Apama Capital Markets Foundation

Version 10.11

October 2021

APAMA

This document applies to Apama Capital Markets Foundation 10.11 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 © 2013-2021 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: PAF-CMF-1011-20211015

Table of Contents

About this Guide **5**
Online Information and Support 6
Data Protection 7

1 Introduction to CMF **9**
Introduction to CMF components 10
Using CMF components 10

2 Session Management **11**
Understanding sessions 12
Connecting to sessions 13
Setting session callbacks 14
Setting session parameters 16
Overriding default error handling for sessions 17
Disconnecting from sessions 18

3 Market Data Management **19**
Overview of using MDA 20
Basic example of working with market data 20
Market data subscribers 21
Synthetic datasources 42
Using MDA with legacy market data components 68
Using low-level MDA interfaces 73

4 Order Management **85**
Order state containers 86
Exchange simulation platform 89
Risk firewall 90
Smart Order Router 175
Order bridging services 178
Trade reporting services 179

5 Analytics **181**
Using the position service framework 182
Creating and configuring a currency converter 216

6 Utilities **233**
Service Framework 234
Configuration Service 237
General Util bundle 241
Status publisher 241
Transaction components 241

User session services....241
Adapter bridging bundle....241
Adapter status bridge service....243
DataView manager....243
Helper events for creating DataViews....245

A Samples....249
Installing and running samples....250
Adapter Support Bundle samples....251
General CMF samples....269

B Legacy Finance Interfaces....275
Understanding event protocols....276
The actors in com.apama.oms....277
Market data package overview....288

C Legacy Market Data Management....297
Market data gateway service....298
Market data bridging service....299
Market data publisher components....300
Depth and tick DataViews....301

About this Guide

- Online Information and Support 6
- Data Protection 7

This guide describes how to use Capital Markets Foundation, which is built on Apama's complex event processing platform.

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at https://documentation.softwareag.com.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at https://empower.softwareag.com/.

You can find product information on the Software AG Empower Product Support website at https://empower.softwareag.com.

To submit feature/enhancement requests, get information about product availability, and download products, go to Products.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the Knowledge Center.

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

Software AG Tech Community

You can find documentation and other technical information on the Software AG Tech Community website at https://techcommunity.softwareag.com. You can:

- Access product documentation, if you have Tech Community credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

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

1 Introduction to CMF

- Introduction to CMF components 10
- Using CMF components 10

The Capital Markets Foundation is built on Apama's complex event processing platform. CMF components provide application building blocks. A custom solution will usually include a number of CMF components plus application-specific logic.

Introduction to CMF components

The CMF provides libraries, components, and services for building a variety of applications, primarily for use in the financial sector. Most CMF components are designed as Apama EPL event objects so that an application can manage their creation and usage. Examples of commonly-used components include the order state containers and market data publishers. For more information on Apama functionality such as EPL, monitors and events, see the Apama documentation.

CMF components fall into the following categories:

- **System integration.** Sessions, order and adapter bridges. A CMF application uses a session to send data to and receive data from the connected datasource.
- **Market data management.** BBA subscribers, Depth subscribers, News subscribers, Orderbook subscribers, Quotebook subscribers, Trade subscribers, Market Data Aggregator, Foreign Exchange Cross Rate Service, Forward to Spot Connector, Price Spreader. Some deployed applications (including CMF solutions) use legacy market data management components or legacy finance support.
- **Order management.** Orderbook Matching Engine, Trade Strategy Framework, Risk Firewall, Forward to Spot Converter, Trade Reporting Service. Components and services for all functionality related to order handlingl.
- **Analytics.** Position Service, statistical libraries such as EPL math functions, QuantLib plug-in. You can also use the MATLAB plug-in in a CMF application. See the Apama documentation for information.
- **Utilities.** Service Framework, ConfigStore, DataView Manager, User Session Manager.

Using CMF components

A CMF application is no different from any other Apama application. It can use all available Apama EPL facilities (such as listeners, event objects, closures, and parallel contexts), it can interact with the outside world through adapters and dashboards.

You can add CMF functionality to an application by writing EPL monitors and event types that use CMF components.

Note:
Some CMF components may not yet support multi-context operation. These limitations are noted where necessary.

CMF components are distributed as Apama correlator deployment packages (CDPs). Once you have installed the CMF, you will be able to add the CMF bundles to a project. If you want to distribute an application that does not use Software AG Designer, a set of Ant macros has been provided to assist with the EPL injection.

2 Session Management

- Understanding sessions 12
- Connecting to sessions 13
- Setting session callbacks 14
- Setting session parameters 16
- Overriding default error handling for sessions 17
- Disconnecting from sessions 18

A session provides the interface between your CMF application and a source of data, such as an external adapter. In this CMF release, you can use sessions only with market data architecture (MDA) subscribers. See "Subscribing to market data" on page 22. It is expected that future CMF releases will extend the use of sessions to other CMF components.

The interfaces for using sessions are:

- `com.apama.session.SessionHandlerFactory`
- `com.apama.session.SessionHandler`

The factory object lets you create a session handler and optionally connect to a session. The session handler lets you manage the session interaction between your application and the external datasource.You use it to connect, disconnect, add parameters, add callbacks to be executed upon successful subscription, and override default error behavior for sessions.

Understanding sessions

A CMF application uses a session to send data to and receive data from the connected datasource. A different session is required for each datasource a CMF application needs to connect to. Examples of datasources:

- An adapter that connects to an exchange, such as NASDAQ, LSE, NYSE
- An adapter that connects to an external system such as a database, a message bus
- The CMF market data simulator

When you know which datasource you want your CMF application to use you must obtain the session ID and the transport ID for that datasource. You need to know these values beforehand in order to connect your CMF application to that datasource. Typically, you obtain this information from the documentation for the adapter you want to connect to.

You can use the same `com.apama.session.SessionHandlerFactory` instance to create multiple `com.apama.session.SessionHandler` instances, which in turn can be used to connect to different datasources or the same datasource.

A `SessionHandler` instance can connect to only a single session at one time. In other words, you cannot use the same session handler instance to connect to more than one session at one time. You can, however, disconnect from one session and reuse the same session handler to connect to a different session.

A CMF application can use multiple session handler instances that all use the same underlying session to connect to the same datasource. This can be done in the same monitor, or in different monitors in the same or different contexts. The actual connection (and login, if that is relevant) is made to the datasource and is created only once. Subsequent instances of the session handler that specify the same session identifiers connect to the same datasource and reuse the original established session.

You cannot route or enqueue an instance of a session handler outside the monitor it was created in. You would have multiple instances of a session handler throughout your application connecting to the same session wherever you need direct access to that session. For example, suppose you

subscribe to depth updates in one monitor, and tick updates from the same datasource in another monitor or context. Each subscription requires a separate session handler instance for that session.

Note:
When an adapter/IAF process starts it automatically connects to and registers with the correlator specified in the adapter's configuration file. (Injection of the Session Manager bundle is required for registration to occur.) Registration makes a session for that adapter/datasource available. After registration, the correlator has the adapter's session ID and transport ID and can use those values to provide a session that a CMF application can connect to in order to communicate with the adapter. A CMF application can connect to only registered sessions. If an application needs a list of all registered sessions, it can obtain that information by execution of the `com.apama.session.SessionManagerInterface.getAllSessionInfo()` action. See the ApamaDoc for details.

Connecting to sessions

The two simplest alternatives for connecting to a session are as follows:

- Execute `SessionHandlerFactory.connect()` to create a session handler and connect to the specified session without notification upon success. See "Sample code for using factory to connect" on page 13.
- Execute `SessionHandlerFactory.create()` to obtain a session handler. Then execute `SessionHandler.connect()` to connect to the specified session. There is no notification upon success unless you added one or more connection callbacks before you called the `connect()` action. See "Sample code for using handler to connect" on page 14.

When you execute `SessionHandlerFactory.connect()` or `SessionHandler.connect()` if the action is successful, then your application is connected to the specified session. If the action is not successful, the default behavior is that CMF logs an error message in the correlator log file. This is the result of the default error callback that is common to all session handlers. The default error callback logs at the `ERROR` level. To add or change error behavior, see "Overriding default error handling for sessions" on page 17.

To be notified of a successful connection, you can call `SessionHandler.connectCb()`, which specifies a connection callback that you provide. See "Setting session callbacks" on page 14.

Sample code for using factory to connect

The simplest way to connect to a session is to execute `com.apama.session.SessionHandlerFactory.connect()`. This action uses the session ID and transport ID that you specify, as well as a reference to the main context, to create a session and connect to that session. For example:

```
monitor SessionExample1 {

   context mainContext := context.current();

   action onload() {
      com.apama.session.SessionHandler sessionHandler :=
         (new com.apama.session.SessionHandlerFactory).connect(
```

```
            mainContext, "MySession", "MyTransport");
        }
      ...
}
```

Sample code for using handler to connect

Alternatively, you can create a `SessionHandler` object without connecting to a session. You then use that session handler to connect to a session. For example:

```
monitor SessionExample2 {

   context mainContext := context.current();

   action onload() {
      com.apama.session.SessionHandler sessionHandler :=
         (new com.apama.session.SessionHandlerFactory).create(mainContext);
      sessionHandler.connect("MySession", "MyTransport");
      ...
   }
}
```

Typically, you obtain a session handler object without connecting when you want to set parameters, set callbacks, or override default error handling before you connect to a session. See "Setting session callbacks" on page 14, "Setting session parameters" on page 16, and "Overriding default error handling for sessions" on page 17.

Session handler objects must be initialized with a factory object before they can be used. For example, the following code does not work:

```
monitor IncorrectSessionExample {

   context mainContext := context.current();

   com.apama.session.SessionHandler sessionHandler;

   action onload() {
      // This will not work:
      sessionHandler.connect(mainContext, "MySession", "MyTransport");
      ...
   }
}
```

Setting session callbacks

When your application successfully connects to a session one or more user-defined callbacks can optionally be invoked. There are several ways to invoke callbacks upon connection to a session:

- Execute `SessionHandlerFactory.connectCb()` and specify the callback that you want to invoke upon session connection.
- Execute `SessionHandler.connectCb()` and specify the callback that you want to invoke upon session connection. See "Sample code for specifying callback when using handler to connect" on page 15.

- Execute `SessionHandler.addConnectedCallback()` one or more times to specify one or more callbacks to be invoked upon subsequent calls to `SessionHandler.connect()`. See "Sample code for adding session connection callbacks" on page 15.

  If you add one or more connected callbacks and then execute `SessionHandler.connectCb()`, this invokes the callback specified in the `connectCb()` action and ignores any callbacks you added with the `addConnectedCallback()` action. If you then call `SessionHandler.connect()`, the callbacks you previously added with `addConnectedCallback()` are used.

Sample code for specifying callback when using handler to connect

One method for connecting to a session is to execute the `connectCb()` action on `SessionHandler`. The following example shows a callback invoked by a successful connection to a session. If there is an error that causes the connection to fail then the specified connection callback is ignored.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;

monitor SessionExample3 {

   context mainContext := context.current();

   action onload() {

      // Create a session and obtain a session handler
      SessionHandler sessionHandler :=
         (new SessionHandlerFactory).create(mainContext);

      // Use the session handler to connect with a callback
      sessionHandler.connectCb(
         "MySession", "MyTransport", onSessionConnected);
   }

   action onSessionConnected(SessionHandler sessionHandler) {

      // Use the session handler to subscribe to market data
   }
}
```

Sample code for adding session connection callbacks

After you obtain a session handler and before you use it to connect to a session you can execute `SessionHandler.addConnectedCallback()` one or more times. This action adds the specified callback to the set of callbacks invoked by CMF when this session handler successfuly connects to a session. Only a subsequent connection, and not an existing connection, is affected. The following sample code shows this.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;

monitor SessionExample4 {
```

```
   context mainContext := context.current();

   action onload() {
      // Create session handler and add a callback
      // to be executed upon successful connection
      SessionHandler sessionHandler :=
         (new SessionHandlerFactory).create(mainContext);
      integer refId :=
         sessionHandler.addConnectedCallback(onSessionConnected);
      sessionHandler.connect("MySession", "MyTransport");
   }

   action onSessionConnected(SessionHandler sessionHandler) {

      //Use the session handler to subscribe to market data
   }
}
```

Setting session parameters

You can specify one or more parameters to be applied when connecting to a session.

To set parameters

1. Use a session handler factory to create a session handler object. For example:

   ```
   com.apama.SessionHandler sessionHandler :=
      (new com.apama.session.SessionHandlerFactory).create(mainContext);
   ```

2. Create a `com.apama.utils.Params` object.
3. Add a parameter name/value pair to the parameters object you created.
4. Repeat the previous step for each parameter you want to add.
5. Execute the `setParameters()` action on the session handler object and pass it the `Params` object you created.
6. Execute the `connect()` or `connectCb()` actions on the session handler object. When you connect to the specified session it applies the parameters you added.

For example:

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.utils.Params;

monitor SessionExample5 {

   context mainContext := context.current();

   action onload() {
      SessionHandler sessionHandler :=
         (new SessionHandlerFactory).create(mainContext);

      // Create a parameters object
      Params params := new com.apama.utils.Params;
```

```
      // Add a parameter and then set it for the session handler
      params.addParam("MyParam","ParamValue");
      sessionHandler.setParams(params);

      // Connection will use the parameter
      sessionHandler.connect("MySession", "MyTransport");
    }
     . . .
  }
}
```

To find out if there are any parameters set for a session handler, execute the `getParameters()` action on the session handler object. This action returns a `Params` object, which you can use if you want to add, change, or remove a parameter. The `getParameters()` action returns only the parameters you explicitly set and not any implicit parameters that have default values.

Any changes you make to parameters do not apply to existing connections. To apply one or more new parameters to an existing connection, you must diconnect and then re-connect.

Overriding default error handling for sessions

The default error handler is invoked if there is an error related to a session. For example, an error can occur when

- Connection to a session fails.
- Disconnection from a session fails.
- You try to remove a previously set callback and you specify an invalid callback Id.

The default error handler sends a message to the correlator log file at the `ERROR` level. To change this behavior, execute the `SessionHandler.addErrorCallback()` action and specify a callback function that handles errors. An error handling callback that you define can use `SessionHandler.defaultErrorCallback()` to invoke the default error callback.

You can execute the `addErrorCallback()` action multiple times on the same session handler to implement multiple error handling callbacks. If you add one or more error callbacks to a session handler then the default error callback is not executed for that session handler.

The parameters of a user-defined error callback include the session handler and also a `com.apama.utils.Error` event. An `Error` event has fields for a message, a dictionary of parameters, and an error type code. The `addErrorCallback()` action adds the specified callback to the set of callbacks executed if there is an error in the operation of the specified session handler.

The `com.apama.session.HandlerErrorConstants` event defines the following error type codes for sessions:

- `CONNECTION_FAILED`
- `DISCONNECTION_FAILED`
- `UNKNOWN_CONNECTED_CALLBACK`
- `UNKNOWN_ERROR_CALLBACK`

When you add an error callback the return value is an integer reference ID that you can specify if you execute `removeErrorCallback()` to discontinue execution of that error callback. To remove all error callbacks, execute the `SessionHandler.clearErrorCallbacks()` action. If you remove all previously set error callbacks then error handling behavior reverts to calling the default error callback.

A callback is unknown if you specify an incorrect reference ID for the connected callback or error callback you are trying to remove with the `SessionHandler.removeConnectedCallback()` or `SessionHandler.removeErrorCallback()` action.

Disconnecting from sessions

To disconnect from a session, execute one of the following actions:

- `SessionHandler.disconnect()` disconnects the session handler from the session it is connected to. There is no notification of success. It is an error to try to disconnect before a successful connection.
- `SessionHandler.disconnectCb()` disconnects the session handler from the session it is connected to and invokes the specified callback upon successful disconnection. It is an error to try to disconnect before a successful connection.

Disconnecting from a session that is no longer needed (such as on system shutdown) is typically not required but is considered good practice.

3 Market Data Management

- Overview of using MDA 20
- Basic example of working with market data 20
- Market data subscribers 21
- Synthetic datasources 42
- Using MDA with legacy market data components 68
- Using low-level MDA interfaces 73

This chapter describes CMF's Market Data Architecture (MDA), which provides components that your application uses to subscribe to market data and to work with synthetic datasources, which are internal to the correlator.

Overview of using MDA

To use CMF's MDA, do the following in your CMF application for each datasource from which you want to obtain market data:

1. Connect to a session, which manages the interaction between your CMF application and a market data source. See "Connecting to sessions " on page 13.
2. Optionally, set parameters for market data subscriptions.
3. Set up one or more subscribers to the market data the connected session provides.
4. Do either or both of the following:
   - Set up listeners for the market data events you expect to receive.
   - Add callbacks that customize default MDA behavior. For example, you can override default error behavior or operate on market data by executing a callback function.

Whether you use listeners or callbacks depends on what you want to do. Use listeners when the underlying event is of use to you as-is without pre-processing. This is especially effective when you are looking for patterns in that event type, such as the price going above a threshold. Use callbacks when you want the pre-processing provided by the subscriber, such as the application of incremental updates to a `Depth` or `OrderBook` event before it is returned by a callback.

Note:
Some deployed applications may use legacy market data management components or legacy finance support as opposed to the MDA. Later chapters provide information on legacy interfaces for reference.

Basic example of working with market data

The following code provides a basic example of how to work with market data. It assumes that an adapter has already been set up and that you know that the session ID for that adapter is `MySession` and that the transport ID for that adapter is `MyTransport`. While code can be in multiple contexts, when you execute an MDA action that requires a reference to a context, the reference must be to the main context.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;

using com.apama.md.TradeSubscriberFactory;
using com.apama.md.TradeSubscriber;

monitor SimpleTradeExample {

   action onload() {

      // Connect to a market data source
```

```
      SessionHandler sessionHandler :=
         (new SessionHandlerFactory).connect(
            context.current(),"MySession", "MyTransport");

      // Subscribe to trades on market data source
      TradeSubscriber tradeSubscriber :=
         (new TradeSubscriberFactory).subscribe(sessionHandler,"SOW");

      com.apama.md.T trade;
      on all com.apama.md.T():trade {
         emit trade;
      }
   }
}
```

To run this code, you can configure the correlator to use the CMF market data simulator or an external MDA adapter, and substitute the appropriate session and transport identifiers and subscription symbol. For a fully working example, see the `samples` folder in your CMF installation directory.

Subscriber objects must be initialized by a factory object before they can be used. For example, the following code does not work:

```
monitor InvalidTradeExample {

   context mainContext := context.current();

   com.apama.session.SessionHandler sessionHandler;
   com.apama.md.TradeSubscriber tradeSubscriber;

   action onload() {
      // This will not work:
      sessionHandler.connect(mainContext, "MySession", "MyTransport");
      tradeSubscriber.subscribe("SOW");
   }
}
```

Market data subscribers

The following topics provide information for subscribing to market data updates.

Understanding subscribers

A CMF application uses a subscriber to obtain market data from the connected datasource and to specify callbacks to be executed upon receiving market data. A different subscriber is required for each type of market data a CMF application requires. There are subscribers for the following types of market data:

- BBA
- Depth
- News
- Orderbook

- Quotebook
- Trade

For each type of market data, an application can create any number of subscribers. In a monitor, or in two different monitors, you can use two different subscriber objects to subscribe to the same symbol with the same parameters. However, you cannot use the same subscriber object to subscribe to the same symbol with the same parameters more than once. If you try to it is a `DUPLICATE_SUBSCRIPTION` error.

You can use the same subscriber object to subscribe to the same symbol more than once only if you specify at least one different parameter for each subscription. The adapter handles each such subscription as a separate and distinct subscription.

Depending on the adapter, data might be sent once for multiple subscriptions for the same symbol, but it has the potential to be processed multiple times - once for each subscription.

Separate monitors cannot share subscriber objects. Each monitor that needs market data must subscribe with its own subscribers.

You can add subscribe-callbacks, update-callbacks, and unsubscribe-callbacks before you subscribe. Subsequent subscriptions execute the callbacks you added at the appropriate times, that is, upon subscription, market data update, or unsubscription. Alternatively, you can specify an update callback when you subscribe. See "Setting subscription callbacks" on page 30.

Subscribing to market data

MDA provides interfaces for subscribing to several types of market data. For each market data type, there is a factory interface for creating its eponymous subscriber interface. In addition, the factory interfaces provide the most commonly used actions provided by the subscriber interfaces. The actions provided by these interfaces are the same for each market data type. Many of these interfaces provide pairs of actions in which one takes a user-defined callback and one does not, for example, `TradeSubscriber.subscribe()` and `TradeSubscriber.subscribeCb()`. See the ApamaDoc for details about using the following interfaces:

Interface	Description
`com.apama.md.BBASubscriberFactory` `com.apama.md.BBASubscriber`	For each type of market data, there is a factory object and a subscriber object.
`com.apama.md.DepthSubscriberFactory` `com.apama.md.DepthSubscriber`	
`com.apama.md.NewsSubscriberFactory` `com.apama.md.NewsSubscriber`	
`com.apama.md.OrderbookSubscriberFactory` `com.apama.md.OrderbookSubscriber`	
`com.apama.md.QuotebookSubscriberFactory`	

Interface	Description
`com.apama.md.QuotebookSubscriber`	The factory object for each type of market data provides the same actions. Likewise, the subscriber object for each type of market data provides the same actions. For example, the `BBASubscriberFactory` object lets you create a BBA subscriber, or create and also subscribe to BBA data. A subscription can be for a single symbol or for multiple symbols. The `BBASubscriber` object lets you manage the subscription to BBA data. You use it to subscribe, unsubscribe, set subscription parameters, override default error behavior, manage callbacks, and obtain information about the subscription.
`com.apama.md.TradeSubscriberFactory` `com.apama.md.TradeSubscriber`	
`com.apama.md.HandlerErrorConstants`	This object provides the possible values for the `errorType` parameter in a `com.apama.utils.Error` event of an error callback of a subscriber. They also provide the possible keys for the `extraParams` dictionary.
`com.apama.utils.Params`	This object provides actions for adding, obtaining and removing parameters.
`com.apama.utils.Error`	This object provides actions for setting and obtaining information associated with an error.

The following table lists actions you can execute to subscribe to market data:

To Do This	Execute
Create a market data subscriber and subscribe to one symbol or subscribe to multiple symbols.	*xxx*`SubscriberFactory.subscribe()` *xxx*`SubscriberFactory.multipleSubscribe()` See "Sample code for using factory to subscribe" on page 24.
Create a market data subscriber and use that subscriber to subscribe to one symbol or to multiple symbols.	*xxx*`SubscriberFactory.create()` *xxx*`Subscriber.subscribe()` *xxx*`Subscriber.multipleSubscribe()`

To Do This	Execute
	See "Sample code for using subscriber object to subscribe" on page 25.

If the action is successful then your application is subscribed to the particular type of market data. If the action is not successful the default behavior is that MDA logs an error message in the correlator log file. This is the result of the default error callback that is common to market data subscribers. The default error callback logs at the ERROR level. See "Overriding default error handling for subscriptions" on page 40.

Subscription examples

The following topics provide examples of subscribing to market data.

