChannelResponse`, it allows the ability to set headers. This can be used, for example, to determine what processing mode Cumulocity IoT will use as shown in the example below.

It is worth noting that when using `withChannelResponse` for measurements, it is not able to achieve the same throughput as sending them without a response. As they are not batched into a single HTTP request, there are just individual create/update requests sent to Cumulocity IoT.

Example of creating a measurement:

```
using com.apama.cumulocity.Measurement;
using com.apama.cumulocity.MeasurementValue;

using com.apama.cumulocity.ObjectCommitFailed;
using com.apama.cumulocity.ObjectCommitted;

monitor Test {
    string deviceId; // Where this is populated from the actual device Id.
    float timestamp; // Where this is populated from the timestamp of the
                    // device.

    action onload {

        monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
        Measurement mo := new Measurement;
        mo.type := "test_measurement";
        mo.source := deviceId;
        mo.time := timestamp;

        //Create a Measurement with two Measurement fragments.
        MeasurementValue mv1 := new MeasurementValue;
        mv1.value := 10.2;
        mv1.unit := "km/hr";

        MeasurementValue mv2 := new MeasurementValue;
        mv2.value := 11.7;
        mv2.unit := "km/hr";
    }
}
```

```

dictionary<string, MeasurementValue> dict :=
    {"speedX": mv1, "speedY": mv2};
mo.measurements.add("c8y_speed", dict);

integer reqId := com.apama.cumulocity.Util.generateReqId();
// Create a new Measurement in Cumulocity, ensuring that a response is
// returned
// and the processing mode, indicating how to process the request,
// sent to Cumulocity is Transient.
send mo.withChannelResponse(reqId, { "X-Cumulocity-Processing-Mode":
    "Transient" }) to Measurement.SEND_CHANNEL;

// If the Measurement creation succeeded do something with the returned
// object or id.
on ObjectCommitted(reqId=reqId) as c and not
    ObjectCommitFailed(reqId=reqId){
    log "New measurement successfully created " + c.toString() at INFO;
}

// If the Measurement creation failed, log an error.
on ObjectCommitFailed(reqId=reqId) as cfail and not
    ObjectCommitted(reqId=reqId){
    log "Creating a new event failed " + cfail.toString() at ERROR;
}
}
}

```

Querying for measurements

To search for a measurement or a collection of measurements, send the `com.apama.cumulocity.FindMeasurement` event to Cumulocity IoT, with a unique `reqId` to the `com.apama.cumulocity.FindMeasurement.SEND_CHANNEL` channel.

The transport will then respond with zero or more `com.apama.cumulocity.FindMeasurementResponse` events and then one `com.apama.cumulocity.FindMeasurementResponseAck` event on the `com.apama.cumulocity.FindMeasurementResponse.SUBSCRIBE_CHANNEL` channel.

Example:

```

integer reqId := com.apama.cumulocity.Util.generateReqId();

com.apama.cumulocity.FindMeasurement request :=
    new com.apama.cumulocity.FindMeasurement;
request.reqId := reqId;

// Filter based on measurement fragment type and series
request.params.add("valueFragmentType", "c8y_MotionMeasurement");
request.params.add("valueFragmentSeries", "motionDetected");

// Subscribe to com.apama.cumulocity.FindMeasurementResponse.SUBSCRIBE_CHANNEL
// to listen for responses
monitor.subscribe(com.apama.cumulocity.FindMeasurementResponse.SUBSCRIBE_CHANNEL);

// Listen for responses based on reqId
on all com.apama.cumulocity.FindMeasurementResponse(reqId=reqId) as response
// Avoid listener leaks by terminating the listener on completion of the request

```

```

and not com.apama.cumulocity.FindMeasurementResponseAck(reqId=reqId)
{
    log "Received Measurement " + response.toString() at INFO;
}

// Listen for com.apama.cumulocity.FindMeasurementResponseAck,
// this indicates that all responses have been received
on com.apama.cumulocity.FindMeasurementResponseAck(reqId=reqId)
as requestCompleted
{
    log "Received all Measurement(s) for request "
        + requestCompleted.reqId.toString() at INFO;

    // Request is completed and we can unsubscribe from this channel
    monitor.unsubscribe(com.apama.cumulocity.FindMeasurementResponse.SUBSCRIBE_CHANNEL);
}

// Send request to find available measurements
send request to com.apama.cumulocity.FindMeasurement.SEND_CHANNEL;

```

Supported query parameters

You can provide the following query parameters with the request:

Parameter	Description
id	Search for a measurement based on measurementId. When searching for a measurement based on Id, all the query parameters listed below are ignored.
source	Search for measurements based on the device identifier or asset identifier of the source.
type	Search for measurements based on the type.
valueFragmentType	Search for measurements based on fragment type (should be used with valueFragmentSeries).
valueFragmentSeries	Search for measurements based on fragment series (should be used with valueFragmentType).
fromDate	Search for measurements from a start date. The date and time is provided as seconds since the epoch.
toDate	Search for measurements within a time range. This is to be used in combination with fromDate. The date and time is provided as seconds since the epoch.
pageSize	Indicates how many entries of the collection are to be retrieved from Cumulocity IoT. This is a batching parameter for getting multiple responses from Cumulocity IoT. A larger pageSize does fewer requests to Cumulocity IoT to retrieve all the measurements, but each request is larger. By default, 1000 measurements are in each page and there is an upper limit of 2000.

Parameter	Description
currentPage	Retrieve a specific page of results for the given pageSize. If currentPage is set, then only a single page is requested. If currentPage is not set (default), all the pages are requested.
revert	Boolean parameter. If a range query is used (that is, the query includes at least one of the fromDate or toDate parameters), you can reverse the order in which the matching measurements are returned by adding the query parameter revert=true. This returns the latest measurements first. By default, Cumulocity IoT returns the oldest measurements first.

Query result paging

Cumulocity IoT queries support paging of data for requesting a specific range of the responses received. For more information, see [“Paging Cumulocity IoT queries” on page 197](#).

Using measurement fragments

A MeasurementFragment event represents a single fragment/series on a measurement.

Creating measurement fragments

You can send a single fragment to Cumulocity IoT to create a single-fragment measurement.

Example of creating a measurement fragment:

```
using com.apama.cumulocity.MeasurementFragment;
using com.apama.cumulocity.ObjectCommitFailed;
using com.apama.cumulocity.ObjectCommitted;

monitor Test {
    string deviceId; // Where this is populated from the actual device Id.
    float timestamp; // Where this is populated from the timestamp of the
                    // device.

    action onload {
        MeasurementFragment mf := new MeasurementFragment;
        mf.type := "test_measurement";
        mf.source := deviceId;
        mf.time := timestamp;

        mf.valueFragment := "c8y_speed";
        mf.valueSeries := "speedX";
        mf.value := 12.0;
        mf.unit := "km/hr";

        integer reqId := com.apama.cumulocity.Util.generateReqId();

        send mf to MeasurementFragment.SEND_CHANNEL;
    }
}
```

Where

- `source` is the source of the measurement.
- `time` is the time at which the measurement was taken.
- `type` is the type of the measurement.
- `valueFragment` is the name of the fragment of the measurement fragment.
- `valueSeries` is the name of the series of the measurement fragment.
- `value` is the value from the sensor.
- `unit` is the units the reading is in, for example, mm, lux, km/hr.

Sending measurement fragments requesting a response and setting headers

When creating a new object, it is recommended that you use the `withChannelResponse` action. This allows your application to receive a response on completion on the `MeasurementFragment.SUBSCRIBE_CHANNEL` channel. You will need to subscribe to the `MeasurementFragment.SUBSCRIBE_CHANNEL` channel first. The response can be one of two possibilities:

- `ObjectCommitted`

This includes the `reqId` which is the identifier of the original request, the `Id` which is the identifier of the newly created or updated object, and the actual object in JSON form.

- `ObjectCommitFailed`

This includes the `reqId` which is the identifier of the original request, the `statusCode` which is the HTTP status code of the failure, and the `body` which is the content of the response from the API (this might be in HTML format).

When using `withChannelResponse`, it allows the ability to set headers. This can be used, for example, to determine what processing mode Cumulocity IoT will use as shown in the example below.

Example of creating a measurement fragment:

```
using com.apama.cumulocity.MeasurementFragment;
using com.apama.cumulocity.ObjectCommitFailed;
using com.apama.cumulocity.ObjectCommitted;

monitor Test {
    string deviceId; // Where this is populated from the actual device Id.
    string timestamp; // Where this is populated from the timestamp of the
                    // device.

    action onload {
        monitor.subscribe(MeasurementFragment.SUBSCRIBE_CHANNEL);
        MeasurementFragment mf := new MeasurementFragment;
        mf.type := "test_measurement";
        mf.source := deviceId;
        mf.time := timestamp;

        mf.valueFragment := "c8y_speed";
    }
}
```

```

mf.valueSeries := "speedX";
mf.value := 12.0;
mf.unit := "km/hr";

integer reqId := com.apama.cumulocity.Util.generateReqId();

// Create a new Measurement in Cumulocity from a single
// MeasurementFragment, ensuring that a response is returned
// and the processing mode, indicating how to process the request, sent
// to Cumulocity is Transient.
send mf.withChannelResponse(reqId, { "X-Cumulocity-Processing-Mode":
  "Transient" }) to MeasurementFragment.SEND_CHANNEL;

// If the Measurement creation succeeded do something with the returned
// object or id.
on ObjectCommitted(reqId=reqId) as c and not
  ObjectCommitFailed(reqId=reqId){
  log "New measurement successfully created " + c.toString() at INFO;
}

// If the Measurement creation failed, log an error.
on ObjectCommitFailed(reqId=reqId) as cfail and not
  ObjectCommitted(reqId=reqId){
  log "Creating a new measurement failed " + cfail.toString()
    at ERROR;
}
}
}

```

Listening for measurement fragments

The Apama mapping codec can turn measurements into measurement fragments, and listeners in EPL can match on the elements of measurement fragments rather than measurements.

Example - matching on measurement fragments:

```

on all MeasurementFragment(valueFragment = 'c8y_speed', valueSeries = 'speedX',
  value > SPEED_LIMIT) as mf {
}

```

Turning measurement fragments on/off

To be able to match based on measurement fragments, you must ensure they are returned by setting the correct measurement format. There are two modes available, `MEASUREMENT_ONLY` and `BOTH`. The default, if it is not set or set incorrectly, is `MEASUREMENT_ONLY`. Set the mode to `BOTH` if you require filtering based on fragments or series.

If you are deploying a custom microservice, connecting to Cumulocity IoT from an external correlator, or using Software AG Designer, you can set the mode in the `CumulocityIoT.properties` file (see also [“Configuring the Cumulocity IoT transport” on page 165](#)) or directly on the command line to start the correlator by setting the `CUMULOCITY_MEASUREMENT_FORMAT` value.

The recommended approach is to set the mode from the `.properties` file. For example, to turn measurement fragments on:

```
CUMULOCITY_MEASUREMENT_FORMAT=BOTH
```

Alternatively, you can set the mode using the command line. For example, to turn measurement fragments off:

```
-DCUMULOCITY_MEASUREMENT_FORMAT=MEASUREMENT_ONLY
```

Note:

As of Apama 10.5, new Apama projects in Software AG Designer have the default set to `BOTH`, but existing projects will retain their previous configuration. If you want to enable fragments in an existing project, you may need to remove and re-add the bundle.

When you deploy (activate) an application directly in Cumulocity IoT using the Apama EPL Apps web application, both measurements and measurement fragments are always available (this is always `BOTH`). See the *Streaming Analytics guide* at <http://cumulocity.com/guides/> for more information.

Handling measurement fragments

It is possible to separate the individual fragments from the contents of a `Measurement` into `MeasurementFragment` objects, which can allow better performance matching in searches. You achieve this by using the sequence `<MeasurementFragment> getFragments()` action on the `Measurement` event. This returns a sequence of `MeasurementFragment` objects.

You can generate a `Measurement` event based on `MeasurementFragment` objects. You can achieve this by using the static `Measurement createFromFragments(sequence<MeasurementFragment> fragments)` action on the `Measurement` event, where `fragments` is the sequence of `MeasurementFragment` objects to create it from, and it returns the created `Measurement`.

Using operations

The `com.apama.cumulocity.Operation` event represents a device operation. If the configuration option `subscribeToOperations` is enabled (see “[Configuring the Cumulocity IoT transport](#)” on [page 165](#)) or if you subscribe to the operations stream, this event is sent to the `com.apama.cumulocity.Operation.SUBSCRIBE_CHANNEL` (same as `cumulocity.operations`) channel. This event contains the unique identifier of the operation (`id`), the identifier of the source (`deviceId`), a status, and optional parameters.

Example of an operation:

```
Operation("12345", "deviceId", "EXECUTING", params)
```

where `params` is a dictionary of string keys and any values (`dictionary<string, any>`).

Make sure to set the `deviceId` field to the identifier of a managed object which has the `com_cumulocity_model_Agent` fragment. The `com_cumulocity_model_Agent` marks devices running a Cumulocity IoT agent. Such devices receive all operations targeted to themselves and their children for routing (see also [Device integration using REST](#) in Cumulocity IoT's *Device SDK guide*).

When creating a new operation, do not supply the `id` field (that is, supply an empty string for the operation identifier).

It is not possible to set the `params` field of an operation to an empty dictionary.

Example

The following is a simple example of how to receive, create and send operations:

```
// Subscribe to receive all the operations published from Cumulocity IoT
monitor.subscribe(Operation.SUBSCRIBE_CHANNEL);

on all Operation() as o {
  log o.toString() at INFO;

  // Update an operation
  o.status := "EXECUTING";
  send o to Operation.SEND_CHANNEL;
}

// Create an operation
Operation operation := new Operation;
operation.source := "<MANAGED_OBJECT_ID>";
operation.status := "PENDING";
operation.params.add("c8y_Message", {"text": "Device Operation"});
send operation to Operation.SEND_CHANNEL;
```

Creating a new operation

```
send com.apama.cumulocity.Operation("", "<SOURCE>", "<STATUS>",
{"c8y_Message":<any> {"text":<any>"Hello Cumulocity device"}} )
to com.apama.cumulocity.Operation.SEND_CHANNEL;
```

Where

- `<SOURCE>` is the source of the operation (same as the `ManagedObject` identifier).
- `<STATUS>` is the status of the operation. This can be `PENDING`.

Sending operations requesting response and setting headers

When creating a new object or updating an existing one, it is recommended that you use the `withChannelResponse` action. This allows your application to receive a response on completion on the `Operation.SUBSCRIBE_CHANNEL` channel. You will need to subscribe to the `Operation.SUBSCRIBE_CHANNEL` channel first. The response can be one of two possibilities:

- `ObjectCommitted`

This includes the `reqId` which is the identifier of the original request, the `Id` which is the identifier of the newly created or updated object, and the actual object in JSON form.

- `ObjectCommitFailed`

This includes the `reqId` which is the identifier of the original request, the `statusCode` which is the HTTP status code of the failure, and the `body` which is the content of the response from the API (this might be in HTML format).

When using `withChannelResponse`, it allows the ability to set headers. This can be used, for example, to determine what processing mode Cumulocity IoT will use as shown in the example below.

Example of creating an operation:

```
using com.apama.cumulocity.Operation;
using com.apama.cumulocity.ObjectCommitFailed;
using com.apama.cumulocity.ObjectCommitted;

monitor Test_Operations {

    action onload {
        monitor.subscribe(Operation.SUBSCRIBE_CHANNEL);
        Operation op := new Operation;
        string name := "CreateOperation";
        op.source := "7104835";
        op.status := "PENDING";
        op.params :=
            {"c8y_Message":any(dictionary<any,any>,
                {any(string,"text"):
                    any(string,"Hello Cumulocity device")
                })
            };

        integer reqId := com.apama.cumulocity.Util.generateReqId();
        // Create a new Operation in Cumulocity, ensuring that a response is
        // returned.
        send op.withChannelResponse(reqId, new dictionary<string, string>) to
            Operation.SEND_CHANNEL;

        // If the Operation creation succeeded do something with the returned
        // object or id.
        on ObjectCommitted(reqId=reqId) as c and not
            ObjectCommitFailed(reqId=reqId){
            log "New operation successfully created " + c.toString() at INFO;
        }

        // If the Operation creation failed, log an error.
        on ObjectCommitFailed(reqId=reqId) as cfail and not
            ObjectCommitted(reqId=reqId){
            log "Creating a new event failed " + cfail.toString() at ERROR;
        }
    }
}
```

Updating an existing operation

You can update the status field.

```
send com.apama.cumulocity.Operation("<OPERATION_ID>","<SOURCE>","<STATUS>",
{"c8y_Message":<any> {"<any>"text":<any>"Updated Cumulocity device"}} )
to com.apama.cumulocity.Operation.SEND_CHANNEL;
```

Where

- `<OPERATION_ID>` is the identifier of the previously created operation. The presence of `<OPERATION_ID>` indicates that the request is for updating an existing operation.

- `<SOURCE>` is the source of the operation (same as the `ManagedObject` identifier).
- `<STATUS>` is the status of the operation. This can be `PENDING`, `EXECUTING`, `SUCCESSFUL` or `FAILED`.

Querying for operations

To search for an operation or a collection of operations, send the `com.apama.cumulocity.FindOperation` event to Cumulocity IoT, with a unique `reqId` to the `com.apama.cumulocity.FindOperation.SEND_CHANNEL` channel.

The transport will then respond with zero or more `com.apama.cumulocity.FindOperationResponse` events and then one `com.apama.cumulocity.FindOperationResponseAck` event on the `com.apama.cumulocity.FindOperationResponse.SUBSCRIBE_CHANNEL` channel.

Example:

```
integer reqId := com.apama.cumulocity.Util.generateReqId();

com.apama.cumulocity.FindOperation request :=
    new com.apama.cumulocity.FindOperation;
request.reqId := reqId;

// Filter based on operation status
request.params.add("status", "PENDING");

// Subscribe to com.apama.cumulocity.FindOperationResponse.SUBSCRIBE_CHANNEL
// to listen for responses
monitor.subscribe(com.apama.cumulocity.FindOperationResponse.SUBSCRIBE_CHANNEL);

// Listen for responses based on reqId
on all com.apama.cumulocity.FindOperationResponse(reqId=reqId) as response
// Avoid listener leaks by terminating the listener on completion of the request
and not com.apama.cumulocity.FindOperationResponseAck(reqId=reqId)
{
    log "Received Operation " + response.toString() at INFO;
}

// Listen for com.apama.cumulocity.FindOperationResponseAck,
// this indicates that all responses have been received
on com.apama.cumulocity.FindOperationResponseAck(reqId=reqId)
as requestCompleted
{
    log "Received all Operation(s) for request "
        + requestCompleted.reqId.toString() at INFO;

    // Request is completed and we can unsubscribe from this channel
    monitor.unsubscribe(com.apama.cumulocity.FindOperationResponse.SUBSCRIBE_CHANNEL);
}

// Send request to find available operations
send request to com.apama.cumulocity.FindOperation.SEND_CHANNEL;
```

Supported query parameters

You can provide the following query parameters with the request:

Parameter	Description
id	Search for an operation based on <code>operationId</code> . When searching for an operation based on <code>id</code> , all the query parameters listed below are ignored.
source	Search for operations based on the device identifier or asset identifier of the source.
status	Search for operations based on the status. The status can be any of <code>SUCCESSFUL</code> , <code>FAILED</code> , <code>EXECUTING</code> or <code>PENDING</code> .
agent	Search for operations based on the agent identifier.
fragmentType	Search for operations based on the fragment type.
pageSize	Indicates how many entries of the collection are to be retrieved from Cumulocity IoT. This is a batching parameter for getting multiple responses from Cumulocity IoT. A larger <code>pageSize</code> does fewer requests to Cumulocity IoT to retrieve all the operations, but each request is larger. By default, 1000 operations are in each page and there is an upper limit of 2000.
currentPage	Retrieve a specific page of results for the given <code>pageSize</code> . If <code>currentPage</code> is set, then only a single page is requested. If <code>currentPage</code> is not set (default), all the pages are requested.
fromDate	Search for operations from a start date. The date and time is provided as seconds since the epoch.
toDate	Search for operations within a time range. This is to be used in combination with <code>fromDate</code> . The date and time is provided as seconds since the epoch.
revert	Boolean parameter. If a range query is used (that is, the query includes at least one of the <code>fromDate</code> or <code>toDate</code> parameters), you can reverse the order in which the matching operations are returned by adding the query parameter <code>revert=true</code> . This returns the oldest operations first. By default, Cumulocity IoT returns the latest operation first.

Query result paging

Cumulocity IoT queries support paging of data for requesting a specific range of the responses received. For more information, see [“Paging Cumulocity IoT queries” on page 197](#).

Receiving update notifications

The Cumulocity IoT transport can receive update notifications on new measurements, events, alarms, managed objects and operations that are processed by the Cumulocity IoT platform. By default, all of these updates are sent. However, using the YAML configuration file, you can configure whether you want to subscribe to them; see [“Configuring the Cumulocity IoT transport” on page 165](#).

Notifications about measurements are only received by the Cumulocity IoT transport if the processing mode in Cumulocity IoT is PERSISTENT or TRANSIENT (and not QUIESCENT or CEP).

When a notification about a managed object, operation, alarm or event is sent, the params dictionary member will contain a property which signals whether the notification is a new object or an update to an existing object. The property name is in the constant PARAM_NOTIFICATION and has the value corresponding to the value of the constants NOTIFICATION_CREATED or NOTIFICATION_UPDATED. The recommended way to distinguish between create and update events is to use the isCreate() and isUpdate() actions which are available on these events, as shown in the example below.

Measurements are not modifiable in Cumulocity IoT, so all measurement notifications are newly-created objects.

Note:

This subscription makes use of the long-polling real-time notifications feature of Cumulocity IoT. Note that this is not recommended for high-throughput use cases. See also the information on real-time notifications in the *Cumulocity IoT - API Specifications* at <https://cumulocity.com/api/>.

Example:

```
using com.apama.cumulocity.ManagedObject;
using com.apama.cumulocity.ManagedObjectDeleted;
using com.apama.cumulocity.Measurement;
using com.apama.cumulocity.MeasurementDeleted;
using com.apama.cumulocity.Event;
using com.apama.cumulocity.EventDeleted;
using com.apama.cumulocity.Alarm;
using com.apama.cumulocity.Operation;
monitor NotificationListener {
    action onload {
        // Subscribe for notification for managed objects
        monitor.subscribe(ManagedObject.SUBSCRIBE_CHANNEL);
        // Subscribe for notification for measurements
        monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
        // Subscribe for notification for events
        monitor.subscribe(Event.SUBSCRIBE_CHANNEL);
        // Subscribe for notification for alarms
        monitor.subscribe(Alarm.SUBSCRIBE_CHANNEL);
        // Subscribe for notification for operations
        monitor.subscribe(Operation.SUBSCRIBE_CHANNEL);
        // Listen for notifications for managed objects
        on all ManagedObject() as managedObject {
            if managedObject.isCreate() {
                log "ManagedObject created" at INFO;
            }
            else if managedObject.isUpdate() {
                log "ManagedObject updated" at INFO;
            }
        }
        // Listen for notifications on deleted managed objects
        on all ManagedObjectDeleted() as managedObjectDeleted {
            log "ManagedObject deleted" at INFO;
        }
        // Listen for notifications for measurements
        on all Measurement() as measurement {
            // Measurements can only be created
        }
    }
}
```

```

    log "Measurement created" at INFO;
  }
  // Listen for notifications on deleted measurements
  on all MeasurementDeleted() as measurementDeleted {
    log "Measurement deleted" at INFO;
  }
  // Listen for notifications for events
  on all Event() as evt {
    if evt.isCreate() {
      log "Event created" at INFO;
    }
    else if evt.isUpdate() {
      log "Event updated" at INFO;
    }
  }
  // Listen for notifications on deleted events
  on all EventDeleted() as eventDeleted {
    log "Event deleted" at INFO;
  }
  // Listen for notifications for alarms
  on all Alarm() as alarm {
    if alarm.isCreate() {
      log "Alarm created" at INFO;
    }
    else if alarm.isUpdate() {
      log "Alarm updated" at INFO;
    }
  }
  // Listen for notifications for operations
  on all Operation() as operation {
    if operation.isCreate() {
      log "Operation created" at INFO;
    }
    else if operation.isUpdate() {
      log "Operation updated" at INFO;
    }
  }
}
}
}

```

Paging Cumulocity IoT queries

Queries support paging when requesting multiple responses from Cumulocity IoT. The number of objects requested by queries are controlled using the following query parameters: `pageSize` and `currentPage`.

- `pageSize` represents a batching parameter for getting multiple responses from Cumulocity IoT. A larger `pageSize` does fewer requests to Cumulocity IoT to retrieve all the objects, but each request is larger. By default, `pageSize` is set to 1000. There is an upper limit of 2000.
- `currentPage` can be set to retrieve a specific page of results for a given `pageSize`. If you set `currentPage`, then only a single page is requested. If `currentPage` is not set (default), all the pages are requested.

Note:

It is not recommended to set a small `pageSize` unless you are requesting a single page. To do this, you *must* set `currentPage`. Set `currentPage` to 1 to retrieve the first `pageSize` results. A warning is logged if `pageSize` is below 50 and `currentPage` is not set.

For more details on query result paging and the above query parameters, see the information on the REST implementation in the *Cumulocity IoT - API Specifications* at <https://cumulocity.com/api/>.

Examples

The following example shows a `FindEvent` query where the first 50 events are requested:

```
// Example 1: A FindEvent query where the first 50 responses are requested.
com.apama.cumulocity.FindEvent request := new com.apama.cumulocity.FindEvent;
// ... adding other query params ...
request.params.add("pageSize", "50");
request.params.add("currentPage", "1");
send request to com.apama.cumulocity.FindEvent.CHANNEL;
```

The next two examples demonstrate how to request a range of events where we are not just interested in the first page of results.

In the second example, `pageSize` is also set to 50, but `currentPage` is set to 3, thus requesting the 101st to 150th events:

```
// Example 2: A FindEvent query where the 101st-150th responses are requested
com.apama.cumulocity.FindEvent request := new com.apama.cumulocity.FindEvent;
//... adding other query params ...
request.params.add("pageSize", "50");
request.params.add("currentPage", "3");
send request to com.apama.cumulocity.FindEvent.CHANNEL;
```

As a general rule, if `currentPage` is greater than 1 and the number of objects you require is *not* a factor of the start of the range (or if the number of responses required is greater than the upper limit of `pageSize`), multiple requests are needed to retrieve the objects of interest. As the second example above retrieves 50 events starting after the 100th response (and as 50 is a factor of 100), only 1 request is required.

The third example illustrates a situation where multiple queries are required. 40 events are to be retrieved, starting after the 60th response. As 40 is not a factor of 60, you should set `pageSize` to 20 (the largest common factor of 60 and 40) and send two requests: one where `currentPage` is set to 4 (this retrieves the 61st-80th events), and another where `currentPage` is set to 5 (this retrieves the 81st-100th events).

```
// Example 3: Two FindEvent queries retrieving the 61st-100th events
// First request retrieves 61st-80th events
com.apama.cumulocity.FindEvent request1 := new com.apama.cumulocity.FindEvent;
// ... adding other query params ...
request1.params.add("pageSize", "20");
request1.params.add("currentPage", "4");
send request1 to com.apama.cumulocity.FindEvent.CHANNEL;
// Second request retrieves 81st-100th events
com.apama.cumulocity.FindEvent request2 := new com.apama.cumulocity.FindEvent;
// ... adding other query params ...
request2.params.add("pageSize", "20");
request2.params.add("currentPage", "5");
```

```
send request2 to com.apama.cumulocity.FindEvent.CHANNEL;
```

Invoking other parts of the Cumulocity IoT REST API

The Cumulocity IoT REST API covers some extra functionality which is not covered with the individual event types. To invoke any other part of the REST API, a generic request/response API is provided which you can use to invoke any part of the Cumulocity IoT API.

To create the `GenericRequest`, always use new `GenericRequest` and then individually set whichever fields are needed by name. You must always set the `reqId` (which is used to tie requests and responses together) to a unique identifier generated by the `com.apama.cumulocity.Util.generateReqId()` action. You will also need to set the HTTP method (also known as the verb) and path. For some APIs, you will also need the `queryParams` (which populates the query string), `body` (typically a sequence or dictionary that will be converted to JSON by the plug-in) and/or additional HTTP headers. The `GenericRequest` event should be sent to the channel specified by the `GenericRequest.SEND_CHANNEL` constant.