Sample code for using factory to subscribe

The simplest way to subscribe to market data is to execute the `subscribe()` or `multipleSubscribe()` actions on a factory instance for the type of market data you are interested in. After a successful subscription to market data, your application starts receiving market data events. Often, you set up listeners for the events of interest and specify actions to be performed when there is a match. For example:

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.TradeSubscriberFactory;

monitor SubscriberExample1 {

   context mainContext := context.current();

   action onload() {
      // Create and connect the session handler
      SessionHandler sessionHandler :=
         SessionHandlerFactory.connect(
            mainContext, "MySession", "MyTransport");

      // Create the trade subscriber factory and specify the session
      // handler plus a symbol to subscribe
      TradeSubscriber subscriber :=
         TradeSubscriberFactory.subscribe(sessionHandler, "SOW");

      on all com.apama.md.T() {
         // Process the trade
      }
   }
}
```

Execution of the *xxx*`SubscriberFactory.subscribe()` or `multipleSubscribe()` actions creates a subscription for receiving market data. In the previous example, note that the `subscribe()` action returns a `TradeSubscriber` object. You need a `TradeSubscriber` object if you want to do any of the following:

- Add one or more callbacks. The default behavior for successful subscription to market data does not invoke any callback functions. You might want to change this so that your application executes some logic upon a successful subscription, or when it receives data, or when it unsubscribes. See "Setting subscription callbacks" on page 30.
- Set one or more subscription parameters. See "Setting subscription parameters" on page 38.
- Override default error behavior. By default, a failed subscription invokes a common error callback, which logs messages at the `ERROR` level. See "Overriding default error handling for subscriptions" on page 40.

Sample code for using subscriber object to subscribe

After you obtain a subscriber object for a market data type, you can use that object to subscribe to receive market data. The following sample code shows this.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.TradeSubscriberFactory;
using com.apama.md.TradeSubscriber;

monitor SubscriberExample2 {

   context mainContext := context.current();

   action onload() {
      com.apama.session.SessionHandler sessionHandler :=
         com.apama.session.SessionHandlerFactory.connect(
            "MySession", "MyTransport");

      // Obtain a trade subscriber and use it to subscribe
      TradeSubscriber tradeSubscriber :=
         TradeSubscriberFactory.create(sessionHandler);
      tradeSubscriber.subscribe("EUR/USD");
   }

   // Process the trade
   on all com.apama.md.T() {
   ...
   }
}
```

Before you use a market data subscriber to subscribe you can execute actions on that subscriber to set parameters or add callbacks to be executed upon subscription, upon receiving market data, and upon unsubscribing. When you subscribe without specifying a callback, any callbacks you previously added are executed at the appropriate points. See "Setting subscription callbacks" on page 30.

Sample code for using multiple contexts to receive market data

When using MDA, when you call an action that takes a context as a parameter you must specify a reference to the main context. However, you can use multiple contexts to receive and operate on market data. The following code provides an example of using multiple contexts.

```
using com.apama.session.SessionHandlerFactory;
```

```
using com.apama.session.SessionHandler;
using com.apama.md.TradeSubscriberFactory;
using com.apama.md.TradeSubscriber;
using com.apama.md.client.CurrentTradeInterface;

monitor SubscriberExample3 {

   action onload() {

      context mainContext := context.current();

      // Spawn to a separate context for each symbol
      spawn init(mainContext,"SYM1") to context("SYM1");
      spawn init(mainContext,"SYM2") to context("SYM2");
      spawn init(mainContext,"SYM3") to context("SYM3");

   }

   action init(context mainContext, string symbol) {

      // Connect to a session
      SessionHandler sessionHandler := SessionHandlerFactory.connect(
         mainContext, "MySession", "MyTransport");

      // Subscribe to depth updates for each context's symbol.
      // This is a way to use the factory to subscribe
      // without returning a subscriber object.
      TradeSubscriberFactory.create(
         sessionHandler).subscribe(symbol);
   }

   // Process the trade
   on all com.apama.md.T() {
   ...
   }
}
```

Sample code for subscribing to multiple symbols

The following code provides an example of invoking the `subscribe()` action multiple times to subscribe to multiple symbols. An alternative that has the same result is to invoke the `multipleSubscribe()` action and specify a sequence of symbols.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.TradeSubscriberFactory;
using com.apama.md.TradeSubscriber;

monitor SubscriberExample4 {

   context mainContext := context.current();

   action onload() {
      SessionHandler sessionHandler :=
         SessionHandlerFactory.connect(
            mainContext, "MySession", "MyTransport");

      TradeSubscriber tradeSubscriber :=
```

```
         TradeSubscriberFactory.create(sessionHandler);
      tradeSubscriber.subscribe("EUR/USD");
      tradeSubscriber.subscribe("EUR/GBP");
      tradeSubscriber.subscribe("USD/GBP");
   }
}
```

Sample code using same session to subscribe to multiple data types

The following code provides an example of using the same session to subscribe to multiple market data types.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.TradeSubscriberFactory;
using com.apama.md.BBASubscriberFactory;

monitor SubscriberExample5 {

   context mainContext := context.current();

   action onload() {
      SessionHandler sessionHandler :=
         SessionHandlerFactory.connect(
            mainContext, "MySession", "MyTransport");

      TradeSubscriberFactory.create(
         sessionHandler).subscribe("SOW");

      BBASubscriberFactory.create(
         sessionHandler).subscribe("SOW");
   }
}
```

Subscribing to news

Working with news events is different from working with the other market data event types in two ways:

- A particular news datasource might send out news stories that have parameters that are not captured in a basic news (`com.apama.md.N`) event. When this is the case there is a schema that provides metadata about extra parameters. You can programmatically access this schema to obtain the format of the extra parameters.
- A news event can be deleted by the source that published it.

In addition to the `com.apama.md.N` event interface, which is the basic news datastream event, CMF provides the following event interfaces for working with news. See the ApamaDoc for details.

Event Interface	Purpose	Description
`com.apama.md.ND`	News deletion	Contains fields for stories that have been deleted. Like the `com.apama.md.N` event, it contains a `newsId` field. In an `ND` event

Event Interface	Purpose	Description
		the value of its `newsId` field might match the value of the `newsId` field in a previous `N` event. It is possible, however, that this is not the case and that the original news item was not received because of filtering. An `ND` event also contains fields for the story's source (`sessionId`), symbol, and data.
`com.apama.md.NSI`	Metadata for one extra parameter	A news schema item describes an extra parameter that is associated with a news event. It contains fields for the parameter data type, parameter description, and whether this item is an addition, change, or deletion from the schema.
`com.apama.md.NS`	Schema that contains metadata for all extra news parameters	A schema is a dictionary of `NSI` events. An `NS` event contains the schema for the extra parameters associated with any news story received from the datasource identified by the `NS.sessionId` field.
`com.apama.md.NSD`	Schema update	A new schema update event also contains a dictionary of `NSI` events. You use `NSD` events to apply schema changes to an `NS` event. Typically, it is expected that the schema for extra parameters will be initialized as part of adapter set-up and you will rarely need to update it.

Because there is a dedicated event type for news stories that have been deleted (`com.apama.md.ND`), it is possible to specifically listen for them. However, when a CMF application receives a news-delete event there is no callback that is automatically invoked. Update callbacks are executed for only news events.

The following sample code shows a subscription to receive all news events, regardless of symbol. When a new story is received, this sample sets up a listener for a deletion event for that story.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.NewsSubscriberFactory;
using com.apama.md.NewsSubscriber;
using com.apama.md.ND;
using com.apama.md.DatastreamConstants;
using com.apama.md.client.CurrentNewsInterface;

monitor NewsSubscriberExample1 {
   context mainContext := context.current();

   action onload() {
```

```
      // The RFA adapter exposes the N2_UBMS feed as a separate session.
      NewsSubscriber newsSubscriber :=
         NewsSubscriberFactory.create(
            SessionHandlerFactory.connect(
               mainContext, "N2_UBMS", "RfaAdapter"));

      // Subscribing to a blank symbol subscribes to all news.
      // The specified update callback is executed each time
      // a news event is received.
      newsSubscriber.subscribeCb("", onSymbolNews);
   }

   action onSymbolNews(CurrentNewsInterface newsIface) {

      // Process new or updated news events.
      ...

      // For new news stories, listen for deletion of that story.

      if (newsIface.getUpdateType() = DatastreamConstants.UPDATETYPE_ADD)
         then {
            ND newsDelete;
            on all ND(newsId=newsIface.getNewsId()):newsDelete {
               // Process newsDelete.
               ...
            }
         }
   }
}
```

A news (`com.apama.md.N`) event has a `dictionary` type `data` field. If the adapter supplies any extra news event parameters the dictionary in the `data` field contains metadata for the extra parameters. This dictionary is referred to as the schema. The schema provides a description for any possible extra parameter for the whole feed, not just individual news stories. You might have entries in the schema that do not appear in every news story. To obtain the schema, execute `CurrentNewsInterface.getNewsSchema()`.

The following sample code shows how to obtain the description for an item in the schema data dictionary.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.NewsSubscriberFactory;
using com.apama.md.NewsSubscriber;
using com.apama.md.NSI;
using com.apama.md.client.CurrentNewInterface;

monitor NewsSubscriberExample2 {

   context mainContext := context.current();

   action onload() {
      NewsSubscriber newsSubscriber := NewsSubscriberFactory.create(
         SessionHandlerFactory.connect(
            mainContext, "N2_UBMS", "RfaAdapter"));

       // Subscribing to all news and specify an updated callback
       newsSubscriber.subscribeCb("", onAllNewsUpdate);
```

```
   }

   action onAllNewsUpdate(CurrentNewsInterface newsUpdate) {

      dictionary<string, string> newsData := newsUpdate.getData();

      string dataKey;
      for dataKey in newsData.keys() {
         NSI newsSchemaItem := newsUpdate.getNewsSchemaField(dataKey);

            // Get the data type and description for the news data field.
            ...
      }
   }
}
```

The next sample uses a connection callback to set up listeners for news stories for a particular symbol at a particular value.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.NewsSubscriberFactory;
using com.apama.md.NewsSubscriber;
using com.apama.md.N;

monitor NewsSubscriberExample3 {

   context mainContext := context.current();

   action onload() {
      // Connect to the session and specify a connection callback.
      SessionHandler handler := SessionHandlerFactory.connectCb(
         mainContext, "N2_UBMS", "RfaAdapter", onSessionConnected);
   }

   action onSessionConnected(SessionHandler handler) {

      string symbol := "496";

      // Subscribe to U.S. Non-farm payrolls.
      NewsSubscriber newsSubscriber := NewsSubscriberFactory.
         subscribe(handler, symbol);

      // Listen for U.S. Non-farm payroll updates on this session
      // with an actual value > 200,000.
      N newsStory;
      on all N(sessionId=handler.getSessionInfo().getSessionId(),
         symbol=symbol, actualValue > 200000):newStory {

         // Process news update.
      }
   }
}
```

Setting subscription callbacks

During execution of a CMF application, MDA can execute subscription callbacks that you define at the following points:

- When your application successfully subscribes to market data.
- When your application receives data updates.
- When your application successfully unsubscribes from market data.
- If there is an error when subscribing, receiving data, or unsubscribing.

There are two ways to specify subscription callbacks:

- Specify an update callback when you subscribe. See "Specifying update callbacks when you subscribe" on page 31.
- Add callbacks to a subscriber object. You can add one or more callbacks to be invoked upon successful subscription, delivery of market data, and/or successful unsubscription. See "Adding callbacks to subscriber objects" on page 32.

In addition, you can add an error callback to a subscription. See "Overriding default error handling for subscriptions" on page 40.

Specifying update callbacks when you subscribe

When you subscribe to market data you specify an update callback, which MDA invokes each time your application receives data. The actions you can execute to do this are as follows:

- *xxx*`SubscriberFactory.subscribeCb()`

  Use a subscriber factory to subscribe to one symbol and specify a callback that MDA invokes each time your application receives data from this subscription. See "Sample code for specifying callback when using factory to subscribe" on page 33.

- *xxx*`SubscriberFactory.multipleSubscribeCb()`

  Use a subscriber factory to subscribe to a sequence of symbols and specify a callback that MDA invokes each time your application receives data from this subscription.

- *xxx*`SubscriberFactory.create()` followed by *xxx*`Subscriber.subscribeCb()`

  Create a subscriber handler and use it to subscribe to one symbol and specify a callback that MDA invokes each time your application receives data from this subscription. If you added any update callbacks with the `addUpdateCallback()` action on this subscriber object they are not invoked. Only the callback you specify is executed when market data from this subscription is received. See "Sample code for specifying callback when using handler to subscribe" on page 33.

- *xxx*`SubscriberFactory.create()` followed by *xxx*`Subscriber.multipleSubscribeCb()`

  Create a subscriber handler and use it to subscribe to a sequence of symbols and specify a callback that MDA invokes each time your application receives data from this subscription. If you added any update callbacks with the `addUpdateCallback()` action on this subscriber object they are not invoked. Only the callback you specify is executed when market data from this subscription is received.

Adding callbacks to subscriber objects

You can add one or more callbacks to a subscription handler. Each callback you add to a subscriber handler applies to subsequent subscriptions made from that subscriber and not to an existing subscription. The following table describes the kinds of callbacks you can add. The actions you execute are all on an *xxx*`Subscriber` object.

Callback Type	Action to Execute	Notes
Subscription	`addSubscribedCallback()`	Adds a callback to be invoked upon subsequent successful subscriptions from this subscriber. Note that `subscribeCb()` and `multipleSubscribeCb()` specify update callbacks and not subscription callbacks.
Update	`addUpdateCallback()`	Adds a callback to be invoked for subsequent subscriptions from this subscriber each time MDA delivers market data to your application. Note that if you add one or more update callbacks and then call the `subscribeCb()` or `multipleSubscribeCb()` action then the update callbacks you added are ignored and only the specified update callback is executed. See "Sample code for adding update callbacks to subscriptions" on page 34.
Unsubscription	`addUnsubscribedCallback()`	Adds a callback to be invoked upon a successful unsubscribe action. Note that if you add one or more unsubscription callbacks and then call the `unsubscribeCb()` or `multipleUnsubscribeCb()` action then the unsubscription callbacks you added are ignored and only the specified callback is executed.

Suppose you add some callbacks with `addUpdateCallback()`, then execute `subscribeCb()`, and then execute `subscribe()`. The `subscribeCb()` action uses the specified update callback and ignores the callbacks you added with the `addUpdateCallback()` action. However, the `subscribe()` action uses the callbacks you added with the `addUpdateCallback()` action even though there was an intervening subscription that specified its own update callback.

Sample code for specifying callback when using factory to subscribe

You can use a market data factory object to subscribe to receive market data and also specify a callback to execute when your application receives market data. The following code shows this. You can execute `subscribeCb()` for one symbol or `multipleSubscribeCb()` for more than one symbol. If there is an error that causes the subscription to fail then the specified update callback is ignored.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.TradeSubscriberFactory;

monitor SubscriberExample6 {

 context mainContext := context.current();

 action onload() {
   // Create and connect to the session
   SessionHandler sessionHandler :=
      (new SessionHandlerFactory).connect(
         mainContext, "MySession", "MyTransport");

    // Use a market data factory to subscribe and specify the
    // update callback to be executed when data is received
    (new TradeSubscriberFactory).subscribeCb(
       sessionHandler, "SOW", onAllTrade);
  }

 action onAllTrade(
   com.apama.md.client.CurrentTradeInterface currTrade ) {
     log "Got Trade using Callback - " at INFO;
     log "  for symbol: " + currTrade.getSymbol() at INFO;
     log "  Quantity:   " + currTrade.getQty().toString() at INFO;
     log "  Price:      " + currTrade.getPrice().toString() at INFO;
     log "  With EP:    " + currTrade.getEPValuesInterface().getRaw().toString()
                              at INFO;
 }
}
```

Sample code for specifying callback when using handler to subscribe

After you obtain a subscriber object for a market data type, you can use that object to subscribe to receive market data and also specify the update callback to be executed when your application receives market data. If you previously added any callbacks to that subscriber handler (with the `addUpdateCallback()` action) they are not executed when your application receives data from this subscription. Only the specified update callback is executed when data is received.

Execute `subscribeCb()` for one symbol or `multipleSubscribeCb()` to receive data for more than one symbol. The following sample code shows a multiple subscription that specifies a callback. If there is an error that causes the subscription to fail then the specified update callback is ignored and the default error callback, or an error callback you added, is executed.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.TradeSubscriberFactory;
```

```
using com.apama.md.TradeSubscriber;

monitor SubscriberExample7 {

 context mainContext := context.current();

 action onload() {

   // Create and connect to the session
   SessionHandler sessionHandler :=
      (new SessionHandlerFactory).connect(
         mainContext, "MySession", "MyTransport");

   // Create the subscriber
   TradeSubscriber subscriber :=
      (new TradeSubscriberFactory).create( sessionHandler );

   // Create a sequence of  symbols and subscribe them
   // and specify the update callback to be executed each time
   // your application receives market data for this subscription
   sequence< string > symbols := [ "EUR/USD", "EUR/GBP" ];
   subscriber.multipleSubscribeCb(symbols, onAllTrade);

   // Unsubscribe after 10 seconds and specify a callback
   on wait ( 10.0 ) {
      subscriber.multipleUnsubscribeCb(symbols, onUnsubscribe);
   }
 }

 action onAllTrade( com.apama.md.client.CurrentTradeInterface currTrade ) {
   log "Got Trade using Callback - " at INFO;
   log "  for symbol: " + currTrade.getSymbol() at INFO;
   log "  Quantity:   " + currTrade.getQty().toString() at INFO;
   log "  Price:      " + currTrade.getPrice().toString() at INFO;
   log "  With EP:    " + currTrade.getEPValuesInterface().getRaw().toString()
                          at INFO;
 }

 action onUnsubscribe( TradeSubscriber subscriber,
   com.apama.md.adapter.ConnectionKey connKey ) {

      // This will be called for each symbol being unsubscribed
 }
}
```

Note:
If you execute `addUnsubscribedCallback()` on a subscriber handler and then execute `unsubscribeCb()` or `multipleUnsubscribeCb()` only the callback specified in the unsubscribe action is executed upon successful unsubscription. Any callbacks that were added with the `addUnsubscribedCallback()` action are ignored.

Sample code for adding update callbacks to subscriptions

After subscription to a market data type, your application begins to receive market data. If you want you can have one or more callback functions executed each time your application receives data. You do this by adding one or more update callbacks to the subscriber handler before

subscription. The addition of update callbacks affects only subsequent subscriptions and not existing subscriptions.

Following is an example of adding an update callback to a subscriber handler:

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.TradeSubscriberFactory;
using com.apama.md.TradeSubscriber;

monitor SubscriberExample8 {

  context mainContext := context.current();

  action onload() {
    SessionHandler sessionHandler :=
       (new SessionHandlerFactory).connect(
          "MySession", "MyTransport");

    // Create a trade subscriber
    TradeSubscriber tradeSubscriber :=
       (new TradeSubscriberFactory).create(sessionHandler);

    // Add an update callback and then subscribe
    integer refId := tradeSubscriber.addUpdateCallback(onAllTrade);
    tradeSubscriber.subscribe("SOW");
  }

  action onAllTrade( com.apama.md.client.CurrentTradeInterface currTrade ) {
    log "Got Trade using Callback - " at INFO;
    log "  for symbol: " + currTrade.getSymbol() at INFO;
    log "  Quantity:   " + currTrade.getQty().toString() at INFO;
    log "  Price:      " + currTrade.getPrice().toString() at INFO;
    log "  With EP:    " + currTrade.getEPValuesInterface().getRaw().toString()
                           at INFO;
  }
}
```

Suppose you add one or more update callbacks to a subscriber handler and then execute the `subscribeCb()` or `multipleSubscribeCb()` action on that handler. MDA ignores the update callbacks that were previously added with the `addUpdateCallback()` action. Only the update callback specified in the subscribe action is invoked when new market data is received.

Sample code for adding update callback between subscriptions

The following code provides another example of adding an update callback to a subscriber object. In this example, an update callback is added and then there is a subscription. Then another update callback is added. This second callback applies to subsequent subscriptions and not existing subscriptions. Consequently, when there is another subscription, both callbacks that were added are invoked when data arrives as a result of that subscription.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.TradeSubscriberFactory;
using com.apama.md.TradeSubscriber;

monitor SubscriberExample9 {
```

```
  context mainContext := context.current();

  action onload() {
    SessionHandler sessionHandler :=
       (new SessionHandlerFactory).connect(
          mainContext, "MySession", "MyTransport");

    TradeSubscriber tradeSubscriber :=
       (new TradeSubscriberFactory).create(sessionHandler);
    integer refId := tradeSubscriber.addUpdateCallback(onAllTrade);
    tradeSubscriber.subscribe("EUR/USD");

    // This update callback affects only subsequent subscriptions.
    refId := tradeSubscriber.addUpdateCallback(onEURGBPTrade);
    tradeSubscriber.subscribe("EUR/GBP");
  }

  action onAllTrade( com.apama.md.client.CurrentTradeInterface currTrade ) {
    log "Got Trade using Callback - " at INFO;
    log "  Quantity:   " + currTrade.getQty().toString() at INFO;
    log "  Price:      " + currTrade.getPrice().toString() at INFO;
    log "  With EP:    " + currTrade.getEPValuesInterface().getRaw().toString()
                           at INFO;
  }

  action onEURGBPTrade( com.apama.md.client.CurrentTradeInterface currTrade ) {
    log "Got EUR/GBP using Callback - " at INFO;
    log "  Quantity:   " + currTrade.getQty().toString() at INFO;
    log "  Price:      " + currTrade.getPrice().toString() at INFO;
    log "  With EP:    " + currTrade.getEPValuesInterface().getRaw().toString()
                           at INFO;
  }
}
```

Sample code for setting subscription callback

The following example creates a trade subscriber and then adds a subscription callback. It also shows how to get a connection key object so that you can filter on the `com.apama.md.T` events that are specific to your subscription. When using a connection key consider the following:

- `sessionId` filters all subscriptions on the specified session.
- `symbol` filters all subscriptions across all sessions for the specified symbol.
- `connectionId` filters for a specific session and symbol subscription, and it is not necessary to additionally specify `sessionId` or `symbol` when using `connectionId`.
- It is possible to subscribe multiple times in different application locations to the same symbol on the same underlying session and using the exact same parameters. Such duplicate subscriptions are treated as the same subscription. They all share the same underlying subscription request, with only a single update being sent for all instances of the duplicate subscriptions. If you subscribe to a symbol in the same way on the same underlying session in multiple places, they all use the same `connectionId`.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
```

```
using com.apama.md.TradeSubscriberFactory;
using com.apama.md.TradeSubscriber;

monitor SubscriberExample10 {

   context mainContext := context.current();

   action onload() {
      // Create and connect the session handler.
      SessionHandler sessionHandler := (new SessionHandlerFactory).
         connect(mainContext, "MySession", "MyTransport");

      // Create the trade subscriber factory and specify the session
      // handler plus a symbol to subscribe.
      TradeSubscriber subscriber := (new TradeSubscriberFactory).
         create(sessionHandler);
      integer refId := subscriber.addSubscribedCallback( onSubscribed );
      subscriber.subscribe("SOW");
   }

   action onSubscribed( TradeSubscriber subscriber,
      com.apama.md.adapter.ConnectionKey connKey ) {

      // The ConnectionId is a unique identifier for the
      // subscribed symbol for the connected session.
      on all com.apama.md.T(
         connectionId=connKey.getConnectionId() ) {
         // Process the trade
      }
   }
}
```

Sample code for subscription update callback invoked in a connection callback

You can define a connection callback in which you subscribe to market data and specify an update callback on the subscription. For example:

```
 using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;

using com.apama.md.DepthSubscriberFactory;
using com.apama.md.DepthSubscriber;
using com.apama.md.client.CurrentDepthInterface;

using com.apama.oms.NewOrder;
using com.apama.oms.OrderUpdate;

monitor SubscriberExample11 {

   context mainContext := context.current();

   action onload() {

      // Connect to a session and specify callback
      SessionHandlerFactory().create(mainContext).connectCb(
         "MySession", "MyTransport", onSessionConnected);

   }
```

```
   action onSessionConnected(SessionHandler sessionHandler) {

      // Subscribe to depth updates for single symbol
      DepthSubscriberFactory().create(sessionHandler).
         subscribeCb("SOW",onDepth);

   }

   action onDepth(CurrentDepthInterface di) {

       // Report updates
      log "Got Depth update using Callback - " at INFO;
      log " Symbol: " + di.getSymbol() at INFO;
      log " Best Bid Quantity: " + di.getBBA().getBidQty().
         toString() at INFO;
      log " Best Bid Price: " + di.getBBA().getBidPrice().
         formatFixed(2) at INFO;
      log " Best Ask Quantity: " + di.getBBA().getAskQty().
         toString() at INFO;
      log " Best Ask Price: " + di.getBBA().getAskPrice().
         formatFixed(2) at INFO;

   }

}
```

Setting subscription parameters

You can specify one or more parameters to be applied when subscribing to market data. For example, you might specify a parameter that is related to MDA, such as `TRANSFER_MODE`, or a custom parameter to be sent as part of the subscription to a datasource.

To set parameters

1. Use a subscriber factory to create a subscriber for the relevant type of market data. For example:

   ```
   com.apama.md.TradeSubscriber tradeSubscriber :=
      (new com.apama.md.TradeSubscriberFactory).create(sessionHandler);
   ```

2. Create a `com.apama.utils.Params` object.
3. Add a parameter name/value pair to the parameters object you created.
4. Repeat the previous step for each parameter you want to add.
5. Execute the `setParameters()` action on the subscriber.
6. Execute one of the subscribe actions on the subscriber. When MDA processes the subscription it applies the parameters you added.

For example:

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.TradeSubscriberFactory;
using com.apama.md.TradeSubscriber;
```

```
using com.apama.utils.Params;

monitor SubscriberExample12 {

   context mainContext := context.current();

   action onload() {
      SessionHandler sessionHandler :=
         (new SessionHandlerFactory).connect(
            "MySession", "MyTransport");

      // Create a trade subscriber helper object and a parameters object
      TradeSubscriber tradeSubscriber :=
         (new TradeSubscriberFactory).create(sessionHandler);
      Params params := new Params;

      // Set a parameter for the subscriber
      params.addParam("MyParam","ParamValue");
      tradeSubscriber.setParams(params);

      // Subscription will use the parameter
      tradeSubscriber.subscribe("SOW");
    }

}
```

To find out if there are any parameters set for a subscription, execute the `getParameters()` action on the subscriber handler object. This action returns a `Params` object, which you can use if you want to add, change, or remove a parameter. The `getParameters()` action returns only the parameters you explicitly set and not any parameters that are implicitly set with a default value.

Any changes you make to parameters do not apply to existing subscriptions. To apply one or more new parameters to an existing subscription, you must end that subscription and then re-subscribe.

Accessing extra parameters

A particular datasource might send out market data that has parameters that are not captured in a basic market data event. When this is the case there is a dictionary of key to value string pairs that are referred to as extra parameters. Also provided is a schema that contains type and description data about extra parameters. The following EPL interfaces provide access to extra parameters:

- `com.apama.md.client.EPSchemaInterface` — Each market data interface provides the `getEPSchemaInterface()` action, which you can call to access the schema for any extra parameters.
- `com.apama.md.client.EPValuesInterface` — Each market data interface and each book entry interface provides the `getEPValuesInterface()` action, which you can call to access the values of any extra parameters.

See the ApamaDoc for details. The code below provides an example of using the `EPSchemaInterface` and `EPValuesInterface`.

```
using com.apama.md.client.CurrentOrderbookInterface;
using com.apama.md.client.EPValuesInterface;
using com.apama.md.client.EPSchemaInterface;
```

```
using com.apama.md.client.OrderEntry;

. . .

   action onAllOrders2( CurrentOrderbookInterface currOrderbook ) {

      log "Got Orderbook using Callback2 -" at INFO;
      log "  for symbol: " + currOrderbook.getSymbol() at INFO;

      // Extract extra parameters.

      // Get the extra parameter interfaces:
      EPValuesInterface valuesIface := currOrderbook.getEPValuesInterface();
      EPSchemaInterface schemaIface := currOrderbook.getEPSchemaInterface();

      // Loop through top level values:
      log "  with EPs: " at INFO;
      logExtraParameters( valuesIface, schemaIface );

      // Loop through extra parameters for top bid order:
      sequence<OrderEntry> bids := currOrderbook.getRawBids();
      if bids.size() > 0 then {
         log "Top Bid EPs: " at INFO;
         logExtraParameters( bids[0].getEPValuesInterface(), schemaIface );
      }
   }

   // Log the value, type and description of all extra parameters:
   action logExtraParameters(
      EPValuesInterface valuesIface,
      EPSchemaInterface schemaIface) {

      // Loop through all values and get the schema (type and description)
      // for each:
      string key;
      for key in valuesIface.getRaw().keys() {
         if schemaIface.hasField( key ) then {
            log "    " + key + " " + valuesIface.getParam( key ) + " "
               + schemaIface.getFieldType( key )
               + " " + schemaIface.getFieldDescription( key ) at INFO;
         }
      }
   }
```

The `AggregatedBook` actions and interfaces are slightly different as they offer access to the extra parameters from the underlying datasources of the Aggregator/FTSC. They require the symbol and source id (known as the XEPKey) of the underlying datasource to access the values and schema.

Overriding default error handling for subscriptions

By default, MDA invokes the default error handler if there is an error related to market data subscribers. For example, if any of the following happen:

- Subscription fails
- Delivery of subscribed data fails

- Unsubscription fails

The default error handler sends a message to the correlator log file at the `ERROR` level. To change this behavior, execute *xxx*`Subscriber.addErrorCallback()`, which adds the specified callback to the set of callbacks executed if there is an error related to that subscriber. An error handling callback that you define can use *xxx*`Subscriber.defaultErrorCallback()` to invoke the default error callback.

You can execute the `addErrorCallback()` action multiple times on the same subscriber to implement multiple error handling callbacks for that subscriber. If you add one or more error callbacks to a subscriber then the default error callback is not executed for that subscriber.

The parameters of a user-defined error callback include the subscriber object and also a `com.apama.utils.Error` event. An `Error` event has fields for a message, a dictionary of parameters, and an error type code. The `addErrorCallback()` action adds the specified callback to the set of callbacks executed if there is an error in the operation of the specified subscriber.

The `com.apama.md.HandlerErrorConstants` event defines the following error type codes for subscriptions:

- `FAILED_TO_SUBSCRIBE`
- `FAILED_TO_UNSUBSCRIBE`
- `SESSION_ERROR`
- `DUPLICATE_SUBSCRIPTION`
- `UNKNOWN_UPDATE_CALLBACK`
- `UNKNOWN_SUBSCRIBED_CALLBACK`
- `UNKNOWN_UNSUBSCRIBED_CALLBACK`
- `UNKNOWN_ERROR_CALLBACK`

When you add an error callback the return value is an integer reference ID that you can specify if you execute `removeErrorCallback()` to discontinue execution of that error callback. To remove all error callbacks, execute the *xxx*`Subscriber.clearErrorCallbacks()` action. If you remove all previously set error callbacks then error handling behavior reverts to calling the default error callback.

A callback is unknown if you specify an incorrect reference ID for a callback you are trying to remove with the *xxx*`Subscriber.removeSubscribedCallback()`, `removeUpdateCallback()` or `removeUnsubscribedCallback()` action.

The following code provides an example of adding an error callback to a subscriber.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.md.TradeSubscriberFactory;
using com.apama.md.TradeSubscriber;

monitor SubscriberExample13 {

   context mainContext := context.current();
```

```
   action onload() {
      SessionHandler sessionHandler :=
         (new SessionHandlerFactory).connect(
            mainContext, "MySession", "MyTransport");

      TradeSubscriber tradeSubscriber :=
         (new TradeSubscriberFactory).create(sessionHandler);
      integer refId := tradeSubscriber.addErrorCallback(onError);
      tradeSubscriber.subscribe("SOW");

      // The following duplicate subscription causes an error.
      tradeSubscriber.subscribe("SOW");
   }

   action onError( TradeSubscriber subscriber,
      com.apama.utils.Error error ) {

      // This error should be a DUPLICATE_SUBSCRIPTION type.
    }
}
```

Unsubscribing from market data

To stop receiving market data, execute one of the unsubscribe actions provided by the subscriber object:

To Do This	Execute
Unsubscribe from one symbol or multiple symbols.	*xxx*`Subscriber.unsubscribe()` *xxx*`Subscriber.multipleUnsubscribe()`
Unsubscribe from one symbol or multiple symbols and invoke the specified callback upon successful unsubscription.	*xxx*`Subscriber.unsubscribeCb()` *xxx*`Subscriber.multipleUnsubscribeCb()`

Specify the symbol or sequence of symbols that you want to unsubscribe. There must be a subscription for each symbol you specify. It does not matter whether the subscription was for an individual symbol or a sequence of symbols. Any subscribed symbol can be unsubscribed individually or as part of a sequence.

If you specify a callback MDA invokes it upon successful unsubscription. An unsubscribe action that specifies a callback ignores any unsubscription callbacks previously added with the *xxx*`Subscriber.addUnsubscribeCallback()` action.

Before disconnecting from a session, it is good practice to terminate any subscriptions serviced by that session.

Synthetic datasources

From an application's point of view, a synthetic datasource looks and acts the same as any other MDA-based datasource. However, these datasources are internal to the correlator rather than

external, such as IAF adapters. Synthetic datasources generally act as filters, modifiers, or aggregators for other datasources (which may be either external adapters or other synthetic datasources).

A CMF application that uses a synthetic datasource must also use the CMF service framework. Every monitor that needs to interact with a synthetic datasource *must* do the following:

1. Create an instance of the service framework interface object.
2. Initialize the service framework instance.
3. Wait for the service framework instance to confirm that it is fully initialized.
4. Create listeners and route, enqueue or emit events.

Failure to initialize the service framework correctly will typically lead to an application that fails intermittently and unpredictably, due to race conditions between the application code and synthetic datasources.

The mechanisms for interacting with the service framework are explained in more detail in the "Service Framework" on page 234. An implementation of the service framework can use the configuration service, which manages CMF and application configuration data stored in an external database. "Configuration Service" on page 237 describes how to use the configuration service.

Market data aggregator

The market data aggregator presents data from multiple sessions as a single composite session. Similar to any other datasource, applications can access this synthetic datasource through the market data manager and client APIs. The aggregator also provides, where available, Extra Params (EP) updates for connected datastreams. The aggregator supports only compound delta updates.

The aggregator supports publication of:

- `com.apama.md.BBA` (best bid and ask, or top-of-book)
- `com.apama.md.X` (aggregated books, or synthetic cross rate books)

The aggregator supports underlying data types of:

- `com.apama.md.O` (orderbooks or Market-by-Order (MBO))
- `com.apama.md.D` (depth or Market-by-Price (MBP))
- `com.apama.md.QB` (Quotebooks)
- `com.apama.md.X` (aggregated books, or synthetic cross rate books)

Creating an aggregator instance

The following code excerpt shows how to create an instance of an aggregator using two underlying sessions (1 and 2):

```
action createAggregator() {
```

```
   // Get the SessionInfo for the two underlying Sessions we want to use
   // In this case, SessionId 1 and 2
   sequence<com.apama.session.SessionInfo> sources := new
      sequence<com.apama.session.SessionInfo>;
   sources.append( sessionManagerIface.getSessionInfo( 1 );
   sources.append( sessionManagerIface.getSessionInfo( 2 );

   // Now create the Aggregator
   com.apama.md.agg.Aggregator aggregator := new
      com.apama.md.agg.Aggregator;
   aggregator.create( mainContext, "MyAggregator", sources,
      new sequence<com.apama.session.SessionConfigParams>,
      aggCreationSuccess, aggCreationFailure );

}
```

Connecting to an aggregator

An application connects to an aggregator in the same way it would to any other MDA datasource. The following code excerpt demonstrates how to connect to an aggregator for the EUR/USD symbol.

```
// This code excerpt assumes that the Aggregator and a MDManagerInterface
// have already been created
aggManager := mgr.createAggregatedBookManager();
com.apama.session.CtrlParams controlParams := new com.apama.session.CtrlParams;
aggManager.connect( "EUR/USD", controlParams,
   onSessionError, onConnectionSuccess, onConnectionFailure );
```

After successful connection, the `AggregatedBookManagerInterface` can be used in the normal way to examine the aggregated book data that is being published by the Aggregator.

The Advanced market data sample project demonstrates a simple use case where one aggregator combines data from up to three different underlying sessions.

Aggregator control parameters

The aggregator supports the control parameters described in the following table.

Name	Type	Default Value
`ENABLE_TIMESTAMPS`	stringified `boolean`	Disabled
If `true`, enables generation of extra timestamps on each published data event: one for the time at which the underlying datastream event entered the aggregator, and other at the point of publication from the aggregator. These are useful to calculate the performance of the aggregator within an application.		
`MIN_SOURCES`	stringified `integer`	Waits for all connections
Defines the minimum number of underlying datastreams that must be connected before the aggregator starts publishing an aggregated book. By default, the aggregator requires all underlying		

Name	Type	Default Value
datasources to be connected before it starts publication. Setting `MIN_VALUES` to a value of `1` causes the aggregator to publish data as soon as it receives a single connection, rather than waiting for all connections to be made.		
`QUEUE_INITIAL_DATA`	stringified `boolean`	Disabled
If `true`, causes all events that the aggregator has calculated, but not published (due to the number of current connections not matching that defined in the `MIN_SOURCES` configuration parameter), to be queued. This option is used in conjunction with the `MIN_SOURCES` parameter. This option allows you to queue all events that the aggregator has calculated but not published. As soon as the correct number of connections complete, all publication events that have been queued will be published. This option is useful to show how the aggregated book has been built up from the underlying datastream events, rather than generating the final snapshot from all the updates that the aggregator has received.		
`CLEAR_STALE_DATA`	stringified `boolean`	Disabled
If `true`, will cause the aggregator to permanently delete any current data from a source that has gone stale (has reached its timeout) rather than temporarily remove it from the aggregated book and cache it.		
`DATA_TIMEOUT`	stringified `float`	Disabled
Specifies a data timeout period in seconds, after which the datasource connection will be temporarily removed from the aggregated book. If no new updates have been received by the aggregator within the timeout period, the data is likely to be out-of-date. Once a new update has been received, the datasource connection will be re-added to the aggregated book.		
`UNDERLYING_DATA_TIMEOUT`	stringified `dictionary<integer,integer>`	Disabled
Similar to `DATA_TIMEOUT`, but this parameter allows setting the timeout on each underlying source separately. The format is a stringified dictionary in which the key is the integer id of the underlying source and the value is the integer timeout for that source. The default is an empty dictionary. If an id for an underlying source is not found, the setting of `DATA_TIMEOUT` is used.		

Forward to Spot Convertor

The Forward to Spot Convertor (FTSC) synthetic datasource connects to an underlying session (which can be an IAF adapter or another synthetic datasource), and converts forward or futures prices and quantities to spot prices and quantities and re-publishes them within the correlator. The CMF contains a helper class to convert spot prices and quantities to forward prices and quantities.

The FTSC supports connections to sessions with the following datastream capabilities, and will attempt to connect to them in the following order:

1. Depth

2. Best Bid Ask
3. Trade

Forward to Spot Convertor parameters

The FTSC uses following configurable parameters:

Parameter	Type	Allowable values	Default value	Mandatory	Description
`publishSpotToForward`	`boolean`	`true`, `false`	`false`	No	If `true`, converts spot prices and quantities to forward prices and quantities. If `false`, converts forward prices and quantities to spot prices and quantities.
`spotSymbol`	`string`	`string`	null `string`	No	If not null, the string will be used as the symbol for spot events. If null or not specified, the original symbol will be used.
`isDirect`	`boolean`	`true`, `false`	`true`	No	If `true`, the currency pair symbol connected and the spot/forward prices and quantities calculated for the symbol do not require inversion. If `false`, the currency pair symbol connected and the spot/forward prices and quantities calculated for the symbol should be inverted. For example, if the symbol connected is EUR/USD and the prices have to be published for USD/EUR, then the `isDirect` flag is set to `false`.
`multiplier`	`float`	Greater than `0`	N/A	Yes	Used in price conversions from Forward to Spot and also from Spot to Forward. If the `isDirect` flag is set to `false`, then the multiplier becomes a divisor. If multiplier is not valid, returns the error: `Multiplier value not valid, SpotConverter instance not created.`
`forwardPointsDivisor`	`float`	`0` or greater than `0`	`0`	No	Specifies a divisor for forward points. If the value specified is neither `0` nor `1`, forward points

Parameter	Type	Allowable values	Default value	Mandatory	Description
					will be divided by the specified value before they are used for conversion.
`isForward PointsInversed`	`boolean`	`true`, `false`	`false`	No	If `false`, the Forward to Spot converter will invert the signs for the given forward points before using them in the calculations. If `true`, the Forward to Spot Converter will use the values as provided. Some venues provide the forward points with the inverted signs, whereas others leave the signs as they are. The logic in the Forward to Spot Converter expects the signs to be inversed.
`forward PointsBid`	`float`	Positive or negative	`0`	No	Depending on the `isForwardPointsInversed` flag and using the `forwardPointsDivisor` parameter value, these points will be normalized and then added or subtracted from the outright bid price.
`forward PointsAsk`	`float`	Positive or negative	`0`	No	Depending on the `isForwardPointsInversed` flag and using the `forwardPointsDivisor` parameter value, these points will be normalized and then added or subtracted from the outright ask price.
`forwardPoints BidRaw`	`float`	Positive or negative	`0`	No	The raw bid forward points before normalized based on divisor and sign inversion.
`forwardPoints AskRaw`	`float`	Positive or negative	`0`	No	The raw ask forward points before normalized based on divisor and sign inversion.
`contractSize`	`integer`	Greater than `0`	N/A	Yes	While converting from Forward to Spot; the forward quantity available is multiplied by the

Parameter	Type	Allowable values	Default value	Mandatory	Description
					contract size. For example, if the EUR/USD contract size is 125,000 and the quantity is 10, then 1.25 million is available. While converting from Spot to Forward, the spot quantity available will be divided by the contract size.
`pipSize`	`float`	Positive or negative	`1`	No	Pip (percentage in point: smallest price increment) is used for rounding the prices in case of Spot to Forward conversion. If the pip value specified is greater than `0` then all the outgoing forward prices are rounded to pip. Values less than `0` result in pip being set to `0`.

`epsSize`	`float`	Positive or negative	`0`	No	Eps is the other component used for rounding the outgoing forward prices. Eps size is only relevant if the pip size is set to a valid value. It is subtracted from the price for the sell side and added in case of the buy side. If the pip value specified is less than `0` eps will be set to `0`. For example, if the side is buy, unrounded forward price is 20.51543999, eps is 0.0001 and pip is 0.0002 then the rounded forward price would be:
					
					rounded price =
					
					`(mathUtil.add (20.51543999,0.0001)/0.0002) .floor().toFloat()* 0.0002 = 20.5154`
					
					If the side is sell the price would be: rounded price =
					
					`(mathUtil.subtract (20.51543999,0.0001)/0.0002) .ceil().toFloat() * 0.0002 = 20.5154`

Parameter	Type	Allowable values	Default value	Mandatory	Description
`spotPrice Precision`	`float`	Positive or negative	`1`	No	The value is used for rounding the outgoing spot prices. The value provided is converted to $10^{(value)}$. For example, if the `spotPricePrecision = 4` then the parameter value will be set to 10^4 `= 1000`. If the `spotPricePrecision` value is not specified or is less than `0` then the parameter is set to `1`. Errors return the following message: `Spot Price Precision value not valid, SpotConverter instance not created.`

Calculations

This section describes the calculations that the FTSC uses to convert from forwards to spots, and vice versa. Before the conversion, if the value of the `pipSize` parameter is greater than 0.0, the value of the `epsSize` parameter is added (for bids) or subtracted (for asks) from the forward price. For example:

- To calculate the bid price on the buy side:

  ```
  forward price = forward price + eps size
  ```

- To calculate the ask price on the sell side:

  ```
  forward price = forward price - eps size
  ```

Converting forwards to spots

When the prices are direct, spot price and quantity are calculated as follows:

```
spot price (unrounded) = mathUtil.add(forward price/multiplier,
   forward points)
spot quantity = quantity * contract size
```

When the prices are indirect, bid price is used to calculate the ask price and quantity and vice-versa. The formulas used are:

```
spot price (unrounded) = mathUtil.add(multiplier/forward price,
   forward points)
spot quantity =(((quantity.toFloat()*contractSize.toFloat()*
   (forward price / multiplier)) * 10000.0).round()/10000.0 ).floor()
```

To round the calculated spot bid price, the `spotPricePrecision` parameter is used in the following way:

```
spot price(rounded) = spotprice(unrounded)*
   spotPricePrecision.floor().toFloat()/spotPricePrecision
```

To round the calculated spot ask price, the `spotPricePrecision` parameter is used in the following way:

```
spot price(rounded) = spotprice(unrounded)*
   spotPricePrecision.ceil().toFloat()/spotPricePrecision
```

Converting spots to forwards

When the prices are direct, forward price and quantity are calculated as follows:

```
forward price(unrounded) =
   mathUtil.subtract(spot price , forward points)*multiplier
forward quantity = spot quantity / contract size
```

- `forward points = forwardPointsBid` or `forwardPointsAsk` depending on the side.
- `forward points = mathUtil.add(forwardPointsBid , forwardPointsAsk) /2` in case of trade price conversions.