To receive responses, you must subscribe to the channel given in the `GenericResponse.SUBSCRIBE_CHANNEL` constant. The response events will contain the `reqId` identifier specified in the request, as well as a body in a `dictionary<any, any>` where the `AnyExtractor` can be used to extract the expected content. This dictionary contains a structure which is equivalent to the JSON payload returned by Cumulocity IoT. For the cases where no body is expected in the response (for example, for a DELETE request), only a `GenericResponseComplete` event will be received for the request identifier.

When using an API which returns a collection, the results are automatically split into multiple `GenericResponse` events, followed by a `GenericResponseComplete`, all with the `reqId` identifier provided in the request.

Here is a simple example of using the API:

```
GenericRequest request := new GenericRequest;
request.reqId := com.apama.cumulocity.Util.generateReqId();
request.method := "GET";
request.isPaging := true;
request.path := "/measurement/measurements";
monitor.subscribe(GenericResponse.SUBSCRIBE_CHANNEL);
on all GenericResponse(reqId=request.reqId) as response
  and not GenericResponseComplete(reqId=request.reqId)
{
  AnyExtractor dict := AnyExtractor(response.getBody());
  AnyExtractor source := AnyExtractor(dict.getDictionary("source"));

  try{
    AnyExtractor speed :=
      AnyExtractor(dict.getDictionary("c8y_SpeedMeasurement")["speed"]);
    log "Found measurement of type: c8y_SpeedMeasurement with id : " +
      dict.getString("id") + " and source id : " + source.getString("id") +
      " and speed " + speed.getFloat("value").toString() +
      " " + speed.getString("unit")
      at INFO;
  }
  catch(Exception e){
    log "Failed to parse unexpected measurement : " +
```

```
        dict.toString() at WARN;
    }
}
on GenericResponseComplete(reqId=request.reqId)
{
    monitor.unsubscribe(GenericResponse.SUBSCRIBE_CHANNEL);
}
send request to GenericRequest.SEND_CHANNEL;
```

Invoking microservices

The Cumulocity IoT transport has a `CumulocityRequestInterface` event which allows you to invoke other microservices within Cumulocity IoT from an EPL application. This can be used from an Apama instance outside of Cumulocity IoT and within Cumulocity IoT, either as an EPL application or a custom microservice.

Before you can create an HTTP request, you need to call a static `connectToCumulocity()` action in order to connect (as shown in the later example). The following is the format of the action on the helper class that you call to create a request:

```
action createRequest(string method, string path, any payload) returns Request
```

Pass the following:

- The specific type of HTTP request that is to be created, such as GET or PUT.
- A specific path that you want to append to your request. For example, the path for a microservice that is running on your desired tenant:
`/service/myMicroService/path/under/microservice`.
- The payload will be encoded as JSON. For example, a dictionary will be converted to a JSON object.

This action will return an instance of a `Request` from the generic HTTP API (see also [“Using predefined generic event definitions to invoke HTTP services with JSON and string payloads” on page 154](#)) with configuration set up on the request. You can later call `execute` on this request, passing in a handler to deal with any response.

The following example shows how to make use of this class, that is, how to make an HTTP request in order to retrieve information from a running microservice:

```
monitor CumulocityTestMonitor {
    action onload() {
        try{
            CumulocityRequestInterface cInterface :=
                CumulocityRequestInterface.connectToCumulocity();
            Request req := cInterface.createRequest("GET",
                "/service/myMicroService/path/under/microservice",
                {"request":"data"});
            req.execute(getHandler);
        }
        catch (com.apama.exceptions.Exception e) {
            log "Error thrown trying to create a Cumulocity Request " +
                e.toString() at ERROR;
        }
    }
}
```



```

    }

    action getHandler(Response resp) {
        AnyExtractor d := resp.payload;
        log "Response output: " + d.toString() at INFO;
        log "Test Done";
    }
}

```

The `CumulocityRequestInterface` will automatically detect if it is running inside or outside of Cumulocity IoT and will automatically connect. If running remotely, it will rely on properties being set, which will be the connection details provided for the transport. You can do this by creating a `.properties` file in your project and specifying it with the `--config` option when starting a correlator (see "Starting the correlator" in *Deploying and Managing Apama Applications*).

Optionally, you can set the following if you are connecting to a Cumulocity IoT server with a self-signed or private certificate. Set this to the path to the certificate authority file by which the server's certificate was signed:

```
CUMULOCITY_TLS_CERT_AUTH_FILE
```

The helper class is included in the **Utilities for Cumulocity** bundle in Software AG Designer. You can also find it in the `monitor/cumulocity` directory of your Apama installation.

Monitoring status for Cumulocity IoT

The Cumulocity IoT component provides status values via the user status mechanism. It provides the following metrics (where `prefix` is `user-CumulocityIoTGenericChain.cumulocityCodec`):

Key	Description
<code>prefix.maxLatencyInLastHourMillis</code>	Maximum request latency observed during the last hour, in milliseconds.
<code>prefix.maxLatencyInLastHourDetails</code>	<p>Details of the maximum latency request. Consists of a tab-separated string containing the following:</p> <ul style="list-style-type: none"> ■ ISO format timestamp in UTC, ■ method, path and parameters truncated to 100 characters (in URL format), and ■ an optional count of the number of objects if this is a batched request (only <code>com.apama.cumulocity.Measurement</code> requests can be batched).
<code>prefix.mostRecentSlowRequestDetails</code>	Details of the most recent slow request. A request is slow if the request-response multiplied by the number of pages (or 1)

Key	Description
	<p>is above CUMULOCITY_LATENCY_SLOW_THRESHOLD_SECS (see “Configuring the Cumulocity IoT transport” on page 165). Consists of a tab-separated string containing the following:</p> <ul style="list-style-type: none">■ ISO format timestamp in UTC,■ method, path and parameters truncated to 100 characters (in URL format), and■ an optional count of the number of objects if this is a batched request (only <code>com.apama.cumulocity.Measurement</code> requests can be batched).
<code>prefix.requestLatencyEWMAShortMillis</code>	A quickly-evolving exponentially-weighted moving average of request latencies, in milliseconds. Uses 0.5 as the weight to calculate this. This puts more importance on recent latencies than <code>requestLatencyEWMALongMillis</code> .
<code>prefix.requestLatencyEWMALongMillis</code>	A longer-term exponentially-weighted moving average of request latencies, in milliseconds. Uses 0.1 as the weight to calculate this.

For more information about monitor status information published by the correlator, see "Managing and Monitoring over REST" and "Watching correlator runtime status", both in *Deploying and Managing Apama Applications*.

When using Software AG Command Central to manage your correlator, see also "Monitoring the KPIs for EPL applications and connectivity plug-ins" in *Deploying and Managing Apama Applications*.

Finding tenant options

In order to find the available tenant options on a tenant, you can send the event `com.apama.cumulocity.FindTenantOptions` to `FindTenantOptions.SEND_CHANNEL`. This results in events of type `com.apama.cumulocity.FindTenantOptionsResponse` being returned on `com.apama.cumulocity.FindTenantOptionsResponse.SUBSCRIBE_CHANNEL`.

The returned event includes a sequence of `com.apama.cumulocity.TenantOption`, which individually include the key/value combinations that represent the available tenant option.

In order to filter the tenant options returned, you can specify values in the category and key fields of the find request. If only the key is specified, then Cumulocity IoT returns all the available tenant options and the correlator filters them by the key.

Getting user details

You can get the details of the current user by sending the event

`com.apama.cumulocity.GetCurrentUser` to `com.apama.cumulocity.GetCurrentUser.SEND_CHANNEL`. This results in events of type `com.apama.cumulocity.GetCurrentUserResponse` being returned on `com.apama.cumulocity.GetCurrentUserResponse.SUBSCRIBE_CHANNEL`.

The `com.apama.cumulocity.GetCurrentUserResponse` returned contains a `com.apama.cumulocity.CurrentUser` event, which in turn contains the id of the user, the `userName`, a sequence of `effectiveRoles` for the user and a dictionary of `userOptions`.

By default, this is the user which the Apama application is running as. This is either the user configured in the Cumulocity IoT connection if it is not running within Cumulocity IoT or the service user of the microservice if it is running in Cumulocity IoT.

The ability to request details of permissions for another user can be done by overriding the authorization or cookies headers in the `com.apama.cumulocity.GetCurrentUser` event. This would normally be used if you are taking the authentication details from a request to your application and using them to determine the roles that user has.

Example - checking a user based on information in a received request:

```
/** Event containing extracted information retrieved from a
 * http request where we want to check the validity of the user */
event ActionRequest {
    string authorization;
    string actionToTake;
    string requestId;
    string channel;
}
/** Response for the HTTP request */
event ActionResponse {
    string requestId;
    string actionResult;
}
/** Response if authorization failed */
event ActionNotAllowed {
    string requestId;
}
...
monitor.subscribe(GetCurrentUserResponse.SUBSCRIBE_CHANNEL);

// Listen for incoming HTTP requests
on all ActionRequest() as ar {
    integer reqId := com.apama.cumulocity.Util.generateReqId();
    // Send a request to check the user from the incoming request
    GetCurrentUser checkUser := new GetCurrentUser;
    checkUser.reqId := reqId;
    checkUser.authorization := ar.authorization;
    send checkUser to GetCurrentUser.SEND_CHANNEL;
```

```
// if authentication passed, check authorization
on GetCurrentUserResponse(reqId=reqId) as res and not
  GetCurrentUserResponseFailed(reqId=reqId) {
  if checkHasRoles("ActionAllowed", res.user.effectiveRoles) {
    send ActionResponse(ar.requestId, performAction(ar.actionToTake))
    to ar.channel;
  } else {
    send ActionNotAllowed(ar.requestId) to ar.channel;
  }
}

// if authentication failed, return an error
on GetCurrentUserResponseFailed(reqId=reqId) as err and not
  GetCurrentUserResponse(reqId=reqId) {
  send ActionNotAllowed(ar.requestId) to ar.channel;
}

}

action performAction(string actiontoTake) returns string{
  // do some action
  return "";
}

}

action checkHasRoles(string role,sequence<Role> effectiveRoles) returns boolean {
  Role r;
  for r in effectiveRoles {
    if r.id = role{
      return true;
    }
  }
  return false;
}
}
```

You can override the current user in one of the following ways:

- By setting the authorization header of the other user. This would be used for basic authentication and returns details of that other user.

If this is invalid, then `GetCurrentUserResponseFailed` is returned.

- By setting both `authCookie` and `xsrftoken` to valid values for another user. This returns details of that other user.

If either `authCookie` or `xsrftoken` are incorrect or not set, then `GetCurrentUserResponseFailed` is returned.

- Not setting all of authorization, `authCookie` or `xsrftoken` returns details of the current user.

Sample EPL

The sample EPL below describes how to subscribe and receive device measurements, device events and device information.

```
package com.apama.sample;

using com.apama.cumulocity.ManagedObject;
using com.apama.cumulocity.Measurement;
```

```

using com.apama.cumulocity.MeasurementValue;
using com.apama.cumulocity.Alarm;
using com.apama.cumulocity.Event;
using com.apama.cumulocity.Operation;

using com.apama.cumulocity.FindManagedObject;
using com.apama.cumulocity.FindManagedObjectResponse;
using com.apama.cumulocity.FindManagedObjectResponseAck;

monitor CumulocityApplication {

    action onload() {

        fetchManagedObjects();

        listenForMeasurements();

        listenForAlarms();

        listenForEvents();

        listenForOperations();
    }

    action fetchManagedObjects() {

        // Subscribe to receive all the devices from Cumulocity IoT
        monitor.subscribe(ManagedObject.SUBSCRIBE_CHANNEL);

        // Consume all the devices from Cumulocity IoT
        on all ManagedObject() as mo {
            log mo.toString() at INFO;

            // Update a managed object
            /*
            mo.params.add("CustomMetadata", {"metadata": "Adding custom data"});
            send mo to ManagedObject.SEND_CHANNEL;
            */
        }

        monitor.subscribe(FindManagedObjectResponse.SUBSCRIBE_CHANNEL);

        // Fetch a list of all available devices
        integer reqId := com.apama.cumulocity.Util.generateReqId();
        on all FindManagedObjectResponse(reqId=reqId) as response
        and not FindManagedObjectResponseAck(reqId=reqId) {
            log "Received managedObject " + response.managedObject.toString() at INFO;
        }

        on FindManagedObjectResponseAck(reqId=reqId) {
            log "Find Managed Objects request completed" at INFO;
            monitor.unsubscribe(FindManagedObjectResponse.SUBSCRIBE_CHANNEL);
        }

        // Retrieve list of all available devices
        send FindManagedObject(reqId, "", {"fragmentType": "c8y_IsDevice"})
            to FindManagedObject.SEND_CHANNEL;
    }

    action listenForMeasurements() {

```

```
// Subscribe to receive all the measurements published from
// Cumulocity IoT
monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);

// Consume all the measurements from Cumulocity IoT
on all Measurement() as m {
    log m.toString() at INFO;
}

// Create a new measurement
/*
Measurement m := new Measurement;
m.source := "<MANAGED_OBJECT_ID>";
m.time := currentTime;
m.type := "TemperatureMeasurement";
MeasurementValue mv := new MeasurementValue;
mv.value := 100.0;
dictionary<string, MeasurementValue> fragment :=
    new dictionary<string, MeasurementValue>;
fragment.add("temperature", mv);
m.measurements.add("TemperatureMeasurement", fragment);
send m to Measurement.SEND_CHANNEL;
*/
}

action listenForEvents() {

    // Subscribe to receive all the events published from
    // Cumulocity IoT
    monitor.subscribe(Event.SUBSCRIBE_CHANNEL);

    // Consume all the events from Cumulocity IoT
    on all Event() as e {
        log e.toString() at INFO;
        // Example for updating an event
        /*
        // update text
        e.text := "This is an updated text";
        send e to Event.SEND_CHANNEL;
        */
    }

    // Create a new event
    /*
    Event evt := new Event;
    evt.source := "<MANAGED_OBJECT_ID>";
    evt.type := "TestEvent";
    evt.time := currentTime;
    evt.text := "This is a sample event";
    send evt to Event.SEND_CHANNEL;
    */
}

action listenForAlarms() {

    // Subscribe to receive all the alarms published from
    // Cumulocity IoT
    monitor.subscribe(Alarm.SUBSCRIBE_CHANNEL);
```

```

// Consume all the alarms from Cumulocity IoT
on all Alarm() as alarm {
    log alarm.toString() at INFO;

    // Example for updating an alarm
    /*
    // set alarm severity to MAJOR
    alarm.severity := "MAJOR";
    send alarm to Alarm.SEND_CHANNEL;
    */
}

// Create a new alarm
/*
Alarm alarm := new Alarm;
alarm.source := "<MANAGED_OBJECT_ID>";
alarm.type := "TestAlarm";
alarm.severity := "MINOR";
alarm.status := "ACTIVE";
alarm.time := currentTime;
alarm.text := "This is a sample alarm";
send alarm to Alarm.SEND_CHANNEL;
*/
}

action listenForOperations() {
    // Subscribe to receive all the operations published from
    // Cumulocity IoT

    // Note: When using the Cumulocity transport, ensure that the
    // subscribeToOperations transport property is set to true
    monitor.subscribe(Operation.SUBSCRIBE_CHANNEL);

    on all Operation() as o {
        log o.toString() at INFO;

        // Update an operation
        /*
        o.status := "EXECUTING";
        send o to Operation.SEND_CHANNEL;
        */
    }

    // Create an operation
    /*
    Operation operation := new Operation;
    operation.source := "<MANAGED_OBJECT_ID>";
    operation.status := "PENDING";
    operation.params.add("c8y_Message", {"text": "Device Operation"});
    send operation to Operation.SEND_CHANNEL;
    */
}
}

```


11 Codec Connectivity Plug-ins

■ The String codec connectivity plug-in	210
■ The Base64 codec connectivity plug-in	211
■ The JSON codec connectivity plug-in	212
■ The Classifier codec connectivity plug-in	216
■ The Mapper codec connectivity plug-in	217
■ The Batch Accumulator codec connectivity plug-in	221
■ The Message List codec connectivity plug-in	222
■ The Unit Test Harness codec connectivity plug-in	225
■ The Diagnostic codec connectivity plug-in	228

The String codec connectivity plug-in

The String codec can be used for transports that are dealing with binary payloads or map payloads with binary data in one or more specific fields of the map which reflect the structure of the event. It provides the ability to perform bidirectional translations between binary format and string format. The codec is able to operate on arbitrary fields of the message, or the entire payload.

- The field of a message going from the host to the transport should be of `java.lang.String` (Java) or `const char*` (C++) type. The String codec translates the fields of such a message to the `byte[]` (Java) or `buffer_t` (C++) type.
- The field of a message going from the transport to the host should be of `byte[]` or `buffer_t` type. The String codec translates the fields of such a message to the `java.lang.String` or `const char*` type.

By default, the String codec does UTF-8 encoding and decoding of a string:

- When converting to a `buffer_t` or `byte[]`, the end result is UTF-8 encoded.
- When converting to a `java.lang.String` or `const char*`, the String codec assumes that the source (`buffer_t` or `byte[]`) is UTF-8 encoded.

To reference the String codec, an entry such as the following is required in the `connectivityPlugins` section of the configuration file (see also [“Configuration file for connectivity plug-ins” on page 26](#)):

```
stringCodec:
  libraryName: connectivity-string-codec
  class: StringCodec
```

You then need to add the String codec into your connectivity chain with the configuration for that instance. An example configuration may look as follows:

```
startChains:
  testChain:
    - apama.eventString
    - stringCodec:
        nullTerminated: false
        eventTypes:
          - com.apamax.test.MyEventType
        encoding: Latin-1
        fields:
          - metadata.foo
          - payload.bar.baz
          - payload.zot
    - myBinaryTransport
```

The following configuration options are available for the String codec:

Configuration option	Description
<code>nullTerminated</code>	It is only permitted to set this option to <code>true</code> when the encoding is UTF-8. It only affects the conversion to bytes for messages sent from the host towards the transport. When

Configuration option	Description
	<p>messages are converted from bytes (on the transport side) to strings (on the host side), a terminating null character is always permitted but never required, regardless of the value of this configuration option.</p> <p>If set to <code>true</code>, a null-terminator character is added to the end of the buffer when sending messages towards the transport.</p> <p>If set to <code>false</code>, messages sent towards the transport do not include a null-terminator.</p> <p>Default: <code>false</code>.</p>
<code>eventTypes</code>	Specifies which event types this codec will handle. Messages with a type that is not listed or where <code>sag.type</code> is not set will be ignored by this codec. If omitted, the codec attempts to encode/decode the payload of all messages.
<code>fields</code>	The list of metadata or payload fields that are to be converted by this codec. Listed field that are not present in the event will be ignored by this codec. It is recommended that you prefix each field with either <code>payload</code> or <code>metadata</code> , for example: <code>payload.myfield</code> or <code>metadata.myfield</code> . If omitted, the codec attempts to encode the entire payload.
<code>encoding</code>	<p>The character set to be used for encoding/decoding.</p> <p>Default: <code>UTF-8</code>.</p> <p>If <code>charset</code> is set in the metadata of a message (in either direction), this will quietly override the <code>encoding</code> option. For example, if the <code>encoding</code> option is set to <code>Latin-1</code> and if the message carries <code>charset</code> in the metadata (for example, <code>metadata.charset=CP1252</code>), then the payload/metadata field or the entire payload of the message is converted using the character-set value of <code>metadata.charset</code>.</p>

The Base64 codec connectivity plug-in

The Base64 codec performs bidirectional translation between binary payloads and a string that is the Base64-encoding of that payload. The codec is able to operate on arbitrary fields of the message, or the entire payload.

- The field of a message going from the host to the transport should be of `byte[]` (Java) or `buffer_t` (C++) type. The Base64 codec encodes the fields of such a message to their Base64-encoding, which will be a `java.lang.String` (Java) or `const char*` (C++) type.

- The field of a message going from the transport to the host should be a `java.lang.String` (Java) or `const char*` (C++) type in Base64 format. The Base64 codec decodes the fields of such a message to the `byte[]` (Java) or `buffer_t` (C++) type.

The Base64 codec follows the MIME (Multipurpose Internet Mail Extensions) specification, which lists Base64 as one of two binary-to-text encoding schemes.

To reference the Base64 codec, an entry such as the following is required in the `connectivityPlugins` section of the configuration file (see also [“Configuration file for connectivity plug-ins” on page 26](#)):

```
base64Codec:
  libraryName: connectivity-base64-codec
  class: Base64Codec
```

You then need to add the Base64 codec into your connectivity chain with the configuration for that instance. An example configuration may look as follows:

```
startChains:
  testChain:
    - apama.eventMap
    - base64Codec:
        eventTypes:
          - com.apamax.MyEvent
        fields:
          - metadata.baz
          - payload.foo.asdf
    - myTransport
```

The following configuration options are available for the Base64 codec:

Configuration option	Description
<code>eventTypes</code>	Specifies which event types this codec will handle. Messages with a type that is not listed or where <code>sag.type</code> is not set will be ignored by this codec. If omitted, the codec attempts to encode/decode the payload of all messages.
<code>fields</code>	The list of metadata or payload fields that are to be converted by this codec. Listed field that are not present in the event will be ignored by this codec. It is recommended that you prefix each field with either <code>payload</code> or <code>metadata</code> , for example: <code>payload.myfield</code> or <code>metadata.myfield</code> . If omitted, the codec attempts to encode the entire payload.

The JSON codec connectivity plug-in

The JSON codec can be used if you have a transport that is dealing with messages in the JSON format and you want your EPL application to be able to interact with it by sending and listening for events. It does not matter whether the transport

- takes JSON-formatted data from an external system and sends it towards the host, or

- wants to be given JSON-formatted data from the direction of the host, and then sends it to an external system, or
- even does both of the above.

The JSON codec does bidirectional translation between JSON documents in the payload of a message (transport side) and an object in the payload of a message (host side). If the JSON codec is adjacent to the eventMap plug-in, then the JSON document on the transport side should be an object with fields for the event members. The Mapper codec can help move fields to match the event structure, map parts of the metadata into the payload, and support JSON values other than objects.

For example, a JSON document like

```
{"a":2,"b":["x","y","z"]}
```

on the transport side corresponds to a `java.util.Map` (Java) or `map_t` (C++) on the host side. This `java.util.Map` or `map_t` maps a string to an integer for one map entry and a string to a list of strings for the other entry. When the `apama.eventMap` host plug-in sees an object like this, it will be able to map it to/from an EPL event type such as the following:

```
event E {
    integer a;
    sequence<string> b;
}
```

The above assumes that either `metadata.sag.type` is set to `E` (see also [“Metadata values” on page 58](#)) or the `apama.eventMap` host plug-in has been configured with `defaultEventType: E`. Remember that this is completely bidirectional.

Taking a closer look at the inbound response, a typical chain would start with the HTTP client transport (see also [“The HTTP Client Transport Connectivity Plug-in” on page 131](#)). This transport returns a byte array containing the response body which is normally transformed into a UTF-8 string using the String codec (see also [“The String codec connectivity plug-in” on page 210](#)). This UTF-8 string, held in the payload, is then passed to the JSON codec and transformed into a data structure that the JSON represents. At this point, the `metadata.http` map holds the headers and other elements of the HTTP response, and is used to set the `metadata.sag.type` and add to the response payload. After the mapping rules have been applied, the payload is passed to the `apama.eventMap` host plug-in and converted to the event that is defined in the `metadata.sag.type`.

The `apama.eventMap` host plug-in can convert a map into an event. This map will either be created from the JSON if it contains an object, or it will need to be created by mapping in the chain. In the example below, we get a JSON object in the response and it maps to the EPL event shown:

```
// Example response payload contains a JSON object.

response = {"hostname":"host","name":"correlator","value":"expected value"}

// Maps to HTTPResponse event putting "value" into extra fields dictionary
// and adding the id from metadata.

HTTPResponse(id,"host","correlator",{value:"expected value"})
```

The top-level value in the JSON will normally be an object, which can be mapped directly to an Apama event. However, it is also possible to use other JSON types such as string, array, boolean or number. In those cases, you will need to map the decoded payload before it can be received by Apama. For example:

```
// Other valid JSON responses that require mapping
// payload = "valid"
// payload = 3.14
// payload = []

HTTPResponse:
  towardsHost:
    mapFrom:
      - payload.contents: payload

// The event can define a field or use @ExtraFieldsDict
// and provide a map for the contents.

event MyEvent{ any contents; }
```

It is not supported to have a JSON null as the top-level value in the payload. Empty payloads are treated as special control messages in connectivity chains and the JSON codec will ignore such messages. Nulls can occur elsewhere in the JSON structure and will be mapped to the empty any type in EPL.

The content of the event that forms the JSON request will be transformed similarly, so care needs to be taken over how the content will end up in the request JSON:

```
event Example{}

// Creates an empty JSON object in the payload.

{}

// Any and Optional need careful handling if the values
// are not set. They default to null.

event Example{ optional<string> test1; any test2 }

// Maps to

{"test1":null,"test2":null}
```

Since the JSON standard does not permit floating point “special” values such as Infinity and NaN (see “Support for IEEE 754 special values” in *Developing Apama Applications*), float fields containing such values are represented as strings when generating JSON using this codec. For example, this is represented as [1.2, 3.4, “Infinity”, “NaN”] in the generated JSON. If you need different handling of special values (for example, representing them as a JSON null), add a custom codec to your connectivity chain before the JSON codec to make any necessary conversions. In the other direction, when parsing JSON, the codec gives an error if requested to parse one of these special values that is not wrapped as a string.

There are two identically behaving versions of the JSON codec, one implemented using C++ and the other implemented using Java. The C++ version of the codec has the same behavior as the Java version, but it usually gives a better performance. In particular, changing between Java and C++

is expensive, so you should match the implementation language of the codec to that of the adjacent codecs. See [“Deploying plug-in libraries” on page 40](#) for more information.

A Java sample is provided in the `samples/connectivity_plugin/java/JSON-Codec` directory of your Apama installation. This provides a more detailed end-to-end example of this codec (along with the source code to this codec), which allows an EPL application to consume and emit JSON as EPL events.

To reference the JSON codec, an entry such as the one below is required in the `connectivityPlugins` section of the configuration file (see also [“Configuration file for connectivity plug-ins” on page 26](#)). You can then just have a JSON codec in a chain in between the host and a transport. No configuration is required for this plug-in.

Configuration file entry for Java:

```
jsonCodec:
  directory: ${APAMA_HOME}/lib/
  classpath:
    - json-codec.jar
  class: com.softwareag.connectivity.plugins.JSONCodec
```

Configuration file entry for C++:

```
jsonCodec:
  libraryName: connectivity-json-codec
  class: JSONCodec
```

The following configuration option is available for the JSON codec:

Configuration option	Description
<code>filterOnContentType</code>	<p>This option can be used in chains which have multiple messages of different types through the chain with a transport or mapper configuration which is setting the <code>metadata.contentType</code> field in the messages. It allows the JSON codec to only process messages which (hostwards) are correctly encoded in JSON or (transportwards) should be encoded in JSON. Other messages need to be handled by another codec in the chain which will handle non-JSON-encoded messages.</p> <p>If set to <code>true</code> and the <code>metadata.contentType</code> is set to anything that does not match the pattern <code>"^application/([^/]*[+])?json(;[^\=;]+\=(\".*\" \"[^\=;]+))\"*\$"</code>, or if it is not set, then the codec will ignore the event in either direction and pass it on unmodified.</p> <p>If set to <code>false</code>, then the codec will attempt to process the message no matter what the value of <code>metadata.contentType</code> is.</p> <p>Default: <code>false</code>.</p>

Note:

Equivalent functionality is available with the JSON EPL plug-in. See "Using the JSON plug-in" in *Developing Apama Applications* for detailed information.

The Classifier codec connectivity plug-in

The Classifier codec can be used to take messages from a transport and assign them message types suitable for the host. Typically this is the type of an EPL event, which corresponds to the metadata field `sag.type` (see also [“Metadata values” on page 58](#)) when you are using the `apama.eventMap` host plug-in. The Classifier codec can inspect the message payload and metadata in order to classify messages to the correct type. If it finds a match, it sets the `sag.type` appropriately, overwriting anything which was there before.