When the prices are indirect, bid price is used to calculate the ask price and quantity and vice-versa. The formula used is:

If the side is bid:

```
forward price(unrounded) = multiplier /mathUtil.subtract(spot price,
   forward pointsAsk)
```

If the side is ask:

```
forward price(unrounded) = multiplier /mathUtil.subtract(spot price,
   forward pointsBid)
```

Quantity for bid and ask:

```
forward quantity = ((spot quantity/contract size.toFloat())/
   (spot price/multiplier)).round()
```

To round the calculated forward price, `pipSize` and `epsSize` parameters are used in the following way:

For buy side, floor function is used:

```
forward price(rounded) =(mathUtil.add( forward price (unrounded), eps size)
    / pip size).floor().toFloat()* pip size
```

For sell side, ceiling function is used:

```
forward price(rounded) =(mathUtil.subtract( forward price (unrounded),
   eps size) / pip size).ceil().toFloat()* pip size
```

Creating a market data manager

Once the source has been created, call the `startSession` action to get the underlying `MDManagerInterface` which can then be used to query and create specific datastream managers.

```
action startSession(com.apama.session.SessionStart sStr,
   action<integer> onSuccess, action<integer,string> onFailure) {
     com.apama.md.user.MDManagerInterface underlyingMDManager :=
     sourceHelper.getUnderlyingMDMangerForSessionId(
         underlyingSession.getSessionId());
}
```

You can query the `MDManagerInterface` for the availability of required datastream and if available, use the relevant datastream manager to access it. The following is a list of the available actions for the built-in datastream types relevant to FTSC:

- `hasDepth(), createDepthManager()`
- `hasOrderbook(), createOrderbookManager()`
- `hasBBA(), createBBAManager()`
- `hasTrade(), createTradeManager()`

For example, to access the BBA datastream, the following code would be required in order to obtain the relevant manager.

```
if underlyingMDManager.hasBBA() then {
   underlyingBBAIFace := underlyingMDManager.createBBAManager();
   conData.add(localConsts.BBA(),new FTSConnectionData);
}
```

After the relevant datastream managers have been created, call the `connect` action to connect to them.

```
action connect(ConnectDatastream cds, string underlyingSymbolName,
   action<com.apama.md.adapter.ConnectionKey,integer> onConnectSuccess,
   action<com.apama.md.adapter.ConnectionKey,integer,string>
   onConnectFailure);
```

Depending on the stream type of the underlying connection, a connection to the appropriate datastream manager is established. For example, to check for stream type BBA and connect to BBA manager:

```
if cds.streamType = localConsts.BBA() then {
   underlyingBBAIFace.connect(underlyingSymbol, underlyingConnectionParams,
      handleUnderlyingSessionError,
      onUnderlyingConSuccessBBA,onUnderlyingConFailure);
}
```

After successful connection to the underlying datastream manager, call the `createConnection()` action to setup data listeners and register the connection.

```
action createConnection(com.apama.md.adapter.ConnectDatastream cds,
   com.apama.md.adapter.ConnectionKey underlyingConnKey);
```

Set up data listeners

Once the connection to the datastream manager has been established, an application should set up listeners appropriate for the underlying stream type. For example, for the stream type BBA:

```
if streamType = localConsts.BBA() then {
   com.apama.md.WrappedBBA wrappedBBA;
   listener l1 := on com.apama.md.WrappedBBA(symbol=underlyingSymbol,
      sessionId=uSessionId):wrappedBBA {
      for conId in
         conData[streamType].pricingParams[wrappedBBA.symbol].keys() {
            transformSendBBA( wrappedBBA.bba, conId );
         }
      }
   com.apama.md.BBA bbaInst;
      listener l2 := on all com.apama.md.BBA(symbol=underlyingSymbol,
         sessionId=uSessionId):bbaInst{
         if enableTimestamps then {
            bbaInst.__timestamps.add(
               constants.AP_TIMESTAMP_DOWNSTREAM_CORRELATOR_FTSC_ENTRY,
               timeMgr.getMicroTime() );
            }
         for conId in
            conData[streamType].pricingParams[bbaInst.symbol].keys() {
               transformSendBBA(bbaInst, conId );
         }
      }
   conData[streamType].dataListeners[underlyingSymbol].append(l1);
   conData[streamType].dataListeners[underlyingSymbol].append(l2);
}
```

Defining the connection parameters

Define the connection parameters that need to be passed while establishing a connection.

For example:

```
        params := new com.apama.md.adapter.FSConvertorParams;
        params.isDirect              := true;
        params.multiplier             := 10000.0;
        params.forwardPointsBid       := 5.0;
        params.forwardPointsAsk       := 5.3;
        params.forwardPointsDivisor   := 10000.0;
        params.contractSize           := 100000;
        params.pipSize                := 0.0002;
        params.epsSize                := 0.0001;
        params.spotPricePrecision     := 4.0;
```

Connecting to a datastream

Once the connection parameters have been defined, you can connect and pass the defined connection parameters.

```
com.apama.md.adapter.FTSCSession FTSCInst;
FTSCInst.create(mainCtx,"Test",uSession, uSession.getConfig(),
   params, onFTSCSuccess, onFTSCFailure);
```

Disconnecting from a datastream

When you are no longer interested in data for a symbol on a specific datastream, disconnect from the datastream by calling the `disconnect()` action on the relevant manager.

```
underlyingBBAIFace.disconnect(
   currConnData.underlyingConnectedSymbols[ symbol ],
   onUnderlyingDisconSuccess, onUnderlyingDisconFailure);
```

Price spreader/skewer

The Price Spreader can spread and skew prices and re-publish them within the correlator. Configurable parameters for spreading and skewing come from an underlying session, which may be either an IAF adapter or another synthetic datasource.

The Price Spreader supports connections to sessions which have the following datastream capabilities, and will attempt to connect to them in the following order of priority:

1. Aggregated Book
2. Depth
3. Best Bid Ask

Spread/skew configuration

The Price Spreader/Skewer uses the following configurable parameters:

Parameter	Type	Allowable values	Default value	Mandatory	Description
`id`	`string`	Any string	None	No	Uniquely identifies a set of spreader parameters.
`skewDirection`	`string`	`bid`, `ask` and `off`	None	Yes	Specifies the direction towards which the prices are to be skewed. An error returns the message: `Skew direction 'given value' is not an option.`
`spread Percentage`	`string`	positive `float`, `NORMAL`, `MODERATE` and `HIGH`	None	No	Percentage by which the prices are to be skewed or to specify the level of volatility. An error returns the message: `Spread percentage 'given value' is not valid,`

Parameter	Type	Allowable values	Default value	Mandatory	Description
					`Spreader instance not created.`
`spreadPercent NormalVolatility`	`float`	`float`	None	No	Percentage spread to be applied to the prices at `NORMAL` volatility level. An error returns the message: `Spread percentage for NORMAL volatility 'given value' is not valid.`
`spreadPercent Moderate Volatility`	`float`	`float`	None	No	Percentage spread to be applied to the prices at `MODERATE` volatility level. An error returns the message: `Spread percentage for MODERATE volatility 'given value' is not valid.`
`spreadPercent HighVolatility`	`float`	`float`	None	No	Percentage spread to be applied to the prices at `HIGH` volatility level. An error returns the message: `Spread percentage for HIGH volatility 'given value' is not valid.`
`spreadMinQty Limit`	`integer`	positive `integer`	`0`	No	Minimum quantity of liquidity required for bid and ask for production of a valid price. If the minimum liquidity is not achieved a zero value BBA is produced. Cannot be negative or greater than `spreadMaxQtyLimit`. An error returns the message: `Minimum`

Parameter	Type	Allowable values	Default value	Mandatory	Description
					quantity for spreading bid/ask prices cannot be greater than the maximum value specified or Minimum quantity for spreading bid/ask cannot be a negative.
spreadMaxQty Limit	integer	positive integer	0	No	Maximum quantity of liquidity to be used for a valid price. An error returns the message: Maximum quantity for spreading bid/asks cannot be a negative value.
skewPercentage	string	positive float, 0 NORMAL, MODERATE and HIGH		No	Specifies the percentage by which prices are to be skewed or the level of volatility. An error returns the message: Skew percentage 'given value' is not valid, Spreader instance not created.
skewPercent NormalVolatility	float	positive float	None	No	Percentage skew to be applied to prices at NORMAL volatility level. An error returns the message: Skew percentage for NORMAL volatility 'given value' is not valid.
skewPercent Moderate Volatility	float	positive float	None	No	Percentage skew to be applied to prices at MODERATE volatility level. An error returns the message: Skew percentage for MODERATE volatility

Parameter	Type	Allowable values	Default value	Mandatory	Description
					`'given value' is not valid.`
`skewPercent HighVolatility`	`float`	positive `float`	None	No	Percentage skew to be applied to prices at `HIGH` volatility level. An error returns the message: `Skew percentage for HIGH volatility 'given value' is not valid.`
`pubMode`	`string`	`TOP_OF_BOOK`, `VWAP`	None	Yes	The spreads calculated are either based on the best prices (top of the book) or the VWAP (Volume Weighted Average Price) for a give volume. An error returns the message: `Publication mode 'given mode' is not an option.`
`vwapVol`	`integer`	`0` or greater than `0`, in case of publication mode VWAP, the value should be greater than `0`	`0`	No	Volume used to calculate the VWAP when the publication mode is VWAP. An error returns the message: `VWAP volume 'given value' must not be negative` or `Negative or zero vwap volume is not valid, Spreader instance not created` – message generated in case publication mode is specified as VWAP and the `vwapVol` value provided is negative or 0.
`spreadAdjuster`	`integer`	`integer`	None	No	Spread adjuster, used to modify spread percentage.

Parameter	Type	Allowable values	Default value	Mandatory	Description
`skewAdjuster`	`integer`	`integer`	None	No	Skew adjuster, used to modify skew percentage.
`minBBASpread`	`float`	`0` or greater than `0`	None	No	Minimum spread that can be applied to the prices. An error returns the message: `Negative minimum BBA spread 'given value' is not valid.`
`decimalPlaces`	`integer`	`integer`	`0`	No	Decimal places used for rounding prices. An error returns the message: `Negative decimal places 'given value' is not valid.`
`publishPrices`	`boolean`	`true`, `false`	`true`	No	Flag to enable or disable publishing of prices.
`acceptOrders`	`boolean`	`true`, `false`	`false`	No	Flag to enable acceptance or rejection of client orders for the instrument.

Calculating spread and skew prices

This section describes the formulas used to spread and skew incoming prices. The spreader can be used in two different modes of operation, Top of Book or VWAP.

Top of Book

To calculate using Top of the Book mode, follow these steps:

1. Check that the bid and ask price are not `0`, if yes then the price and the quantity for both the sides is set to `0`.
2. To spread direction towards bid:
    a. Check whether bid and ask quantities are greater than the `spreadMinQtyLimit`. If either is not, the price and quantity for both the sides are set to `0`, if both are greater than the `spreadMinQtyLimit`, the prices are spread.

b. Spread the prices:

```
spread bid price = mathUtil.subtract(bid price,
   ((mathUtil.subtract(ask price,bid price)*(spread percentage/100))/2)
```

```
spread ask price = mathUtil.add(ask price,(mathUtil.subtract
   (ask price,bid price)*(spread percentage/100))/2)
```

c. Check whether bid and ask quantities exceed `spreadMaxQtyLimit`. If yes, the exceeding quantity is set to the `spreadMaxQuantityLimit` parameter value.

3. To skew the prices towards bid:

```
skewed bid price = mathUtil.subtract(spread bid price,
   mathUtil.subtract(spread ask price, spread bid price )
   *skew percentage /100)
```

```
skewed ask price = mathUtil.substract(spread ask price,
   mathUtil.subtract(spread ask price, spread bid price )
   *skew percentage /100)
```

4. To spread direction towards ask:

a. Check whether bid and ask quantities are greater than the `spreadMinQtyLimit`. If either of them is not, the price and quantity for both the sides are set to 0, if both are greater than the `spreadMinQtyLimit`, the prices are spread.

b. Spread the prices:

```
spread bid price =
   mathUtil.subtract(bid price,(mathUtil.subtract(ask price,
   bid price)*spread percentage /100)/2
```

```
spread ask price =
   mathUtil.add(ask price,(mathUtil.subtract(ask price,
   bid price)*spread percentage/100)/2
```

c. Check that the bid and ask quantities do not exceed `spreadMaxQtyLimit`. If yes, the exceeding quantity is set to the `spreadMaxQtyLimit` parameter value.

5. To skew the prices towards bid:

```
skewed bid price = mathUtil.add(spread bid price,
  mathUtil.subtract(spread ask price,spread bid price)*skew percentage/ 100)
```

```
skewed ask price = mathUtil.add (spread ask price,
  mathUtil.subtract(spread ask price,spread bid price)*skew percentage/ 100)
```

6. Skew direction is set off:

a. Check whether bid and ask quantities are greater than the `spreadMinQtyLimit`. If either of them is not, the price and quantity for both the sides are set to 0, if both the quantities are greater than the `spreadMinQtyLimit` then the prices are spread.

b. Spread the price as following :

```
spreaded bid price =
   mathUtil.subtract(bid price, (mathUtil.subtract(ask price,
```

```
   bid price)*spread percentage /100)/2)
```

```
spreaded ask price =
   mathUtil.add(ask price, (mathUtil.subtract(ask price,
   bid price)*spread percentage/100)/2)
```

c. Check that the bid and ask quantities do not exceed `spreadMaxQtyLimit`. If they do, the exceeding quantity is set to the `spreadMaxQtyLimit` parameter value.

7. If calculated spreaded bid and ask prices are greater than `0` adjust the prices using the `spreadAdjuster` and the `skewAdjuster` parameters:

```
Bid price = spreaded Bid price -
  (0.00001 * 5.0 * spreadAdjuster.toFloat())
```

```
Ask price = spreaded Ask price +
  (0.00001 * 5.0 * spreadAdjuster.toFloat())
```

```
Bid price = spreaded Bid price +
   (0.00001 * 5.0 * skewAdjuster.toFloat())
```

```
Ask price =
   spreaded Ask price + (0.00001 * 5.0 * skewAdjuster.toFloat())
```

8. If the value of the parameter `minBBASpread` is not `0` then check that the applied spread to the bid and ask prices is appropriate.

```
spread = Ask price - Bid price
minSpread = minBBASpread / base.pow(decimalPlaces.toFloat()
```

Now if the `minSpread` and spread is greater than `0` and spread is less than the minSpread then the bid and ask prices are calculated as following:

```
Bid price = (Bid Price + Ask price)/2 - (minSpread /2)
Ask price = (Bid Price + Ask price)/2 + (minSpread /2)
```

9. Finally, adjust the bid and ask prices:

```
Bid price = round(10.0.pow(-1.0 * decimalPlaces.toFloat()), true, Bid price)
Ask price = round(10.0.pow(-1.0 * decimalPlaces.toFloat()), false, Ask price)
```

Reverting Top of the Book

Reverting Prices from the Spreaded/Skewed Prices

Spread and/or skewed prices can be reverted back to original values using the parameters provided. The only mode which supports reverting the prices is Top of the Book and the prices cannot be reverted if either the bid or the ask price is `0`.

The calculations are as follows:

1. Skew direction is towards bid :

```
unskewed bid price =  mathUtil.add(skewed bid price, mathUtil.subtract
   ( skewed ask price, skewed bid price*skew percentage/100)
```

```
unskewed ask price = mathUtil.add(skewed ask price, mathUtil.subtract
```

```
   ( skewed bid price, skewed ask price*skew percentage/100)
```

2. Skew direction is towards ask:

```
unskewed bid price = mathUtil.subtract(skewed bid price,
   mathUtil.subtract(skewed ask price, skewed bid price)*
   skew percentage/100)
```

```
unskewed ask price = mathUtil.subtract(skewed ask price,
   mathUtil.subtract(skewed bid price,
   skewed ask price*skew percentage/100)
```

3. Revert spread:

```
Original bid price = mathUtil.add(unskewed bid price,
   mathUtil.subtract(unskewed ask price, unskewed bid price) /
   mathUtil.add(1.0, spread percentage/100)*spread percentage/200.0)
```

```
Original ask price = mathUtil.subtract(unskewed ask price,
   mathUtil.subtract (unskewed ask price, unskewed bid price) /
   mathUtil.add(1.0 , spread percentage/100)* spread percentage/200.0)
```

VWAP

To spread or skew using VWAP mode, follow these steps:

1. Iterate over the bid side and keep a counter for the quantity and if the total quantity is less than the `vwapVol` then:

```
mathUtil.add (Total bid price, quantity.toFloat() * bid price)
```

   When the total quantity becomes greater than the `vwapVol`, break the loop and the total bid price is calculated as:

```
mathUtil.add(Total bid price, mathUtil.subtract(bid quantity,
   mathUtil.subtract(quantity counter, VwapVol)).toFloat()*
   bid price)
```

2. If the available liquidity for the bid side is less than the specified VWAP volume, the VWAP price and quantity is set to `0`. Otherwise, the VWAP bid price is calculated as:

```
VWAP bid price = Total bid price / vwapVol.toFloat()
```

3. Iterate over the ask side and keep a counter for the quantity and if the total quantity is less than the `vwapVol`, then:

```
Total ask price = quantity.toFloat() * ask price
```

   When the total quantity becomes greater than the `vwapVol`, break the loop and the total ask price is calculated as:

```
mathUtil.add(Total ask price, (mathUtil.subtract(ask quantity,
   (mathUtil.subtract(quantity counter, VwapVol)).toFloat()* ask price)
```

4. If the available liquidity for the ask side is less than the specified VWAP volume then the VWAP the price and quantity is set to `0` otherwise the VWAP ask price is calculated as:

```
VWAP ask price = Total ask price / vwapVol.toFloat()
```

5. Check that the bid and ask prices are not `0`, if yes then the price and quantity for both the sides is set to `0`.

6. Skew direction towards bid:

   a. Check the bid and ask quantities are greater than the `spreadMinQtyLimit`. If either of them is not then the price and quantity for both the sides are set to `0`, if both the quantities are greater than the `spreadMinQtyLimit` then the prices are spread.

   b. Spread the prices as following :

```
spread bid price = mathUtil.subtract(bid price,
   (mathUtil.subtract(ask price, bid price)
   *spread percentage /100)/2)
```

```
spread ask price = mathUtil.add(ask price,
   (mathUtil.subtract(ask price, bid price)*spread
   percentage/100)/2)
```

   c. Check the bid and ask quantities do not exceed `spreadMaxQtyLimit`. If yes, the exceeding quantity is set to the `spreadMaxQtyLimit` parameter value.

7. Skew the prices towards bid:

```
Skewed bid price = mathUtil.subtract(spread bid price,
   (mathUtil.subtract(spread ask price, spread bid price )*
   skew percentage / 100)
```

```
Skewed ask price = mathUtil.subtract(spread ask price,
   (mathUtil.subtract(spread ask price, spread bid price )*
   skew percentage / 100)
```

8. Skew direction towards ask:

   a. Check the bid and ask quantities are greater than the `spreadMinQtyLimit`. If either of them is not then the price and quantity for both the sides are set to 0, if both the quantities are greater than the `spreadMinQtyLimit` then the prices are spread.

   b. Spread the price as following:

```
spread bid price = mathUtil.subtract(
        bid price, (mathUtil.subtract(ask price, bid price)
*spread percentage/100)/2)
```

```
spread ask price = mathUtil.add(ask price,(mathUtil.subtract(
        ask price, bid price)*spread percentage/100)/2
```

   c. Check that the bid and ask quantities do not exceed `spreadMaxQtyLimit`. If yes, the exceeding quantity is set to the `spreadMaxQtyLimit` parameter value.

9. Skew the prices toward ask:

```
skewed bid price = mathUtil.add(spread bid price, mathUtil.subtract(
   spread ask price, spread bid price)*skew percentage/100)
```

b. skewed ask price =

```
mathUtil.add(spread ask price, mathUtil.subtract(
   spread ask price, spread bid price)*skew percentage/100)
```

10. Skew direction is set off:

a. Check the bid and ask quantities are greater than the `spreadMinQtyLimit`. If either of them is not then the price and quantity for both the sides are set to 0, if both the quantities are greater than the `spreadMinQtyLimit` then the prices are spread.

b. Spread the price as following :

```
spread bid price = mathUtil.subtract(bid price,
   (mathUtil.subtract(ask price, bid price)*spread
   percentage/100)/2)
```

```
spread ask price = mathUtil.add(ask price,
   (mathUtil.subtract(ask price, bid price)*spread
   percentage/100)/2)
```

c. Check that the bid and ask quantities do not exceed `spreadMaxQtyLimit`. If they do then the exceeding quantity is set to the `spreadMaxQtyLimit` parameter value.

Price spreader example

Once the source has been created, call the `startSession()` action to get the underlying `MDManagerInterface` which can then be used to query and create specific datastream managers.

```
action startSession(com.apama.session.SessionStart sStr,
   action<integer> onSuccess,
   action<integer,string> onFailure) {
      sessionStarted := true;
      com.apama.md.user.MDManagerInterface underlyingMDManager :=
         sourceHelper.getUnderlyingMDMangerForSessionId(
         underlyingSession.getSessionId());
}
```

You can query the `MDManagerInterface` for the availability of required datastreams and if available, use the relevant datastream manager to access that specific datastream. The following is a list of the available actions for the built-in datastream types relevant to the Price Spreader:

- `hasA(),createAggregatedBookManager()`
- `hasDepth(),createDepthManager()`
- `hasBBA(), createBBAManager()`

For example, to access the BBA datastream, the following code would be required in order to obtain the relevant manager.

```
if underlyingMDManager.hasBBA() then {
    underlyingBBAIFace := underlyingMDManager.createBBAManager();
```

```
    underlyingStreamType := localConsts.BBA();
}
```

After the relevant datastream managers have been created, the `connect()` action is called to connect to them.

```
action connect(
   ConnectDatastream cds,
   string underlyingSymbolName,
   action<com.apama.md.adapter.ConnectionKey, integer> onConnectSuccess,
   action<com.apama.md.adapter.ConnectionKey, integer, string> onConnectFailure)
```

Depending on what the stream type is, a connection to the datastream manager is established. For example, to check for stream type depth and connect to the depth manager:

```
if underlyingStreamType = localConsts.DEPTH()
   then {
      underlyingDepthIFace.connect( underlyingSymbolName,
      underlyingConnectionParams, handleUnderlyingSessionError,
      onUnderlyingConSuccess, onUnderlyingConFailure );
}
```

After the successful connection to the underlying datastream manager call the `createConnection()` action to setup callback references and register connection.

```
action createConnection( com.apama.md.adapter.ConnectDatastream cds,
   com.apama.md.adapter.ConnectionKey underlyingConnKey )
```

Defining the connection parameters

Define the connection parameters which need to be passed while establishing a connection. For example:

```
params := new com.apama.md.adapter.SpreaderParams;
params.spreadPercentage         := 2.0;
params.spreadMaxQtyLimit        := 20;
params.skewDirection            := "OFF";
params.skewPercentage           := 5.0;
params.pubMode                  := "TOP_OF_BOOK";
params.vwapVol                  := 0;
```

Connecting to a spreader

Once the connection parameters have been defined, you can connect and pass the defined connection parameters.

```
com.apama.md.adapter.SpreaderSession SpreaderInst;
SpreaderInst.create( mainCtx,"Test",uSession, uSession.getConfig(), params,
   onSpreaderSuccess, onSpreaderFailure);
```

Disconnecting from a spreader

When you are no longer interested in data for a symbol on a specific datastream, disconnect from the datastream by calling the `disconnect()` action on the relevant manager.

```
underlyingBBAIFace.disconnect(
   currConnData.underlyingConnectedSymbols[symbol],
   onUnderlyingDisconSuccess,
   onUnderlyingDisconFailure);
```

Foreign Exchange cross rate service

The Foreign Exchange (FX) Cross Rate Service synthetic datasource, connects to either one or two underlying sessions (which may be either an IAF adapter or another synthetic datasource), and calculates a "synthetic cross" based on two separate currency pairs and re-publishes them within the correlator. For example, use EUR/USD and USD/JPY as a means of publishing the synthetic cross rate of EUR/JPY.

The Synthetic Cross Rate Service supports connections to sessions which have the following datastream capabilities, and will attempt to connect to them in the following order of priority:

1. AggregatedBook
2. Depth
3. Orderbook
4. Best Bid Ask

Cross rate service configuration

The Cross Rate Service supports the following control parameters that should be passed in on a connection:

Parameter	Type	Default value	Mandatory	Description
`legSymbols`	`sequence <string>`	None	Yes	Specifies the set of underlying legs to use for calculating the synthetic cross rate. Only two legs are supported for the calculation.
`legSessions`	`sequence <integer>`	None	Yes	Specifies the set of sessions for the underlying legs to use for calculating the synthetic cross rate. This must exactly match the `legSymbols` sequence, even if the `SessionIds` are the same for all legs. Only two legs are

Parameter	Type	Default value	Mandatory	Description
				supported for the calculations.
`maxQuantityLimit`	`integer`	None	No	Specifies the maximum quantity to publish on both sides of the calculated synthetic cross rate. Note that this published quantity will include all available quantity at a particular price level of the calculated synthetic cross rate, rather than truncating it to the specified limit. The specified limit is inclusive of the quantity to publish.
`maxLevelsLimit`	`integer`	None	No	Specifies the maximum number of levels to publish on both sides of the calculated synthetic cross rate. The specified limit is inclusive of the number of levels to publish.
`LegParams_x`	`string`	None	No	Specifies a stringified `com.apama.session.CtrlParams` object for the underlying leg connection, where `x` is a zero-based index into the sequence of legs to use for the synthetic cross rate calculation.
`enableTimestamps`	`boolean`	`false`	No	Specifies that timestamps will be recorded on receipt of the underlying leg data and at the point of publication, and will be added to the

Parameter	Type	Default value	Mandatory	Description
				`__timestamps` parameter of the event being published. This is useful for performance testing, but it does add a small overhead caused by the extra calls to `getMicroTime()` in the time manager plug-in.

Cross rate calculations

A synthetic cross rate can be calculated from any two FX currency pairs that share a common currency. Consider a trader who wishes to exchange GBP for RMB through the intermediate currency of USD. The currency pairs that will make up the legs of the cross will be GBP/USD, and USD/RMB. As the two currencies are given in different terms, the exchange rate will be determined by multiplying in the cross. For example:

```
GBP/USD0.62039 - 0.62041
USD/RMB0.15170 - 0.15175
```

The quote for the bid price for the exchange GBP/RMB is calculated by multiplying the bid price from leg 1, by the bid price from leg 2:

```
0.62039 x 0.15170 = 0.09411
```

The quote for the ask price is likewise calculated by multiplying the ask price from leg 1 with the ask price from leg 2:

```
0.62041 x 0.15175 = 0.09415
```

Which results in a calculated cross rate for GBP/RMB of:

```
GBP/RMB0.09411 - 0.09415
```

There are 4 different ways in which a cross rate can be calculated from two legs. For example, using USD as a common currency, the following combinations can be used to calculate the EUR/JPY cross rate:

1. EUR/USD + USD/JPY = EUR/JPY

```
output.bidPrice := leg1.bidPrice * leg2.bidPrice
output.askPrice := leg1.askPrice * leg2.askPrice
output.bidQty := (min((leg1.bidPrice * leg1.bidQty),
   (1* leg2.bidQty)) / leg1.bidPrice).floor()
output.bidQty := (min((leg1.askPrice * leg1.askQty),
   (1* leg2.askQty)) / leg1.askPrice).floor()
```

2. EUR/USD + JPY/USD = EUR/JPY

```
output.bidPrice := leg1.bidPrice * (1/leg2.bidPrice)
output.askPrice := leg1.askPrice * (1/leg2.askPrice)
output.bidQty := (min((leg1.bidPrice * leg1.bidQty),
   (leg2.askPrice * leg2.bidQty)) / leg1.bidPrice).floor()
output.bidQty := (min((leg1.askPrice * leg1.askQty),
   (leg2.bidPrice * leg2.askQty)) / leg1.askPrice).floor()
```

3. USD/EUR + USD/JPY = EUR/JPY

```
output.bidPrice := (1/leg1.askPrice) * leg2.bidPrice
output.askPrice := (1/leg1.bidPrice) * leg2.askPrice
output.bidQty := (min(leg1.askQty, leg2.askQty)) / (1/leg1.askPrice).floor()
output.bidQty := (min(leg1.bidQty, leg2.bidQty)) / (1/leg1.bidPrice).floor()
```

4. USD/EUR + JPY/USD = EUR/JPY

```
output.bidPrice := (1/leg1.askPrice) * (1/leg2.bidPrice)
output.askPrice := (1/leg1.bidPrice) * (1/leg2.askPrice)
output.bidQty := (min(leg1.askPrice,
   (leg2.askPrice * leg2.askQty))) / (1/leg1.askPrice).floor()
output.bidQty := (min(leg1.bidPrice,
   (leg2.bidPrice * leg2.bidQty))) / (1/leg1.bidPrice).floor()
```

Creating an instance of the cross rate service

The following code excerpt shows how to create an instance of the synthetic Cross Rate Service using two underlying Sessions (1 and 2):

```
action createCrossRateService() {
  // Get the SessionInfo for the two underlying Sessions we want to use
  // In this case, SessionId 1 and 2
  sequence<com.apama.session.SessionInfo> sources := new
     sequence<com.apama.session.SessionInfo>;
  sources.append( sessionManagerIface.getSessionInfo( 1 );
  sources.append( sessionManagerIface.getSessionInfo( 2 );
  // Now create the Cross Rate Service
  com.apama.md.adapter.CrossRateServiceSession crossRateService := new
     com.apama.md.adapter.CrossRateServiceSession;
  crossRateService.create( mainContext, "MyCrossRateSession", sources,
     new sequence<com.apama.session.SessionConfigParams>,
     xrateCreationSuccess, xrateCreationFailure );
}
```

Connecting to the cross rate service

Making a connection to the Cross Rate Service is done in exactly the same way as any other MDA datasource. The following code excerpt demonstrates connecting to the synthetic cross rate EUR/JPY using two underlying legs, EUR/USD from SessionId 1, and USD/JPY from SessionId 2.

```
// This code excerpt assumes that the Cross Rate Service and a
// MDManagerInterface have already been created
aggManager := mgr.createAggregatedBookManager();
// We want to use "EUR/USD" from Session 1 and "USD/JPY" from Session 2
// for the cross rate calculation
```

```
sequence<string> legSymbols := ["EUR/USD","USD/JPY"];
sequence<integer> legSessions:= [1,2];
com.apama.session.CtrlParams controlParams := new com.apama.session.CtrlParams;
controlParams.addParam("legSymbols",legSymbols);
controlParams.addParam("legSessions", legSessions);
aggManager.connect( "EUR/JPY", controlParams,
   onSessionError, onConnectionSuccess, onConnectionFailure );
```

If the connection was successful, the `AggregatedBookManagerInterface` can be used in the normal way to examine the aggregated book data that is being published by the Cross Rate Service.

Using MDA with legacy market data components

Since many existing adapters and applications have been built using legacy market data components, you might find it necessary to use a hybrid approach. This section describes how the Market Data Architecture can be used together with applications and adapters that use legacy components.

The CMF includes a Market Data Bridge service that allows you to implement applications with all the benefits of the Market Data Architecture, and still use an adapter that uses legacy interfaces.

After the `MDBridge` has been created, the service handles all the connection/communication protocol for both session management and datastream management. The legacy adapter will appear just as if it were an MDA Session. This does add a small overhead to the system latency, as the events need to be translated from the new to the old architecture and back.

Note:
The `MDBridge` only supports single subscriptions. Subscriptions after the data has started to flow will not work.

Creating a Market Data Bridge

There are two ways to create a Market Data Bridge for legacy adapters:

- Market Data Bridge Service – allows you to simply send one configuration event (`com.apama.md.bridge.ConfigMDBridge`) to the main context to create a new instance of a Market Data Bridge.
- Market Data Bridge Interface object – creates an instance of the Market Data Bridge as an event object (`com.apama,md,bridge.MDBridgeInterface`) in an application's monitor. This is useful if you want to use the Market Data Bridge Extensions Interface (`com.apama.md.bridge.MDBridgeExtensionInterface`) to override some of the behavior of the Market Data Bridge in your application.

Regardless of which method an application uses, the same event object provides initial configuration. The `com.apama.md.ConfigMDBridge` event is either sent to the Market Data Bridge Service, or is passed in as a parameter to the Market Data Bridge Interfaces `createBridge()` action. This event specifies the `ServiceId` and `MarketId` of the legacy adapter that is to be bridged, as well as what DataStreams the Market Data Bridge should publish.

The following configuration parameters allow you to alter some of the behavior of the Market Data Bridge:

- `DISABLE_STATUS_SUPPORT` – if `true`, disables the dependency on the underlying adapter publishing `com.apama.statusreport.Status` messages to inform the Market Data Bridge that it is connected. This option also prevents the Market Data Bridge from publishing any `com.apama.statusreport.Status` events that come from the underlying adapter. This is useful for legacy synthetic adapters such as the Exchange Simulator. If this parameter is not defined, or is set to `false`, the Market Data Bridge will check for `com.apama.statusreport.Status` events from the underlying adapter before the session can be started or stopped.

- `SESSIONCONTROL_IGNORES_AVAILABLE` – if `true`, relaxes the dependency on the underlying adapter publishing `com.apama.statusreport.Status` messages to inform the Market Data Bridge that it is connected. Specifically, it means that the Market Data Bridge will respond to Session Control commands such as Start/Stop on the receipt of a relevant `com.apama.statusreport.Status messages`. But will ignore the `available` flag on this message. This is useful for adapters that are slow to start, or that you wish to pend starting them until later. Unlike the `DISABLE_STATUS_SUPPORT` parameter, this option allows the Market Data Bridge to respond to cases where the underlying adapter goes down, and republishes the `com.apama.statusreport.Status` events.

- `CONNECTION_TIMEOUT` – specifies how long (in seconds) the Market Data Bridges will wait for a response from the underlying adapter before it times out and generates an error. By default, this time out period is 20 seconds. If the Market Data Bridge was created with `DISABLE_STATUS_SUPPORT` set to `true`, then this option is ignored. This option can also be configured for the Market Data Bridge on Session Start/Reconfiguration to allow a more dynamic setup.

- `ENABLE_THROTTLING` – enables the built-in throttling support of the Market Data Bridge. The throttling mechanism that is currently supported uses a "chaser-token" design pattern where, on receipt of a market data event, the Market Data Bridge will enqueue a Throttle-Token event to the contexts input queue. All market data events that are already on the queue will be ignore until the Market Data Bridge receives this Throttle- Token event. This simple mechanism allows the Market Data Bridge to effectively self-regulate the rate at which it needs to throttle (IE it will only throttle when input event rates exceed what it can handle in a timely manner).

- `ORDERIDKEY` – defines what the `OrderId` should be when converting the legacy market data events to the MDA Orderbook datastream type. If the parameter defined exists in the `extraParams` dictionary in the legacy market data event, this value will be used for the orderId in the MDA event. If the parameter contains the string `[0-9]` then this will be replaced with the current (1-based) book level when checking the `extraParams` dictionary. For example, setting the `ORDERIDKEY` to `Level[0-9]_OrderId`, would cause the Market Data Bridge to look for `Level1_OrderId` for the Best-Bid/Ask value, and `Level2_OrderId` for the next level. If this parameter was defined as an empty string, or was not provided, the `OrderId` will be created.

- `STATUS_SUBSERVICEID` – specifies the `subServiceId` parameter in the `com.apama.statusreport.Status` events that the Market Data Bridge is listening for from the underlying adapter. This parameter is ignored if the `DISABLE_STATUS_SUPPORT` parameter has been set to `true`. By default, the Market Data Bridge will be looking for an empty string.

- `STATUS_OBJECTID` – specifies the `ObjectId` parameter in the `com.apama.statusreport.Status` events that the Market Data Bridge is listening for from the underlying adapter. This parameter is ignored if the `DISABLE_STATUS_SUPPORT` parameter has been set to `true`. By default, the Market Data Bridge will be looking for `com.apama.statusreport.Status` events with an ObjectId of `Adapter`.

Session configuration

Once the Market Data Bridge has been created, it will register itself with the Session Manager Service, and will use the following information:

- `SessionName` - an amalgamation of the `ServiceId` and `MarketId` that was provided at creation time separated by a colon (for example, `ServiceId:MarketId`).
- `TransportName` - all sessions that use the Market Data Bridge will have the transport name of `MDBridge`.
- `Session Configuration` - an application can use the following session configuration parameters to alter the behavior of the session:
  - `CONNECTION_TIMEOUT` - amount of time (in seconds) the Market Data Bridge will wait for a response from the underlying adapter before it times out and generates an error. By default, this time out period is 20 seconds. If the Market Data Bridge was created with `DISABLE_STATUS_SUPPORT` set to `true`, then this option is ignored.
  - `QUOTEIDKEY` - a key name to look for in the underlying adapters `com.apama.marketdata.Depth` event to use as the `QuoteID` field in order to correctly publish a Quotebook DataStream. By default, the Market Data Bridge will look for the `QuoteID` as the key. This parameter is ignored if the Market Data Bridge was created without defining a Quotebook DataStream in the list of available StreamTypes.
  - `QUOTETTL` - defines the duration before the quote is removed from the book after publication. By default, this parameter is set to 5 seconds. This parameter is ignored if the Market Data Bridge was created without defining a Quotebook DataStream in the list of available stream types.

Market Data Bridge extension interface

The `MDBridgeExtensionInterface` interface offers more control over interaction with a legacy adapter. This interface allows you to define actions that will be called at various key points in the Market Data Bridge. These extension points allow you to modify the legacy Market Data events before they are sent/received by the Market Data Bridge for processing.

These actions are called just before the corresponding event is sent to the underlying adapter. In the case of `depthTransform()` and `tickTransform()`, the action will be called immediately after the Market Data Bridge has received the corresponding event.

All of the actions allow the return of a `Boolean` value. This is used to indicate whether the event should be processed further by the Market Data Bridge. For example, if you decide that an incoming

com.apama.marketdata.Depth event is not valid, and should not be published by the Market Data Bridge, you can return a value of false from that function.

The following actions can be overridden:

Action	Description
subscribeDepth()	Allows you to customize a SubscribeDepth request before it is sent out of the MDBridge.
subscribeTick()	Allows you to customize a SubscribeTick request before it is sent out of the MDBridge.
unsubscribeDepth()	Allows you to customize an UnsubscribeDepth request before it is sent out of the MDBridge.
unsubscribeTick()	Allows you to customize an UnsubscribeTick request before it is sent out of the MDBridge.
depthTransform()	Allows you to modify or ignore the incoming Depth event before it is converted to the MDA event types.
tickTransform()	Allows you to modify or ignore the incoming Tick event before it is converted to the MDA event types.

The legacy application bridge

The CMF also includes a Legacy Application Bridge that allows a legacy application that consumes the older style com.apama.marketdata event types, to connect to adapters that only offer the MDA market data event com.apama.md. The legacy bridge is an EPL layer which translates subscription, unsubscription and market data update events between the two architectures.

The Legacy Application bridge translates:

- com.apama.md.BBA, com.apama.md.D and com.apama.md.O MDA events into com.apama.marketdata.Depth events.
- com.apama.md.T MDA events into com.apama.marketdata.Tick events.

To use the Apama Legacy Application Bridge in a Software AG Designer project, add the Legacy Application Bridge bundle (from APAMA_FOUNDATION_HOME) as a dependency of the project.

If you use the CMF-macros.xml Ant script to start your project, the legacyapplication-bridge-bundle target can be used as a dependency to ensure the correct EPL files are injected into the correlator during start-up.

To enable the Apama Legacy Application Bridge to start translating events for the adapter session, send a configuration event:

```
com.apama.md.user.SessionConfiguration("@TRANSPORT_NAME@",{
  "adapterName":"@TRANSPORT_NAME@",
  "channelId":"@CHANNEL@",
  "SERVICEID":"@SERVICEID@"})
```

Example:

```
com.apama.md.user.SessionConfiguration("ActivTransport",{
  "adapterName":"ActivTransport",
  "channelId":"Activ",
  "SERVICEID":"Activ"})
```

Parameters:

- `TRANSPORT_NAME` - specifies the name of the IAF transport
- `adapterName` - specifies the name of the IAF transport
- `channelId` - specifies the ID of the IAF transport channel
- `SERVICEID` - Specifies the ID of the service for market data.

Data Subscription Translation:

- `com.apama.marketdata.SubscribeTick` events are translated into `com.apama.md.user.Connect` events with the "`streamType`" set to "`TRADE`".
- `com.apama.marketdata.SubscribeDepth` events are translated into `com.apama.md.user.Connect` event. The "`streamType`" is extracted from the "`subscriptionType`" member of the extra params of the `com.apama.marketdata.SubscribeDepth` if present, else defaults to "`MBP`".
- `com.apama.marketdata.UnsubscribeTick` and `com.apama.marketdata.UnsubscribeDepth` are translated into `com.apama.md.user.Disconnect` events with the same rules as the subscriptions above.
- The complete list of the `subscriptionType` are as follows:-
  - `TRADE` - trade subscription
  - `MBO` - market by order, order subscription by order
  - `MBP` - market by price, order subscription by price
  - `BBA` - best bid ask, top of the book subscription

For subscription management, required extra parameters:

- `Exchange` - specifies the session name for the MDA adapter.
- `subscriptionType` - specifies the type of *datastream* such as, `QUOTE` or `TRADE`.

Example tick subscriptions:

```
com.apama.marketdata.SubscribeTick("AdapterService",
  "ActivTransport",
  "MSFT.Q",
  {"subscriptionType":"TRADE","Exchange":"Activ"})
com.apama.marketdata.UnsubscribeTick("AdapterService",
  "ActivTransport",
  "MSFT.Q",
  {"subscriptionType":"TRADE","Exchange":"Activ"})
```

Market by price subscriptions:

```
com.apama.marketdata.SubscribeDepth("AdapterService",
  "ActivTransport",
  "MSFT.Q",
  {"subscriptionType":"MBP","Exchange":"Activ"})
com.apama.marketdata.UnsubscribeDepth("AdapterService",
  "ActivTransport",
  "MSFT.Q",
  {"subscriptionType":"MBP","Exchange":"Activ"})
```

Market by order subscriptions:

```
com.apama.marketdata.SubscribeDepth("AdapterService",
   "ActivTransport","MSFT.Q",{"subscriptionType":"MBO","Exchange":"Activ"})
com.apama.marketdata.UnsubscribeDepth("AdapterService",
  "ActivTransport",
  "MSFT.Q",
  {"subscriptionType":"MBO","Exchange":"Activ"})
```

Best bid ask subscriptions:

```
com.apama.marketdata.SubscribeDepth("AdapterService",
   "ActivTransport",
   "MSFT.Q",
   {"subscriptionType":"BBA","Exchange":"Activ"})
com.apama.marketdata.UnsubscribeDepth("AdapterService",
  "ActivTransport",
  "MSFT.Q",
  {"subscriptionType":"BBA","Exchange":"Activ"})
```

Using low-level MDA interfaces

This section describes how to use low-level MDA interfaces to make queries and connections and obtain the required data from a specific session. For most applications, the MDA subscriber factory and subscriber handler interfaces described in "Overview of using MDA" on page 20 provide all needed functionality. You might want to use low-level MDA interfaces when you want to do the following:

- Build custom subscribers
- Have greater control over the setup of the underlying services
- Continue previous use of synthetic services such as the internal CMF aggregator, crossing service, pricing service

The `SessionManagerFactory` event object is the main starting point for all applications that want to use the low-level MDA interfaces. This provides the application with an interface to query and control the session that it needs to use. The `MDManagerFactory` event object is the starting point for applications to create all of the necessary MDA Manager objects used to query and connect to the specific session datastreams.

Introduction to use of low-level MDA interfaces

Applications can use the low-level MDA interfaces to manage sessions and access datastreams. These interfaces provide the same features as provided by the factory and helper MDA interfaces.

The difference is that the factory and helper interfaces do more default set-up while the low-level interfaces require you to explicitly set up each MDA component.

Sessions are separate connections to an internal or external datasource such as a simulator or an adapter. A datasource can register one or more sessions. For example, a single datasource that supports connections to two exchanges would expose two sessions. Sessions have a unique ID, eliminating the need for unique IDs for the legacy service, market and exchange IDs. Sessions also have self-describing capabilities.

Sessions are automatically registered with the Session Manager by the datasource when they become available. Session Manager Interfaces provide a mechanism to query registered sessions and what they support. Sessions can be started, stopped, and reconfigured or de-registered if not required by the application. Session management usually occurs once at application startup.

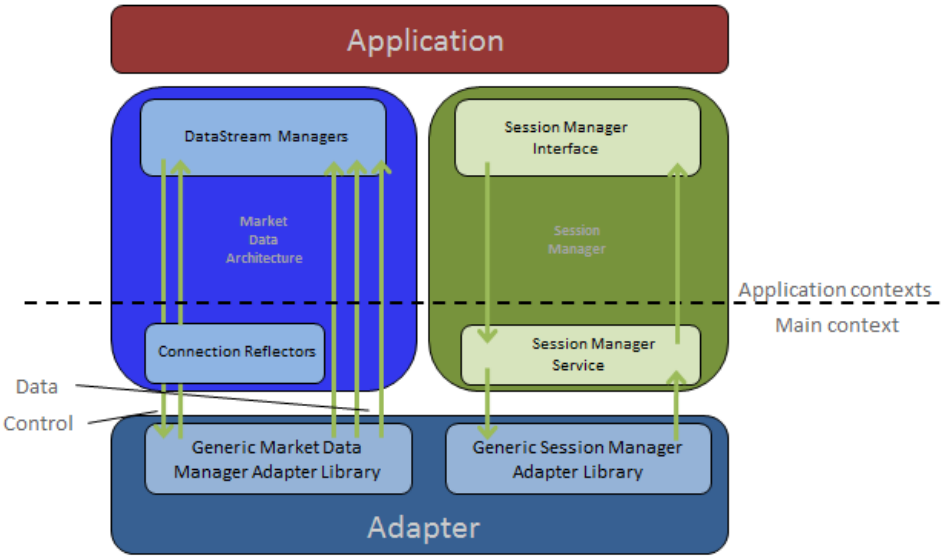
The control flow for using a session includes the following:

- Find a suitable session
- Get list of available registered sessions
- Register callbacks for new sessions being added/updated/removed
- Control a session

  A session can have self-describing control parameters, which can include information such as the name, type, description, allowable values, a default value, and a flag indicating if the parameter is for information only.
- Start a new session with specified control parameters
- Reconfigure an existing session with different control parameters
- Stop an existing session if it is no longer required

Session Management represents an activity separate from that of accessing the data from sessions. An example of the latter is using MDA to get market data from the session. The following figure shows the use of the MDA and sessions in an application. The left arrows on the MDA block illustrate control signals, such as connect and disconnect. The right arrows represent data and responses to control requests. The Generic Market Data Manager adapter library uses channels to send data directly to one or more requesting Manager instances, which may be in one or more contexts. (In release prior to 5.2, this was accomplished by means of data reflectors.)