To reference the Classifier codec, an entry such as the following is required in the `connectivityPlugins` section of the configuration file (see also [“Configuration file for connectivity plug-ins” on page 26](#)):

```
classifierCodec:
  libraryName: ClassifierCodec
  class: ClassifierCodec
```

You then need to add `classifierCodec` into your connectivity chains with the configuration for that instance. An example configuration may look as follows:

```
classifierCodec:
  rules:
    - SomeType:
      - payload.someField: someValue
      - payload.anotherField:
    - AnotherType:
      - metadata.metadataField: true
    - ThirdType:
      - regex:payload.field2: /bar/([0-9]{3})([a-zA-Z]{3})/![@#%\^&\*]+
      - payload.field3: something
    - FourthType:
      - regex:payload.field1: Valid PLAIN String
    - FallbackType:
```

The top-level configuration element is just `rules`. It contains a list of one or more types which can be assigned to messages. Each type contains a list of rules to match against the type with the following properties:

- Types are evaluated in order and the first one to match is assigned to a message.
- If the list of rules for a type is empty, then it matches any message. There should be only one such type and it must be last. With `FallbackType` in the above example configuration, it is always guaranteed that a type is set on a message (because `FallbackType` is a last resort).
- Rules within a type are evaluated in order, and comparisons stop on the first failure so that common cases are evaluated first.
- Field names must start with `“metadata.”` or `“payload.”`, or with `“regex:metadata.”` or `“regex:payload.”` if you are using regular expressions.

- Metadata and payload field names which contain a period (.) refer to nested map structures within the metadata or payload. For example, `metadata.http.headers.accept` refers to a map called “http” within the metadata, which contains a map called “headers”, which contains an element called “accept”.
- Empty rules match if the field is present (even if empty) with any value. With `SomeType` in the above example configuration, this rule matches if `anotherField` in the payload contains *any* value. This does not apply for regular expressions where empty rules are not allowed.
- A non-empty rule usually looks for an exact string match, unless the field name begins with “regex:”. In this case, the rule looks for a regular expression match against the entire field value. For example, if `field2` of the payload in the above example configuration is equal to `/bar/123aBc/` or another matching string, and `field3` contains something, then the message can be classified as being of type `ThirdType`.
- Regular expression matches are performed using the ICU library (see the ICU User Guide at <http://userguide.icu-project.org/strings/regex> for detailed information) with the default flags - single line and case sensitive.
- All rules within a type must be true to match that type. For the above example configuration, this means that if `anotherField` in the payload exists, but `someField` does not contain `someValue`, then `SomeType` does not match.
- If no types match, then the `sag.type` metadata field remains unchanged.
- For “payload.” rules to match, the payload must be a `java.util.Map` (Java) or `map_t` (C++) with string keys.
- Messages coming *from* the direction of the host do not interact with the Classifier codec at all. Use the `apama.eventMap` host plug-in, which always sets a `sag.type` for messages going from the host towards the transport.

If you want to encode an *or* rule, then you can list the same type twice with different sets of rules.

The Mapper codec connectivity plug-in

The Mapper codec can be used to take messages from a transport which do not match the schema expected by the host and turn them into messages which are suitable for the host. Typically this means making sure that the fields in the message have the same names as the fields in the corresponding EPL event type if you are using the `apama.eventMap` host plug-in (see also “[Translating EPL events using the apama.eventMap host plug-in](#)” on page 32). This codec can move fields around between the payload and the metadata and set the values of fields which have no value from the transport. It is bidirectional and can also map messages coming from the host into a format suitable for the transport.

The source code for this plug-in is also shipped as a sample in the `samples/connectivity_plugin/cpp/mapper` directory of your Apama installation.

To reference the Mapper codec, an entry such as the following is required in the `connectivityPlugins` section of the configuration file (see also “[Configuration file for connectivity plug-ins](#)” on page 26):

```
mapperCodec:
  libraryName: MapperCodec
  class: MapperCodec
```

You then need to add `mapperCodec` into your connectivity chains with the configuration for that instance. If you are also using the Classifier codec to assign types to incoming messages, then you must have that codec on the transport side of the Mapper codec. An example configuration may look as follows:

```
- mapperCodec:
  allowMissing: true
  SomeType:
    towardsHost:
      mapFrom:
        - metadata.targetField1: metadata.sourceField1
        - payload.myField2: metadata.myField2
        - metadata.targetField3: payload.sourceField3
        # to set the correlator channel on a per-message basis:
        - metadata.sag.channel: payload.mychannel
        # to move all fields from payload to metadata, use:
        # - metadata: payload
      copyFrom:
        - metadata.targetField4: metadata.sourceField4
      forEach:
        - payload.listA:
            mapFrom:
              - targetFieldA: sourceFieldA
            copyFrom:
              - targetFieldB: sourceFieldB
      set:
        - payload.fieldName: An overridden value
      defaultValue:
        - payload.targetFieldX: A default value

    towardsTransport:
      mapFrom:
        - metadata.myField2: payload.myField2
        - payload.sourceField3: metadata.targetField3

  "*":
    towardsHost:
      defaultValue:
        - payload.targetFieldY: A different value
```

An example message input and output for the above mapping a `SomeType` event towards the host, as logged by the Diagnostic codec (with extra spacing to make it clearer), is:

```
[premap] Towards Host: {myField2:Field2,
                        sag.type:SomeType,
                        sourceField4:Field4,
                        sourceField1:Field1} /
                        {listA:[{sourceFieldB:Beta,
                                sourceFieldA:Alpha},
                                {sourceFieldB:Brian,
                                sourceFieldA:Andrew}],
                        fieldName:Gamma,
                        sourceField3:Field3}
```

```
[postmap] Towards Host: {targetField3:Field3,
                          sag.type:SomeType,
                          targetField1:Field1,
                          sourceField4:Field4,
                          targetField4:Field4} /
                          {listA:[{targetFieldA:Alpha,
                                    sourceFieldB:Beta,
                                    targetFieldB:Beta},
                                    {targetFieldA:Andrew,
                                    sourceFieldB:Brian,
                                    targetFieldB:Brian}],
                          fieldName:An overridden value,
                          myField2:Field2,
                          targetFieldX:A default value,
                          targetFieldY:A different value}
```

The configuration of the Mapper codec is nested, with a map of type names, each containing directions, each containing actions, each containing rules. The type name corresponds to the `sag.type` metadata field (see [“Metadata values” on page 58](#)). Instead of the name of a type, the special symbol "*" (which must include the quotes so that it can be parsed in the YAML format correctly) can be used to list rules to apply to all types. Messages are first processed with any rules matching their specific type, then any rules under "*" are applied.

The rules for a type are split into two directions:

- `towardsHost` - Messages going from the transport to the host.
- `towardsTransport` - Messages going from the host to the transport.

If you are writing a bidirectional chain, these rules will usually be the converse of each other.

Within a direction, the following actions can be applied. Each action operates on one or two fields. Each field can be the entire payload, a field within the payload or metadata, or a nested field; see below for details of field names.

- `mapFrom` - Move a value to a new field from the specified field (*target*: *source*).

For the above example configuration, this means that if a message of type `SomeType` is being passed from the transport towards the host, then the field `sourceField1` from the metadata is removed and its value is put into the field `targetField1` of the metadata. The second `mapFrom` rule moves the value of `myField2` from the metadata to the payload, using the same field name. It is always the field on the left-hand side of the rule which is the target of the move operation, regardless of whether messages are moving towards the transport or towards the host. For bidirectional message types, it is quite common to have rules in `towardsTransport` that are the inverse of the `towardsHost` rules, as is the case for `myField2` in this example.

- `copyFrom` - This action is exactly the same as the `mapFrom` action except that the source field is not removed after the copy.
- `forEach` - Iterate through all elements of a given sequence and apply the supplied mapping actions.

For the above example configuration, this means that if a message of type `SomeType` is being passed from the transport towards the host and if the message payload contains a sequence field `listA`, then for each element of `listA` the subrules of `mapFrom` and `copyFrom` are applied.

That is, for every child element of *listA*, the field *sourceFieldA* is mapped to *targetFieldA* (mapFrom) and the value of field *sourceFieldB* is copied to *targetFieldB* (copyFrom).

Note that the rules can only be applied if the child elements of the sequence are another event or a dictionary (with string keys).

- **set** - Set a metadata field, payload field or the top-level payload to a specific value, regardless of whether that field already exists or not. For the above example configuration, this means that if a message of the type *SomeType* is being passed from the transport towards the host, then the field *fieldName* will be set to *An overridden value*.
- **defaultValue** - If a metadata field, payload field or top-level payload is unset or empty/null, then set it to a specified value. For the above example configuration, this means that if a message of any type is being passed from the transport towards the host, then the field *targetFieldY* of its payload is set to *A different value* if - and only if - that field does not already exist in the map. The following applies:
 - A top-level payload can be mapped to a string or map.
 - A payload field or metadata field can be mapped to a string, list or map.

Each of the above actions has a list of rules of the form *target_field: source* (where *source* is either a field name or a string value). Notes:

- The actions are applied in the following order: *copyFrom*, *mapFrom*, *forEach*, *set*, *defaultValue*.
- Rules are applied in order within each action section, so you can move the contents out of a field and then replace it with a different value.
- The left-hand side is the field whose value is being set.

In the case of *forEach*, the left-hand side field corresponds to the sequence to which the subrules are applied.

- Field names must start with “metadata.” or “payload.”, or must be the string “payload” or “metadata” - except for those within a *forEach* action, in which case they only name a field within an element of the sequence.
- Field names which contain a period (.) refer to nested map structures within the metadata or payload. For example, *payload.http.headers.accept* refers to a map called “http” within the payload, which contains a map called “headers”, which contains an element called “accept”.

Special cases: in metadata source expressions, a field name with a period (.) in it is looked up at the top-level and used if it is found, otherwise as a nested name. Using the *sag.* prefix as a target does not create a new map within the metadata.

- A *copyFrom* or *mapFrom* rule where the source field does not exist uses the default value if the *defaultValue* exists or if a subsequent *copyFrom* or *mapFrom* rule exists for the same destination field. If none of these fallback options exist (like a *defaultValue*), then the message is discarded with an error.
- The Mapper codec also accepts an *allowMissing* configuration item at the top level. This affects all rules in the Mapper codec and defaults to *false*. If *allowMissing* is set to *true*, an error is not raised when a *defaultValue* (or a subsequent *copyFrom* or *mapFrom* rule) has not been set

and a source field is missing. `allowMissing` needs to be defined at the same level as the event types.

- If setting a payload field on a payload that is not a map, the payload is first overwritten with an empty map.
- `payload` in the left-hand side or right-hand side of a rule (rather than `payload.fieldname`) refers to the entire payload object. This allows you, for example, to map an entire string payload into a field in a map in the resulting message's payload, or vice-versa.
- Any rules that mention `payload.fieldname` assume that the payload is a `java.util.Map` (Java) or `map_t` (C++) with string keys.

The Batch Accumulator codec connectivity plug-in

Events being written from the correlator to a connectivity transport are automatically batched for performance reasons. Many transports also read batches of incoming messages and send them into the correlator in a batch as well. However, some transports do not perform this batching and deliver messages one at a time. For such transports, performance can be increased by adding the batching before it is parsed by the correlator. This can be done with the Batch Accumulator codec.

For the transports that are provided out of the box, all the message-bus transports such as MQTT and Kafka already provide batching. The HTTP client, however, does not benefit from it because of the request/response nature. If you are using the HTTP server in submission-only mode and want to achieve a high rate of requests with multiple clients, then the Batch Accumulator codec can be useful. It may also be useful with any custom transports you write.

The Batch Accumulator codec automatically tunes batch sizes from one up, depending on the rate of incoming requests, and requires no manual tuning. It does not wait to accumulate a batch for a certain period of time and so does not introduce unnecessary latency. The batching is performed for messages going from the transport to the host. Messages from the host to the transport are passed through verbatim since they are already in batches.

To load the Batch Accumulator codec, an entry such as the following is required in the `connectivityPlugins` section of the configuration file (see also [“Configuration file for connectivity plug-ins” on page 26](#)):

```
batchAccumulatorCodec:
  libraryName: connectivity-batch-accumulator-codec
  class: BatchAccumulator
```

You then need to insert the `batchAccumulatorCodec` in your connectivity chain just before the transport. For example:

```
dynamicChains:
  http:
    - apama.eventMap
    - mapperCodec:
      ...
    - classifierCodec:
      ...
    - jsonCodec
    - stringCodec
```

```
- batchAccumulatorCodec
- httpServer:
    automaticResponses: true
```

The Batch Accumulator codec can be inserted anywhere in the chain, but it is better to insert it close to the transport. It is entirely transparent to the plug-ins either side.

By default, the Batch Accumulator codec has a maximum batch size of 1000. This means if more than 1000 messages are waiting to be processed by the host-bound thread, requests from the transport will block. It also means you can be using up to 1000 times your message size in memory in outstanding events. You can configure a different batch size with the `maxBatchSize` configuration option (see below).

The Batch Accumulator codec exposes the actual size of the queue via a user-defined status value. This is available through the various monitoring tools for the correlator with the name `user-chain.batchAccumulator.queueSize`. This will be the most recent batch size removed from the queue. See also [“User-defined status reporting from connectivity plug-ins” on page 63](#).

The following configuration option is available for the Batch Accumulator codec:

Configuration option	Description
<code>maxBatchSize</code>	Optional. The maximum number of messages in a batch. This must be a positive integer, at least 1. Default: 1000.

The Message List codec connectivity plug-in

The Message List codec can be used with a service which supports processing a batch of events in a single request by combining multiple requests into a single list. This requires support from the connected service since the Message List codec changes the type of events being sent.

At high event rates, the correlator will produce batches of messages rather than a single message. Some transports, such as the HTTP client, are inherently single-event based and the maximum rate they can send at depends on the speed of processing a single message. The Message List codec can combine a batch of messages being sent from the correlator into a single message whose body is a list of the original messages in the batch. If the destination service supports this, then the whole batch can be delivered in a single round-trip from the transport.

If the service produces replies which are also a list of replies, then the Message List codec splits them up and delivers them back to the correlator as separate messages.

You need a transport or downstream codec which expects the lists produced by the Message List codec as well as a service which supports them. Often this will be by encoding the lists in something like JSON and then using the String codec to produce a binary message for the transport.

To load the Message List codec, an entry such as the following is required in the `connectivityPlugins` section of the configuration file (see also [“Configuration file for connectivity plug-ins” on page 26](#)):

```
messageListCodec:
  libraryName: connectivity-message-list-codec
  class: MessageListCodec
```

You then need to add the `messageListCodec` to your connectivity chain in the appropriate place. A typical use might be for accessing a web service which has been configured to process lists of messages in JSON format. The HTTP client chain for such a service might look like this:

```
startChains:
  webServiceChain:
    - apama.eventMap
    - mapperCodec:
      ...
    - classifierCodec:
      ...
    - messageListCodec:
      metadataMode: first
    - jsonCodec
    - stringCodec
    - httpClient:
      host: ${WEBSERVICE_HOST}
      port: ${WEBSERVICE_PORT}
```

With the above example, the lists produced by the Message List codec are being encoded into JSON to be consumed by the web service.

The following configuration options are available for the Message List codec:

Configuration option	Description
<code>maxBatchSize</code>	<p>Optional. The maximum number of events that are to be combined into a single list. The actual number will depend on the correlator's accumulation of messages to send.</p> <p>This must be a positive integer.</p> <p>Default: 1000.</p>
<code>metadataMode</code>	<p>Required. The strategy for handling the metadata of multiple requests. This must be one of the following:</p> <ul style="list-style-type: none"> ■ <code>first</code> - Just use the metadata from the first message in the batch as the metadata for the list message. ■ <code>splitBatch</code> - Only add items whose metadata is identical from the batch to a list. Create a new list message when the metadata changes. ■ <code>requestIdList</code> - Use the metadata from the first message, but set <code>metadata.requestId</code> to be a list containing the <code>requestId</code> of each message ■ <code>member</code> - Instead of creating a list of payloads, create a list of maps, each with two members, where <code>metadata</code> refers

Configuration option	Description
	to the metadata for that message and payload refers to the payload of that message.
	When converting a list back to separate messages, the above mapping is performed in reverse to create the individual messages.

The main choice to make is how to handle the metadata when combining multiple messages into a single message. Which choice you will make depends on your application. Let us assume that we have a batch of messages of the following form:

```
metadata = { requestId: 5, http: { method: PUT, path: /add } }  
payload = { name: "Matt", age: 30 }
```

The payload values and the `requestId` vary with each message. The examples below show how the Message List codec combines two messages in a batch using the different `metadataMode` strategies.

metadataMode: first

```
metadata = { requestId: 5, http: { method: PUT, path: /add } }  
payload = [ { name: "Matt", age: 30 }, { name: "James", age: 21 } ]
```

metadataMode: requestIdList

```
metadata = { requestId: [5, 6], http: { method: PUT, path: /add } }  
payload = [ { name: "Matt", age: 30 }, { name: "James", age: 21 } ]
```

metadataMode: splitBatch

```
metadata = { requestId: 5, http: { method: PUT, path: /add } }  
payload = [ { name: "Matt", age: 30 } ]  
metadata = { requestId: 6, http: { method: PUT, path: /add } }  
payload = [ { name: "James", age: 21 } ]
```

metadataMode: member

```
metadata = { }  
payload = [  
  {  
    metadata: { requestId: 5, http: { method: PUT, path: /add } },  
    payload: { name: "Matt", age: 30 }  
  },  
  {  
    metadata: { requestId: 6, http: { method: PUT, path: /add } },  
    payload: { name: "James", age: 21 }  
  }  
]
```


You need to construct your application and the web service it is calling for the strategy that you have chosen. In some cases, you may need an additional Mapper codec to set some of the metadata for the combined message on the transport side of the Message List codec.

The Unit Test Harness codec connectivity plug-in

The Unit Test Harness codec and the Null Transport transport make it easy to send test messages into a connectivity chain and/or to write out received messages to a text file, without the need to write any EPL in the correlator, which is very useful for writing unit tests for connectivity plug-ins, and especially codecs.

- **Unit Test Harness.** This is a Java codec that is useful for unit testing your plug-ins in isolation, by writing messages from the chain to a text file, and/or sending test messages into the chain from a text file (in either direction) without the need to use or configure the host or transport at either end of the chain.
- **Null Transport.** This is a trivial Java transport that does not send any messages towards the host and ignores any messages received from the host direction. If you are unit-testing a codec plug-in, no sending or receiving functionality is required for the transport. However, as a transport needs to exist at the end of a connectivity chain, you should use the Null Transport as the chain's transport.

The following example configuration shows the how the harness could be used for testing the behavior of a codec plug-in in the “towards host” direction, by sending test messages through the plug-in being tested and writing the messages from the plug-in out to a file:

```
connectivityPlugins:
  unitTestHarness:
    classpath: ${APAMA_HOME}/lib/connectivity-unit-test-harness.jar
    class: com.softwareag.connectivity.testplugins.UnitTestHarness
  nullTransport:
    classpath: ${APAMA_HOME}/lib/connectivity-unit-test-harness.jar
    class: com.softwareag.connectivity.testplugins.NullTransport
  # plug-in being tested would also be defined here

startChains:
  MyTestChain:
    - apama.eventMap
    # this is a unit test, so the host is not used

    - unitTestHarness:
      pluginUnderTest: towardsTransport
      writeTo: output-towards-host.txt
    - myCodecPluginBeingTested
    - unitTestHarness:
      pluginUnderTest: towardsHost
      readFrom: ${TEST_INPUT_DIR}/input-towards-host.txt

    - nullTransport
    # this is a codec unit test, so no transport functionality is required
```

Apama ships an example of some PySys test cases using the Unit Test Harness codec as part of the JSON codec. You can find it in the `samples\connectivity_plugin\java\JSON-Codec\tests` directory of your Apama installation.

The following configuration options are available for the Unit Test Harness codec:

Configuration option	Description
<code>pluginUnderTest</code>	<p>Required. Either <code>towardsTransport</code> or <code>towardsHost</code>, indicating which direction the plug-in being tested is along the chain relative to this <code>unitTestHarness</code> instance.</p> <p>Messages from the <code>readFrom</code> file are sent down the chain towards the direction identified by <code>pluginUnderTest</code>, and messages received from that direction of the chain (that is, that were sent towards the opposite of the <code>pluginUnderTest</code> direction) are written to the <code>writeTo</code> file.</p> <p>For example, a chain might have the host plug-in followed by a <code>unitTestHarness</code> with <code>pluginUnderTest=towardsTransport</code> followed by a codec plug-in that you are testing, followed by a <code>unitTestHarness</code> with <code>pluginUnderTest=towardsHost</code> followed by a <code>nullTransport</code> instance.</p>
<code>writeTo</code>	<p>The path of a UTF-8 text file to which messages from the plug-in under test are written.</p> <p>If empty, no messages are written to a text file.</p> <p>Default: empty.</p>
<code>readFrom</code>	<p>The path of a UTF-8 text file from which messages are read and sent towards the plug-in under test, or a directory containing such text files. When this plug-in is started, messages are read from the file and sent towards the plug-in under test. If a directory is specified, then the same is done for any new files in that directory, including any files that are subsequently written to that directory while the plug-in is running.</p> <p>If empty, no messages are sent by the <code>unitTestHarness</code>.</p> <p>Default: empty.</p>
<code>logOutput</code>	<p>By default, the <code>unitTestHarness</code> writes a single-line log message to the host's log file for each message from the plug-in under test, using the same format as <code>writeTo</code>. This may be useful as an alternative to <code>writeTo</code>, or as a way to create test cases that block until either the host log file contains the final expected message or an error message.</p> <p>If you want to disable this behavior because the log file is becoming too large, then set this to <code>false</code>.</p> <p>Default: <code>true</code>.</p>

Configuration option	Description
<code>passThrough</code>	<p>By default, any messages received by the <code>unitTestHarness</code> are not passed on to the next plug-in in the chain (typically the transport or the host), as usually writing or logging the messages is all that is required and passing them on would require extra configuration in the host or transport to avoid error messages.</p> <p>If you want to include the host or transport in your testing, then set this to <code>true</code>.</p> <p>Default: <code>false</code>.</p>
<code>echoBack</code>	<p>If set to <code>true</code>, messages received from the plug-in under test are automatically sent back in the opposite direction towards the plug-in under test. This is useful for testing the round-trip mapping behavior of a codec plug-in.</p> <p>Default: <code>false</code>.</p>

The file format for `readFrom` and `writeTo` is identical, containing a metadata line followed by a payload line, repeating until the end of the file. Blank lines and lines starting with a hash (#) are ignored. The file encoding for both is always UTF-8 (which is the same as ASCII for the most common alphanumeric and punctuation characters).

A metadata line is encoded as a single-line JSON string. For example:

```
metadata={"aim":"purpose of this test message", "other stuff":[1, 2, 3]}
```

A payload line can use one of the following formats, depending on the type:

- Any message payload except for `byte[]` can be encoded using as a single-line JSON string. For example:

```
payload_json={"a\nb":123, "c":[true]}
```

There are some special JSON formats that can be used:

- JSON does not allow non-string keys. A special format of a JSON object with a `".Map"` key and a value of a list of size 2 lists will be converted to a map. For example:

```
{".Map": [[123, "abc"], [987, "zyx"]]}
```

This would be converted to `{123:"abc", 987:"zyx"}` if JSON allowed it. These may also be nested, for example:

```
{".Map": [{"".Map": [[333, "abc"], [555, "zyx"]]}, "value"]}
```

Any other keys in the `".Map"` object will be ignored.

- A special format of a JSON map object with a `".NamedMap"` key to a string value and a `".Map"` key will create a `NamedMap` Java class which can be used with the EPL any type. For example:

```
{"NamedMap":"MyEvent","Map":{"i":123}}
```

The contents of ".Map" will be named with the `MyEvent` name which can be used to determine the type of the converted any type variable. The ".Map" value may also use the special formatting above.

- Although JSON can be used to represent simple string payloads, it is sometimes simpler to use `payload_string` format for these as it removes the need to follow JSON quoting rules. For example:

```
payload_string=a " b
```

Note that the above can only be used if there are no new lines in the string. If there are new lines, use a JSON string instead. For example:

```
payload_json="my\nstring"
```

- For binary payloads (that is, a message whose payload is a byte array), use `payload_byte[]`, which takes a base64-encoded representation of binary data. For example:

```
payload_byte[]=SGVsbG8gV29ybGQ=
```

The Diagnostic codec connectivity plug-in

The Diagnostic codec can be used to diagnose issues with connectivity plug-ins. It logs the events that go through a connectivity chain in either direction.

To reference the Diagnostic codec, an entry such as the following is required in the `connectivityPlugins` section of the configuration file (see also [“Configuration file for connectivity plug-ins” on page 26](#)):

```
diagnosticCodec:
  libraryName: DiagnosticCodec
  class: DiagnosticCodec
```

You can then add the `diagnosticCodec` at any point in a connectivity chain. With no further configuration, the codec logs to the correlator log file at `INFO` level.

An example configuration may look as follows:

```
startChains:
  myChain:
    - apama.eventMap
    - diagnosticCodec:
      tag: host
      output: logger
      logLevel: DEBUG
    - myCodec # the codec being inspected
    - diagnosticCodec:
      tag: transport
      output: logger
      logLevel: DEBUG
    - myTransport
```

The following configuration options are available:

Configuration option	Description
<code>tag: string</code>	If a chain has multiple <code>DiagnosticCodec</code> instances, you can specify a tag for each instance to distinguish it from other instances. <i>string</i> is the tag that is used to prefix the messages from the current instance. Default: empty.
<code>output: mode</code>	Defines the file to which the codec logs its output. <i>mode</i> can be one of the following: <ul style="list-style-type: none"> ■ <code>logger</code> - Default. The codec logs to the correlator log file at the log level that is defined with <code>logLevel</code>. ■ <code>file</code> - The codec logs to the file that is defined with <code>fileName</code>.
<code>logLevel: level</code>	Applies when the <code>logger</code> mode is defined. <i>level</i> can be any correlator log level. Default: <code>INFO</code> .
<code>fileName: file</code>	Applies when the <code>file</code> mode is defined. <i>file</i> is either the path to a file or one of the special strings <code>stdout</code> or <code>stderr</code> . Default: <code>stdout</code> .

When writing to the correlator log file, the Diagnostic codec replaces any non-printable ASCII characters (those with ASCII values less than 0x20, which includes tab and newline) with an underscore (`_`) character.

Output to files

If the mode of the output configuration option is set to `file`, the Diagnostic codec formats the output messages as follows:

```
[tag] direction: metadata / payload
```

where:

- *tag* is optional. This is the tag that is defined in the configuration (or omitted completely if no tag is configured). If a tag is present, it is enclosed in square brackets.
- *direction* is either “Towards Host” or “Towards Transport”.
- *metadata* is the content of the metadata at the point in the chain where the Diagnostic codec has been placed.
- *payload* is the content of the payload at the point in the chain where the Diagnostic codec has been placed.

The following is an example of an output message that is written to a file:

```
[host] Towards Transport: {sag.type:test.EventType, sag.channel:myChain} /
{t:Isabelle,isB:true}
```

Output to loggers

If the mode of the output configuration option is set to `logger`, the Diagnostic codec formats the output messages with an additional prefix:

```
timestamp loglevel [threadID] - <connectivity.codecname.chainname>
```

where:

- *timestamp* is the date and time that the output was logged.
- *loglevel* is configured in the Diagnostic codec configuration.
- *threadID* is the unique integer identifier of the thread that logged the message.
- *codecname* is the name of the Diagnostic codec listed in the configuration (usually “`diagnosticCodec`”).
- *chainname* is the name of the codec chain listed in the configuration.

The following is an example of an output message that is written to a log file:

```
2020-03-16 17:45:14.472 DEBUG [18744] - <connectivity.diagnosticCodec.myChain> [host]  
Towards Transport: {sag.type:test.EventType,sag.channel:myChain} /  
{t:Isabelle,isB:true}
```

III Correlator-Integrated Support for the Java Message Service (JMS)

12	Using the Java Message Service (JMS)	233
----	--------------------------------------------	-----

12 Using the Java Message Service (JMS)

■ Overview of correlator-integrated messaging for JMS	234
■ Getting started with simple correlator-integrated messaging for JMS	236
■ Getting started with reliable correlator-integrated messaging for JMS	246
■ Mapping Apama events and JMS messages	248
■ Dynamic senders and receivers	275
■ Durable topics	276
■ Receiver flow control	276
■ Monitoring correlator-integrated messaging for JMS status	277
■ Logging correlator-integrated messaging for JMS status	278
■ JMS configuration reference	285
■ Designing and implementing applications for correlator-integrated messaging for JMS .	296
■ Diagnosing problems when using JMS	311
■ JMS failures modes and how to cope with them	313

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. In this documentation, this support is referred to as "correlator-integrated messaging for JMS".

Note:

Apama supports a number of different JMS providers. For provider-specific limitations, see the Apama readme, which is available from <http://documentation.softwareag.com/apama/index.htm>.