The arrows in the Session Manager block illustrate control signals. Control signals are also reflected to multiple Session Manager Interfaces if they are created in multiple contexts. This is not shown in the diagram below.



Self-describing datasources

When a datasource registers a session with the Session Manager, it provides the name of the session being registered and defines session capabilities. These capabilities include the types of datastreams supported, and the schema for any supported control parameters. The control parameters schema provides information about each control parameter, such as the name, a description, and the type of the value, whether required or optional.

Datasources use channels to send events directly to requesting contexts.

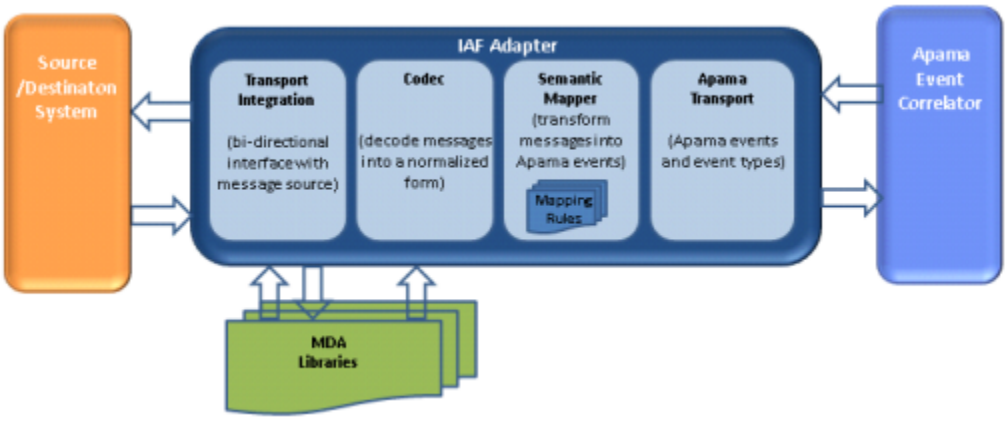
Capability events include the following information:

- Session identifier: a unique identifier for each datasource, provided by the Session Manager
- Client request identifier: a unique identifier for every connection request, allocated by the client layer of the platform
- Datastream name: a unique name distinguishing each datastream type
- Control parameters: the schema of modifiable parameters for connecting to the datastream

Processing data close to the source

Performing the majority of the data processing at the datasource level makes it possible to achieve increased event throughput and decreased event latency. The Integration Adapter Framework (IAF) transport/codec handles complex processing. This allows for more efficient processing and also facilitates the transmission of data in deltas instead of using full data snapshots. In addition to increasing efficiency, this design eliminates datasource-specific service monitors in most cases.

The MDA libraries support the IAF with performance enhancements and filterability; they do not replace it. All common functionality for processing market data from sessions is included in a generic data source library, which can be used by all datasources. The following illustration depicts the MDA relationship to the IAF framework.



Starting a session

First, obtain a `SessionManager` object, which can then be used to query and connect to a specific session.

```
// Create a new Session Manager
(new com.apama.session.SessionManagerFactory).createSessionManager(
  context.current(),
  onCreated);
```

Once the Session Manager has been created, query about the available sessions using one of the following actions:

- `getAllSessionInfo()` - returns a dictionary containing information about all the sessions that are currently registered with the Session Manager.
- `getSessionInfo()` - returns information about a specific session that has been registered with the Session Manager. See the ApamaDoc for information on parameters.

A session must be started before an application can use it. If the session requires configuration parameters, specify them as part of the start action. The configuration schema object returned in the session information object contains the configuration parameters. In the following example, the `sessionID` parameter comes from the `SessionInfo` object that is returned from `getAllSessionInfo()` or `getSessionInfo()`.

```
// This action is called when the Session Manager is created
action onCreated( com.apama.session.SessionManagerInterface sessionIFace ) {

// Now start the session
com.apama.session.SessionConfigParams config :=
new com.apama.session.SessionConfigParams;
sessionIFace.startSession(sessionId,
  config,
  onSessionStartSuccess,
  onSessionStartFailure);
}
```

If you need to reconfigure the session after it has been started, call the following action with the new configuration parameters:

```
// Reconfigure the session
sessionIFace.reconfigureSession(sessionId,
  newConfig,
  onSessionReconfigSuccess,
  onSessionReconfigFailure);
```

Creating a market data manager

First obtain an `MDManager` object, which can then be used to query and connect to a specific session.

```
// This action is called when the Session was successfully started
action onSessionStartSuccess(integer sessionId) {

// Create a new Market Data Manager for this session
(new com.apama.md.user.MDManagerFactory).createMDManager(
   context.current(),
   SessionInfo,
   onCreationSuccess,
   onFailureToCreate);
}
```

Once the manager has been created successfully, you can query the availability of the required datastream and if available, use the relevant datastream manager to access that specific datastream. The following is a list of the available actions for the built-in datastream types:

- `hasBBA(), createBBAManager()`
- `hasTrade(), createTradeManager()`
- `hasDepth(), createDepthManager()`
- `hasOrderbook(), createOrderbookManager()`
- `hasQuote(), createQuoteManager()`
- `hasQuotebook(), createQuotebookManager()`
- `hasEP(), createEPManager()`

Alternatively, you can query session capabilities to see if a specific datastream type is available. This can be done by using the `com.apama.session.SessionInfo` object for the session and calling `sessionInfo.getCapabilities().hasDatastream( streamType)` on its interface. This method means that you can query a session's datastream capabilities quickly without needing to create `MDManagers` first. For example, to access the best-bid-ask (BBA) datastream, the following code would be required in order to obtain the relevant manager.

```
// this action is called if the creation of the MD manager was successful
action onCreationSuccess( com.apama.md.user.MDManagerInterface mdManager ) {
  // Now create a new interface to the BBA Manager
  if mdManager.hasBBA() then {
    BBAManagerInterfacebba bbaManagerIface :=
    mdManager.createBBAManager();
    connect( bbaManagerIface );
  }
```

Connecting to a datastream

After an application obtains a specific datastream manager, the manager can be used to connect and receive the required data from the specified session. The connect action makes a request to connect to a datastream and accepts two callbacks, one to be called on success and the other to be used if the connection fails. Upon a successful connection, the callback function will provide a unique connection key for that connection, a `com.apama.md.adapter.ConnectionKey` object. An application must connect to each symbol of interest. If an attempt is made to connect to the same symbol twice, the datasource implementation determines connection key that will be returned. Generally, a connect with the same control parameters returns the same connection key.

```
// this action is called if the creation of the BBA manager was successful
action connect( BBAManagerInterface bbaIface ) {
// Connect to a BBA Manager for a given symbol
  bbaManagerIface.connect(symbol,
    new com.apama.session.CtrlParams,
    onConnectionSuccess,
    onFailureToConnect);

    . . .
}
```

Connecting to a quotebook datastream

Connecting to a quotebook datastream is similar to connecting to other built-in Apama datastreams. The process differs in the way you provide the information necessary to identify the exact quote you want the adapter to subscribe to connecting to the datasource. After you have found and started your session (see "Starting a session" on page 76) and created a `MDManagerInterface` object (see "Creating a market data manager" on page 50) connect to a quotebook datastream as follows:

1. Using a `MDManagerInterface` object, call the `hasQuotebook()` action to see if the quotebook datastream is available.

2. If the quotebook datastream is available, create a `QuotebookManager` using the `QuotebookInterfaceManager` instance's `createQuotebookManager()` action. For example:

   ```
   // this action is called if the creation of the MD manager was successful
   action onCreationSuccess( com.apama.md.user.MDManagerInterface mdManager ) {
     // Now create a new interface to the Quotebook Manager
     if mdManager.hasQuotebook() then {
       QuotebookManagerInterface quotebookManagerIface :=
         mdManager.createQuotebookManager();
       connect( quotebookManagerIface );
     }
     //...
   }
   ```

3. Connect to the quotebook datastream using the `QuotebookInterfaceManager` instance's `connect()` action. For example:

   ```
   // this action is called if the creation of the Quotebook manager was
   // successful
   ```

```
action connect( QuotebookManagerInterface quotebookIface ) {
  // Set the desired Connect Control Params
  com.apama.session.CtrlParams connParams :=
    new com.apama.session.CtrlParams;
  connParams.addParam("Currency", "USD");
  // Connect to a Quotebook Manager for a given symbol
  quotebookIface.connect(symbol,
      connParams,
      onConnectionSuccess,
      onFailureToConnect);
 //. . .
}
```

The second argument to the `connect()` action contains the various Control Parameters that indicates the quote you want to connect to in the quotebook. For more information on how to specify the Control Parameters, see "Specifying control parameters" on page 79

Specifying control parameters

When subscribing to a quotebook datastream, in order to identify the exact quote you want the adapter to subscribe to, you provide information by passing a `com.apama.session.CtrlParams` object as the second argument to the `QuotebookInterfaceManager` instance's `connect()` action.

Many adapters use standard control parameters. For a list of commonly used standard control parameters, see "Standard control parameters" on page 79. For information about specialized control parameters used by individual adapters, refer to the adapter documentation.

You create the `CtrlParams` object and specify its contents as follows:

1. Create a `com.apama.session.CtrlParams` object, for example:

   ```
   com.apama.session.CtrlParams connParams := new com.apama.session.CtrlParams
   ```

2. Add the names and values of the control parameters to the `CtrlParams` object using the `addParam()` action. For example:

   ```
   connParams.addParam("Currency", "USD");
   connParams.addParam("VolumeBands", "1000000,3000000,5000000,10000000");
   ```

Standard control parameters

The standard control parameters listed in this table are used by many adapters.

Control Parameter Key	Description
`VolumeBands`	A comma separated list of the volumes the adapter should get Quotes for, for example, "1000000,3000000,5000000,10000000".

Control Parameter Key	Description
`Currency`	The three letter Currency Code the Quote prices should be expressed in. If this is not specified, this is set by default to the base currency of the Quote.
`FutSettDate`	The settlement date of the Quote in the format expected by the adapter.
`CFICode`	The CFI Code (Classification of Financial Instruments) six character code. (The default for this can be read in from the adapter configuration file).
`INCLUDE_INDICATIVE_PRICE`	Whether to include indicative quotes (tradeable=`false`). The default is `false` so indicative prices are filtered out.
`<FixTab>`	Any FIX tag integer can be added and will be processed by the adapter and passed on to the exchange, for example, `"6363:"B1"`.

Accessing datastream information

The sample in this section shows how to access the datastream information assuming the connection has been successful. The example shows two methods of accessing the data, one uses an update callback that is registered with the manager object and triggers when any new data is available. These update handlers can be added and removed at will and as many times as required, as shown in the sample.

It is worth noting that in the case of snapshot event-only data, events that include all the information about the current state of the market, such as best-bid-ask (BBA) or trade data, using listeners would seem very reasonable. However, large data structures such as the orderbook and depth events are usually not sent as whole snapshot events for obvious efficiency reasons. Therefore, the manager access functions are generally the best way to obtain the most recent data. These functions collate and cache the data and provide them to the user.

```
// this action is called if the creation of the BBA manager was successful
action onConnectionSuccess(com.apama.md.adapter.ConnectionKey connKey) {
   log "connected BBA for: "+ connKey.getSymbol();

   // 1. Demonstrating call back setup procedure
   // Add an update callback for all BBA if we were successful
   integer updateRef := bbaManagerIface.addUpdateCallback( connKey, onAllBba);

   // 2. Demonstrating access to datastream via listener
   com.apama.md.BBA bbaInst;
     on all com.apama.md.BBA(symbol=connKey.getSymbol()):bbaInst {
              handleBba(bbaInst);
       }

   // Wait a while then disconnect
     on wait(30.0) {
       boolean success := bbaManagerIface.removeCallback(
         connKey,
```

```
        updateRef );
      bbaManagerIface.disconnect(connKey,
        onDisconnectSuccess,
        onDisconnectFailure);
    }
  }
}
```

The following actions are used in the code above to handle the BBA data. Note the difference in the input argument of the two actions.

```
// This callback is called whenever we get a new BBA using the callback
// mechanism
action onAllBba( com.apama.md.client.CurrentBBAInterface currBba ) {
  log "Got BBA using Callback- " at INFO;
  log " for symbol: " + symbol at INFO;
  log " Bid Quantity: " + currBba.getBidQty().toString() at INFO;
  log " Bid Price: " + currBba.getBidPrice().toString() at INFO;
  log " Ask Quantity: " + currBba.getAskQty().toString() at INFO;
  log " Ask Price: " + currBba.getAskPrice().toString() at INFO;
  log " With EP: " + currBba.getEPValuesInterface().getRaw().toString() at INFO;
}

// This callback is called whenever we get a new BBA using listeners
action handleBba( com.apama.md.Bba currBba ) {
  log "Got BBA using Listener - " at INFO;
  log " for symbol: " + symbol at INFO;
  log " Bid Quantity: " + currBba.bidQuantity.toString() at INFO;
  log " Bid Price: " + currBba.bidPrice.toString() at INFO;
  log " Ask Quantity: " + currBba.askQuantity.toString() at INFO;
  log " Ask Price: " + currBba.askPrice.toString() at INFO;
}
```

The following shows an example of how to select the datastream transfer mode.

```
// Modes for connecting to an Orderbook datastream
com.apama.session.CtrlParams conParams := new com.apama.session.CtrlParams;
// Snapshot mode:
conParams.addParam(TRANSFER_MODE, SNAPSHOT_ONLY);
// Atomic Delta mode:
conParams.addParam(TRANSFER_MODE, ATOMIC_DELTA);
// Compound Delta mode:
conParams.addParam(TRANSFER_MODE, COMPOUND_DELTA);
orderbokManagerIface.connect(symbol, conParams, onSuccess, OnFailure);
```

Accessing quotebook datastream information

This section contains details about accessing quotebook datastream information along with an example.

- The prices in a quotebook use the Apama `decimal` data type, introduced in the Apama 5.0 release.
- The quotebook can be accessed and iterated over by both Price ordering (the `*PriceLevel()` actions) and by Quantity ordering (the `*QuantityLevel()` actions.
- You can use the following helper actions:

- `getBestPriceForQuantity()` - Get the best price for a quantity with a matching quote in the book.
- `getPriceForQuantity()` - Get the worst price for any quantity by accumulating the quote quantities.
- `getVWAP()` - Calculate the VWAP for the whole book.
- `getVWAPForQuantities()` - Calculate the VWAP for a given sequence of quantities.

The following example uses the `getVWAP()` helper action and then iterates over the datastream using the `getQuotesByQuantityLevel()` action.

```
action onAllQuotes(com.apama.md.client.CurrentQuotebookInterface
 currQuotebook) {
  log "Got Quotebook using Callback -" at INFO;
  log "  for symbol: " + currQuotebook.getSymbol() at INFO;
  log "  Ask VWAP: " + currQuotebook.getAsks().getVWAP().toString() at INFO;
  log "  Bid VWAP: " + currQuotebook.getBids().getVWAP().toString() at INFO;

  // Loop through all the Ask quotes in Quantity order
  integer maxLevels := currQuotebook.getAsks().getMaxKnownQuoteQuantityLevel();
  integer level := 0;
  while (level < maxLevels) {
    sequence<com.apama.md.client.QuoteEntryInterface> quotes :=
        currQuotebook.getAsks().getQuotesByQuantityLevel(level);
    com.apama.md.client.QuoteEntryInterface quote;

    log "  Asks for Level: " + level.toString() at INFO;

    for quote in quotes {
      log "    Quantity: " + quote.getQuantity().toString() at INFO;
      log "    Price:    " + quote.getPrice().toString() at INFO;
      log "    Party:    " + quote.getParty().toString() at INFO;
      log "    QuoteId:  " + quote.getQuoteId().toString() at INFO;
      log "    RequestId:" + quote.getRequestId().toString() at INFO;
      log "    Tradeable:" + quote.getTradeable().toString() at INFO;
    }

    level := level + 1;
  }
}
```

Disconnecting from a datastream

When an application is no longer interested in data for a symbol on a specific datastream, it can disconnect from the datastream by calling the `disconnect` action on the relevant manager. The specified `connKey` must be the same `com.apama.md.adapter.ConnectionKey` returned by the connection success callback.

```
bbaManagerIface.disconnect(connKey,
  onDisconnectSuccess,
  onDisconnectFailure);
```

Stopping a session

When a specific session is temporarily not of interest, an application can stop it as follows:

```
// Stop the session
sessionIFace.stopSession(sessionId,
  onSessionStopSuccess,
  onSessionStopFailure );
```

If you are no longer interested in the session, and will not require it for the rest of the lifetime of the application, deregister it from the list of available sessions. This will remove all information about that session from the Session Manager.

```
// Deregister the session
sessionIFace.deregisterSession(sessionId,
  onSessionDeregisterSuccess,
  onSessionDeregisterFailure );
```

Precautions when spawning to contexts

You should take care when spawning monitors to a new context if you are using the new Market Data API and `SessionManagerInterface`. Spawning a monitor to a new context that has already created a `SessionManagerInterface` will cause problems. On creation of the `SessionManagerInterface`, the current context is registered with the Session Manager Service. Spawning a monitor that contains the `SessionManagerInterface` will make a new copy of the interface, but the Session Manager Service will not communicate with it.

If you need to spawn a monitor to a new context and still require the `SessionManagerInterface` in that context, a new `SessionManagerInterface` should be created for each new context.

Similarly, once a datastream manager interface, such as `BBAManagerInterface`, or `TradeManagerInterface`, has been created, the monitor it was created in should not be spawned to a new context. If you need to spawn their monitor to a new context and still require the datastream manager interface in that context, a new datastream manager interface should be created for each new context.

4 Order Management

- Order state containers 86
- Exchange simulation platform 89
- Risk firewall 90
- Smart Order Router 175
- Order bridging services 178
- Trade reporting services 179

The bundles in the order management category include components and services to handle orders for a financial application. The components and services in order management-related bundles contain functionality for managing and monitoring orders created by CMF-based applications. These components and services should be used by any application that needs to:

- Send new orders to an external order routing adapter or an internal order routing venue.
- Manage (for example, amend and cancel) orders already submitted to a venue.
- Monitor the status of orders, including those submitted by other applications.
- Transparently "bridge" order management events between multiple correlators.
- Protect against invalid orders being placed by using the Risk Firewall.
- Allow strategies to place orders for multiple venues using the SOR framework.
- Simulate an exchange.

Order state containers

The order state container components provide an action-based interface to the standard `com.apama.oms.*` event interface for order management. For a CMF-based application, order state containers implement and enforce well-defined semantics on top of the event protocol. The application does not need to directly route and listen for `com.apama.oms.*` events. This means that all users of the containers agree on the meaning of, for example, a specific `OrderUpdate` event. This level of agreement is very difficult to achieve if each application constructs and interprets the events according to its own rules.

There are three different order state container objects in the `com.apama.oms` package:

1. The `OrderPublisherStateContainer` allows applications to create new orders, submit them to an execution venue and manage them after successful submission.
2. The `OrderReceiverStateContainer` allows applications to act as execution venues, for example, to receive new orders and provide updates on the status of those orders back to the submitting applications.
3. The `OrderMonitorStateContainer` allows applications to track the state of other orders in the same correlator.

In a typical algorithmic trading application, where the application strategy is submitting orders to an external venue via an adapter, the application will use an `OrderPublisherStateContainer` to encapsulate each order it submits, while the adapter will use an `OrderReceiverStateContainer` to represent the state of each order on the external venue and to report updates back to the application. In the case of an application accepting client orders, the application itself should use an `OrderReceiverStateContainer` while the adapter delivering the orders to the application is likely to use an `OrderPublisherStateContainer` for each client order. The `OrderMonitorStateContainer` is typically used by services to track the state of all the orders in the system, including those that were submitted through other applications.

As the APIs provided by the order state container objects are all quite similar, this section focuses on the `OrderPublisherStateContainer` and mentions significant differences to the other containers when these are relevant.

The `OrderPublisherStateContainer`

The `OrderPublisherStateContainer` object can be used to submit new orders, amend or cancel submitted orders, and query order state. To create a new order, an application should first construct a new `com.apama.oms.OrderPublisherStateContainer` then call either the `submit()` or `submitQuick()` action. Orders can also be submitted through the risk firewall. The `submit()` action takes a fully-initialized `com.apama.oms.NewOrder` event as its argument whereas `submitQuick()` avoids the need to construct the `NewOrder` event but only allows a limited set of order parameters to be controlled. The following example how to submit a new order using the order publisher state container. This example uses the `submitQuick()` action to create a new market order to sell 1 million EUR/USD at a price of 1.20365, and submit the order to the venue with the `serviceId` of `FIX` and `marketId` of `CNX`:

```
action sendOrder() {
  com.apama.oms.OrderEventConstants const
     :=new com.apama.oms.OrderEventConstants;
  com.apama.oms.OrderPublisherStateContainer pub
     := new com.apama.oms.OrderPublisherStateContainer;
  pub.submitQuick("EUR/USD", "FIX", "CNX", "", const.MARKET_ORDER(),
     const.SELL_SIDE(), 1.20365, 10000000, false);

  // Monitor changes to the order
  pub.addUpdateListener("myCallback", monitorHandler);
}

// This action will be called every time the order state changes
action monitorHandler(integer id, com.apama.oms.OrderState state) {
  if( state.isFinal() ) then {
     log "Order Complete!";
  }
}
```

Applications can register a callback with the publisher which will be called when the state of the order changes. A string is used to uniquely identify an update callback, so that it may be removed at a later date. The callback that you define will contain the unique identifier of the publisher that has called the action, and the current state of the order. The unique id will match that obtained by calling `getPubId()` on the publisher, so an application can identify the publisher that sent the update. This object encapsulates the complete state of the order and is used by all of the order state containers.

The `OrderState` object provides a large set of actions for querying the internal state of the order being handled. These are described in detail in the ApamaDoc for the `OrderState` object. In order to determine what the current state of the order is in, and to determine when the order makes a state transition, two sets of query actions are provided. The `is*()` and `just*()` actions allow an application to query the status of an order, for example, whether the order is "final" or not. The results of the `is*()` actions indicate whether or not the order is currently in that state, so `isCancelled()` will return `true` if the order has been canceled. The `just*()` actions are similar but their results indicate whether the queried status was `true` only for the most recent update processed by the container. For example, `justAcknowledged()` will return `true` for the update immediately

after the order is acknowledged, but not on any subsequent updates, whereas `isAcknowledged()` will always return `true` after the order has been acknowledged.