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.

Note:

If a license file cannot be found, the correlator is limited to `BEST_EFFORT` only messaging. See "Running Apama without a license file" in *Introduction to Apama*.

In Software AG Designer, you can specify configuration for JMS, 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 Apama's adapter editor.

Samples for using correlator-integrated messaging for JMS

Apama provides the following example applications in Software AG Designer 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.

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 Software AG Designer, 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.

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 provides an example application in Software AG Designer 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. From the **File** menu, choose **New > Apama Project**. 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. From the **Select required bundle instances** list box, select the **JMS (Correlator-integrated support for the Java Message Service)** bundle.
4. Click **Finish**.


The correlator-integrated adapter for JMS is added to the project's **Connectivity and Adapters** node. In addition, all the necessary resources to support correlator-integrated messaging for JMS are generated. 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.

Adding and configuring connections

When you first add the correlator-integrated messaging for JMS bundle to an Apama 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 **Connectivity and Adapters** node and then expand the **JMS (Correlator-integrated support for the Java Message Service)** node.
2. Double-click the adapter's 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. In the JMS Configuration Wizard, specify the following:
 - a. **JMS Provider**, select from the drop-down list.
 - b. **Connection ID** must be unique. The connection ID is 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 ID is not exposed to the JMS provider in any way.
 - c. **Description** is optional and currently unused.
5. Click **Next**.
6. The Classpath details page of the JMS Configuration Wizard displays the default classpath details for the JMS provider that you selected in the previous step. Add or modify the values as appropriate for your environment.

To add or modify the classpath details:

- a. The **Select Installation Directory** field lists the default directory where the JMS provider's JAR files are located. You can change this directory by clicking the browse button (...).
 - b. To add an entry to the CLASSPATH, click the **Add Classpath** button (+) and add the new value in the Add Classpath Variable dialog. You can also remove an entry by selecting it and clicking the **Remove Classpath** button (x).
7. Click **Next** to proceed to the Connection Properties page of the JMS Configuration Wizard. If necessary, add or modify the values as appropriate.
 - a. The **Use JNDI** checkbox indicates the usage of JNDI by JMS providers. For JMS providers that use JNDI, the checkbox is selected. For providers that do not support JNDI, the checkbox is not selected. It is not possible to change the value of the **Use JNDI** checkbox. If necessary, you can edit the generated XML file after completing the wizard to change how the connection factory is instantiated. For more information about customizing the XML, see ["XML configuration bean reference" on page 288](#) and Spring Beans documentation.

- 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, select 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.

Note: If you are using JNDI to get the connection factory, it is usually necessary to first add and configure a JNDI name for the connection factory you wish to use using the administration tools provided by the JMS implementation you are using. For example, if using Universal Messaging, this would be the Enterprise Manager tool. A common mistake when configuring the JNDI connection factory binding is to use `localhost` rather than a fully qualified host name or IP address. For many JMS implementations, this will not permit connections from hosts other than the one that the server is running on.

8. Click **Finish**.

The adapter editor is updated 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.

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 Apama's 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.

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 Apama's 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 240](#).

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 248](#).

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.

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, Software AG Designer 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 266](#).

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 266](#).

Adding a classpath variable entry

You can add an entry to the JMS provider's connection classpath using the New Classpath Variable dialog.

➤ To add a classpath variable entry

1. In the Project Explorer, double-click the project's correlator-integrated adapter for JMS instance. This opens the instance configuration in Apama's adapter editor.
2. Select the desired JMS connection.
3. Expand the **Classpath** section and click the **Add Classpath Variable** button (.

This displays the New Classpath Variable dialog.

4. From the **Choose Group** drop-down list, select the group that represents the JMS connection.
5. If desired, add a **Variable Name** and **Variable Value** (either both fields must be filled in or both must be blank).


When you create a variable in this dialog, you can use it as a shorthand way of specifying locations when you want to add several JAR files from the same location. If you specify the name of a previously defined variable in the **Variable Name** field, the **Variable Value** field is automatically filled in.

6. In the **Jar Name** field, enter the name of the JAR file or click the **Browse** button and select the JAR file.
7. Click **OK**.

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 Apama's 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**.

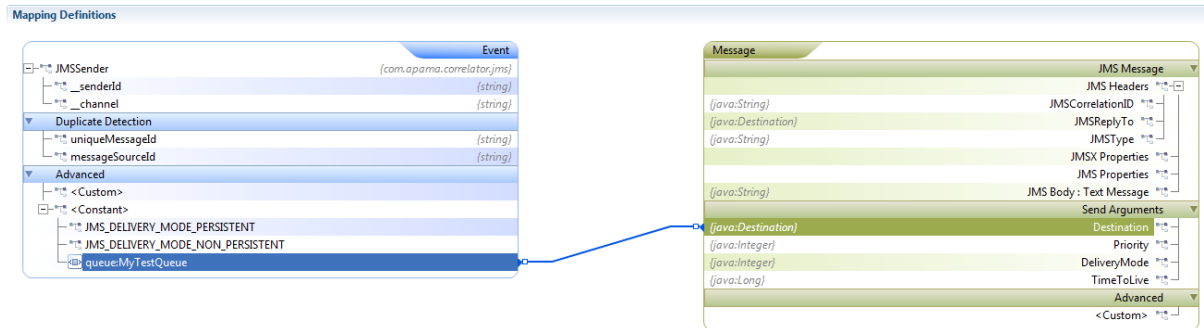
This updates the display in the adapter editor's **Event Mappings** tab to show a hierarchical view of the Apama event on the left (the mapping source) and a hierarchical view of the JMS message on the right (the mapping target).

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 248](#).

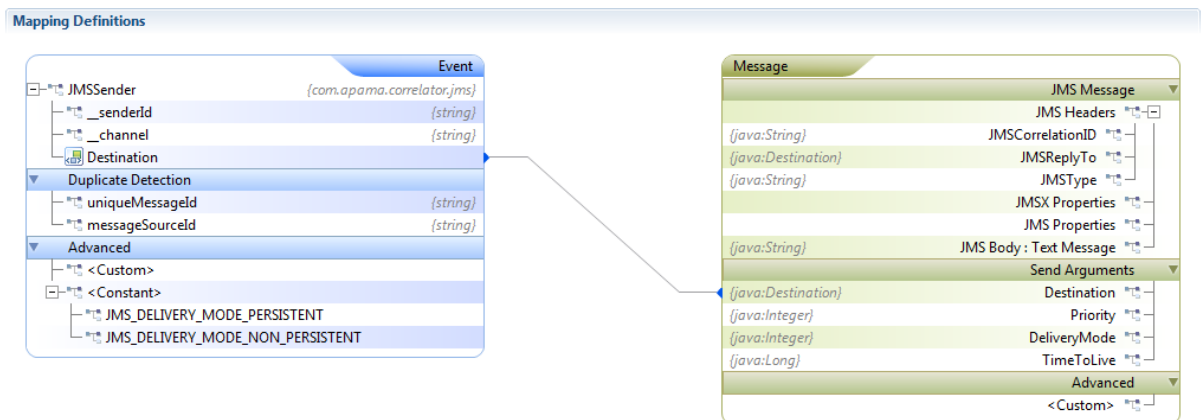
11. Specify the message's JMS destination in either of two ways:

- 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 249](#).

- 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*.

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.

Software AG Designer, engine_deploy and other tools all ensure that .evt files are sent in after all EPL has been injected.

- 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() {
    com.apama.correlator.jms.JMS.onApplicationInitialized();
    // Any other post-injection startup logic goes here too.
}
```

Note:

For simple applications, you can add the EPL bundle **Automatic onApplicationInitialized** to your project (see also "Adding bundles to projects" in *Using Apama with Software AG Designer*). This bundle will ensure that `onApplicationInitialized` is called as soon as the entire application has been injected into the correlator. However, in cases where you need to wait for a `MemoryStore`, database or another resource to be prepared before your application is able to begin to process incoming messages, you should not use the bundle. In these cases, you should write your own start event and application logic.

- 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 that `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 244](#).

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 299](#).

The steps described in this section build on the example created in [“Getting started with simple correlator-integrated messaging for JMS” on page 236](#).

Note:

If a license file cannot be found, the correlator is limited to BEST_EFFORT only messaging. See *“Running Apama without a license file” in Introduction to Apama*.

➤ **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 236](#).
2. If necessary, in the Project Explorer expand the project's **Connectivity and 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 the 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*.

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. Apama's adapter editor in Software AG Designer provides a visual mapping tool to create the mapping rules. There are several approaches for how to map Apama events to the JMS messages - these are explained in the topics below.

In addition, you can also specify transformation types:

- **XSLT transformation type.** Use this approach when receiving JMS messages containing XML to change or simplify the structure of the XML document.
- **XPath XML transformation type.** Use this approach 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.
- **XMLDecode transformation type.** Use this approach when receiving JMS messages containing XML and multiple rules are working off of the same XML source.

For more information, see ["Specifying transformation types" on page 274](#).

Simple mapping for JMS messages

Use this approach when a simple Apama event field can be associated with a corresponding value in the JMS message.

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 described below.

➤ To drag a line

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.

Each rule is represented 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.

Using expressions in mapping rules

Use this approach when sending or receiving JMS messages and you need to write a customized JUEL expression for a mapping rule.

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 266](#).

Template-based XML generation

Use this approach 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.

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.

Adding multiple XPath mapping rules for a received XML document

Use this to configure a set of XPath mappings, based on an XML schema or sample XML document using the Treat as dialog.

Using Treat As on JMS Body dialog - Sender

In the **Mapping Definitions** section of the **Sender Mapping Configuration**, when you right-click **JMS Body: Text Message** and select the **Treat as...** option, the Treat As dialog appears. The Treat As dialog allows you to select the type of JMS message to be sent.

➤ To select a base type for the JMS body

1. Select the base type of the JMS body type in the **Body Type** field. If you are mapping to a bytes message, the UTF-8 encoding is used to convert the character string to a bytes message.
2. Click **OK**.

Using Treat As on JMS Body dialog - Receiver

In the **Mapping Definitions** section of the **Receiver Mapping Configuration**, when you right-click **JMS Body: Text Message** and select the **Treat as...** option, the Treat As dialog appears. The Treat As dialog allows you to select a base type for the JMS body to treat a node in the mapper.

➤ To select a base type for the JMS body

1. Select the base type of the JMS body type in the **Body Type** field. If you select **Text Message** or **Bytes Message**, you must also select one of the following options:
 - Select the **String** option if you have selected **Text Message** in the **Body Type** field.
 - Select the **Bytes** option if you have selected **Bytes Message** in the **Body Type** field. If the JMS body is a bytes message, the UTF-8 encoding is used to convert it into a character string.
 - Select the **XML** option and browse for the required XML file.

- Select the **XML using schema** option and browse for the required schema file using the Type Chooser dialog. See [“Using the Type Chooser dialog” on page 251](#) for more information on using the Type Chooser dialog. The **Element name** is automatically populated if it is defined for the chosen schema file.

2. Click **OK**.

Using the Type Chooser dialog

The Type Chooser dialog allows you to choose and display types from files in different source locations. It supports:

- The file types XML, XSD, WSDL, and Schema
- Selecting a source file from a local and remote URL
- Retrieving and displaying all types from the selected file
- Global searches of specific types
- WSDL service files and multilevel imports

> To choose a type

1. Click the drop-down arrow beside the **Source** field to select a location where the system can find the type definitions. The following options are available:

Option	Description
Recent	Displays the recently selected file locations.
Local File System	Selects a file in your local file system. Selecting this option displays the Open dialog; it also determines which dialog is opened when you subsequently select ... (Open).
Workspace	Selects a file in your workspace. Selecting this option displays the Select Resource dialog in which you can filter and select a Schema, XML, XSD, or WSDL file extension. You can also select a custom file extension (if any).
Remote URL	Selects a file from a web-based URL. Selecting this option displays the Select Remote URL dialog in which you can specify the remote URL for the file.
XML Schema	Selects a built-in type defined by the XML schema specification. Selecting this option displays the appropriate choices in the bottom pane; it also determines the system behavior when you subsequently select ... (populates the XML schema types again).

You can search for these types of files: XML, XSD, and WSDL. In addition, you can use custom file types (if any) and built-in XML Schema types.

2. Optionally, click ... to search for a file that you recently selected.

The bottom pane of the dialog displays the types available in the selected file.

3. To filter the types and display a specific type, enter the first few characters of the type that you want to select in the type filter text field. The supported pattern characters are: ? for single character and * for a string.

Using the XPath Helper

The XPath Helper enables you to generate and evaluate XPath expressions. It implements all features defined in the W3C Recommendation for the XML Path Language (XPath). See also <http://www.w3.org/TR/xpath/> (Version 1.0) and <http://www.w3.org/TR/xpath20/> (Version 2.0).

Launching the XPath Helper

When you open an XML file in the XML editor, an **XML** menu is shown in the Software AG Designer menu bar.

Note:

In some cases, there might be different ways of launching the XPath Helper. Depending on the context you launch it in you can use the **Input document** field to select a different XML document.

➤ To launch the XPath Helper

- Do one of the following:
 - Open an XML document in the XML Editor, and from the **XML** menu, choose **XPath Helper**.
 - Or select the XML document in the **Project Explorer** view, invoke the context menu and then choose **XML > XPath Helper**.

The XPath Helper dialog is shown in both cases. The content of the current XML document is shown in the left pane in the form of a tree, consisting of XML elements and their values.

The path to the current XML document is shown in the **Input document** field. If you want, you can also select a different XML document using this field.

Note:

In some cases, you might not be able to use the **Input document** field, as its accessibility depends on the context in which you launch the XPath Helper.

Setting node properties

The XPath Helper enables you to set a node in the document tree (shown in the left pane of the XPath Helper dialog) as the target, key, or root node.

➤ To set node properties

1. Right-click a node in the left pane of the XPath Helper dialog.
2. From the resulting context menu, choose one of the following commands:

Command	Description
Set Target	<p>Sets the node as the target (indicated with a red circle). Enables you to test whether the selected node is present in the result set of an XPath expression. It also enables you to test whether that node appears once or multiple times in the result set.</p> <p>When you evaluate the XPath expression, the number of hits found for the selected node is shown in the Results pane. If the target node appears in the result set only once, the Results pane displays the message “Reached target uniquely”, and the target indicator (red circle) in the left pane of the XPath Helper dialog changes to a check mark. If the node appears more than once, it displays the message “Reached target” with the target indicator changed to a check mark. If it does not appear at all, it displays the message “Did not reach target” and the target indicator remains as it is.</p>
Remove Target	Removes the current target.
Set Key	<p>Sets the selected node as the key (indicated by a yellow key symbol) and displays the appropriate XPath expression in the text area of the XPath Expression tab.</p> <p>Each time you set a key, a text box is shown next to the key symbol where you can enter a condition or constraint.</p> <p>You can also enter a condition or constraint in the text area of the XPath Expression tab.</p>
Remove Key	Removes the current key.
Set Root	Sets the selected node as the root. Only the subtree starting from this node will be considered as the input parameter for evaluating and generating XPath expressions.
Remove Root	Removes the current root.

Generating XPath expressions





The XPath Helper enables you to generate XPath expressions. XPath expressions are used to select nodes or node sets in an XML document.

> To generate an XPath expression

1. In the document tree in the left pane of the XPath Helper dialog, double-click a node to display the expression on the **XPath Expression** tab.
2. Select one of the following XPath generation options from the corresponding button at the top of the **XPath Expression** tab (initially **Prefix** is shown as the button name, indicating that this option is currently selected):

Option	Description
Prefix	Includes prefixes in the generated XPath expression. The XPath Helper resolves these prefixes to their corresponding namespaces, depending upon the namespace entries listed on the Namespaces tab.
Namespace	Includes the required namespaces in the generated XPath expression. This generated XPath expression is independent of the list of namespaces on the Namespaces tab.
Local Name	Generates an XPath expression that is independent of the prefixes and namespaces used in the input document.

3. In addition, you can select the following options:

Icon	Option	Description
	Recent XPath Expressions	Displays 10 recent XPath expressions. You can select a different expression for evaluation.
	Use Index	Inserts an index in the XPath expression to identify the position of the element.
	Invoke Text Function	Generates an XPath expression that invokes the <code>text()</code> function in the last location step.
	Select Node Depth	Limits the number of location steps in an XPath expression and determines whether the expression is relative or absolute.

With each selection, the appropriate XPath expression is shown in the text area of the **XPath Expression** tab, ready to be evaluated.

Note:

You can also modify the default generated XPath expression in the text area.

Adding and removing namespaces


The XPath Helper dialog displays a list of namespaces on the **Namespaces** tab. They are used for evaluating the XPath expression. You can add or remove namespace entries from this tab.

> To add or remove a namespace


1. Select the **Namespaces** tab in the right pane of the XPath Helper dialog.

A list of all namespaces found in the root node of the document tree is shown.

2. To add a namespace:

- a. Click .
- b. In the resulting Add Namespace dialog, specify a prefix and a namespace URI.
- c. Click **OK**.


3. To remove a namespace:

- a. On the **Namespaces** tab, select the namespace you want to remove.
- b. Click .
- c. Click **OK** to confirm the removal of the namespace.

Evaluating XPath expressions

After you have generated an XPath expression, you can evaluate it.

> To evaluate an XPath expression

1. Either use the expression that is currently shown on the **XPath Expression** tab, or click  and select another expression.
2. Select either Version 1.0 or Version 2.0 of the W3C Recommendation for the XML Path Language (XPath). To do so, use the drop-down list of the corresponding button at the top of the **XPath Expression** tab (initially **Version 2.0** is shown as the button name, indicating that this option is currently selected).
3. Do one of the following:
 - Click **Evaluate** to evaluate the generated expression.

- Or select **Auto Evaluate** to enable XPath Helper to continuously display results as you double-click a node in the document tree or manually enter the XPath expressions.

The XPath Helper evaluates the expression according to the selected criteria and shows the results in the **Results** pane, in a tree format. You can expand and collapse the tree if it has subnodes.

Using convention-based XML mapping with JMS messages

Use this approach 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.

The topics below explain how to use convention-based XML mapping with JMS messages.

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` or `integer`. 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` or `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` or `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`.
- A field of the `optional` type is processed in the same way as the contained type. If the `optional` value is empty, then it is not processed when creating XML. Similarly when creating an Apama

event from XML, if a node corresponding to an optional field is absent, then the field will have an empty value.

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).
- 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 that are Apama EPL keywords (such as `<return>`) must be escaped in the event definition using a hash character, for example, `string #return;`.
- 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`.

You can generate event type definitions automatically from an XML schema using Software AG Designer. See "Creating new event definition files for EPL applications" in *Using Apama with Software AG Designer*.

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.
- Sequences of optional types are not supported.

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>
    <e1>Hello</e1>
    <e1>there</e1>
    <e-2 e2att='value1'><subElement>e2-sub-value1</subElement></e-2>
  <e-2 e2att='value2'><subElement>e2-sub-value2</subElement></e-2>
  <e1>world</e1>
  <namespacedElement xmlns='urn:xmlns:foobar'>My namespaced
    text</namespacedElement>
  <p:namespacedElement>My namespaced text 2</p:namespacedElement>
  <namespacedElement>My non-namespaced text 3</namespacedElement>
  <return>Element whose name is an EPL keyword</return>
</myroot>
```

Define the Apama event `MyEvent` as follows:

```
event MyElement2
{
    string _myattribute;
    boolean _myboolattribute;
    string xmlTextNode;
}
event E2
{
    string _e2att;
    string subElement;
}
event MyRoot
{
    string myelement1;
    MyElement2 myelement2;
    sequence<string> e1;
    sequence<string> namespacedElement;
    string #return;
    sequence<E2> e$002d2;
}
event MyEvent
{
    string destination;
    MyRoot myroot;
}
```

Note that the field names and types matter but the event type names do not.

The document above would be parsed to the following Apama event string:

```
MyEvent("queue:MyQueue",
  MyRoot("An element value",
    MyElement2("123",true,"456"),
    ["Hello","there","world"],
    ["My namespaced text","My namespaced text 2","My non-namespaced text 3"],
    "Element whose name is an EPL keyword",
    [E2("value1","e2-sub-value1"),E2("value2","e2-sub-value2")]
  ))
```

The exact same event definitions could be used in the other direction for creating an XML document, although the node order will be slightly different from that of the document shown above (based on the field order) and everything would be in the same XML namespace.

For the above example, the following XML is generated with `http://www.example.com/myevent` as the namespace and `p` as the namespace prefix:

```
<p:myroot xmlns:p="http://www.example.com/myevent">
  <p:myelement1>An element value</p:myelement1>
  <p:myelement2 myattribute="123" myboolattribute="true">456</p:myelement2>
  <p:e1>Hello</p:e1>
  <p:e1>there</p:e1>
  <p:e1>world</p:e1>
  <p:namespacedElement>My namespaced text</p:namespacedElement>
  <p:namespacedElement>My namespaced text 2</p:namespacedElement>
  <p:namespacedElement>My non-namespaced text 3</p:namespacedElement>
  <p:return>Element whose name is an EPL keyword</p:return>
  <p:e-2 e2att="value1">
    <p:subElement>e2-sub-value1</p:subElement>
  </p:e-2>
  <p:e-2 e2att="value2">
    <p:subElement>e2-sub-value2</p:subElement>
  </p:e-2>
</p:myroot>
```

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.

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. 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 dialog.
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**.

In the mapping tree, an entry of type **Convert To XML** is added to the selected event node.

6. Drag a mapping line from the **Convert To XML** entry to the desired node in the XML message, for example, to **JMS Body**.

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.

Add Computed Node dialog

The Add Computed Node dialog provides an efficient means of adding customized entries to the mapping rules in the **Mapping Definitions** section. This feature helps you take advantage of the following benefits:

- You do not need to re-enter the custom entries each time you want to map a node.
- You can modify the entry dynamically for all mappings rules generated from a node at once, by simply modifying the entry.

➤ **To add customized entries to the mapping rules**

1. By default, the **Type** is selected as **Computed Node**. Depending on the customization you want to add before the mapping, select one of the following from the drop-down list in the **Select method** field:
 - **String.concat** to concatenate the specified string to the end of this string. The **Display name** field appears with a default value. Edit the value if you want. Enter the String in the String field that you want to concatenate in the **Method Arguments** section.
 - **String.contains** to determine whether a specific substring is contained within the string. A Boolean value, true is returned if the substring is contained within the string; false otherwise. The **Display name** field appears with a default value. Edit the value if you want. Enter the Substring in the Substring field in the **Method Arguments** section.
 - **String.endsWith** to determine whether the string ends with a specific suffix. A Boolean value, true is returned if the string ends with the specified suffix; false otherwise. The **Display name** field appears with a default value. Edit the value if you want. Enter the suffix string in the Suffix field in the **Method Arguments** section.
 - **String.replaceAll** to replace each substring of this string that matches the given regular expression with the given replacement String. The **Display name** field appears with a default value. Edit the value if you want. Enter the regular expression and the replacement Strings in the Regex and Replacement fields of the **Method Arguments** section.
 - **String.startsWith** to determine whether the string starts with a specific prefix. A Boolean value, true is returned if the string starts with the specified prefix; false otherwise. The **Display name** field appears with a default value. Edit the value if you want. Enter the suffix string in the Suffix field in the **Method Arguments** section.
 - **String.substring** to obtain a specific substring within a given string. The substring is specified by a beginIndex (inclusive) and an endIndex (exclusive). The **Display name** field appears with a default value. Edit the value if you want. Enter the begin index in the Begin index field and end index in the End index field in the **Method Arguments** section.
 - **String.toLowerCase** to convert all of the characters in this String to lower case. The **Display name** field appears with a default value. Edit the value if you want.
 - **String.toUpperCase** to convert all of the characters in this String to upper case. The **Display name** field appears with a default value. Edit the value if you want.
 - **String.trim** to return a copy of the String, with leading and trailing whitespace omitted. The **Display name** field appears with a default value. Edit the value if you want.
 - **XPath** if your mapping requires an XPath transformation. The **Display name** field appears with a default value. Edit the value if you want. In the **Method Arguments** section, enter

the expression manually or 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. Build the desired XPath expression using the XPath Helper.

- **XML transformation** if your mapping requires an XSLT transformation. The **Display name** field appears with a default value. Edit the value if you want. In the **Method Arguments** section, enter the expression manually or click **Browse** to locate the file of the stylesheet to use. You can also use the drop-down arrow to create a new stylesheet or select from the local file system or workspace.
- **XMLDecode** if your mapping requires parsing information from an XML document, and you want to use Apama's XMLDecode transformation which offers higher performance than XPath transformation. See [“XMLDecode” on page 263](#) for more information on XMLDecode properties. The **Display name** field appears with a default value. In the **Method Arguments** section:
 - In the **Node path** field, specify a valid node path. For example, `/root/someelement[2]/text()`.
 - In the **Properties** field, specify the properties as `property=value`. If you want to specify multiple properties, specify the properties separated by a semicolon.
- **Convert To XML** if you want to convert an event to XML, or to convert an event field that is an event type or sequence type to XML. The **Convert to XML** method uses convention-based mapping. See [“Using convention-based XML mapping with JMS messages” on page 256](#) for more information. 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. The **Display name** field appears with a default value.

In the **Method Arguments** section:

- Optionally, type a namespace for the generated XML document in the **Namespace** field. For example, `http://www.example.com/myevent`.
- Optionally, type a namespace prefix for the generated XML document in the **Prefix** field. See [“Convention-based JMS message mapping example” on page 257](#) for more information.
- 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.

The expression that will be used in the mapping rule is displayed in the **Expression value** field.

2. Click **OK**.

XMLDecode

The XMLDecode is similar to XML codec functionality intended to be used in the correlator JMS. The XMLDecode uses the Least Recently Used (LRU) cache to avoid repeating the decode when multiple rules are working off of the same XML source. For example, JMS body.

The XMLDecode supports the following properties:

- skipNullFields - Default value is True.
- trimXMLText - Default value is False.
- generateTwinOrderSuffix - Default value is True.
- generateSiblingOrderSuffix - Default value is False.
- logFlattenedXML - Default value is False.
- namespaceAware - Default value is False.
- xmlField - This property needs to be set only if nested XML within a CDATA section needs to be parsed.
- sequenceField
- ensurePresent
- separator
- parseNode

See [“The XML codec IAF plug-in” on page 507](#) for a detailed description of XMLDecode properties.

For properties with multiple values, use a comma (,) to separate the values.

For multiple properties, use a semicolon (;) as a separator and the equals sign (=) to separate the key and value.

The XMLDecode functionality can be used in two ways:

- As a mapping action. See [“Specifying transformation types” on page 274](#).
- As a mapping method using the Add Computed Node dialog. See [“Add Computed Node dialog” on page 260](#).

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, "".

Using custom EL mapping extensions

Apama's correlator-integrated adapter for JMS uses 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: 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.

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 Software AG Designer 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.

Use a `<jms:classpath>` element to enclose the `ELMappingExtensionProviderList` bean.