Many of the `OrderState` actions of the `OrderPublisherStateContainer` are mirrored on the container itself for convenience.

The `OrderReceiverStateContainer`

The `OrderReceiverStateContainer` differs mainly in that the application is responsible for setting the state of the order rather than querying it. An instance of the `OrderReceiverStateContainer` object should be created whenever a service receives a `NewOrder` event with appropriate addressing information. In the example below, a new order receiver state container object is created for every new order that is received and `receiveOrder` is called to start handling the state of that order. The quantity of the order is then checked for a maximum limit, and the order is either filled or rejected accordingly.

```
action handleNewOrders() {
  com.apama.oms.NewOrder newOrder;
  on all com.apama.oms.NewOrder(serviceId="FIX", marketId="CNX"):newOrder {
    com.apama.oms.OrderReceiverStateContainer recv :=
       new com.apama.oms.OrderReceiverStateContainer;
    recv.receiveOrder(newOrder, false);

    // Test if the order received was valid
    if( recv.getQty() < 1000 ) then {
       // Order was valid, so acknowledge the order and fill it
       recv.acknowledge( integer.getUnique() );
       recv.fill( recv.getPrice(), recv.getQuantity() );
    } else {
       // Order was invalid, reject it
       recv.reject( "Order rejected! Quantity requested was more than 1000!" );
    }
  }
}
```

There are a number of actions that can be called on the order receiver to change the state of the order. These include actions to handle amends, cancels, rejections, fills, etc. State-changing actions such as `acknowledge()` will cause the container to route an `OrderUpdate` event. If the application used an `OrderPublisherStateContainer` object to submit the order, any update handler callbacks it registered will be called. The `OrderReceiverStateContainer` also has the same set of `is*()` and `just*()` actions as the `OrderPublisherStateContainer` object that can be used to query the current state of the order being managed.

The `OrderMonitorStateContainer`

The `OrderMonitorStateContainer` is very similar to the both the `OrderPublisherStateContainer` and the `OrderReceiverStateContainer` but does not have the ability to actually submit, amend or cancel orders. The `OrderMonitorStateContainer` also has `is*` and `just*` actions that can be used to query the current state of the order being monitored.

In the example below, a new order monitor state container object is created for every new order that is received and `monitorOrder()` is called to start monitoring the state of that order. The `addUpdateListener()` action is called to register a callback action with the monitor state container

that will be called whenever the state of the order changes. A string is used to uniquely identify an update callback, so that it may be removed at a later date. The callback that the user defines will contain the unique identifier of the order monitor that has called the action, and the current state of the order.

```
com.apama.oms.NewOrder newOrder;
on all com.apama.oms.NewOrder(serviceId="FIX", marketId="CNX"):newOrder {
   com.apama.oms.OrderMonitorStateContainer mon :=new
      com.apama.oms.OrderMonitorStateContainer;
   mon.monitorOrder(newOrder,  false);
   mon.addUpdateListener("myCallback", monitorHandler);
}
// ...
action monitorHandler(integer id, com.apama.oms.OrderState state) {
   if state.justAcknowledged() then {
   log "Order acknowledged!";
   }
}
```

See the ApamaDoc for the order state container and `OrderState` objects for more details on the available actions.

Exchange simulation platform

A market exchange commonly consists of an order management unit and a matching engine. The order management component receives orders and subsequently rejects or accepts them based on its validation conditions. Accepted orders are either placed in the order-book or directly passed on to the matching engine for immediate transaction. The matching engine carries out executions of the orders in the book in accordance with the market's trading mechanism.

The Exchange Simulator bundle contains the code base for the simulation platform. The Exchange Simulator bundle supports creating a basic instance of the exchange simulator where only the minimal set of parameters. More advanced users may choose to create the ExchangeEngine component directly for more direct control. An event needs to be routed that identifies the symbol, service, market, and exchange being used for market data and the order management (OMS). This event also includes a fixed latency value which is added to all operations and a flag for choosing whether a simulated order flow should be generated or not and also another flag identifying whether DataViews should be created to publish market data events.

The exchange simulation platform can be described in terms of its two fundamental components, the exchange engine, and the order book.

Exchange engine

The exchange engine handles all order management requests and communicates with the order-book to decide the fate of a request and execute required actions. The exchange engine contains the order handling logic and also the matching logic. All book related activity is contained within the order-book component which is visible to the exchange engine. The matching logic is based on price. The exchange engine is also responsible for calling on publishing objects to send out events containing the depth of the order-book at the end of handling any request which has been accepted and executed. The exchange engine also triggers publishing of trade data.

Order book

The order-book maintains a complete record of the orders available for trading in the market. The order-book consists of a buy (bid) and a sell (offer) side, where each buy or sell order indicates the amount and price of an asset available for trade. In the basic case, orders are prioritized based on their price and time of arrival. Orders with more competitive prices are placed on the top, followed by orders at same price but with a later arrival time, and consecutively, orders with less competitive prices.

Exchange Simulator example

The Exchange Simulator bundle supports creating a basic instance of the exchange simulator where only the minimal set of parameters. More advanced users may choose to create the ExchangeEngine component directly for more direct control. The following is an example of an event routed to request a market with no latency, generate order flow, and set up DataViews.

```
route com.apama.oms.CreateBasicExchangeSimulator (true, "marketDataSymbol",
   "marketDataService", "marketDataExchange", "marketDataMarket", "omsSymbol",
   "omsService", "omsExchange", "omsMarket", 0.0, 0.0,
   true, new dictionary<string,string>);
```

Note that the order flow simulator is mainly used for testing the system and demonstration purposes. Given an initial price and a set of other parameters which are all documented in the `com.apama.oms.OrderFlowParams` event, the data generator produces orders with prices based on a random walk. These orders are then amended or canceled based on the other user-supplied parameters. Setting these parameters on the fly is possible via the `com.apama.oms.SimulateOrderFlow` event.

Risk firewall

The CMF risk firewall provides a mechanism to evaluate and potentially block orders based on a set of rules/criteria.

The CMF risk firewall is not designed to be a security firewall, and will not prevent malicious users of an application from circumventing the risk firewall to place orders directly.

To use the risk firewall, you must add the Risk Firewall bundle to your application. To use the risk firewall default rule classes, you must also add the Risk Firewall Rules bundle to your application.

Understanding risk firewalls

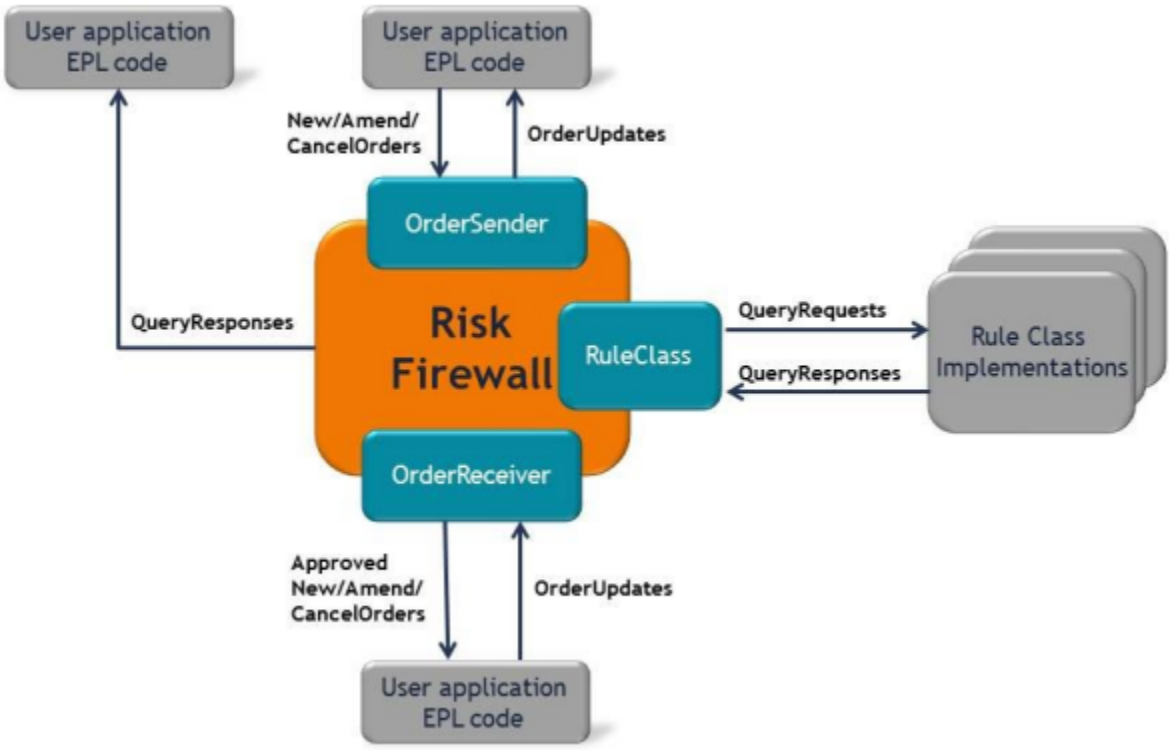
To ensure that order parameters fall within desired limits and comply with regulations, you create one or more risk firewalls. For each risk firewall, you register one or more rule classes, add rule class instances, unlock the risk firewall to make it available to be used, and use risk firewall order sender components to send orders into the risk firewall. The risk firewall evaluates each order against its rule class instances.

After evaluation, the risk firewall determines whether the order should be approved, pended, or rejected. This determination depends on both evaluation results and configuration parameter settings.

When an order is approved the risk firewall forwards it to an order receiver component in your application. It is up to the application to determine what to do with an approved order. For example, the application can have the order receiver forward the approved order to an order management system outside your application, or to a smart order router, or to a trading algorithm. Likewise, if an order is rejected, your application determines how to handle the order.

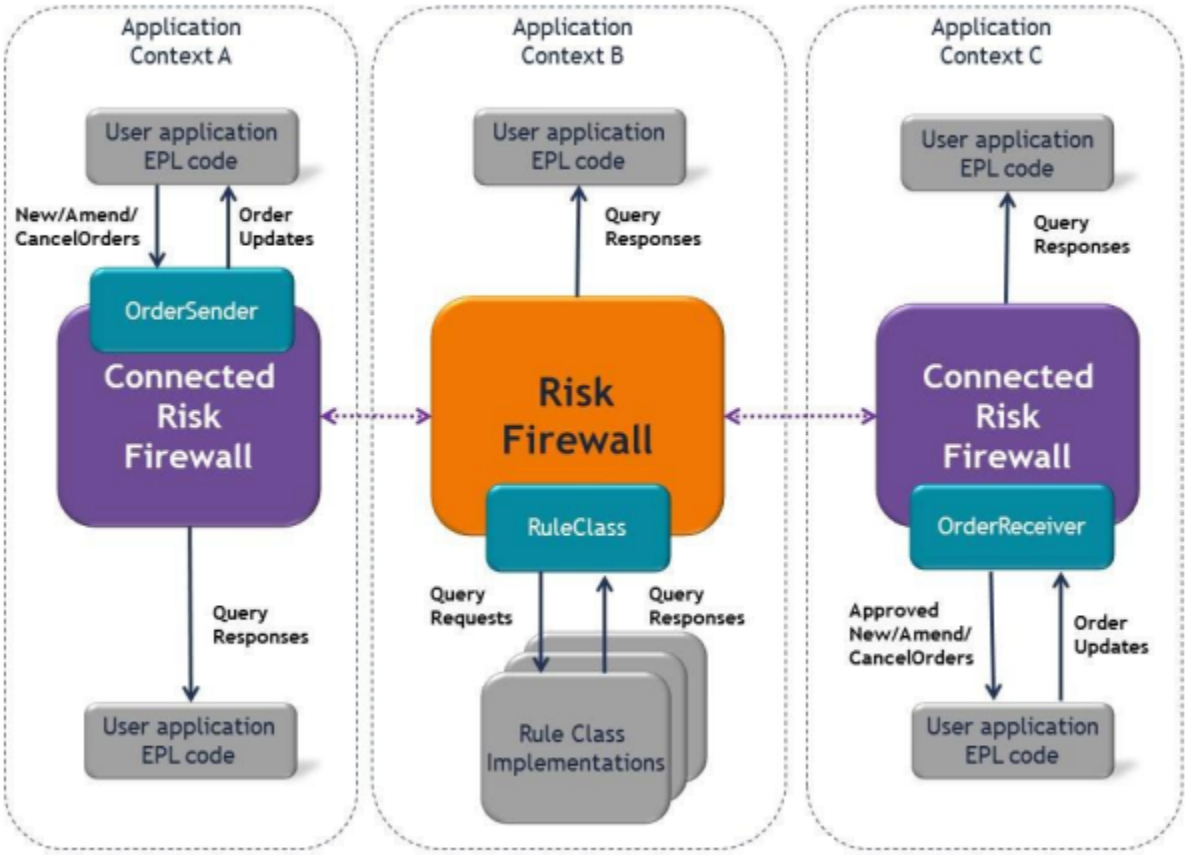
The risk firewall provides a framework for defining custom, pre-trade risk rules, and also provides a number of default rule classes that are ready to register with a risk firewall. The default rule classes provide a structure for determining, for example, whether an order price is above a specified ceiling, whether a trader has placed a cumulative order value beyond a particular limit, and whether the overall position for a set of orders has exceeded specified parameters. For each registered rule class, you add one or more rule instances. Each rule instance can specify different parameters and order filters. Note that for custom rule classes, the addition of at least one rule class instance is not always a requirement. See "Implementing custom risk firewall rule classes" on page 168.

After an order is processed by an external order management system, that system returns an order update. The risk firewall order receiver passes the order update back through the risk firewall so that it can be processed by the rule instances. This enables the rule instances to update any cumulative values they maintain. The risk firewall then sends the order update to the risk firewall order sender that originated the order. The following figure illustrates this.



`NewOrder`, `AmendOrder`, and `CancelOrder` events go from the `OrderSender` component into the risk firewall. If approved, the risk firewall sends these events to the `OrderReceiver` component. `OrderUpdate` events go from external order-processing components to the `OrderReceiver`, which sends the order updates into the risk firewall. The risk firewall sends any order state changes to all registered rule classes so that they can update their internal state if required. The risk firewall then sends the `OrderUpdate` to the `OrderSender` component.

You can create any number of uniquely named risk firewalls. You can use multiple contexts to implement your risk firewall architecture. You can create a risk firewall in one context and use it from one or more monitors in that context as well as use it from other contexts. The following figure illustrates this.



You must always explicitly create risk firewall instances and register rule classes. However, by default, a risk firewall is configured to persist rule class instances. Upon correlator re-start, they are automatically re-added to the risk firewall using previously set per-instance parameters. This happens regardless of whether persistence is enabled for the correlator.

Sample code for implementing a risk firewall can be found in the `samples` directory of your CMF installation directory. Also in that directory, is the source code for the default rule classes, which you can use as a reference for implementing custom rule classes.

Risk firewall components

The main components in any risk firewall architecture are:

- **Risk firewall factory.** You use `com.apama.firewall.RiskFirewallFactory` to create firewall instances in the current context and to connect to remote firewall instances, which are in other contexts or monitors.
- **Risk firewall.** You obtain an instance of a risk firewall (`com.apama.firewall.RiskFirewall`) by either creating a risk firewall with a `RiskFirewallFactory` instance or by connecting to a risk firewall with a `RiskFirewallFactory` instance. You use a risk firewall instance to register the rule classes and add the rule instances that specify the criteria against which to evaluate orders. Also, a `RiskFirewall` instance provides actions for setting callbacks to be executed upon changing the lock state of the risk firewall and upon rule query responses that indicate whether an order was approved. When you connect to a remote risk firewall, there are some `RiskFirewall` actions that are not available because the risk firewall you are connected to is in another context or monitor. See "Connecting to risk firewalls" on page 100.
- **Order sender.** A `RiskFirewall` instance provides an action for obtaining an instance of an order sender (`com.apama.firewall.OrderSender`). You use an order sender to send orders into the risk firewall for evaluation. When there is an update to a submitted order the risk firewall sends it to the order sender that originated the order.
- **Order receiver.** A `RiskFirewall` instance provides an action for obtaining an instance of an order receiver (`com.apama.firewall.OrderReceiver`). You use an order receiver to receive approved orders from the risk firewall, and process them according to your application logic. For example, an order receiver can send approved orders to external order management systems. A risk firewall order receiver also receives order updates back from those systems, and sends order updates back through the risk firewall.
- **Rule class factory.** You use `com.apama.firewall.RuleClassFactory` to create a default implementation of a custom rule class. You can then customize the default implementation to evaluate orders against particular, specified parameters.
- **Rule class.** The rule class factory returns a `com.apama.firewall.RuleClass` instance. Each rule class instance is a specific implementation of a rule class that evaluates the orders sent into the risk firewall.

The default rule classes provided with CMF are defined in the `com.apama.firewall.rules` package. For each default rule class, there is an event interface, also in the `com.apama.firewall.rules` package, that defines the configuration parameters you can specify for that rule class.

A risk firewall application also uses:

- `com.apama.oms.NewOrder`, `AmendOrder`, and `CancelOrder` — to send new orders to an external order management system, or to amend or cancel orders submitted to an OMS.
- `com.apama.oms.OrderUpdate` — for updates to submitted orders.
- `com.apama.utils.Params` — to set parameters for risk firewall factories, risk firewall instances, and rule class instances.
- `com.apama.utils.ParamsSchema` — to define the configuration options that are available to be set for a rule class and to define the possible values for each configuration option.

- `com.apama.utils.Error` — to override default error handling behavior for risk firewall factories and risk firewall instances.

See the ApamaDoc for details about these interfaces.

Slice filters for risk firewall rules

Rule classes can identify order fields of interest using a concept known as slicing, that is, splitting a stream of events into sub-streams along several dimensions. Rule class slices can use the following dimensions:

- Symbol
- Service identifier
- Exchange identifier
- Market identifier
- Owner (trader) identifier
- Extra parameters

A slice can match a set of values in each dimension, or all values. Examples of slices that can be specified include:

- Symbol `EUR/USD`, for example, to ensure that the organization-wide net order position for this instrument does not exceed a given value.
- Service identifier `FIX` and market identifier `CNX`, for example, to ensure that the total quantity of orders from all traders across all symbols on a single market does not exceed a specified level.
- Symbol `EUR/USD`, Service identifier `FIX`, market identifier `CNX` and trader identifier `A` or `B`, for example, to ensure that the net position of this group of two traders for the `EUR/USD` instrument on a single market is within specified limits.

Overview of steps for using a risk firewall

The typical steps for setting up and using a risk firewall are as follows:

1. Add the Risk Firewall bundle to your application. To use the risk firewall default rule classes, you must also add the Risk Firewall Rules bundle to your application.
2. Create a risk firewall. See "Creating risk firewalls" on page 97.
3. Register one or more rule classes with the risk firewall you created. See "Registering rule classes with a risk firewall" on page 125.
4. Add one or more risk firewall rule instances for each rule class you registered with the risk firewall. See "Adding rule instances to rule classes" on page 146. Note that for custom rule classes, the addition of at least one rule class instance is not always a requirement. See "Implementing custom risk firewall rule classes" on page 168.

5. Unlock the risk firewall, which is locked by default so your application can ensure that initialization has been done before trying to use the risk firewall. See "Unlocking and locking risk firewalls" on page 157.
6. Send orders into the risk firewall. See "Sending orders into a risk firewall" on page 158.
7. Process orders after evaluation by the risk firewall. This is application-specific. For example, your application can
   - Forward approved orders to an order management system outside your application. See "Receiving approved orders from a risk firewall" on page 162.
   - Process rejected orders by sending notifications to relevant users. See "Handling orders rejected by a risk firewall" on page 165.
   - Amend pending orders by allowing human intervention.
8. Receive order updates from external order management systems and pass them back into the risk firewall. See "Processing order updates" on page 165.
9. Process order updates from the risk firewall.

Optionally, you can also do the following:

- Configure a risk firewall factory instance by setting parameters and/or overriding default error handling behavior. These configurations apply to subsequent execution of the `RiskFirewallFactory.create()`, `createCb()`, `connect()`, or `connectCb()` action on that instance. See "Configuring risk firewall factories" on page 104.
- Configure a risk firewall instance by:
  - "Setting risk firewall query response callbacks" on page 107
  - "Setting risk firewall lock callbacks" on page 108
  - "Setting risk firewall parameters" on page 110
  - "Overriding default error handling for risk firewalls" on page 123
- Specify the order in which to evaluate registered rule classes. See "Setting rule class priority" on page 152.
- Connect to a risk firewall that is in another monitor or context. See "Connecting to risk firewalls" on page 100.
- Set order update callbacks that process order updates that the risk firewall sends. See "Setting order update callbacks" on page 160.