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 <code>jms.headers</code>	<code>dictionary<string, string></code>	<code>dictionary<string, string></code>
JMSDestination <code>jms.header['JMSDestination']</code>	<code>string</code> (with <code>jndi:/topic:/queue: prefix</code>)	<code>string</code> (with <code>topic:/queue: prefix</code>)
JMSReplyTo <code>jms.header['JMSReplyTo']</code>	<code>string</code> (with <code>jndi:/topic:/queue: prefix</code>)	<code>string</code> (with <code>topic:/queue: prefix</code>)
JMSCorrelationID <code>jms.header['JMSCorrelationID']</code>	<code>string</code>	<code>string</code>
JMSType <code>jms.header['JMSType']</code>	<code>string</code>	<code>string</code>
JMSPriority	<code>integer, string</code>	<code>integer, string</code>

JMS message element / JMS EL expression	Compatible Apama field type(s) when sending	Compatible Apama field type(s) when receiving
<code>jms.header['JMSPriority']</code>		
JMSDeliveryMode <code>jms.header['JMSDeliveryMode']</code>	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 <code>jms.header['JMSTimeToLive']</code>	integer, string (in milliseconds from the time JMS sends the message)	N/A when receiving
JMSExpiration <code>jms.header['JMSExpiration']</code>	N/A when sending	integer, string (in milliseconds since the epoch)
JMSMessageID <code>jms.header['JMSMessageID']</code>	N/A when sending	boolean, string
JMSTimestamp <code>jms.header['JMSTimestamp']</code>	N/A when sending	integer, string (in milliseconds since the epoch)
JMSRedelivered <code>jms.header['JMSRedelivered']</code>	N/A when sending	string
Dictionary of all message properties <code>jms.properties</code>	dictionary<string, string>	dictionary<string, string>
String Message Property <code>jms.property['propName']</code>	string	string
Boolean Message Property <code>jms.property['propName']</code>	boolean	boolean, string
Long Message Property <code>jms.property['propName']</code>	integer	integer, string
Double Message Property <code>jms.property['propName']</code>	float	float, string
Byte Message Property	Not supported	string

JMS message element / JMS EL expression	Compatible Apama field type(s) when sending	Compatible Apama field type(s) when receiving
<code>jms.property['propName']</code>		
Short Message Property	Not supported	string
<code>jms.property['propName']</code>		
Integer Message Property	Not supported	string
<code>jms.property['propName']</code>		
Float Message Property	Not supported	string
<code>jms.property['propName']</code>		
JMSX Property	Same as other properties	Same as other properties
<code>jms.xproperty['propName']</code>		
Dictionary of all JMSX properties	dictionary<string, string>	dictionary<string, string>
<code>jms.xproperties</code>		
TextMessage Body	string, event ^[1]	string, event ^[1]
<code>jms.body.textmessage</code>		
MapMessage Body	dictionary<string, string>	dictionary<string, string>
<code>jms.body.mapmessage</code>		
MapMessage Body Entry	string	string
<code>jms.body.mapmessage['mapKey']</code>		
ObjectMessage Body with a serializable <code>java.util.Map<Object, Object></code>	dictionary<string, string>	dictionary<string, string>
<code>jms.body. objectmessage</code>		
ObjectMessage Body with a serializable <code>java.util.List<Object></code>	sequence<string>	sequence<string>
<code>jms.body. objectmessage</code>		
ObjectMessage Body with any serializable Object	N/A	string
<code>jms.body. objectmessage</code>		

JMS message element / JMS EL expression	Compatible Apama field type(s) when sending	Compatible Apama field type(s) when receiving
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 [“Using convention-based XML mapping with JMS messages” on page 256](#) 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.

For receiving messages	Description
<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.

String method	Description
<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 Java 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> .
<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.

Binary method	Description
<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.

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, bidirectional 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 identifiers 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 identifier (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.

Specifying transformation types

In the **Mapping Element Details** section, in the **Transformation Type** field select the desired type from the drop-down list.

➤ To specify the transformation details

1. 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.
3. Under **Mapping Element Details** on the **Properties** tab, in the **Transformation Type** drop-down list:
 - Select **XSLT Transformation**. In the **Stylesheet URL** field, click **Browse** to locate the stylesheet file.
 - Select **XPath**. 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:
 1. Click the **Browse** button [...] to the right of the **XPath Expression** field.
 2. 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. See also [“Using the XPath Helper” on page 252](#).

3. 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`.
4. In the XPath Helper, click **OK**. The XPath Builder closes and the **XPath Expression** field displays the XPath expression you built.

■ Select **XMLDecode**.

1. In the **Node path** field, specify a valid node path.
2. In the **Properties** field, specify the properties as `property=value`. If you want to specify multiple properties, specify the properties separated by a semicolon.

For more information, see [“Specifying XML codec properties” on page 510](#).

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 236](#), 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 287](#).)

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.

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.

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 Software AG Designer, 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;
...
on all JMSReceiverFlowControlMarker() as 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 in ApamaDoc format 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 *API Reference for EPL (ApamaDoc)*.

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.

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 `JMSConnectionStatus`, `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.

Occasionally, it may be useful 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.

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 "Descriptions of correlator status log fields" (in *Deploying and Managing Apama Applications*, 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 pm=956240 runq=0
INFO [20032:Status] - JMS Status: s=1 tx=6 sRate=1,200 sOutst=9,005 r=2
rx=4 rRate=1,180 rWindow=-1 rRedel=3 rMaxDeliverySecs=2.1
rDupsDet=2 rDupIds=2,005,023 connErr=2 jvmMB=49

```

Status information logged by correlators that are configured for integrated messaging is described in the following sections.

The following table describes the correlator 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

Field	Full name	Description
		window sizes. Note that even if this value is greater than 0 there could still be one or more receivers which have exhausted their own windows and are blocked, so consider enabled <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

Field	Full name	Description
		detector. Does not include dynamic receivers that have now been removed.
rDupIds	Duplicate ids in memory	The total number of uniqueMessageIds being kept in memory for duplicate detection purposes by EXACTLY_ONCE reliable receivers. This is the total of the size of all per-message-source fixed-size expiry queues plus the unbounded time-based expiry queue. If this becomes too large it is possible the correlator could run out of memory.
connErr	Connection errors	The total number of times a valid JMS connection has gone down. Note that this tracks errors in existing connections and does not include repeated failures to establish a connection.
jvmMB	JVM used memory	The amount of memory used by the JVM in Megabytes (heap plus non-heap), which can be compared with the maximum memory size provided for the JVM to check how much spare memory there is and ensure that the correlator is not close to running out of memory. This is particularly useful for checking peak memory consumption, such as testing when any EXACTLY_ONCE duplicate detectors are fully populated with the maximum likely number of uniqueMessageIds 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 objects waiting to be garbage collected, so inevitably this will go up and down a certain amount as garbage collections occur.
...	onApplication Initializedindicator	The suffix <waiting for onApplicationInitialized> will be added to the status lines if the EPL application has not yet called <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 pm=956240 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 maxMsgKB=1,650.9
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 maxMsgKB=1,245.7
INFO [19276:Status] - JMSReceiver myConnection-receiver-apama-queue-01:
      rx=5 rRate=0 rWindow=96 rRedel=0
      rMaxDeliverySecs=0.0 msgErrors=1 rDupsDet=1
      perSourceDupIds=4 timeExpiryDupIds=0 maxMsgKB=1,000.2
```

The JMS connector-specific status lines contain:

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

Field	Full name	Description
		decremented when the previous session is closed.

The JMS sender-specific status lines contain:

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 WARN and ERROR messages that will provide more details.

The JMS receiver-specific status lines contain:

Field	Full name	Description
rx	Received messages	The number of messages received from JMS, including messages received but not yet mapped to Apama events and added to the input queue of each public context.
rRate	Receive throughput rate	The number of JMS messages received per second by this receiver, calculated over the interval since the last status line (typically 5 seconds).
rWindow	Flow control window size	If receiverFlowControl is disabled this has the special value "-1". If it is enabled, this gives the current flow control window size, that is the number of successfully mapped events that can still be received before the receiver will block. A zero value indicates that the flow control window has been exhausted and the application should call

Field	Full name	Description
		JMSReceiverFlowControlMarker. updateFlowControlWindow() to unblock the receiver. A negative value indicates that the window has been updated to a negative value, which has the same effect as a window of 0.
rRedel	Redelivered messages	The number of JMS messages received with the JMSRedelivered flag set to true.
rMaxDeliverySecs	Maximum delivery time	The highest time taken by the JMS broker to deliver a message to this receiver, based on the difference between the time when each message is received and the value of the JMSTimestamp message header field which indicates the time when it was sent.
rMsgErrors	Per-message error count	The number of received JMS messages that could not be passed to the Apama application due to some error, typically a mapping failure. See the log file for WARN and ERROR messages that will provide more details.
rDupsDet	Duplicate messages detected	The number of duplicate messages detected by this EXACTLY_ONCE receiver and suppressed because their uniqueMessageId was already present in this receiver's duplicate detector. Only displayed for EXACTLY_ONCE receivers.
perSourceDupIds	Per-source duplicate ids in memory	The total number of uniqueMessageIds being kept in memory for duplicate detection purposes by all per-message-source fixed-size expiry queues. Only displayed for EXACTLY_ONCE receivers.
timeExpiryDupIds	Time-based duplicate ids in memory	The total number of uniqueMessageIds being kept in memory for duplicate detection purposes by the unbounded time-based expiry uniqueMessageId queue.
maxMsgKB	Maximum message size received so far	The maximum size in kilobytes of JMS messages that have been received so far.

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.

Configuration files for JMS

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` option (see also "Starting the correlator" in *Deploying and Managing Apama Applications*), 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.)

Note:

When the correlator is started, any properties that are specified with the `--config file or -Dkey=value` option take precedence and override the properties defined in a `connectionId-spring.properties` file. An INFO message is then logged for all Spring properties that are being ignored.

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 in Software AG Designer, 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). Software AG Designer 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 Software AG Designer 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 Software AG Designer for all mapping configuration tasks.

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 are located in different XML files.

Setting property values

Most bean properties have primitive values (such as string, number, boolean) which are set like this:

```
<property name="propName" value="my value"/>
```

However, there are also a few properties that reference other beans, such as the `reliableReceiveDatabase` property on `jms:connection` and the `receiverSettings` property on `jms:receiver`. These property values can be set by specifying the ID of a top-level bean like this (where it is assumed that `globalReceiverSettings` is the ID of a `JmsReceiverSettings` bean):

```
<property name="receiverSettings" ref="globalReceiverSettings"/>
```

Any top-level bean may be referenced in this way, that is, any bean that is a child of the `<beans>` element and not nested inside another bean. Referencing a bean that is defined in a different configuration file is supported, and the `jms-global-spring.xml` file is intended as a convenient place to store top-level beans that should be shared across many different JMS connections.

Instead of referencing a shared bean, it is also possible to configure a bean property by creating an “inner” configuration bean nested inside the property value like this:

```
<property name="receiverSettings">
  <bean class="com.apama.correlator.jms.config.JmsReceiverSettings">
    <property name="logJmsMessages" value="true"/>
  </bean>
</property>
```

(Note, advanced users may want to exploit Spring's property inheritance by using the `parent=` attribute on an inner bean to inherit most properties from a standard top-level bean while overriding some specific subset of properties or by type-based “auto-wiring” - any non-primitive property of `jms:connection/receiver/sender` for which no value is explicitly set will implicitly reference a top-level bean of the required type. This is how `jms:connection` beans get a reference to the `reliableReceiveDatabase` and `defaultSender/ReceiverSettings` beans. Most configuration can just ignore this detail and use the automatically wired property values, 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 Software AG Designer.

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.

XML 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 264](#).

jms:connection

This bean defines the information needed to establish a JMS Connection to a single JMS broker instance. Its required properties are: `connectionFactory` or `connectionFactory.jndiName`, and (if JNDI is used to locate the connection factory), `jndiContext`.

Example:

```
<jms:connection id="MyConnection">
  <property name="staticReceiverList"
    value="{staticReceiverList.MyConnection}" />
```



```

<property name="defaultReceiverReliability"
  value="${defaultReceiverReliability.MyConnection}"/>
<property name="defaultSenderReliability"
  value="${defaultSenderReliability.MyConnection}"/>
<property name="connectionFactory.jndiName"
  value="${connectionFactory.jndiName.MyConnection}" />
<property name="jndiContext.environment">
  <value>
    ${jndiContext.environment.MyConnection}
  </value>
</property>
<property name="connectionAuthentication.username"
  value="${connectionAuthentication.username.MyConnection}" />
<property name="connectionAuthentication.password"
  value="${connectionAuthentication.password.MyConnection}" />
</jms:connection>

```

Supported properties:

- `connectionFactory.jndiName` - the JNDI lookup name for the `ConnectionFactory` object that should be used for this `jms:connection`.
- `connectionFactory` - a JMS provider bean that implements the JMS `ConnectionFactory` interface, if the `ConnectionFactory` is to be instantiated directory by the Spring framework (rather than using JNDI to lookup the `ConnectionFactory`). The bean value that is provided will usually required properties and/or constructor arguments to be specified in order to fully initialize it.
- `connectionAuthentication.username` - the name of the user/principal to be used for the JMS connection (note that this is often different from the username/password needed to login to the JNDI server, which is part of the JNDI environment configuration). Default value is "".
- `connectionAuthentication.password` - the password/credentials to be used for the JMS connection.
- `jndiContext.environment` - the set of properties that specify the environment for initializing access to the JNDI store. Typically includes some standard JNDI keys such as `java.naming.factory.initial`, `java.naming.provider.url`, `java.naming.security.principal` and `java.naming.security.credentials`, and maybe also some provider-specific keys. The usual way to specify a properties map value is `key=value` entries delimited by newlines and surrounded by the `<value>` element, for example, `<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 Apache 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 Apache 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 connections 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.

To protect the security of personal data, see "Protecting Personal Data in Apama Applications" in *Developing Apama Applications*.

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"/>
  <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 unique `messageIds` 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`.

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.

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 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.

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 using any reliability mode other than `BEST_EFFORT`, 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. In addition, messages sent using any of the reliable modes are sent with the JMS `PERSISTENT` delivery mode flag by default, and are guaranteed to remain in the correlator's persistence store until they have been successfully sent to the JMS broker (or until the send failed for a reason other than connection loss).

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 have 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.

When sending messages in a persistent correlator using `BEST_EFFORT`, the behavior is different. In this case, messages are passed to JMS immediately without waiting for a correlator persistence cycle. This results in lower latency, but also means it is possible for a client receiving JMS messages sent by the correlator to see inconsistent output in the event of a correlator failure. For example, the correlator might send one set of messages with unique identifiers (for example, from `integer.incrementCounter()`) but on restart send similar messages but in a different ordering, while responses from the first set of messages may then be received, resulting in mismatches between the requests and the responses being processed.

Important:

Consider carefully what behavior is required by your application, and use one of the reliable modes instead of `BEST_EFFORT` if you need to avoid inconsistent output.

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).

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 299](#).

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 301](#).

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 296](#)

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.

Note:

If a license file cannot be found, the correlator is limited to `BEST_EFFORT` only messaging. See [“Running Apama without a license file” in *Introduction to Apama*](#).

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. Note that regardless of the value of `maxBatchIntervalMillis`, the receiver will not be suspended while no events are being received. See [“Advanced configuration bean properties” on page 295](#).

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 301](#). 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 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, Apache 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 {
        // 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 288](#).

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 an EPL 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.

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.

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`.

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.

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 288](#).

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.

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 **TRANSACTION** 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.

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 Software AG Designer, edit the Apama launch configuration for your project as described below.

> To edit the launch configuration

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**.

Diagnosing problems when using JMS

This topic contains several approaches for diagnosing JMS issues you may encounter.

Note:

If the correlator log indicates it has connected to JMS but you are not receiving any messages, this is usually due to a missing call to `JMS.onApplicationInitialized`. This call is required before the correlator will attempt to receive any messages from JMS.

- 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 context 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 context 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.
- If you are using JNDI to get the connection factory, it is usually necessary to first add and configure a JNDI name for the connection factory you wish to use using the administration tools provided by the JMS implementation you are using. For example, if using Universal Messaging, this would be the Enterprise Manager tool. A common mistake when configuring the JNDI connection factory binding is to use localhost rather than a fully qualified host name or IP address. For many JMS implementations, this will not permit connections from hosts other than the one that the server is running on.

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 276](#) 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.

IV Working with IAF Plug-ins

13	The Integration Adapter Framework	319
14	Using the IAF	327
15	C/C++ Transport Plug-in Development	369
16	C/C++ Codec Plug-in Development	375
17	C/C++ Plug-in Support APIs	387
18	Transport Plug-in Development in Java	391
19	Java Codec Plug-in Development	397
20	Plug-in Support APIs for Java	405
21	Monitoring Adapter Status	409
22	Out of Band Connection Notifications	423
23	The Event Payload	429

13 The Integration Adapter Framework

■ Overview	320
■ Architecture	321
■ The transport layer	323
■ The codec layer	323
■ The Semantic Mapper layer	324
■ Contents of the IAF	325

This section 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.

There are two ways of integrating with Apama through software. The first is to use the low-level Client Software Development Kits (SDKs) to write your own custom software interface (see). The second is to instantiate an adapter with the higher-level Integration Adapter Framework (IAF). This is described in the topics below.

Note:

The IAF architecture is superseded by connectivity plug-ins. Therefore, Software AG strongly recommends choosing connectivity plug-ins over the IAF when creating new adapters and connectivity.

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 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.

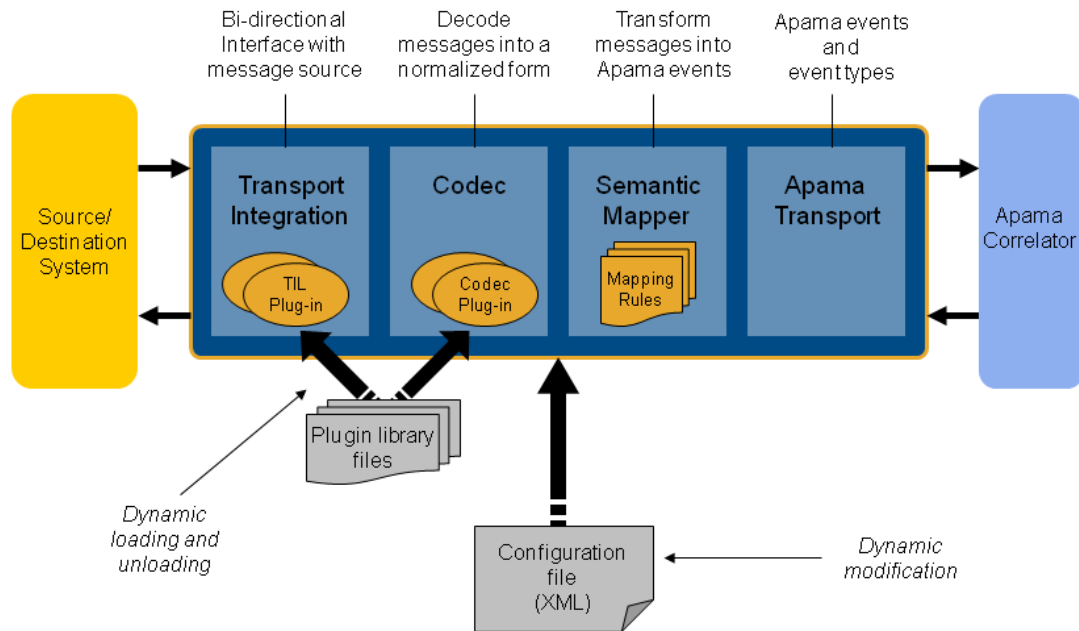
These message sources and/or sinks vary extensively. Typically each comes with its own proprietary communications paradigm, communications protocol, message representation and programming interfaces. Interfacing any software like Apama with a source/sink of messages like a message bus therefore requires 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.



As illustrated above, the IAF acts as the interface between the messaging middleware and the 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 correlator. It injects EPL definitions and event instances into the 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 369](#) and [“Transport Plug-in Development in Java” on page 391](#) 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 339](#).

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 375](#) and [“Java Codec Plug-in Development” on page 397](#) 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 339](#).

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 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 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 343](#).

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 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.
- 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.

14 Using the IAF

■ The IAF runtime	328
■ IAF Management – Managing a running adapter I	335
■ IAF Client – Managing a running adapter II	336
■ IAF Watch – Monitoring running adapter status	337
■ The IAF configuration file	339
■ IAF samples	364

This section describes how to start and manage the Integration Adapter Framework and how to specify an adapter's configuration file.

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 339](#) – 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`, in directories on `JAVA_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.

See [“Java configuration \(optional\)” on page 363](#) for information about how the location of Java plug-in classes are determined.

IAF command-line options

The `iaf` executable starts the IAF. This is located in the `bin` directory of the Apama installation.

Synopsis

To start the adapter, run the following command:

```
iaf [ options ] [ config.xml ]
```

where *config.xml* is the name of a configuration file using the format described in [“The IAF configuration file” on page 339](#). A configuration file must be provided unless the `-h` or `-v` options are used.

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

Description

Unless the `-e` or `--events` options 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` options are used, the `iaf` executable generates event definitions that can be saved to a file and injected during your application's startup sequences as specified by Software AG Designer or Apama command-line tools. If either of these options 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 options. 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.

For more information about the service configuration file, see "Using the Apama component extended configuration file" in *Deploying and Managing Apama Applications*.

If the `--logfile` and `--loglevel` options 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.

Options

The `iaf` executable takes the following options:

Option	Description
<code>-v</code> <code>--version</code>	Displays version information for the <code>iaf</code> executable.
<code>-h</code> <code>--help</code>	Displays usage information.

Option	Description
<code>-p port --port port</code>	Specifies the port on which the IAF should listen for commands. The default is 16903.
<code>-f file --logfile file</code>	Specifies the name of the file to which to write log information. The default is <code>stderr</code> . See also "Text internationalization in the logs" in <i>Deploying and Managing Apama Applications</i> .
<code>-l level --loglevel level</code>	Sets the level of information that is sent to the log file. Available levels are TRACE, DEBUG, INFO, WARN, ERROR, FATAL, CRIT and OFF. The default is INFO.
<code>-t --truncate</code>	Specifies that if the log file already exists, the IAF should empty it first. The default is to append to the log file.
<code>-N name --name name</code>	Sets a user-defined name for the component. This name is used to identify the component in the log messages.
<code>-e --events</code>	Dumps event definitions to <code>stdout</code> and then exits.
<code>--logQueueSizePeriod p</code>	Sends information to the log every <i>p</i> seconds. See also “IAF log file status messages” on page 331 .
<code>--pidfile file</code>	<p>Specifies the name of the file that contains the process identifier. This file is created at process startup and can be used, for example, to externally monitor or terminate the process. The IAF will remove that file after a clean shutdown.</p> <p>It is recommended that the file name includes the port number to distinguish different IAFs (for example, <code>iaf-16903.pid</code>).</p> <p>The IAF takes an exclusive lock on the pidfile while it is running. This means that if you have another IAF or correlator running using the same pidfile, the second process will fail to start up. You should not run multiple components from the same configuration using the same pidfile. In other cases, existing pidfiles will be overwritten, even if they contain a process identifier of a running process.</p>
<code>-Xconfig file --configFile file</code>	Reserved for usage under guidance by Apama support. See also “Using the Apama component extended configuration file” on page 333 .

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. For example:

```
Status: ApEvRx=589 ApEvTx=2056000 TrEvRx=2056008 TrEvTx=587 vm=407200 pm=956240 si=0.0
so=0.0 oq=10
```

Where the fields have the following meanings:

Field	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.
pm	Number of kilobytes of physical/resident 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.
oq	Sum total of events across all output queues of the IAF.

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 the following:

- 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.

Note:

Regularly rotating log files and storing the old ones in a secure location may be important as part of your personal data protection policy. For more information, see "Protecting and erasing data from Apama log files" in *Developing Apama Applications*.

On Windows, to automate log file rotation, you can set up scheduled tasks that run the following tools:

- 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. If the name of the file contains blanks, be sure to enclose it in quotation marks.

```
iaf_management --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). If the name of a file contains blanks, be sure to enclose it 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 --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 --reopenLog
```

Move the IAF log file before you execute the `--reopenLog` option. 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` option applies to both of them. Consequently, you want to move both log files before you issue the `--reopenLog` option.

- 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.

Note:

Some people use the term “log rolling” instead of “log rotation”.

Using the Apama component extended configuration file

The Apama component extended configuration file is an optional file that can be used by the IAF. You can use it to do the following:

- Bind Apama server components to a particular set of addresses.
- Specify that Apama client connections must be from a particular IP address or range of IP addresses.
- Set environment variables for Apama components.

In an extended configuration file, if a line includes a special character that you want to be treated as a literal, you must escape it by inserting a backslash just before it. A character that follows two consecutive backslashes (`\\`) is treated as a literal. Single quotes inside double quotes are treated as a literal. Double quotes inside single quotes are treated as a literal.

Note:

To configure the correlator, see “Configuring the correlator” in *Deploying and Managing Apama Applications*.

Binding server components to particular addresses

To bind Apama server components to a particular address or set of addresses, specify a `BindAddress` line for each address. Specify this in the `[Server]` section of the extended configuration file. For example:

```
[Server]
BindAddress=127.0.0.1:15903
BindAddress=10.0.0.1
```

You can specify as many `BindAddress` lines as you want. Clients can connect to any of the listed addresses.

An IP address is required. If you do not specify a port, the Apama server components use the port that is specified when the correlator is started. This makes it possible to share extended configuration files if you want to restrict connections according to only addresses.

If you do not specify an extended configuration file when you start the correlator, or there are no `BindAddress` entries in the extended configuration file, the Apama components bind to the wildcard address (`0.0.0.0`).

Ensuring that client connections are from particular addresses

To ensure that client connections are from particular addresses, add one or more `AllowClient` entries to the extended configuration file in the `[Server]` section. For example:

```
[Server]
AllowClient=127.0.0.1
AllowClient=192.168.128.0/17
```

An `AllowClient` entry takes an IP address, as in the first example above, or a CIDR (Classless Inter-Domain Routing) address range, as in the second example above. With these example entries in the extended configuration file, the Apama components allow connections from either the localhost (`127.0.0.1`) or IP addresses where the first 17 bits match the first 17 bits of `192.168.128.0`. The Apama components do not accept connections from any other IP addresses.

If you specify an extended configuration file when you start the correlator, and if there are any `AllowClient` entries in the extended configuration file, then the Apama components do not allow connections from any IP address that does not fall within one of the `AllowClient` ranges specified. If you do not specify an extended configuration file when you start the correlator, or there are no `AllowClient` entries in an extended configuration file that you do specify, the Apama components accept connections from any client.

Important:

This feature is intended to prevent mistakenly connecting to the wrong server. It is not intended to prevent malicious intruders since it provides no protection against address spoofing.

Setting environment variables for Apama components

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

```
[Environment]
MY_ENV_VAR=myvalue
```

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

Note:

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

Sample extended configuration file

Save the extended configuration file as a text file. Blank lines are ignored. For example, the contents of `ApamaExtendedConfig.txt` might be as follows:

```
[Server]
BindAddress=127.0.0.1:15903
BindAddress=10.0.0.1
AllowClient=127.0.0.1
AllowClient=10.0.0.0/24
[Environment]
LD_LIBRARY_PATH=/usr/local/mydir
```

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. The executable for this tool is located in the `bin` directory of the Apama installation. Any output information is displayed on `stdout`.

See also [“IAF Client – Managing a running adapter II” on page 336](#) for IAF-specific management information, as opposed to this generic component management tool.

Synopsis

To manage a running adapter, run the following command:

```
iaf_management [ options ]
```

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

Description

For historical reasons, `iaf_management` does the same as `engine_management`. The only difference in behavior is that `iaf_management` defaults to the default IAF port (16903).

For detailed descriptions of all options and exit values, see "Shutting down and managing components" in *Deploying and Managing Apama Applications*.

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 correlator, and to request an adapter to dynamically reload its configuration. The executable for this tool is located in the `bin` directory of the Apama installation.

Synopsis

To manage a running adapter, run the following command:

```
iaf_client [ options ]
```

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

Description

If the adapter is listening for control connections on a non-standard port (specified with the `--port` option to the `iaf` executable), you must pass the same port number to the `iaf_client` tool.

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.

You can use the `--reload` option to dynamically reconfigure a running adapter from a changed configuration file without restarting the IAF. When this option is used, 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 cannot 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`.

Options

The `iaf_client` tool takes the following options:

Option	Description
<code>-h --help</code>	Displays usage information.
<code>-n host --hostname host</code>	Name of the host to which you want to connect. Non-ASCII characters are not allowed in host names.
<code>-p port --port port</code>	Port on which the IAF is listening.
<code>-r --reload</code>	Instructs the IAF to reload its configuration file.
<code>-s --suspend</code>	Instructs the IAF to suspend the sending of events to the correlator (not to the external transport).
<code>-t --resume</code>	Instructs the IAF to resume the sending of events to the correlator (not to the external transport).
<code>-q --quit</code>	Instructs the IAF to shut down.
<code>-v --verbose</code>	Requests verbose output during execution.
<code>-V --version</code>	Displays version information for the <code>iaf_client</code> tool.

IAF Watch – Monitoring running adapter status

The `iaf_watch` tool allows you to monitor the live status of a running adapter. The executable for this tool is located in the `bin` directory of the Apama installation.

Synopsis

To monitor the status of a running adapter, run the following command:

```
iaf_watch [ options ]
```

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

Description

If the adapter is listening for control connections on a non-standard port (specified with the `--port` option to the `iaf` executable), you must pass the same port number to the `iaf_watch` tool.

By default, the tool collects status information from the adapter once per second and displays this in a human-readable form.

Options

The `iaf_watch` tool takes the following options:

Option	Description
<code>-h --help</code>	Displays usage information.
<code>-n host --hostname host</code>	Name of the host to which you want to connect. Non-ASCII characters are not allowed in host names.
<code>-p port --port port</code>	Port on which the IAF is listening.
<code>-i ms --interval ms</code>	Specifies the poll interval in milliseconds.
<code>-f file --filename file</code>	Writes status information to the named file. The default is to send status information to <code>stdout</code> .
<code>-r --raw</code>	Indicates that you want raw output format, which is more suitable for machine parsing. Raw output format consists of a single line for each status message. Each line is a comma-separated list of status numbers. This format can be useful in a test environment. If you do not specify that you want raw output format, the default is a multi-line, human-readable format for each status message.
<code>-t --title</code>	If you also specify the <code>--raw</code> option, you can specify the <code>--title</code> option so that the output contains headers that make it easy to identify the columns.
<code>-o --once</code>	Outputs one set of status information and then quits. The default is to indefinitely return status information at the specified poll interval.
<code>-v --verbose</code>	Displays process names and versions in addition to status information. The default is to display only status information.
<code>-u --utf8</code>	Writes the output in UTF-8 encoding.
<code>-V --version</code>	Displays version information for the <code>iaf_client</code> tool.

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 correlator(s).
- `<universal-messaging>` - Defines how the IAF connects to Software AG's Universal Messaging message bus.

Note:

Use of Universal Messaging from the IAF is deprecated and will be removed in a future release.

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 “[The File IAF Adapter \(JMultiFileTransport\)](#)” on page 483.

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>
  <xi:include href="market-static.xml"
    xpointer="xpointer(/static/codecs)"/>
  <xi:include href="market-static.xml"
    xpointer="xpointer(/static/mapping)"/>
</adapter-config>
```

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 363](#) 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 335](#) 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 343](#) describes the semantic translation and transformation rules. [“IAF samples” on page 364](#) 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 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 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`, 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 normalized 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 [“The Event Payload” on page 429](#) 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` option of the `iaf` executable 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 Software AG Designer 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 429](#) 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.

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 form 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 401](#) as well as the Apama Javadoc. For more information on the C/C++ functions, see [“Codec utilities” on page 381](#).

- 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 `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.

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 correlator configuration

The adapter configuration file requires an `<apama>` element, which configures how the IAF connects to the Apama 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 a 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 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 "Correlator pipelining" in *Deploying and Managing Apama Applications*.

Disabling Apama messaging and using Universal Messaging instead

Note:

Use of Universal Messaging from the IAF is deprecated and will be removed in a future release. It is recommended that you now change any IAF-based adapter configurations using Universal Messaging with a `<universal-messaging>` element in the configuration file to use an `<apama>` element to talk directly to the correlator.

A deployed adapter can use Software AG's Universal Messaging message bus in place of connections specified in the `<apama>` element. When you want to use Universal Messaging instead of explicitly set connections do the following:

- Add a `<universal-messaging>` element in place of or after the `<apama>` element. See [“Configuring IAF adapters to use Universal Messaging” on page 356](#).
- 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 Universal Messaging 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.

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 IAF adapters to use Universal Messaging

Note:

Use of Universal Messaging from the IAF is deprecated and will be removed in a future release. It is recommended that you now change any IAF-based adapter configurations using Universal Messaging with a `<universal-messaging>` element in the configuration file to use an `<apama>` element to talk directly to the correlator. See [“Apama correlator configuration” on page 354](#).

If you are configuring your Apama application to use Universal Messaging, you can configure an IAF adapter to use the Universal Messaging 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 Universal Messaging realm names (RNAME) to connect to. You can use commas or semicolons as separators.

Commas indicate that you want the adapter to try to connect to the Universal Messaging realms in the order in which you specify them here. Semicolons indicate that the adapter can try to connect to the specified Universal Messaging realms in any order.

If you specify more than one RNAME, each Universal Messaging realm you specify must belong to the same Universal Messaging cluster. Specification of more than one Universal Messaging realm lets you benefit from failover features. See the [Universal Messaging documentation](#) for information on Communication Protocols and RNAMEs.

The `um-properties` attribute can be set to the name or path of a file that contains Universal Messaging configuration settings. See also [“Defining Universal Messaging properties for the IAF” on page 359](#).

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 Universal Messaging 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 Universal Messaging 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 Universal Messaging must send each event to a named channel. An adapter that is configured to use Universal Messaging 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 Software AG Designer to create the adapter configuration file, then the `defaultChannel` attribute is set to "*adapter_name adapter_instance_id*". For example: "File Adapter instance 3".
5. If none of `transportChannel`, `presetChannel`, or `defaultChannel` are set and if you did not use Software AG Designer to create the adapter configuration file, then the adapter fails if it tries to use Universal Messaging.

All events sent by the adapter on channels that are Universal Messaging 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 tool.

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 Universal Messaging. The deployed adapter uses only its explicitly set connections.

Subscribing to receive events from an adapter that is using Universal Messaging

In each context, in any correlator, that is listening for events from an adapter that is using Universal Messaging, 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>
```

Defining Universal Messaging properties for the IAF

Note:

Use of Universal Messaging from the IAF is deprecated and will be removed in a future release. It is recommended that you now change any IAF-based adapter configurations using Universal Messaging with a `<universal-messaging>` element in the configuration file to use an `<apama>` element to talk directly to the correlator. See [“Apama correlator configuration” on page 354](#).

The IAF 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 in Software AG Designer to add Universal Messaging configuration to a project, Software AG Designer copies the `um-config.properties` file to the `config` folder in your project.

A Universal Messaging properties file for the IAF can contain entries for the following properties:

Property name	Description
<code>um.channels.escaped</code>	<p>Specifies whether channel names are escaped (<code>true</code>) or not (<code>false</code>). When set to <code>false</code>, the IAF passes channel names directly to Universal Messaging without escaping. In addition, when the slash (/) and backslash (\) characters are not escaped, they can be used to create nested channels.</p> <div> <p>CAUTION:</p> <p>The IAF treats slash (/) and backslash (\) as different characters while Universal Messaging treats them as identical characters (Universal Messaging generally changes a backslash to a slash). You must choose to use one of these characters in your application and standardize on this. Use of both characters as path separators will result in undefined behavior.</p> </div> <p>When escaping is disabled (<code>false</code>), you must be careful not to use characters which are not supported by Universal Messaging (see the Universal Messaging documentation for the most up to date list of supported characters and character sets).</p> <p>Default: <code>true</code>.</p>
<code>um.channels.mode</code>	<p>Indicates whether Universal Messaging channels can be dynamically created. Specify one of the following:</p>

Property name	Description
	<ul style="list-style-type: none"> ■ <code>autocreate</code> The IAF 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 <code>um_</code> for the channel to be a Universal Messaging channel. ■ <code>mixed</code> The IAF looks up each channel to determine if it is a Universal Messaging channel. If the channel does not exist, it is created only if it has the prefix specified by the <code>um.channels.prefix</code> property. ■ <code>precreate</code> Requires Universal Messaging channels to be created by using Universal Messaging Enterprise Manager or Universal Messaging client APIs. The IAF looks up all channels (except the default <code>""</code> channel) to determine whether they are Universal Messaging channels. If a channel does not exist as a Universal Messaging channel, it is not created. <p>Default: <code>precreate</code>.</p>
<code>um.channels.prefix</code>	<p>Specifies a prefix for channel names. Channel names must have this prefix to allow dynamic creation.</p> <p>Default: <code>um_</code>.</p>
<code>um.realms</code>	<p>List of <code>RNAME</code> values (URLs). You can use commas or semicolons as separators.</p> <p>Commas indicate that you want the adapter to try to connect to the Universal Messaging realms in the order in which you specify them here. Semicolons indicate that the adapter can try to connect to the specified Universal Messaging realms in any order.</p> <p>Every <code>RNAME</code> you specify must belong to the same Universal Messaging cluster.</p>

Property name	Description
	Default: Required.
<code>um.security.certificatefile</code>	Security certificate used to connect to Universal Messaging.
	Default: None.
<code>um.security.certificatepassword</code>	Password for the specified security certificate file.
	Default: None.
<code>um.security.truststorefile</code>	Certificate authority file for verifying server certificate.
	Default: None.
<code>um.security.user</code>	User name supplied to the Universal Messaging realm.
	Default: Current user name from the operating system.
<code>um.session.pool</code>	Configures how many Universal Messaging sessions to use. More sessions can increase throughput by allowing events to be sent in parallel, but may consume more CPU.
	Note that if you are using the SHM protocol to communicate with the broker, you will probably want to limit the number of sessions to 1 or 2, as SHM connections will consume 2 CPU cores for each session.
	Default: 8.

For example, a Universal Messaging 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
```

The Universal Messaging configuration file for the IAF is encoded in UTF-8.

Communicating with the correlator over Universal Messaging

Note:

Use of Universal Messaging from the IAF is deprecated and will be removed in a future release. It is recommended that you now change any IAF-based adapter configurations using Universal

Messaging with a `<universal-messaging>` element in the configuration file to use an `<apama>` element to talk directly to the correlator. See [“Apama correlator configuration” on page 354](#).

The correlator must be using the Universal Messaging transport connectivity plug-in (see [“The Universal Messaging Transport Connectivity Plug-in” on page 73](#)), and this connectivity plug-in must be configured to be equivalent to the Universal Messaging properties you have configured for the IAF.

You should use a single `dynamicChains` definition:

```
dynamicChains:
  UMString:
    - apama.eventString:
        suppressLoopback: true
        description: "@{um.rnames}"
        remoteAddress: "@{um.rnames}"
    - stringCodec
        nullTerminated: true
    - UMTransport:
        channelPattern: ".*"
```

The configuration of the chain manager should be equivalent to certain properties defined in the IAF:

IAF configuration	Connectivity plug-in configuration
um.channels.prefix=UM_ um.channels.mode=precreate	managerConfig: channel: prefix: UM_ includePrefixOnUM: true missingChannelMode: ignore
um.channels.prefix=UM_ um.channels.mode=autocreate	managerConfig: channel: prefix: UM_ includePrefixOnUM: true missingChannelMode: create
um.channels.prefix=UM_ um.channels.mode=mixed	managerConfig: channel: prefix: UM_ includePrefixOnUM: true missingChannelMode: create
um.channels.mode=precreate	managerConfig: channel: prefix: "" missingChannelMode: ignore

Logging configuration (optional)

The optional `<logging>` and `<plugin-logging>` 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` options 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 340](#)), 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. For example:

```
<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 328](#) 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 [“Codec IAF Plug-ins” on page 499](#) 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 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 (for example, `"", "=", ";"`).
- 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 correlator, provides a simple event mapping, and configures the IAF for sending and receiving events to and from the 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 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.

Plug-ins need to be placed in a location where they can be picked up by the 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 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 (for example, `"", ",", "=", ";"`).
- 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 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 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.

15 C/C++ Transport Plug-in Development

■ The C/C++ transport plug-in development specification	370
■ Transport example	373
■ Getting started with transport layer plug-in development	373

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 319](#) 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 section 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 391](#) 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 `apiaf.lib`. To do this in Visual Studio, select your project and then select **Project Properties > Linker > Input > Additional Dependencies** and add `apiaf.lib;apcommon.lib`.

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

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. You can find these files in the `include` directory of your Apama installation.

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. See the `AP_EventTransport_Functions` structure in the *API Reference for C++ (Doxygen)* for detailed information on these functions.

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 the `addEventDecoder` and `removeEventDecoder` functions. 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 375](#).

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.

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. See the `AP_EventTransport_Functions` structure in the *API Reference for C++ (Doxygen)* for more information.

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 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:

```
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, and one such object needs to be created for every plug-in within its constructor function. See the `AP_EventTransport_Functions` structure in the *API Reference for C++ (Doxygen)* for more information.

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 definitions:

- `AP_EventTransportCtorPtr` points to the constructor function. 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.
- `AP_EventTransportDtorPtr` points to the related destructor function.
- `AP_EventTransportInfoPtr` points to the info function.

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.

See the *API Reference for C++ (Doxygen)* for more information on the above definitions.

Other transport definitions

`EventTransport.h` also provides some additional definitions that the event transport author needs to be aware of:

- `AP_EventTransportError` defines the set of error codes that can be returned by the transport's functions.
- The `AP_EventTransportProperty` structure 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.
- Properties are passed to the event transport within an `AP_EventTransportProperties` structure.
- The status of a transport is reported in an `AP_EventTransportStatus` structure.

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. These files are located in the `include` directory of your Apama installation. See the contents of these files for more information.

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`
- `getLastError`

See `AP_EventTransport_Functions` in the *API Reference for C++ (Doxygen)* for more information on these functions.

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);
```

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 364](#) for more information about this sample. [“The Basic File IAF Adapter \(FileTransport/JFileTransport\)” on page 497](#) describes how the `FileTransport` plug-in can be used in practice.

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.

16 C/C++ Codec Plug-in Development

■ The C/C++ codec plug-in development specification	376
■ Transport example	384
■ Getting started with codec layer plug-in development	384

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 319](#) 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 397](#) 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. [“Codec IAF Plug-ins” on page 499](#) 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 `apiaf.lib`. To do this in Visual Studio, select your project and then select **Project Properties > Linker > Input > Additional Dependencies** and add:

```
apiaf.lib;apcommon.lib
```

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

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`. 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. You can find these files in the `include` directory of your Apama installation.

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`
- `getLastError`
- `getStatus`

It is recommended that `updateProperties` is invoked by the codec constructor.

See the `AP_EventCodec_Functions` structure in the *API Reference for C++ (Doxygen)* for detailed information on these functions.

Codec encoder functions

If the codec is to implement an encoder, implementations need to be provided for the following functions:

- `sendNormalisedEvent`
- `flushUpstream`
- `getLastError`
- `addEventTransport`
- `removeEventTransport`

See the `AP_EventEncoder_Functions` structure in the *API Reference for C++ (Doxygen)* for detailed information on these functions.

Codec decoder functions

If the codec is to provide a decoder, implementations need to be provided for the following functions:

- `sendTransportEvent`
- `setSemanticMapper`
- `flushDownstream`
- `getLastError`

See the `AP_EventDecoder_Functions` structure in the *API Reference for C++ (Doxygen)* for detailed information on these functions.

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 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 with the following functions:

- `updateProperties`
- `getLastError`
- `getStatus`

where the library functions `updateProperties`, `getLastError` and `getStatus` are being defined as being the implementations of the Codec Development Specification's `updateProperties`, `getLastError` and `getStatus` function definitions respectively.

See the `AP_EventCodec_Functions` structure in the *API Reference for C++ (Doxygen)* for detailed information.

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` structure with the following functions:

- `sendNormalisedEvent`
- `flushUpstream`
- `getLastError`
- `addEventTransport`
- `removeEventTransport`

See the `AP_EventEncoder_Functions` structure in the *API Reference for C++ (Doxygen)* for detailed information.

In the implementation of an encoding codec, this function table could be implemented as follows:

```
static struct AP_EventEncoder_Functions EventEncoder_Functions = {
    sendNormalisedEvent,
    flushUpstream,
    getLastErrorEncoder,
    addEventTransport,
    removeEventTransport
};
```

This time, the library functions `sendNormalisedEvent`, `flushUpstream`, `getLastError`, `addEventTransport` and `removeEventTransport` are being defined as the implementations of the Codec Development Specification's `sendNormalisedEvent`, `flushUpstream`, `getLastError`, `addEventTransport` and `removeEventTransport` function definitions respectively.

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 with the following functions:

- `sendTransportEvent`
- `setSemanticMapper`
- `flushDownstream`
- `getLastError`

See the `AP_EventDecoder_Functions` structure in the *API Reference for C++ (Doxygen)* for detailed information.

In the implementation of a decoding codec, this function table could be implemented as follows:

```
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.

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. See the *API Reference for C++ (Doxygen)* for detailed information on these structures.

An `AP_EventCodec` object needs to be created for every plug-in within its constructor function. 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`.

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 definitions:

- `AP_EventCodecCtorPtr` points to the constructor function.
- `AP_EventCodecDtorPtr` points to the destructor function.
- `AP_EventCodecInfoPtr` points to the info function. Every codec needs to implement an info function. This is called by the IAF to obtain information as to the capabilities (encoder/decoder) of the codec.

The IAF will search for these functions by the names `AP_EventCodec_ctor` and `AP_EventCodec_dtor` when the library is loaded, and it will search for and call `AP_EventCodec_info`. So you must use these exact names when implementing a codec plug-in.

See the *API Reference for C++ (Doxygen)* for more information on the above definitions.

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 info function to define whether the codec can decode or encode messages.

```
#define AP_EVENTCODEC_CAP_ENCODER 0x0001
#define AP_EVENTCODEC_CAP_DECODER 0x0002
```

- `AP_EventCodecError` defines the set of error codes that can be returned by the codec's functions.
- The `AP_EventCodecProperty` structure 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.
- Properties are passed to the event transport within an `AP_EventCodecProperties` structure.
- The status of a codec is reported in an `AP_EventCodecStatus` structure.

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.

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. These files are located in the `include` directory of your Apama installation. See the contents of these files for more information.

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`, where functions, (of type `AP_SemanticMapper_Functions*`) points to the definitions for two functions:

- `sendNormalisedEvent`
- `getLastError`

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`.

`AP_SemanticMapperError` defines the error codes that can be returned by `sendNormalisedEvent`.

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 369](#). 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`
- `getLastError`

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`.

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 Writer 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);
```

An any field and optional field can be added as follows:

```
AP_EventWriter* event = AP_EventWriter_ctor(AP_EVENT, "MyEvent");

AP_EventWriterValue val;
val.refValue = NULL;

//add an 'empty' any
event->addAny(event, AP_EMPTY, val, NULL);

//add an 'empty' optional as the second field
event->addOptional(event, AP_EMPTY, val);

val.intValue = 100;
event->addAny(event, AP_INTEGER, val, NULL);
val.intValue = 200;
event->addOptional(event, AP_INTEGER, val);

//Add an 'any' field containing 'optional'
AP_EventWriter *opt = AP_EventWriter_ctor(AP_EVENT, "optional");
opt->addInt(opt, 1);

val.refValue = opt;
event->addAny(writer, AP_EVENT, val, "optional<integer>");
```

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. It contains two main structures:

■ AP_NormalisedEvent

This structure represents a single normalized event. It 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.

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 called when the event is no longer required to free up resources.

■ `AP_NormalisedEventIterator`

This structure can be used to step through the contents of a normalized event structure, in forwards or reverse order. It contains a function table defined by `AP_NormalisedEventIterator_Functions`, which includes all of the functions exported by a normalized event iterator.

`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.

In both `AP_NormalisedEvent` and `AP_NormalisedEventIterator` functions, there is always a pointer to the corresponding structure. This is analogous to the implicit `this` pointer passed to a C++ object when a member function is invoked on it.

See the `NormalisedEvent.h` header file for more information about the structures and functions.

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 364](#) for more information about this sample. [“The Basic File IAF Adapter \(FileTransport/JFileTransport\)” on page 497](#) describes how the `FileTransport` plug-in can be used in practice.

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 [“Codec IAF Plug-ins” on page 499](#) 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.

17 C/C++ Plug-in Support APIs

■ Logging from IAF plug-ins in C/C++	388
■ Using the latency framework	388

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.

Logging from IAF 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 362](#) 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 your 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`

`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`

Along with the other logging functions 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`

- `AP_LogInfo`

- `AP_LogWarn`

- `AP_LogError`

- `AP_LogCrit`

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 reconfiguration requests.

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++ 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 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”.

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.

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-ins to write the code that makes use of them.

See the `IAF_TimestampConfig` structure in the *API Reference for C++ (Doxygen)* for detailed information on the fields.

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.

See the `AP_TimestampSet_Functions` structure in the *API Reference for C++ (Doxygen)* for detailed information on the available functions.

18 Transport Plug-in Development in Java

■ The transport plug-in development specification for Java	392
■ Example	395
■ Getting started with Java transport layer plug-in development	395

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 319](#) 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 section 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 369](#) provides analogous information about developing transport plug-ins using C/C++.

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 `ap-iaf-extension-api.jar`, so this file must always be on the classpath during plug-in development. It is located in the `lib` directory of your Apama installation.

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`.

Java transport functions to implement

HTML Javadoc documentation for `AbstractEventTransport` and related classes is provided as part of the Apama documentation set. See the *API Reference for Java (Javadoc)* for detailed information on the functions that a transport plug-in author needs to implement.

`AbstractEventTransport` is the constructor. A typical constructor would create a logger using the plug-in name provided (see [“Logging from IAF plug-ins in Java” on page 406](#)), 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.

See [“Communication with the codec layer” on page 393](#) for information on how the transport layer communicates with the codec layer in both the upstream and downstream directions.

Communication with the codec layer

This section discusses how the transport layer communicates with the codec layer in both the upstream and downstream directions.

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`.

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 (for example, `<property name="decoderName" value="MyCodec"/>`), and saving it in an instance field (for example, `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). See the *API Reference for Java (Javadoc)* for more information on this method.

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.

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.

`TransportException` defines a number of constructors, to make it easy to set up the exception's information quickly in different situations.

See the `TransportException` class in the *API Reference for Java (Javadoc)* for more information on these constants and constructors.

Logging

See [“Logging from IAF plug-ins in Java” on page 406](#) for information about how transport plug-ins should log error, status and debug information.

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 364](#) for more information about this sample. The section [“The Basic File IAF Adapter \(FileTransport/JFileTransport\)” on page 497](#) describes how the `JFileTransport` plug-in can be used in practice.

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.

19 Java Codec Plug-in Development

■ The codec plug-in development specification for Java	398
■ Java codec example	403
■ Getting started with Java codec layer plug-in development	403

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 *decoding*. Decoding involves translating *downstream* messages retrieved by the transport layer into the standard “normalized 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 319](#) 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 375](#) 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. [“Codec IAF Plug-ins” on page 499](#) 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.

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 `ap-iaf-extension-api.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 functions to implement

HTML Javadoc documentation for `AbstractEventCodec` and related classes is provided as part of the Apama documentation set. See the *API Reference for Java (Javadoc)* for detailed information on the functions that a codec plug-in author needs to implement.

`AbstractEventCodec` is the constructor. A typical constructor would create a logger using the plug-in name provided (see [“Logging from IAF plug-ins in Java” on page 406](#)), 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.

See [“Communication with other layers” on page 399](#) for information on how the codec layer communicates with the transport layer and Semantic Mapper in upstream and downstream directions.

See also [“Working with normalized events” on page 401](#) for help working with `NormalisedEvent` objects.

Communication with other layers

This section discusses how the codec layer communicates with the transport layer and Semantic Mapper in upstream and downstream directions.

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 (for example, `<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. See the `EventTransport` interface in the *API Reference for Java (Javadoc)* for more information on this method.

For example, the implementation of the event codec's `sendNormalisedEvent` could look something like this:

```
// Select EventTransport using saved plug-in property value
EventTransport transport = eventTransports.get(currentTransportName);
// Encode message
MyCustomMessageType message = myEncodeMessage(event);
// Send to Transport layer plug-in
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.

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. See the `SemanticMapper` interface in the *API Reference for Java (Javadoc)* for more information on this method.

For example, the implementation of the event codec's `sendTransportEvent` 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.

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.

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.

See the `CodecException` class in the *API Reference for Java (Javadoc)* for more information on these constants and constructors.

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 difference is the set of error codes.

See the `SemanticMapperException` class in the *API Reference for Java (Javadoc)* for more information on the constructors and error codes.

Logging

See [“Logging from IAF plug-ins in Java” on page 406](#) for information about how codec plug-ins should log error, status and debug information.

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 `ap-util.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`

The `NormalisedEvent` class represents a single normalized event. This class the most important part of the interface, and encapsulates the data and operations that can be performed on a single normalized event.

Normalized 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.

- `NormalisedEventIterator`

Several of the `NormalisedEvent` methods return an instance of the `NormalisedEventIterator` class, which provides a way to step though the name-value pairs making up the normalized event, forwards or backwards.

There is no public constructor. Iterators are created and returned only by `NormalisedEvent` methods.

■ `NormalisedEventException`

Any errors encountered by `NormalisedEvent` result in instances of `NormalisedEventException` being thrown.

See the *API Reference for Java (Javadoc)* for detailed information on these classes.

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 364](#) for more information about this sample. The section [“The String codec IAF plug-in” on page 500](#) describes how the `JStringCodec` plug-in can be used in practice.

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.

20 Plug-in Support APIs for Java

■ Logging from IAF plug-ins in Java	406
■ Using the latency framework	407

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.

Logging from IAF 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 362](#) 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`

It is recommended that you do not use the `FATAL` or `CRIT` log levels provided by the `Logger` class, which are present only for historical reasons. It is better to use `ERROR` for all error conditions regardless of how fatal they are, and `INFO` for informational messages. By default, the JMon classes log at `WARN` level. See "Setting correlator and plug-in log files and log levels in a YAML configuration file" in *Deploying and Managing Apama Applications* for information about configuring log levels in the correlator.

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 *API Reference for Java (Javadoc)* for more information about the `Logger` class.

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.

Javadoc documentation for `com.apama.util.TimestampSet` and `com.apama.util.TimestampConfig` classes is provided as part of the Apama documentation set. See the *API Reference for Java (Javadoc)*.

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.

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.

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. See the *API Reference for Java (Javadoc)* for more information on the fields of the `TimestampConfig` class.

Java latency framework API

The Java interface for the latency framework is declared in the header file `com.apama.util.TimestampSet` class.

See the `TimestampSet` class in the *API Reference for Java (Javadoc)* for detailed information on the available functions.

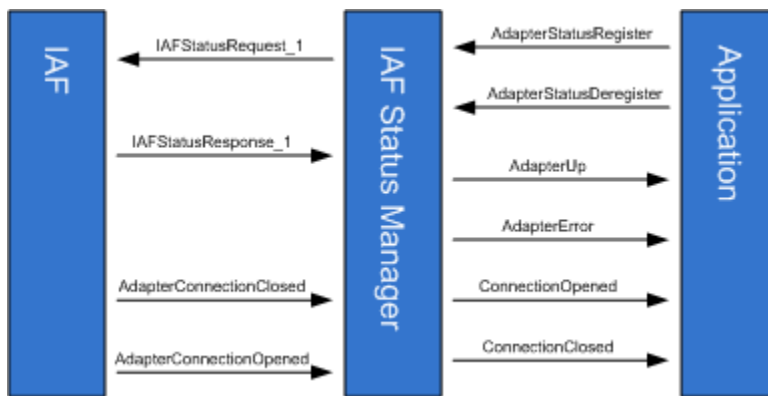
21 Monitoring Adapter Status

■ IAFStatusManager	411
■ Application interface	411
■ Returning information from the getStatus method	412
■ Connections and other custom properties	413
■ Asynchronously notifying IAFStatusManager of connection changes	414
■ StatusSupport	417
■ DataView support	420

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.

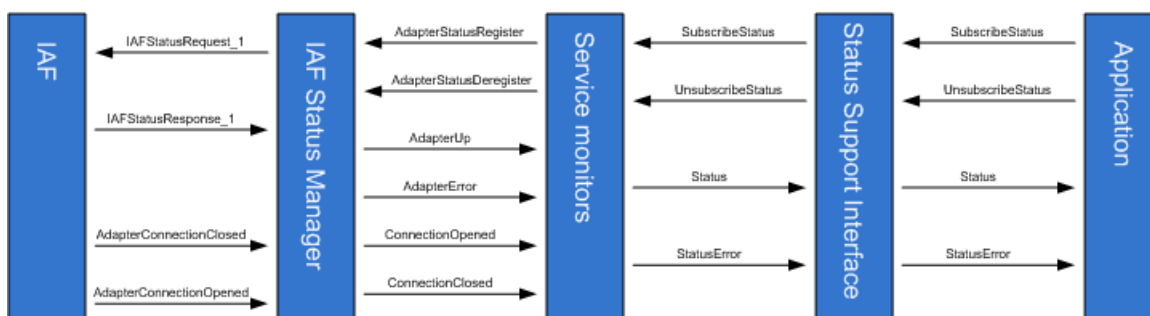
Apama provides the following two mechanisms for handling IAF Adapter status information:

- **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 411](#)

- **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 417](#).

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 listed below to register or deregister as a consumer for adapter status information.

The IAFStatusManager defines the following input events:

- The `AdapterStatusRegister` event is sent by a client that is interested in receiving status events from the specified codec and transport. The fields of this event uniquely identify a subscription.

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 below).

- The `AdapterStatusDeregister` event is sent by a client that wants to remove its subscription for status events.

See the *API Reference for EPL (ApamaDoc)* for more details on the `IAFStatusManager` and the above events.

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 413](#).

The `IAFStatusManager` defines the following output events:

- The `AdapterUp` event is used to notify registered clients that the specified adapter process is running.
- The `AdapterError` event is used to notify registered clients that there is a problem with the subscription.
- The `ConnectionOpened` event is used to notify registered clients that a connection between the adapter and the external system it communicates with has successfully been established.
- The `ConnectionClosed` event is used to notify registered clients that a connection between the adapter and the external system it communicates with has been closed.

See the *API Reference for EPL (ApamaDoc)* for more details on the `IAFStatusManager` and the above events.

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. See the *API Reference for Java (Javadoc)* for detailed information on the `ExtendedTransportStatus` and `ExtendedCodecStatus` classes.

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. See the *API Reference for C++ (Doxygen)* for detailed information on the `AP_EventTransportStatus` and `AP_EventCodecStatus` structures.

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.

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 414](#).

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:

Field name	Field value
AdapterConnectionOpenEvent or AdapterConnectionClosedEvent	No value (empty string). This will either represent a connection opened or connection closed and be translated into AdapterConnectionOpened or AdapterConnectionClosed events respectively for the IAFStatusManager to consume.
codecName	Name of codec.
transportName	Name of transport.
connectionName	No value (empty string) if there is only one connection. If there is more than one connection, this should contain CONNECTION_<id> and one event should be sent for every connection the adapter is concerned with.
connectionGeneration	Connection generation identifier. This identifies a successful connection attempt with a connectionName. 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 416](#).

Mapping AdapterConnectionClosed and AdapterConnectionOpened events

As described in [“Asynchronously notifying IAFStateManager of connection changes” on page 414](#), 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=""/>
  </mapping-rules>
</event>

```



```

    <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>

```

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) 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 in [“StatusSupport events” on page 418](#).

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 service ID 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 title case, for example, "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.

DataView support

The `IAFStatusDataViewService` provides support for publishing the status of an IAF adapter as a `DataView` item. It can be used to easily monitor the adapter status using Apama's Scenario Browser or to visualize the adapter status using an Apama dashboard. The `IAFStatusDataViewService.mon` file, which can be found in the `adapters/monitors` directory of the Apama installation, needs to be injected to make use of the `IAFStatusDataViewService`. Note that the monitor is not included in any bundle in Software AG Designer. For more information on the `DataViewService`, see "Making Application Data Available to Clients" in *Developing Apama Applications*.

The status of every adapter is published as a `DataView` item of a single `DataView` definition (`IAF_ADAPTER_STATUS`). The `IAFStatusDataViewService` is dependent on the `IAFStatusManager` and can only publish the status of adapters which support the `IAFStatusManager`.

The `IAFStatusDataViewService` uses the following events:

- The `AdapterDataViewRegister` event is used to register an adapter for publishing its status as a `DataView` item. You have to route the `AdapterDataViewRegister` event with the appropriate adapter name to start publishing its status.
- The `AdapterDataViewDeregister` event is used to de-register an adapter from publishing its status as a `DataView` item. You have to route the `AdapterDataViewDeregister` event with the appropriate adapter name to stop publishing its status.
- The `AdapterDataViewResponse` event is sent in response of the `AdapterDataViewRegister` and `AdapterDataViewDeregister` events to indicate the success or failure of the operation.

See the *API Reference for EPL (ApamaDoc)* for more details about the `IAFStatusDataViewService` and the events.

Once an adapter is registered to publish its status information as a `DataView` item, the `IAFStatusDataViewService` starts consuming `AdapterUp` and `AdapterError` events from the `IAFStatusManager`. Each registered adapter has a corresponding `DataView` item for the status. This `DataView` item is periodically updated with the current status from the `AdapterUp` and `AdapterError` events.

After an adapter has been registered for publishing its status, the adapter status can be viewed in the Scenario Browser or visualized in a dashboard:

- To view the adapter status in the Scenario Browser, open the Scenario Browser as described in "Displaying the Scenario Browser" (in *Using Apama with Software AG Designer*), and then click or expand the IAF Adapter Status node to see the current status of the adapter.

- To visualize the adapter status in a dashboard, use the `IAF_ADAPTER_STATUS` DataView as the data source. For more information, see "Attaching Dashboards to Correlator Data" in *Building and Using Apama Dashboards*.

22 Out of Band Connection Notifications

■ Mapping example	424
■ Ordering of out of band notifications	425

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 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*. Keep in mind that the `OutOfBandConnections` event, which is mentioned in that topic, is used to get only the senders and receivers that are connected to correlator.

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 349](#).

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

23 The Event Payload

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 (for example, 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.

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 343](#), one of the attributes of the event-mapping element `<event>` is `copyUnmappedToDictionaryPayload`.

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 this attribute is set to `false`, any unmapped fields are discarded.

If this attribute 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.

Accessing the payload in the correlator

Using `copyUnmappedToDictionaryPayload` puts all the payload fields in a standard EPL dictionary.

V Standard IAF Plug-ins

24	The Database Connector IAF Adapter (ADBC)	433
25	The File IAF Adapter (JMultiFileTransport)	483
26	The Basic File IAF Adapter (FileTransport/JFileTransport)	497
27	Codec IAF Plug-ins	499

24 The Database Connector IAF Adapter (ADBC)

■ Overview of using ADBC	434
■ Registering your ODBC database DSN on Windows	435
■ Adding an ADBC adapter to an Apama project	436
■ Configuring the Apama database connector	437
■ The ADBCHelper application programming interface	444
■ The ADBC Event application programming interface	456
■ The Visual Event Mapper	478
■ Playback	480
■ Sample applications	481
■ Format of events in .sim files	481

The Apama Database Connector (ADBC) is an 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 Apama with Software AG Designer*.

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 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 481](#).

Apama provides JDBC database drivers for the following Apama-certified databases:

- Microsoft SQL Server
- Oracle

Apama does not provide any ODBC drivers. You need to use your own ODBC drivers to use ODBC. Any bugs in driver need to be directly resolved with the driver vendor. Use of JDBC rather than ODBC is recommended.

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 project in Software AG Designer, the adapter instance is automatically configured with appropriate JDBC settings.

The Apama JDBC drivers are licensed to be used with any Apama component.

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 444](#).

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 456](#).

Apama's ADBC Adapter editor in Software AG Designer includes an Event Mapping tab that lets you quickly specify the mapping rules for storing events in existing database tables. Software AG Designer 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.

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.

➤ To register your ODBC database DSN

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

Adding an ADBC adapter to an Apama project

When you add an ADBC adapter to an Apama project in Software AG Designer, all the resources associated with the adapter such as service monitors and configuration files are automatically included.

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 (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:
 1. From the **File** menu, choose **New > Apama Project**.
 2. Give the project a name, and click **Next**.
 - If you are adding an ADBC adapter to an existing project:
 1. In the Project Explorer view, right-click the **Connectivity and Adapters** node and select **Add Connectivity and Adapters**.
 2. Enter a new name for the adapter instance or accept the default instance name. Software AG Designer prevents you from using a name that is already in use.
2. Select the ADBC adapter bundle that is appropriate to the kind of data source your application will use. Click **OK**.

When you add a data source-specific adapter, the **ADBC Adapter (Common Apama database connector adapter)** bundle will be added to the project automatically.

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 Software AG Designer, 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 Software AG Designer, 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 441](#).

Configuring an ADBC adapter

In Software AG Designer, an adapter's configuration file is opened in Apama's 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 (**ODBC Adapter**, **JDBC Adapter**, or **Sim File Adapter**).
2. Double-click the entry for the adapter instance you want to configure. The configuration file opens in the adapter editor.

The **Settings** tab of the editor's graphical display presents configuration information. For an instance of the ADBC-JDBC adapter, the following tabs are shown:

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

For an instance of the ADBC-ODBC adapter, the display is similar but with fewer items in the above sections. For an instance of the ADBC-Sim file 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 the description of the **Variables** section below.
- **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_NAME` variable; for more information on this variable, see the description of the **Variables** section below.
- **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; for more information on this variable, see the description of the **Variables** section below.
- **Store batch size** — 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.

- **Store commit interval** — 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.
- **Auto commit** — This controls the use of the ODBC/JDBC driver autocommit flag. The default value is `false`.
- **Login timeout** — This is a JDBC-specific property that allows you to change the default login timeout when `com.apama.database.Connection.OpenDatabase` or `com.apama.database.Connection.OpenDatabaseShared` are called.
- **Query timeout** — This is a JDBC-specific property that allows you to set the timeout for queries. The default value is 0. Keep in mind that different database vendors define a query timeout differently; see the documentation for these databases for more information.

Note:

For more information on the interaction of the **Auto commit**, **Store commit interval** and **Store batch size** properties, see [“Committing database changes” on page 450](#).

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.
- **Log commands** — 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.
- **Log queries** — This property behaves identically to the **Log commands** property except that it specifies whether or not to log the start and completion of queries.
- **Flow control low water** — This defines a threshold for the number of query responses not acknowledged by the ADBC flow control monitor before a query paused by **Flow control high water** 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.
- **Flow control high water** — 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 **Flow control low water** 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.
- **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 project.

You can add or edit values of the following additional advanced properties by clicking the **XML Source** 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 442](#).
- `CloseDatabaseIfDisconnected` — This controls automatic closing of databases whose connection is found to be invalid. See [“Configuring ADBC Automatic Database Close” on page 442](#).

- **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:

- **ADAPTER_CONFIG_DIR** — JDBC and ODBC adapters only. This specifies the directory where the adapter's configuration files are located. This is automatically set by default.
- **ADAPTER_INSTANCE_ID** — This refers to the instance ID that is given to the adapter by Software AG Designer.
- **ADAPTER_INSTANCE_NAME** — This refers to the adapter name that is displayed in Software AG Designer.
- **ADAPTERS_DIR** — JDBC and ODBC adapters only. Specifies the directory where the adapter will look for the adapter files. By default, this is the Apama installation's `adapters` directory.
- **ADAPTERS_JAR_DIR** — JDBC adapter only. This specifies the directory where the Apama adapter jar files are located. By default, this is the Apama installation's `adapters\lib` directory.
- **APAMA_HOME** — This refers to the location of the Apama installation directory.
- **APAMA_MSG_ENABLED** — Apama elements can be enabled to define how IAF connects to the Apama correlator(s). The `enabled` attribute has valid values as `true` or `false`. This is based on the launch configuration.
- **BUNDLE_DIR** — Sim file adapter only. This specifies the directory where the adapter bundle is located.
- **BUNDLE_DISPLAY_NAME** — This refers to the adapter bundle name that is displayed in Software AG Designer.
- **BUNDLE_INSTANCES** — Sim file adapter only. This refers to the bundle instances files that Software AG Designer has copied to the project.
- **CORRELATOR_HOST** — This specifies the name of the host machine where the project's default correlator runs. This is automatically set by default.
- **CORRELATOR_PORT** — This specifies the port used by the correlator. This is automatically set by default.
- **DATABASE_LOCATION** — This specifies the location of the database for use with the ADBC Discovery API, for example, `jdbc:mysql://localhost/trades`.

- `JDBC_DRIVER_JARFILE` — JDBC adapter only. This specifies the name of the data source driver file, for example, a MySQL version X driver path might be specified at `C:/Program Files/MySQL/mysql-connector-java-X/mysql-connector-java-X.jar`.
- `JDBC_DRIVER_NAME` — JDBC adapter only. This specifies the class name of the driver, such as `com.mysql.jdbc.Driver`.
- `MAPPING_INSTANCE_FILE` — JDBC and ODBC adapters only. This refers to the file name for the adapter's mapping configurations when configured through Software AG Designer.
- `UM_CONFIGURATION_FILE` — This refers to the configuration file that is used when configuring the IAF to communicate via Universal Messaging. See also "Configuring IAF adapters to use Universal Messaging" in *Connecting Apama Applications to External Components*.

Note:

Use of Universal Messaging from the IAF is deprecated and will be removed in a future release.

- `UM_MSG_ENABLED` — This refers to the flag that indicates whether the IAF is configured to use Universal Messaging.

Note:

Use of Universal Messaging from the IAF is deprecated and will be removed in a future release.

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 478](#).

Manually editing a configuration file

If you are not using Software AG Designer to develop your Apama 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 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 Software AG Designer, you can access this property by using the **XML Source** tab in the adapter editor.

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) .

Service monitors

If your Apama application uses ADBC, you need to inject several required service monitors. In Software AG Designer, 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 437](#). If you are not using Software AG Designer to develop your application, you need to manually inject the following required service monitors in the order they are listed:

- `ADBCAdapterEvents.mon` — Provides definitions for all events sent to or from the ADBC Adapter.
- `ADBCEvents.mon` — Provides the public API for ADBC, implemented as actions on the following events:
 - `Discovery` — This event types defines the actions for discovering ADBC resources. It is used to find the available data sources (ODBC, JDBC, Sim, etc.) and the default databases and query templates configured for those data sources.
 - `Connection` — This event type defines actions for performing all operations on a database except those involving queries
 - `Query` — This event type defines actions for performing queries on a database.
- `ADBCAdapterService.mon` — Provides actions for the following:
 - Forwarding database request events to the adapter.
 - Forwarding database response events to the ADBC Service API layer.
 - Provides support for using the ADBC event API from contexts other than the main context (see also `setPrespawnContext` in [“Setting context” on page 454](#)).
- `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

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 375](#) and [“Java Codec Plug-in Development” on page 397](#).

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 456](#).

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 acknowledgment. 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 described below.

➤ To use the ADBCHelper API

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

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

2. Call the `setAdapterInstanceName` action of `DBUtil` to identify the adapter instance, for example:

```
db.setAdapterInstanceName("JDBCAdapter1")
```

For more information, see [“Specifying the adapter instance” on page 448](#).

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 `isOpen` action of `DBUtil`. 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 448](#).
4. Call one of the `DBUtil` open actions to open the database. For more information on open actions, see [“Opening databases” on page 446](#).
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 449](#).
 - 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 450](#).
 - 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 450](#).
 For more information on rollback actions, see [“Performing rollback operations” on page 451](#).
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 451](#).
7. For action calls that request an acknowledgment, your application needs to do the following:
 - a. Create an instance of the `com.apama.database.DBAcknowledge` event.
 Note, if your code contains the following line, you do not need to use the fully qualified name for `DBAcknowledge`.

```
using com.apama.database.DBAcknowledge;
```
 - b. Create a listener for the `DBAcknowledge` event that indicates when the `DBUtil` event action call is complete.
 For more information, see [“Handling acknowledgments” on page 452](#).
8. Create an action that handles errors that could occur during execution of a `DBUtil` event action call. For more information, see [“Handling errors” on page 453](#).
9. Call the `DBUtil` event's close action to close the database. For information, see [“Closing databases” on page 454](#).

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 453](#) 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 437](#). 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.

For more information on the interaction of the `AutoCommit`, `StoreCommitInterval` and `StoreBatchSize` properties, see [“Committing database changes” on page 450](#).

- `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 open action.

```
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 )
```

Specifying the adapter instance

Use the `setAdapterInstanceName()` action of `DBUtil` to specify an adapter instance.

Note:

For adapters created in versions of Apama prior to 9.12, the call to `setAdapterInstanceName()` is optional if the adapter was added as `INSTANCE_1`.

➤ 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);  
    // ...
```

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() {  
    db.openQuickODBC(dbUrl,""," ",handleError );  
}
```


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 451](#).

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.

Committing database changes

There are several approaches to defining when ADBC should commit changes. Although it is possible to use more than one of these, it is usually best to stick to just one:

- **EPL-controlled committing** by calling `DBUtil.doSQLCommit` (or `doSQLCommitAck`). This is the recommended approach for applications that update the database, as it gives maximum control of transactions to the EPL application. Do not use this when `autoCommit` is enabled.
- **Automatic committing per SQL statement** by setting `autoCommit=true`, which configures the underlying database driver to automatically perform a commit after the completion of each SQL statement (by default, this is disabled). This is useful for simple applications that only perform read-only queries.
- **Timed committing** by setting `commitStoreInterval`, which configures ADBC to batch up SQL statements over the specified time window into a single commit (the default is 0 which means that timed committing is disabled). Do not use this when `autoCommit` is enabled.

Note:

Whether you are using queries to get data from the database or to put data in, it is essential to use at least one of these commit mechanisms, as queries will not complete until a commit is issued.

There is also a property called `StoreBatchSize` which allows multiple SQL statements to be batched together into a single call to the database, in order to reduce communication overhead. The batching of statements does not itself result in any extra commits, but does still have some impact on which statements are committed together:

- If using EPL-controlled committing, the partial batch that has been received so far will be committed.
- If `autoCommit` is enabled, statements are not committed until the batch size is reached.
- If using timed committing, a partial batch is committed when the time expires based on what has been uncommitted to that point.

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.

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;
}
```

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's Visual Mapper in Software AG Designer, which automatically saves the mapping information in a project file. For more information, see [“Using the Visual Event Mapper” on page 478](#).

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("exampleldb","thomas","thomas123",handleError);
    db.doSQLEventQuery("SELECT * FROM network_info", NetworkInfo);
    //...
}
on all NetworkInfo := netInfo {
    // Code to do something with the returned event...
}
```

Handling acknowledgments

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

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

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

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

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

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

For example:

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

Handling errors

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

The following EPL code snippet shows a simple error handler.

```
//...
com.apama.database.DBUtil db;
//...
action onload() {
  db.openQuickODBC("exampledb","thomas","thomas123", handleError);
  //...
}

action handleError( string reason ) {
  log "DB Error: " + reason;
}
```

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_AND_RETRY_LAST_REQUEST` — Try to reconnect and leave the pending requests unchanged, retry the last request on a successful database reconnection.
 - `DO_NOT_RECONNECT` — Do not try to reconnect. This is the default reconnect policy.
- `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.

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 )
```

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 >
```

Setting context

By default the ADBCHelper API sends requests to an internal service monitor running in the main context with the EPL route statement. 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 how to send events.

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

```
setPrespawnContext( context c )
```

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.

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 (
      '02', 'FR', 101 );");
    db.doSQLCmd( "insert into NetworkInformation values (
      'Three', 'FR', 102 );");
    db.doSQLCmdAck( "insert into NetworkInformation values (
      'Orange', 'FR', 103 );", 100, false);
    on com.apama.database.DBAcknowledge(ackNum = 100) as ack {
      if ack.error.length() = 0 {
        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 ADBC Event application programming interface

The ADBC (Apama Database Connector) Event application 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`

Discovering data sources

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

➤ To discover data sources

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

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

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


and

```
action findAvailableDataSourcesFull(
    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 service ID 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.findAvailableDataSources(TIME_TO_WAIT, handleAvailableServers);
action handleAvailableServers(string error,
    sequence<com.apama.database.DataSource> results)
{
    if error.length() != 0 {
        log "Error occurred getting available data sources: " +
            error at ERROR;
    }
    else {
        if results.size() > 0 {
            // Save off first service ID found.
            // Assumes first data source has at least one db
            if getDbServiceId() = "" {
                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;
        }
    }
}
```

Discovering databases

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

> To discover databases

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 {  
        log "Error occurred getting available data sources: " +  
            error at ERROR;  
    }  
    else {
```

```

    if results.size() > 0 {

        // Save off first service ID found.
        // Assumes first data source has at least one db
        if getDbServiceId() = "" {
            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 {
        log "Error occurred getting available databases: " +
            error at ERROR;
    }
    else {
        if results.size() > 0 {
            // Save name of first db found
            if getDbName() = "" {
                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;
        }
    }
}
}

```

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 service ID, 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 437](#). 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 {
    log "Error occurred getting available databases: " +
      error at ERROR;
  }
  else {
    if results.size() > 0 {
      // Save name of first db found
      if getDbName() = "" {
        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;
    }
  }
}
```

Note:

If reusing a database connection, rather than calling `openDatabase` again, it is advised to use `reopenWithAck` instead. In cases where there are issues using the current connection, for example, the call to `closeDatabase` is not succeeding following an incident where the IAF went down, then you should call `reopenWithAck` to recover the connection.

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 {  
        log "Error closing database " + getDbName() + ": " +  
            error at ERROR;  
    }  
    else {  
        log "Database " + getDbName() + " closed." at INFO;  
    }  
}
```

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 464](#) 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 acknowledgment, 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 acknowledgments 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 465](#).

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 464](#) 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 acknowledgment, 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 acknowledgment callback string contains any errors reported as well as the returned token, an empty acknowledgment 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 acknowledgments 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 465](#).

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)
```

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.

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.

> To commit a transaction manually

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
```

and:

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

Rolling back transactions

To roll back 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.

> To roll back a transaction

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
```

and:

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

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.

➤ To run a command

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)
```

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 468](#).
- **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 471](#) and [“Stored procedures” on page 472](#).

- **Named query** — An SQL query that you write in an XML file as part of the Apama project in Software AG Designer. 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 474](#).

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. The relevant `init` action should be called each time before calling the query's `start` action.
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 469](#) 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 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.

Stopping queries

The following action cancels all outstanding queries in the queue.

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

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.

➤ To edit the `ColumnNameCase` property

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 following 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`.

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.

Using a prepared statement

Note that `PreparedQuery` events support only ODBC/JDBC data types. Vendor-specific data types are not allowed.

➤ To use a prepared statement

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

The definition for the `init()` action is:

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

The arguments for the `init()` action are:

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

For example:

```
sequence<string> inputTypes := ["INTEGER","INTEGER"];
myPreparedQuery.init (
    myConnection,
```

```
"SELECT * FROM mytable WHERE inventory > ? and inventory <?",  
inputTypes, new sequence<string>);
```

4. Call the new PreparedQuery event's create() action, passing in the name of the callback action.
5. In the callback action's code, call the Query event's initPreparedQuery() action (instead of the initQuery() action), passing in the name of the PreparedQuery event. See [“Executing standard queries” on page 468](#). As with any query, the relevant init action should be called each time before calling the query's start action.
6. Call the Query event's setInputParams() action, passing in the values to be used for the replaceable parameters. This should always be called before starting a query that is using a prepared query.

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 468](#).

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.

Using a stored procedure

Queries in Apama applications 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 468](#).
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 468](#).

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.

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 {
            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("token", handleResultEvent);
    query.start(handleQueryComplete);
}
action handleResultEvent(Query q, ResultEvent result, string token) {
    log result.toString() at INFO;
}
action handleQueryComplete(Query query, string error,
    integer eventCount, float lastEventTime) {
    if error.length() != 0 {
        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 {
            log " Last Event Time: " + lastEventTime.toString()
                at INFO;
        }
    }
}
conn.closeDatabase(handleCloseDatabase);
}
action handleCloseDatabase(Connection conn, string error) {
    if error.length() != 0 {
        log "Error closing database : " + error at ERROR;
    }
    else {
        log "Database closed." at INFO;
    }
}
}
}

```

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 have added, 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.

The Visual Event Mapper

Note:

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, Software AG Designer 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 Software AG Designer 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 Software AG Designer generates events that cause Software AG Designer 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 Visual Event Mapper

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.

➤ **To map an Apama event to a table in a database**

1. Add a new instance of the ADBC Adapter or open an existing instance and select the adapter editor's **Event Mapping** tab.
2. If you want Software AG Designer 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, all the EPL code that interacts with the database is generated: 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, mapping rules are generated that specify the associations between event fields and database columns. A monitor that listens for events of the specified type is also generated. The monitor allows the Apama application to manage when the events are written to the database.

Playback

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

For full information on the Data Player, see "Using the Data Player" in *Using Apama with Software AG Designer*.

Command line tools

When you have stored event data in a database and created queries in Software AG Designer, you can also launch a playback session using the Data Player command line tool, `adbc_management`.

The `adbc_management` tool is described in *Deploying and Managing Apama Applications*, in the section "Using the data player command-line interface".

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.

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's Data Player in Software AG Designer 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 time zone for the series. The time-zone identifiers can be any supported by Java. 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 time zone:

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

Comments in .sim files

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

25 The File IAF Adapter (JMultiFileTransport)

■ File adapter plug-ins	484
■ File adapter service monitor files	485
■ Adding the File adapter to an Apama project	485
■ Configuring the File adapter	486
■ Overview of event protocol for communication with the File adapter	487
■ Opening files for reading	488
■ Specifying file names in OpenFileForReading events	490
■ Opening comma separated values (CSV) files	491
■ Opening fixed width files	492
■ Sending the read request	493
■ Requesting data from the file	493
■ Receiving data	493
■ Opening files for writing	494
■ LineWritten event	495
■ Monitoring the File adapter	496

The File adapter 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 Apama application in Software AG Designer, when you add a standard adapter to the project, Software AG Designer 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 project.

If you are not using Software AG Designer, 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 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 topics below.

`JMultiFileTransport` is the recommended way to read/write files, the `FileTransport/JFileTransport` is just a sample for which binaries are also included. See [“The Basic File IAF Adapter \(FileTransport/JFileTransport\)” on page 497](#) for more information.

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 IAF plug-in” on page 521](#) and [“The Fixed Width codec IAF plug-in” on page 524](#).
- `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 project in Software AG Designer, these plug-ins are automatically added to the configuration file. If you are not using Software AG Designer, you can use the `File.xml.dist` template file as the basis for the configuration file. See [“Configuring the File adapter” on page 486](#) for more information about adding the necessary settings to the adapter's configuration file.

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 Software AG Designer, the project's default run configuration automatically injects them. If you are not using Software AG Designer, 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`

For detailed information on the file adapter, see the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)*.

Adding the File adapter to an Apama project

If you are developing an application with Apama in Software AG Designer, add the File adapter as described below.

➤ To add the File adapter

1. In the Project Explorer, right-click the **Connectivity and Adapters** node and select **Add Connectivity and Adapters**.
2. Select **File Adapter (File adapter for reading and writing to ASCII files)**. A default name is added to the **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. You will be prevented from using a name already in use.
3. Click **OK**.

A File adapter entry that contains the new instance is added to the project's **Connectivity and Adapters** node and the instance's configuration file is opened in Apama's adapter editor.

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 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.

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 project, a configuration file for each instance of the adapter is automatically created. Double-click the name of the adapter instance to open the configuration file in the adapter editor.

If you are using the Apama adapter editor in Software AG Designer, 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 Software AG Designer, 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 codecs and mapping rules. Normally you do not need to change any information in this file. The default channel for File adapter events is `FILE` and the default for `transportName` is `JMultiFileTransport`. See [“The IAF configuration file” on page 339](#) for more information on the contents on an adapter's configuration file.

In Software AG Designer, adapters are configured using Apama's 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 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 Software AG Designer, 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"/>`

- Set the `<classpath>` elements:

```
<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 Software AG Designer, 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 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 Software AG Designer, 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

See also the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)*.

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 sending an event to the channel specified in the adapter's configuration file, typically `FILE`, for example:

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

See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `OpenFileForReading` event.

Parameter	Description
<code>transportName</code>	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.
<code>requestId</code>	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.
<code>codec</code>	Name of the codec to use with the file. This must match one of the codecs specified in the <code>adapter-static.xml</code> 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 payload field of the <code>OpenFileForReading</code> event. The codecs needs this information to correctly interpret the data. For details about adding to the payload field, see “Opening comma separated values (CSV) files” on page 491 or “Opening fixed width files” on page 492 .
<code>filename</code>	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:

Parameter	Description
	<p><code>c:\logfiles*.log</code></p> <p><code>/user/local/jcasablancas/logfiles/*.log</code></p> <p>For more information, see “Specifying file names in OpenFileForReading events” on page 490.</p>
<code>linesInHeader</code>	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.
<code>acceptedLinePattern</code>	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.
<code>payload</code>	<p>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.</p> <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> ■ <code>boolean isLeftAligned</code> <p>Whether the data in the field is aligned to the left or not (that is, right aligned)</p> ■ <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>

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 488](#)), 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 sent directly to the adapter, but rather sent to the correlator's main context, where the adapter service monitor runs. The File adapter's service monitor is responsible for sending the events that are sent from other contexts to the File adapter, and for sending the events received from the File adapter to whichever context should process them (as specified in the `OpenFileForReadingToContext` event).

See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `OpenFileForReadingToContext` event.

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 sending openFileForReading to "FILE", wrap it in
// the OpenFileForReadingToContext event and send it to the service
// monitor in the main context.
send 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 sending to the "FILE" channel, send it to the main
    // context
    send com.apama.file.ReadLines(openFileForReading.transportName, -1,
        readHandle.sessionId, 20) to mainContext;
    ...
}
```

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. See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `EndOfFile` event.

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. See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `NewFileOpened` event.

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.

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.incrementCounter("FileTransport.requestId");

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

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

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

//send event to channel in config.
send 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 IAF plug-in” on page 521](#).

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.incrementCounter("FileTransport.requestId");

//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"] := "_";

//send event to channel in config.
send 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 IAF plug-in” on page 524](#).

Sending the read request

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

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

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

Sending 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.

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.

See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `FileHandle` and `FileError` events.

Requesting data from the file

After your application receives a `FileHandle` event, it can send a `ReadLines` event, which signals the adapter to start reading lines from the file. See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `ReadLines` event.

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.

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. See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `FileLine` event.

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. See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `CloseFile` event.

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.

Opening files for writing

To open a file for writing, send an `OpenFileForWriting` event. The definition of the `OpenFileForWriting` event is similar to the definition of the `OpenFileForReading` event. See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `OpenFileForWriting` event.

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, send the `OpenFileForWriting` event to the "FILE" channel, for example:

```
send OpenFileForWriting(...) to "FILE"
```

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 492](#).

Again, as with reading a file, the File adapter sends a `FileHandle` or `FileError` event (see [“Sending the read request” on page 493](#)), 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. See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `FileLine` event.

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 acknowledgment

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.

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 494](#)), 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 sent directly to the adapter, but rather sent to the correlator's main context, where the adapter service monitor runs. The File adapter's service monitor is responsible for sending the events that are sent from other contexts to the File adapter, and for sending the events received from the File adapter to whichever context should process them (as specified in the `OpenFileForWritingToContext` event).

See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `OpenFileForWritingToContext` event.

Using the `OpenFileForWritingToContext` event is similar to using the `OpenFileForReadingToContext` event. See [“Opening files for reading with parallel processing applications” on page 489](#) for an example use of the `OpenFileForReadingToContext` event.

LineWritten event

After the File adapter writes a line to a file, the adapter sends a `LineWritten` event. See the `com.apama.file` package in the *API Reference for EPL (ApamaDoc)* for detailed information on the `LineWritten` event.

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 send 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 send a `CloseFile` event. See also [“Opening files for reading” on page 488](#).

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.

26 The Basic File IAF Adapter (FileTransport/JFileTransport)

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 340](#).

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 364](#) for more information.

`JMultiFileTransport` is the recommended way to read/write files, the `FileTransport/JFileTransport` is just a sample for which binaries are also included. See [“The File IAF Adapter \(JMultiFileTransport\)” on page 483](#) for more information.

27 Codec IAF Plug-ins

■ The String codec IAF plug-in	500
■ The Null codec IAF plug-in	501
■ The Filter codec IAF plug-in	503
■ The XML codec IAF plug-in	507
■ The CSV codec IAF plug-in	521
■ The Fixed Width codec IAF plug-in	524

Apama provides several standard codec IAF plug-ins for your convenience, which can be used for testing or in combination with custom plug-ins. They are described in the topics below.

The compiled binaries for all the standard plug-ins are available in the `\bin` and `\lib` directories (for the C and Java versions respectively).

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 364](#).

The String codec IAF 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 340](#).

This is one of the sample plug-ins for which source code is available – see the [“IAF samples” on page 364](#) for more information.

The Null codec IAF 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.JNullCodec">
```

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 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 Filter codec IAF 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 topics below.

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 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" />
    <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" />
```

Syntax Element	Description
<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 IAF 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.

XML codec plug-in	
Received From or Sent To Transport Layer	Received From or Sent To Semantic Mapping Layer
<ul style="list-style-type: none"> ▪ Normalised event ▪ Each field is a string ▪ Each string contains XML data 	<ul style="list-style-type: none"> ▪ Normalised event ▪ Each field is a name/value pair ▪ Values are the elements, attributes, CDATA, and text ▪ Unordered

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 topics below to help you configure the XML codec.

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 510](#).

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.

```
ap-iaf-extension-api.jar
ap-util.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 341](#).

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 343](#).

Properties are described in the topics below.

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 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_JARDIR/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_JARDIR/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 516](#) and [“Examples of conversions” on page 518](#).

You can set both node order properties to `true`. For sample output when both are set to `true`, see [“Examples of conversions” on page 518](#). 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 521](#).

`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 520](#).

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 slash (/) 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 &lt;string&gt;" default="[]"/>
  <map transport="Test.root/prices/info/@type"
    apama="types" type="reference"
    referencetype="sequence &lt;string&gt;" default="[]"/>
  <map transport="Test.root/prices/info/text()"
    apama="prices" type="reference"
```



```
referencetype="sequence &lt;string&gt;" 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 IAF 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; see [“The Fixed Width codec IAF plug-in” on page 524](#)) can be used with the standard Apama File adapter to read data from files and to write data to files.

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 [“Multiple configurations and the CSV codec”](#) on page 522.
- 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.

Optionally, you can also set the `excelCompatible` property in the IAF configuration file. By default, this is set to `false`. If set to `true`, Excel compatibility mode is enabled, and double quotes are then used to match the behavior of Excel. The separator property is still required when using the `excelCompatible` property. For example:

```
<property name="separator" value="," />
<property name="excelCompatible" value="true" />
```

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 pairs stored as strings that will be parsed in the codec:

Key	Value
separator	The character that is to be used as the separator character, for example, a comma (,) or semicolon (;).

Key	Value
<code>excelCompatible</code>	Optional. If set to <code>true</code> , Excel compatibility mode is enabled. Double quotes are then used to match the behavior of Excel. Default: <code>false</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 IAF 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 IAF plug-in” on page 521](#).

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:

```
TSC0--L---392.25---400.25---382.25---
```

The following is an example of a right-aligned line:

```
--TSC0---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 [“Multiple configurations and the Fixed Width codec” on page 525](#).
- 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 525](#)), 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.

VI Developing Custom Clients

28	The Client Software Development Kits	531
29	Engine Management API	537
30	Engine Client API	541
31	Event Service API	545
32	Scenario Service API	547

28 The Client Software Development Kits

■ The client libraries	532
■ Working with event objects	534
■ Logging	534
■ Exception handling and thread safety	534

Apama applications that are to run within the correlator can either be built natively in the Apama Event Processing Language (EPL) or in JMon.

Although Apama includes a suite of tools to allow EPL code to be submitted to the correlator interactively, as well as submit events from text files, it is often necessary to go further and integrate the 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 correlator (like market data and order management).

In environments that require the 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 319](#).

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

- **Engine Management API.** Low-level base API on which other APIs are built. It provides facilities to inject/delete EPL, send/receive events, inspect and monitor engines. See [“Engine Management API” on page 537](#) for detailed information.

Note:

In most cases, we recommend using one of the higher-level APIs listed below in preference to the Engine Management API.

- **Engine Client API.** More powerful API built on top of the Engine Management API. It provides all the functionality provided by the Engine Management API along with functionality such as auto-reconnection or listeners for property changes. See [“Engine Client API” on page 541](#) for detailed information.
- **Event Service API.** More powerful API focused around sending and receiving events to and from channels. It provides synchronous or asynchronous pseudo-RPC mechanisms. See [“Event Service API” on page 545](#) for detailed information.
- **Scenario Service API.** Provides an external interface to DataViews and queries. See [“Scenario Service API” on page 547](#) for detailed information.

The Engine Management API is available for C++, Java and .NET. The other APIs are available only for Java and .NET.

The client libraries

The client libraries can be found in the following locations of your Apama installation:

- For C++, in the `lib` directory: `libapclient.so` (`-lapclient`) on UNIX, or `apclient.lib` on Windows.
- For Java, in the `lib` directory: `ap-client.jar`.

- For .NET, in the `bin` directory: `apclientdotnet.dll` (and its dependency `apclient.dll`).

Using the C++ client library

To program against the C++ SDK, you must use the definitions from the `engine_client_cpp.hpp` header file, which is located in the `include` directory of your 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++ 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/>.

To configure the build for an Apama C++ client:

- On UNIX, 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++ > General > Additional Include Directories**.

Also, link against `apclient.lib`. To do this in Visual Studio, select your project and then select **Project Properties > Linker > Input > Additional Dependencies** and add `apclient.lib`.

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

Using the Java client library

Add the `ap-client.jar` library to your classpath when compiling and running.

Using the .NET client library

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

To run an application using the wrapper:

1. Copy the following libraries from the `bin` directory of your Apama installation into the directory that contains your compiled .NET assembly:
 - `apclient.dll`
 - `apclientdotnet.dll`
 - `log4net.dll`
2. Ensure that the `bin` directory of your Apama installation is in the `PATH` environment variable.

Working with event objects

To create event objects to use with the Client Software Development Kits (and also to delete them), use the following:

- For C++, use `com::apama::event::createEvent` and related functions.
- For Java, use `com.apama.event.parser.EventType` and its associated classes.
- For .NET, use `Apama.Event.Parser.EventType` and its associated classes.

Logging

Logging in C++

The C++ API can output extensive information. This information can be useful in diagnosing connectivity issues or problems that you may encounter when writing the software that interfaces with the engine. As an author of a C++ client, you need not bother with the standard logging unless you 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` (that is, 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, the C++ API provides various functions which can be found in the header file (`.hpp`). These functions generally have the term “log” in their names.

Logging in Java and .NET

The logging facilities in Java and .NET are significantly more powerful than in C++. The Java API makes use of Log4j, a publicly available logging library for Java. For your convenience, Apama provides a wrapper class that abstracts the logging capabilities provided, and it is this interface that is used by the Client API for Java. See the Javadoc of the `com.apama.util.Logger` class for more information about the logging facility.

The .NET API makes use of log4net, a publicly available logging library for .NET. For your convenience, Apama provides a wrapper class that abstracts the logging capabilities provided, and it is this interface that is used by the Client API for .NET. See the documentation of the `Apama.Util.Logger` class for more information about the logging facility.

Exception handling and thread safety

Exception handling

Several of the methods and functions of the API can throw exceptions if they fail or encounter exceptional circumstances. All of these are of the following type:

- `com::apama::EngineException` in C++
- `com.apama.EngineException` in Java
- `Apama.EngineException` in .NET

An exception contains a text message indicating the nature of the problem encountered.

Thread safety

All client APIs are thread-safe, unless otherwise specified, in the sense that you can call API methods from any thread. Several background threads are created during the usage of a client API. Events received from a correlator will be handled in one of background threads, so you cannot assume that events will be delivered to you on any particular thread.

29 Engine Management API

The Apama Engine Management API is available in the following languages: C++, Java and .NET.

The Engine Management API provides capabilities to interact with an Apama engine, typically a correlator. It provides the ability to send and receive events, inject and delete EPL entities (such as EPL, JMon and CDP files), connect Apama engines together, inspect what was injected into the correlator and monitor the status of the correlator.

Note:

We recommend that where possible Java and .NET developers should use the Engine Client API instead of the more basic low-level Engine Management API. Also consider using the higher-level Event Service API for clients that primarily involve sending and receiving events.

For a full reference of the Engine Management API in the different languages, refer to the following in your Apama installation directory:

- For C++, refer to the header file `engine_client_cpp.hpp` in the `include` directory.
- For Java, refer to the *API Reference for Java (Javadoc)*.
- For .NET, refer to the *API Reference for .NET*.

For examples of using the Engine Management API, see the following directories of your Apama installation:

- `samples\engine_client\cpp`
- `samples\engine_client\java\EngineManagement`
- `samples\engine_client\dotnet\EngineManagement`

Differences between the different API languages

The APIs for C++, Java and .NET are packaged through a set of classes that model a set of entities.

The C++ API differs slightly from the Java and .NET APIs in the way it supports construction and destruction of objects. While the C++ API provides a set of static library methods that must be called to create and delete objects of the main classes, the Java and .NET APIs either provide factory classes or else have no restrictions on directly constructing objects. The C++ API generally has to manually delete the objects (structure instances) created using the methods provided. The methods

to create objects (structure instances) in the C++ API generally have the term “create” in their names, and methods to delete objects (structure instances) have the term “delete” in their names.

Note:

In C++ applications, any strings passed by the application to the 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, you have to use the methods provided by the C++ Engine Management API for the conversion to/from UTF-8 encoding. The methods for UTF-8 conversion generally have the term “UTF8” in their names.

The APIs for Java and .NET are almost identical except for the naming/capitalization of classes, methods and packages.

Using the Engine Management API

Before using the API for C++ or .NET Engine Management, the API must be initialized exactly once:

- The C++ API can be initialized by calling the `com::apama::engine::engineInit()` function.
- The .NET API can be initialized by calling the `Apama.Engine.Api.EngineInit()` method.

The Java API does not need to be initialized before using it.

The key object of the API is the `EngineManagement` object. The `EngineManagement` object connects to the engine and allows clients to interact with it. In each language, the API provides a mechanism to create an `EngineManagement` object:

- In the C++ API, the `EngineManagement` object is an instance of the `com::apama::engine::EngineManagement` class.
- In the Java API, it is an instance of the `com.apama.engine.EngineManagement` interface.
- In the .NET API, it is an instance of the `Apama.Engine.EngineManagement` interface.

The following method needs to be called to get an `EngineManagement` object:

- For the C++ client, you have to call `com::apama::engine::connectToEngine()`.
- For the Java client, you have to call the static `connectToEngine()` method of the `com.apama.engine.EngineManagementFactory` class.
- For the .NET client, you have to call the static `ConnectToEngine()` method of the `Apama.Engine.Api` class.

Once the `EngineManagement` object is created, it can be used to send/receive events, inject/delete EPL entities, monitor and inspect engines, and connect two engines together.

If an `EngineManagement` object is no longer needed, it should be disconnected. The methods for disconnecting generally have the term “disconnect” in their names.

Once done with the Engine Management API, it should be shut down if using the API for C++ or .NET. The API for Java does not need to be shut down. The methods for shutting down the API generally have the term “shutdown”, “close” or “dispose” in their names.

Sending events

Clients wishing to use the `EngineManagement` object to send events into the engine should use various methods provided by it. The methods to send events generally have the term “send” in their names. The “send” methods/functions require an event object as input.

Receiving events

Clients wishing to use the `EngineManagement` object to receive events should do so by adding one or more consumers with channels from which it wants to receive events. A consumer with multiple subscriptions to the same channel will receive only a single copy of each event emitted to the subscribed channel. The methods to add consumers generally have the term “consumer” in their names.

- In the C++ API, consumers are instances of the `com::apama::event::EventConsumer` class.
- In the Java API, consumers are instances of the `com.apama.event.EventConsumer` interface.
- In the .NET API, consumers are instances of the `Apama.Event.EventConsumer` class.

The methods/functions to add consumers return an `EventSupplier` object which acts as the unique interface between the correlator and that particular `EventConsumer` instance. The supplier should be disconnected and disposed before the consumer is disconnect and disposed.

If a consumer needs to be notified when it is disconnected, `DisconnectableEventConsumer` should be used when adding a consumer. The `disconnect()` method of the consumer will be called when the connection to the engine is lost.

Note:

If you need more advanced send/receive functionality, you should also consider using the higher-level `EventService` API (for Java and .NET) rather than the Engine Management API.

Injecting and deleting EPL

The `EngineManagement` object provides the capability to gather information from a remote correlator about the set of monitors and event types that it is currently working with. This information can be retrieved by calling `inspectEngine()` or a similar method on the `EngineManagement` object.

The `inspectEngine` method/function returns the `EngineInfo` object which encapsulates various information about the engine such as the number of monitors, the number of event types, or the number of contexts.

Status monitoring

The `EngineManagement` object provides the capability to get status information from the engine. The status information can be retrieved by calling `getStatus()` or a similar method on the `EngineManagement` object.

The `getStatus()` function returns an `EngineStatus` object which encapsulates various status information about the engine such as uptime, the number of consumers, or input and output queue

sizes. For more information, see "List of correlator status statistics" in *Deploying and Managing Apama Applications*.

Connecting correlators

The `EngineManagement` object provides the capability to connect one engine to another engine so that it receives events emitted on the specified channels. The methods to attach/detach two engines generally have the term “attach” or “detach” in their names.

30 Engine Client API

The Apama Engine Client API is available in the following languages: Java and .NET.

The Engine Client API provides capabilities to interact with an Apama engine, typically a correlator. It provides the ability to send and receive events, inject and delete EPL entities (such as EPL, JMon and CDP files), connect Apama engines together, inspect what was injected into the correlator and monitor the status of the correlator.

Note:

We recommend that where possible Java and .NET developers should use the Engine Client API instead of the more basic low-level Engine Management API. Also consider using the higher-level Event Service API for clients that primarily involve sending and receiving events.

For a full reference of the Engine Client API in the different languages, refer to the following:

- *API Reference for Java (Javadoc)*
- *API Reference for .NET*

For examples of using the Engine Client API, see the following directories of your Apama installation:

- `samples\engine_client\java\EngineClient`
- `samples\engine_client\dotnet\EngineClient`

Using the Engine Client API

An `EngineClient` instance can be acquired by calling the `createEngineClient` method of the `EngineClientFactory` class. The factory class is the following:

- For Java, it is `com.apama.engine.beans.EngineClientFactory`.
- For .NET, it is `Apama.Engine.Client.EngineClientFactory`.

The host and port of the target engine can be provided when creating the `EngineClient` instance or later using the `setHost(string)` and `setPort(int)` methods.

Once the `EngineClient` instance has been configured as needed, a connection can be established either synchronously using `connectNow()` (which throws an exception on error) or asynchronously using `connectInBackground()` (which automatically retries until a connection is established; see `setConnectionPollingInterval(int)` and `setReconnectPeriod(long)`). Changes to the values of

the host or port will cause the bean to attempt to re-connect to the correlator running on the new host/port.

Once the client connected successfully, a background thread will keep pinging it to ensure that the connection is still up (see the `setConnectionPollingInterval(int)` method for Java and the `ConnectionPollingInterval` property for .NET). If the entire client connection is lost, the following properties will change to false to notify the user of the `EngineClient`:

- `PROPERTY_BEAN_CONNECTED` and `PROPERTY_ALL_CONSUMERS_CONNECTED` for Java.
- `PropertyConnected` and `PropertyAllConsumersConnected` for .NET.

The client will continue to try to reconnect in the background, with the retry interval that can be configured using the `setConnectionPollingInterval(int)` method for Java or the `ConnectionPollingInterval` property for .NET.

If the client itself remains connected but its named consumers are disconnected by the engine (due to being slow if slow consumer disconnection is enabled), then it is the caller's responsibility to force the reconnection of the consumers by disconnecting and reconnecting the client (although applications with this requirement may be better off using the `EventService` API instead which does this automatically; see [“Event Service API” on page 545](#) for more information). To prevent reconnecting slow consumers too soon after a disconnection, a minimum time between the reconnection attempts can be configured using the `setReconnectPeriod(long)` method for Java or the `SetReconnectPeriod` property for .NET.

The `EngineClient` object can be disconnected using `disconnect()` whenever required.

If the `EngineClient` object is no longer needed, a final cleanup should be performed by calling `close()` for Java or `Dispose()` for .NET, which will disconnect it and also ensure that any background threads started by the client have been terminated.

Sending events

Clients wishing to use the `EngineClient` object to send events into the engine should use various methods provided by it. The methods to send events generally have the term “send” in their names.

Receiving events

Clients wishing to use the `EngineClient` object to receive events should do so by adding one or more named consumers, each of which has its own independent event listener and list of channels to receive from. The named consumer methods generally have the term “consumer” in their names, and are defined by `com.apama.engine.beans.interfaces.ReceiveConsumerOperationsInterface` for Java and `Apama.Engine.Client.IMessagingClient` for .NET.

Note:

If you need more advanced send/receive functionality, you should also consider using the higher level `EventService` API (for Java and .NET) rather than the `Engine Client` API.

Injecting and deleting EPL

The Engine Client API provides the capability to inject and delete EPL entities. The methods to inject normally have the term “inject” in their names. The methods to delete normally have the term “delete” or “kill” in their names.

A normal deletion works only if the EPL element being deleted is not referenced by any other element. For example, you cannot delete an event type that is used by any monitors.

The Engine Client API also provides the capability to force the deletion, which 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 the C events, the operation also deletes monitor A and thus also the listener for B events.

Inspecting the correlator

The Engine Client API provides the capability to gather information from a remote correlator about the set of monitors and event types that it is currently working with. This information can be retrieved by calling the `getRemoteEngineInfo()` method on the `EngineClient` object.

The Engine Client API also provides the capability to use a background thread to periodically collect engine information and make available the last known collected information. The polling interval can be configured using the `setInspectPollingInterval(integer)` method for Java or the `InspectPollingInterval` property for .NET. The last collected information can be retrieved with the `getEngineInfo()` method.

Status monitoring

The Engine Client API provides the capability to get status information from the remote correlator. The status information can be retrieved by calling the `getRemoteStatus()` method on the `EngineClient` object.

The Engine Client API also provides the capability to use a background thread to periodically collect status information and make available the last known collected status. The polling interval can be configured using the `setStatusPollingInterval(integer)` method for Java or the `StatusPollingInterval` property for .NET. The last collected status can be retrieved with the `getStatus()` method for Java or the `Status` property for .NET.

The Engine Client API also provides the capability to get user-defined status information from a client EPL application. This information can be retrieved using actions such as `getUserString()` on the `EngineStatus` object. For more information on these actions, see the corresponding section in “Using the Management interface” in *Developing Apama Applications*.

Connecting two engines

The Engine Client API provides the capability to connect one engine to another engine so that it receives events emitted on the specified channels. The methods to connect two engines generally have “attachAsConsumerOfEngine” in their names.

Thread-safety

The Engine Client API provides locking so that concurrent use by multiple threads is safe.

To avoid deadlock problems, clients should be careful not to call methods on the object from within synchronous event listeners. Instead, if this is required, the consumer should be added as asynchronous (see the `addConsumer` method), or the `EventService` API should be used instead. However, it is safe to call `EngineClient` methods (except `close` or `Dispose`) from within property change listeners, which are asynchronous by default.

31 Event Service API

The Apama Event Service API is layered on top of the Engine Client API which is described in [“Engine Client API” on page 541](#).

The Event Service API allows client applications to focus on events and channels.

For a full reference of the Event Service API in the different languages, refer to the following:

- *API Reference for Java (Javadoc)*
- *API Reference for .NET*

For examples of using the Event Service API, see the following directories of your Apama installation:

- `samples\engine_client\java\EventService`
- `samples\engine_client\dotnet\EventService`

Using the Event Service API

An `EventService` instance can be created by calling the `createEventService()` method of the `EventServiceFactory` class. The factory class is the following:

- For Java, it is `com.apama.services.event.EventServiceFactory`.
- For .NET, it is `Apama.Services.Event.EventServiceFactory`.

The host and port of the target engine is provided when creating the `EventService` instance.

Once the `EventService` instance has been created, it automatically takes care of connecting to the correlator in the background and of reconnecting when disconnected.

If the `EventService` is no longer needed, a cleanup should be performed by calling `close()` for Java or `Dispose()` for .NET, which will disconnect it and also ensure that any started background threads have been terminated.

Sending events

Clients wishing to use the `EventService` to send events into the correlator should use the `send` methods provided by it. The methods to send events generally have the term “send” in their names.

Receiving events

To use the Event Service API, clients first need to create one or more `EventServiceChannel` objects to receive events from one or more channels. The methods to add/remove channels have the term “channel” in their names. Once an `EventServiceChannel` object is created, it can be used to receive events from the correlator. To receive events, clients need to add one or more event listeners on an `EventServiceChannel` object to receive either events of certain event types or all events from the channel.

The Event Service API also provides an advanced mechanism to emulate request-response functionality. The `EventServiceChannel` object provides both synchronous and asynchronous mechanisms for request-response. With the synchronous mechanism, the client sends an event to the correlator and waits for the matching response event. With the asynchronous mechanism, the client sends an event to the correlator, and callback is invoked when a matching response event is received. The methods to emulate request-response functionality generally have “`RequestResponse`” in their names.

32 Scenario Service API

Scenarios are a simple abstraction for interacting with an Apama application using create/read/update/delete (CRUD) operations.

Each scenario is defined by a unique identifier and display name, and a list of named input and/or output parameters and their types. Multiple instances of each scenario can be created, each with their own input parameter values. Scenario instances can subsequently be edited or deleted. Instances can produce a series of updates as their output parameter values change, and also have an “owner” and a “state” which can be RUNNING, ENDED or FAILED.

Apama queries use the scenario abstraction to support creating, editing and deleting instances of each query, but do not generate any outputs. Scenarios that provide a read-only view of data - generating outputs but not allowing external clients to create, edit or delete instances - are called DataViews. DataViews can be used to expose data from many parts of an Apama application, including from MemoryStore tables (see "Exposing in-memory or persistent data as DataViews" in *Developing Apama Applications*), and from EPL applications that use Apama's DataView EPL API, which is also known as the DataView Service (see "Making Application Data Available to Clients" in *Developing Apama Applications*).

Apama provides a Scenario Service API in Java and .NET for working with scenarios programmatically in external clients such as custom user interfaces. For manually interacting with scenarios, Apama provides a Scenario Browser view in Software AG Designer (see also "Using the Scenario Browser view" in *Using Apama with Software AG Designer*) and a standalone tool. The Apama Scenario Service API is layered on top of the EventService API which is described in [“Event Service API” on page 545](#).

If you have scenarios that update frequently, you might need to reduce the frequency of update events sent by the correlator. See "Controlling the update frequency" in *Building and Using Apama Dashboards*.

For a full reference of the Scenario Service API in the different languages, refer to the following:

- *API Reference for Java (Javadoc)*
- *API Reference for .NET*

For examples of using the Scenario Service API, see the following directories of your Apama installation:

- `samples\engine_client\java\ScenarioService`
- `samples\engine_client\dotnet\ScenarioService`

The key elements

The Scenario Service API mainly consists of the following interfaces:

- `IScenarioService`

For Java, this is `com.apama.services.scenario.IScenarioService`.

For .NET, this is `Apama.Services.Scenario.IScenarioService`.

- `IScenarioDefinition`

For Java, this is `com.apama.services.scenario.IScenarioDefinition`.

For .NET, this is `Apama.Services.Scenario.IScenarioDefinition`.

- `IScenarioInstance`

For Java, this is `com.apama.services.scenario.IScenarioInstance`.

For .NET, this is `Apama.Services.Scenario.IScenarioInstance`.

ScenarioService

The `IScenarioService` interface is used to establish communication with the correlator and to provide access to the scenario definitions in the correlator. The `ScenarioServiceFactory` class is used to create the `ScenarioService` object that implements the interface:

- For Java, the factory class is `com.apama.services.scenario.ScenarioServiceFactory`.

- For .NET, the factory class is `Apama.Services.Scenario.ScenarioServiceFactory`.

The API also provides a helper `ScenarioServiceConfig` class that is used to build a properties map used by the `ScenarioServiceFactory` when creating a new `ScenarioService` object:

- For Java, the helper class is `com.apama.services.scenario.ScenarioServiceConfig`.

- For .NET, the helper class is `Apama.Services.Scenario.ScenarioServiceConfig`.

The `ScenarioService` object provides methods to get the IDs/names of all known scenarios (not instances) in the correlator. It also provides methods to get the `ScenarioDefinition` instances for all known scenarios (not instances) or for specific scenarios.

The `ScenarioService` object provides the capability to register listeners to get notified of all scenarios in the correlator as they are discovered. The listener can be passed when creating a `ScenarioService` object or it can be manually added later. If the listener is manually added after the `ScenarioService` object has been created, then the application must manually call methods to discover any scenarios that the service discovered before the application listener was added. Listeners are also notified for other properties. Listeners can be added which get notified only when some specific property is changed. See the Javadoc or .NET documentation for a full list of the supported properties.

If the `ScenarioService` is no longer needed, a cleanup should be performed by calling `close()` for Java or `Dispose()` for .NET, which will disconnect it and also ensure that any started background threads have been terminated.

ScenarioDefinition

The `IScenarioDefinition` interface is used to represent a scenario (not an instance) that is running in the correlator. `ScenarioDefinition` instances are obtained by calling the appropriate methods on the `ScenarioService`.

The `ScenarioDefinition` object provides methods to obtain meta-information about the scenario, such as the scenario's input and output parameter names and types. The object also provides methods to access all or specific instances, create new instances, and add and remove listeners for property changes.

ScenarioInstance

The `IScenarioInstance` interface is used to represent a single instance of a scenario. The `ScenarioInstance` is returned by methods of the `ScenarioDefinition` object that are used to discover and create new scenario instances. The `ScenarioInstance` has methods to get/set the values of the instance's parameters as well as to delete the instance. The interface also has methods to add and remove listeners for property changes.

To protect the security of personal data of the users who created instances of scenarios, see "Protecting Personal Data in Apama Applications" in *Developing Apama Applications*.

Thread safety

The Scenario Service API is thread-safe. Note that unlike other parts of the Apama API, Scenario Service listener callbacks may be invoked while locks internal to the Scenario Service are held. For this reason, it is not safe to call Scenario Service methods while holding any application-defined lock that might also be acquired within a Scenario Service property change listener, as this may result in a deadlock.