Basic example of using a risk firewall

The following code is a simple example of how to use a risk firewall.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;
using com.apama.firewall.rules.OrderPriceLimitRiskFirewallRule;
```

```
using com.apama.firewall.rules.OrderPriceLimitRiskFirewallRuleConsts;

// This sample demonstrates how to create and configure a risk firewall.
// It also demonstrates placing orders through the risk firewall,
// and receiving approved orders from the risk firewall.
//
// After running this sample, the correlator log displays two approved
// orders received, and the third order is rejected as it would breach
// the price limit that has been set.
monitor BasicRiskFirewallSample {
  // Store the main context.
  context mainContext := context.current();

  action onload() {
    // Create the risk firewall in the main context, and
    // call it "MyFirewall".
    RiskFirewall rfw :=
       (new RiskFirewallFactory).create( mainContext, "MyFirewall" );

    // Add a callback to receive approved order events from the risk firewall.
    // This callback is executed for only new orders. Similar actions are
    // provided for adding callbacks for amendments and cancellations.
    // This action returns a reference Id that can be used to remove the
    // callback later if required.
    integer refId1 := rfw.getOrderReceiver().
    addAcceptedOrderCallback( cbOnAcceptedOrderReceived );

    // Handle any order updates that come back.
    integer refId2 := rfw.getOrderSender().
    addOrderUpdateCallback( cbOnOrderUpdates );

    // Create and register the Order Price Limit default rule class.
    rfw.registerRuleClass( (new OrderPriceLimitRiskFirewallRule).create() );

    // Add a new instance of the Order Price Limit rule class.
    OrderPriceLimitRiskFirewallRuleConsts priceLimitConsts :=
       new OrderPriceLimitRiskFirewallRuleConsts;

    // Add some basic configuration for the Price Limit rule class.
    // In this case, issue a warning when the price goes over 15.0
    // and reject orders with a price over 20.0.
    Params ruleParams := new Params;
    ruleParams.addFloatParam( priceLimitConsts.PRICE_LIMIT_PARAMETER,   20.0 );
    ruleParams.addFloatParam( priceLimitConsts.PRICE_WARNING_PARAMETER, 15.0 );

    // Add the new rule class instance to the risk firewall.
    integer instanceId := rfw.addRuleInstance(
    priceLimitConsts.RULE_CLASS_NAME, ruleParams );

    // The risk firewall is now set up. Unlock it to allow orders
    // through. The risk firewall will not unlock until the Service
    // Framework has been activated, and all of the above operations
    // have completed.
    rfw.unlockCb( cbOnRiskFirewallUnlocked );
  }

  // This action is called when the risk firewall has been unlocked.
  action cbOnRiskFirewallUnlocked( RiskFirewall rfw ) {
    // Send orders to the risk firewall using the OrderSender interface.
    log "SENDING ORDERS";
```

```
    // This order will be approved.
    rfw.getOrderSender().sendOrder( NewOrder(
       "orderId_1","APMA",
       10.0, "BUY", "LIMIT", 10,
       "TargetService","","",
       "TargetMarket","", "TraderA",
       new dictionary<string,string> ) );

    // This order will issue a warning as it breaches the price
    // warning level that has been set, but is under the price limit.
    rfw.getOrderSender().sendOrder( NewOrder(
       "orderId_2","APMA",
       16.0, "BUY", "LIMIT", 10,
       "TargetService","","",
       "TargetMarket","", "TraderA",
       new dictionary<string,string> ) );

       // This order will be rejected as it breaches the price limit
       // that has been set.
       rfw.getOrderSender().sendOrder(
          NewOrder( "orderId_3","APMA",
             21.0, "BUY", "LIMIT", 10,
             "TargetService","","",
             "TargetMarket","", "TraderA",
             new dictionary<string,string> ) );
       }

  // This action is called whenever the risk firewall has approved
  // a new order.
  action cbOnAcceptedOrderReceived( NewOrder order ) {
     log "RECEIVED APPROVED NEW ORDER: "+order.toString();
  }

  // This action is called whenever the risk firewall has received
  // an update to an order.
  action cbOnOrderUpdates( OrderUpdate update ) {
     log "RECEIVED ORDER UPDATE: "+update.toString();
  }
}
```

Creating risk firewalls

To create a risk firewall, execute either of the following actions:

- `com.apama.firewall.RiskFirewallFactory.create()` constructs a new risk firewall instance in the current context.
- `com.apama.firewall.RiskFirewallFactory.createCb()` constructs a new risk firewall instance in the current context and then executes the specified callback.

Both actions take these two parameters:

- *`serviceManagerCtx`* is a reference to the context that the Service Framework manager has been created in. Currently, this is always the main context.

- *rfwName* is a string that contains the name for the risk firewall. In an application, two risk firewalls cannot have the same name.

In addition, the `createCb()` action takes a callback action as its third parameter.

Creation of a risk firewall is asynchronous. If your application requires notification when the risk firewall has been constructed, execute the `createCb()` action. After the CMF is activated and the risk firewall is constructed, the specified callback will be executed.

You may perform actions immediately after the call to create the risk firewall, for example you can register rule classes, add rule class instances, or connect to a the new risk firewall. In this case, the create callback will be executed after the rule classes are registered, the rule class instances are added, and the connections from remote contexts or monitors are set up. After execution of this callback, you can of course register additional rule classes, add more rule class instances, and connect from additional contexts and monitors.

If any parameters are set on a `RiskFirewallFactory`, these parameters have the same settings in a risk firewall instance created by that factory. If default error handling is changed for a `RiskFirewallFactory` instance and then you use it to create a risk firewall, if there is an error in the creation of the risk firewall then the updated error handling behavior is followed. Also, a risk firewall instance created by this factory handles errors in the same way as the factory. See "Configuring risk firewall factories" on page 104.

Any parameters you set for a risk firewall factory instance and any error handling behavior you update for that factory apply to only subsequently created risk firewalls.

Sample code for creating risk firewall

The following code executes the `RiskFirewallFactory.create()` action to create a risk firewall in the main context.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;

monitor RiskFirewallExample1 {
   context mainContext := context.current();

   action onload() {
      // Use a risk firewall factory to create a new risk firewall instance
      // called "MyFirewall" in the current context.
      // In this case, use the default configuration parameter settings.
      RiskFirewall rfw :=
         (new RiskFirewallFactory).create( mainContext, "MyFirewall" );
      ...
   }
}
```

In the following example, the risk firewall is created in a non-main context. The use of the risk firewall factory is the same as in the previous example. The only changes are those necessary for spawning an action to a new context.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;

monitor RiskFirewallExample2 {
```

```
   context mainContext := context.current();

   action onload() {
      spawn createRiskFirewall() to context( "FirewallCtx" );
   }

   action createRiskFirewall() {
      // Use a risk firewall factory to create a new risk firewall instance
      // called "MyFirewall" in the current context.
      // In this case, use the default configuration parameter settings.
      RiskFirewall rfw :=
         (new RiskFirewallFactory).create( mainContext, "MyFirewall" );
      ...
     }
}
```

Sample code for creating risk firewall and specifying a callback

The following code executes the `RiskFirewallFactory.createCb()` action to create a risk firewall in the main context. The `createCb()` action executes the callback you specify when the new risk firewall is fully constructed.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;

monitor RiskFirewallExample3 {
   context mainContext := context.current();

   action onload() {
      // Use a risk firewall factory to create a new risk firewall
      // instance called "MyFirewall" in the current context.
      // This example uses default configuration options.
      RiskFirewall rfw := (new RiskFirewallFactory).
         createCb( mainContext, "MyFirewall", onFirewallCreated );

      // Use the risk firewall to perform any needed operations.
      // Some operations may pend until the risk firewall has been
      // fully constructed.
   }

   action onFirewallCreated( RiskFirewall myFirewall ) {
      log "The RiskFirewall is now fully constructed";
      ...
   }
}
```

Deleting risk firewalls

To delete a risk firewall, execute the `com.apama.firewall.RiskFirewall.delete()` action. This action cleans up data and listeners created by the risk firewall.

If you call the `delete()` action on a risk firewall that you created with `RiskFirewall.create()` (or `createCb()`) then this action deletes the risk firewall itself. Also, any risk firewall instances that were connected to the deleted risk firewall no longer operate correctly.

If you call the `delete()` action on a risk firewall that you connected to with `RiskFirewall.connect()` (or `connectCb()`) then this action cleans up any resources and disconnects from the remote risk firewall, but does not delete the remote risk firewall itself.

The risk firewall you are deleting must not have been deleted previously.

The `delete()` action is not asynchronous.

Connecting to risk firewalls

After you create a risk firewall, you can use that risk firewall instance from other contexts and from other monitors. For example, you might want to register rule classes and add rule class instances in one context, send orders to that risk firewall from a different context or monitor, and handle orders approved by that risk firewall in yet another context or monitor. When you want to use a risk firewall that was created in another context or monitor you connect to that risk firewall.

To connect to a risk firewall, execute either of the following actions:

- `com.apama.firewall.RiskFirewallFactory.connect()` connects to a risk firewall instance in another context or monitor and returns a `com.apama.firewall.RiskFirewall` instance for use in the current context. Not all risk firewall actions are available on a risk firewall that is connected to a remote risk firewall instance. See the table later in this topic for details.
- `com.apama.firewall.RiskFirewallFactory.connectCb()` connects to a risk firewall instance in another context or monitor, returns a `com.apama.firewall.RiskFirewall` instance for use in the current context, and executes the specified callback. Not all risk firewall actions are available on a risk firewall that is connected to a remote risk firewall instance. See the table later in this topic for details.

Both actions take these two parameters:

- *`serviceFrameworkCtx`* is a reference to the context in which the Service Framework manager was created. Currently, this is always the main context.
- *`rfwName`* is a string that contains the name of the risk firewall you want to connect to. The name of the risk firewall must be available in the context or monitor from which you are connecting. Note that in the context that contains the risk firewall you want to connect to, to obtain the name of that risk firewall, you can call `com.apama.firewall.RiskFirewall.getFirewallName()`.

In addition, the `connectCb()` action takes a callback action as its third parameter. This callback is invoked after connection to the remote risk firewall.

The `RiskFirewall` instance returned by the `connect()` or `connectCb()` action provides a subset of the actions provided by `RiskFirewall` instances that are returned by `create()` or `createCb()` actions. The following table shows which actions are available from a risk firewall that is connected to a risk firewall instance in another context or monitor. Only a locally created risk firewall may register rule classes, add rule instances, and modify the configuration of the risk firewall instance. A remote connection to a risk firewall is limited to only essential actions, such as sending and receiving orders. This policy ensures a level of protection from modifications by risk firewall connections to remote risk firewall instances.

Risk Firewall Action	Available in Locally Created Risk Firewall	Available in Remotely Connected Risk Firewall

`addErrorCallback()`
`clearErrorCallbacks()`
`removeErrorCallback()`
`defaultErrorCallback()`
	✓	✓

`addLockStateChangedCallback()`
`clearLockStateChangedCallbacks()`
`removeLockStateChangedCallback()`
	✓	✓

`addQueryResponseCallback()`
`clear QueryResponseCallbacks()`
`removeQueryResponseCallback()`
	✓	✓

`delete()`	✓	✓

`getAllRuleClassInfo()`
`getRuleClassInfo()`
`getAllRuleClassInstanceInfo()`
`getAllRuleInstanceInfo()`
`getRuleInstanceInfo()`
	✓	✓

`getFirewallName()`	✓	✓
`getOrderReceiver()`	✓	✓
`getOrderSender()`	✓	✓
`getParams()`	✓	✓
`isLocked()`	✓	✓
`isRemote()`	✓	✓
`overrideSoftReject()`	✓	✓
`clearObjection()`	✓	✓

`addRuleInstance()`
`addRuleInstanceCb()`
`modifyRuleInstance()`
`modifyRuleInstanceCb()`
	✓	

Risk Firewall Action	Available in Locally Created Risk Firewall	Available in Remotely Connected Risk Firewall
`removeRuleInstance()` `removeRuleInstanceCb()`		

`lock()`
`unlock()`
`unlockCb()`		

`registerRuleClass()` `registerRuleClassCb()`		
`setParams()`		
`setRuleClassPriority()`		

To determine whether the risk firewall in the current context is connected to a remote risk firewall instance, execute the `RiskFirewall.isRemote()` action.

Connection to a remote risk firewall is asynchronous. Consequently, some risk firewall actions that are normally available on a risk firewall that is connected to a remote risk firewall might not be available until the connection is completely in place. If your application requires notification when the connection to the remote risk firewall is in place, execute `connectCb()` and specify a callback.

You can execute `connect()` or `connectCb()` to connect to a remote risk firewall that has not yet been created. However, a risk firewall you connect to must be created before the configurable timeout period has expired, otherwise an error will be returned. The configuration parameter that specifies the timeout period is `CONFIG_TIMEOUT_DURATION` and the default value is `5.0` seconds.

There can be any number of connections to a risk firewall instance.

Any error callbacks that were set on the `RiskFirewallFactory` prior to using it to connect to the remote risk firewall are used if there is an error when connecting to the risk firewall.

However, any parameters that were set on the `RiskFirewallFactory` prior to using it to connect to the remote risk firewall are ignored. The only exception to this is the `CONFIG_TIMEOUT_DURATION` parameter. You can set this on a factory and its value is used for the connection to the remote risk firewall.

A risk firewall that is connected to a remote risk firewall instance has the same configuration parameter settings as the risk firewall instance it is connected to. You cannot set parameters on a risk firewall that is connected to a remote risk firewall. This protects remote risk firewall instances from modifications by risk firewall connections. To change the configuration parameters for a risk firewall that is connected to a remote risk firewall you must change the configuration parameters directly on the remote risk firewall instance itself. When you update the configuration parameters

for a risk firewall instance any risk firewalls connected to that remote instance inherit the updated settings.

Connecting to a risk firewall instance in the same context is useful when you have two or more EPL monitors in the same context. For example, you can create a risk firewall instance in `monitorA` and in `monitorB` you can connect to the risk firewall in `monitorA`. You can then send orders from `monitorB`. A connected risk firewall takes care of the communication between the two monitors.

Sample code for connecting to risk firewall

The following code provides an example of connecting to a risk firewall.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;

monitor RiskFirewallExample4 {
   context mainContext := context.current();

   action onload() {
      // Use a risk firewall factory to connect to
      // a remote risk firewall instance called "MyFirewall".
      RiskFirewall rfwRemote := (new RiskFirewallFactory).
         connect( mainContext, "MyFirewall" );

      // Use the remotely connected risk firewall to
      // perform any required operations.
      // Some operations may pend until the remotely connected
      // risk firewall has been fully constructed and connected.
   }
}
```

Sample code for connecting to risk firewall and specifying callback

The following code provides an example of connecting to a risk firewall by executing the `connectCb()` action. After connecting to the remote risk firewall instance, the `onFirewallConnected()` callback is executed.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;

monitor RiskFirewallExample5 {
   context mainContext := context.current();

   action onload() {
      // Use a risk firewall factory to connect to
      // a remote risk firewall instance called "MyFirewall"
      // and specify a callback to be executed upon connection.
      RiskFirewall rfwRemote := (new RiskFirewallFactory).
         connectCb( mainContext, "MyFirewall", onFirewallConnected );

      // Use the remotely connected risk firewall to
      // perform any needed operations.
      // Some operations may pend until the remotely connected
      // risk firewall has been fully constructed and connected.
   }
```

```
   action onFirewallConnected(
      RiskFirewall rfwRemote ) {
         log "The RiskFirewall is now fully connected";

      // Use the remote risk firewall to perform
      // any required operations.
   }
}
```

Disconnecting from risk firewall

To disconnect from a remote risk firewall, execute the `com.apama.firewall.RiskFirewall.delete()` action. This action cleans up data and listeners that were created in the current context by the risk firewall. When you are connected to a remote risk firewall instance, the `delete()` action deletes only the risk firewall in the current context and not the risk firewall instance in the remote context.

The risk firewall you are disconnecting from must not have been previously deleted.

The `delete()` action is not asynchronous.

Configuring risk firewall factories

You use a `com.apama.firewall.RiskFirewallFactory` instance to create new `com.apama.firewall.RiskFirewall` instances. Before you use a risk firewall factory to create a new risk firewall, you can configure the risk firewall factory as described in the topics below.

Setting risk firewall factory parameters

There are many configuration parameters associated with risk firewall instances. If you want, you can set these parameters on a risk firewall factory so that subsequent risk firewalls created from that factory have the same parameter settings.

The constant values used to specify these configuration parameters are defined in `com.apama.firewall.Consts`.

When you use a risk firewall factory to create a new risk firewall instance, the created risk firewall instance has the same configuration parameter values as the risk firewall factory that created it. However, when you use a risk firewall factory to connect to a remote risk firewall instance, the returned risk firewall has the same configuration parameter values as the risk firewall instance it is connected to.

If the default configuration parameter settings meet your needs, you do not need to set risk firewall factory parameters. However, if you want to configure the risk firewall instances you create to have one or more common, non-default configuration parameter settings, you can set the appropriate parameter values on a risk firewall factory instance. Any risk firewalls you create from that factory will have the parameter values you previously set on that factory.

To set a parameter on a risk firewall factory:

1. Create a risk firewall factory instance.
2. Create a `com.apama.utils.Params` object.

3. Add a parameter name/value pair to the parameters object you created.
4. Repeat the previous step for each parameter you want to set.
5. Execute the `setParams()` action on the risk firewall factory instance and pass it the `Params` object you created.

After you set one or more configuration parameters on a risk firewall factory instance, you can obtain the parameter values you set on that instance by executing `com.apama.firewall.RiskFirewallFactory.getParams()`. This action returns the factory configuration parameters you explicitly set and not any default settings for configuration parameters.

Setting risk firewall factory parameters applies only to subsequently created risk firewall instances.

After you create a risk firewall instance, you can still change its configuration parameters. You do this by executing `com.apama.firewall.RiskFirewall.setParams()` on the risk firewall instance. See "Setting risk firewall parameters" on page 110.

The following example shows how to set a configuration parameter on a risk firewall factory instance.

```
using com.apama.firewall.RiskFirewallFactory;
com.apama.firewall.RiskFirewall;
com.apama.utils.Params;

monitor RiskFirewallExample6 {
   context mainContext := context.current();

   action onload() {
      // Use the risk firewall factory to define any non-default
      // risk firewall configuration. Also create a new risk firewall
      // instance called "MyFirewall" in the current context.
      // In this case, use non-default configuration parameter settings.
      RiskFirewallFactory factory := new RiskFirewallFactory;

      // Set up the configuration parameter object.
      Params params := new Params;
      params.addParam( "REJECT_MODE", "DEFAULT_ACCEPT" );
      factory.setParams( params );

      // Create the risk firewall.
      RiskFirewall myFirewall := (
         new RiskFirewallFactory).create( mainContext, "MyFirewall" );
    }
}
```

Overriding default error handling for risk firewall factories

The default error handler is invoked if there is an error related to risk firewall factories. For example, if any of the following happen:

- Risk firewall creation or connection fails.
- You try to remove an error callback and specify an incorrect error callback reference identifier.
- You try to set a parameter and specify an invalid parameter name.

The default error handler sends a message to the correlator log file at the `ERROR` level. To change this behavior, execute `com.apama.firewall.RiskFirewallFactory.addErrorCallback()`, which adds the specified callback to the set of callbacks executed if there is an error related to that risk firewall factory instance.

You can execute the `addErrorCallback()` action multiple times on the same factory instance to implement multiple error handling callbacks for that factory instance. If you add one or more error callbacks to a factory instance then the default error callback is not executed for that factory instance.

The parameters of a user-defined error callback include the risk firewall factory instance and also a `com.apama.utils.Error` event. An `Error` event has fields for a message, a dictionary of parameters, and an error type code. The `addErrorCallback()` action adds the specified callback to the set of callbacks executed if there is an error in the operation of the specified risk firewall factory instance.

The `com.apama.firewall.ErrorConstants` event defines the following error type codes, which can apply to risk firewall factory instances:

- `FAILED_TO_CREATE_RISKFIREWALL_INTERFACE`
- `FAILED_TO_REMOVE_ERROR_CALLBACK`
- `INVALID_PARAMETER`

When you add an error callback the return value is an integer reference ID that you can specify if you execute `RiskFirewallFactory.removeErrorCallback()` to discontinue execution of that error callback. To remove all error callbacks, execute the `RiskFirewallFactory.clearErrorCallbacks()` action. If you remove all previously set error callbacks then error handling behavior reverts to calling the default error callback.

Suppose you add one or more error callbacks to a risk firewall factory instance. A risk firewall instance created by that factory has the same error callbacks as that factory. However, if you use that factory to connect to a remote risk firewall instance, the returned risk firewall uses the same error handling as the risk firewall instance it is connected to.

In the following example, the application adds an error callback to be executed if the risk firewall factory fails to create a new risk firewall instance. For example, perhaps the application needs to send the error to a custom location.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;
using com.apama.utils.Error;

monitor RiskFirewallExample7 {
   context mainContext := context.current();

   action onload() {
      // Use a risk firewall factory to define an error callback
      // that will be called if risk firewall creation fails.
      // Then create a new risk firewall instance called "MyFirewall"
      // in the current context.
      // In this case, use the default configuration parameter settings.
      RiskFirewallFactory factory := new RiskFirewallFactory;

      // Set the error callback action that will be called should an error
```

```
      // occur when creating the risk firewall.
      integer refId := factory.addErrorCallback( onRiskFirewallErrors );

      // Create a risk firewall.
      RiskFirewall rfw := factory.create( mainContext, "MyFirewall" );
   }

   // This action is called if an error occurs creating the risk firewall.
   action onRiskFirewallErrors ( RiskFirewall rfw, Error error ) {
      // The application can query the Error object to return the error message,
      // and to determine the type of error and any extra parameter information
      // that is provided, to assist the application to programmatically
      // determine the cause of the failure.
      ...
   }
}
```

In the following example, the application adds an error callback to be executed if an attempt to connect to a remote risk firewall instance fails.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;
using com.apama.utils.Error;

monitor RiskFirewallExample8{
   context mainContext := context.current();

   action onload() {
      // Use a risk firewall factory to define an error callback that
      // will be called if a connection attempt fails.
      // Use the same risk firewall factory to connect to a risk firewall
      // instance called "MyFirewall".
      RiskFirewallFactory factory := new RiskFirewallFactory;

      // Set the error callback that will be called if an error occurs
      // when creating or connection a risk firewall.
      integer refId := factory.addErrorCallback( onRiskFirewallErrors );

      // Connect to a remote risk firewall.
      RiskFirewall rfwRemote := factory.connect( mainContext, "MyFirewall" );
   }

   // This action is called if an error occurs while trying to connect
   // to the remote risk firewall.
   action onRiskFirewallErrors( RiskFirewall rfwRemote, Error error ) {
      // The application can query the Error object to return the error message,
      // and to determine the type of error and any extra parameter information
      // that is provided, to assist the application to programmatically
      // determine the cause of the failure.
   }
}
```

Configuring risk firewall instances

You can configure risk firewall instances as described in the topics below.

Setting risk firewall query response callbacks

When you send an order into a risk firewall, the risk firewall queries the registered rule classes to determine whether or not to approve the order. If you want, you can add one or more callbacks to be executed each time the query results for an order are available.

To set a query response callback, execute `com.apama.firewall.RiskFirewall.addQueryResponseCallback()`. This action registers the specified callback with the risk firewall instance it is executed on. The registered callback is executed for every order that enters the risk firewall. The `addQueryResponseCallback()` action returns a unique integer reference Id that you can use to remove the callback at a later date if required. This action is not asynchronous and is always available.

When a risk firewall instance executes a query response callback it provides the query request as well as the query results. By default, query results contain only failure and warning responses, and not approval responses. This is determined by the settings of these configuration parameters:

- `CONFIG_ADD_QUERY_RESPONSE_FAIL` — default is `true`.
- `CONFIG_ADD_QUERY_RESPONSE_PASS` — default is `false`.
- `CONFIG_ADD_QUERY_RESPONSE_WARN` — default is `true`.

You may want to modify these settings, for example, to record the results of all risk firewall queries for auditing purposes. To do this, you need to set the `CONFIG_ADD_QUERY_RESPONSE_PASS` parameter to `true`.

In a query response callback, you cannot override the result of the query. You can, however, publish DataViews that show the warnings or failures that have been issued by the risk firewall.

To remove a previously-added query response callback, execute `RiskFirewall.removeQueryResponseCallback()` and specify the integer reference Id that was returned when you added the callback. To remove all previously-added query response callbacks, execute the `RiskFirewall.clearQueryResponseCallback()` action.

Setting risk firewall lock callbacks

A risk firewall instance can be locked so that it cannot accept incoming orders for evaluation. This can be useful to immediately prevent any more trading from taking place. For example, to prevent a runaway trading algorithm.

By default, when you create a risk firewall instance it is locked until you explicitly unlock it. This ensures that no orders can enter the risk firewall before all required rule classes have been registered and rule class instances have been added. Consequently, it can be important to indicate to parts of your application when the lock state (locked or unlocked) of a risk firewall changes.

To set a lock state changed callback, execute `com.apama.firewall.RiskFirewall.addLockStateChangedCallback()`. This action registers a callback with the risk firewall instance it is executed on. The registered callback is executed whenever the lock status of the risk firewall instance changes. For example, this can be useful for delaying certain risk firewall operations until the risk firewall is unlocked. This action is not asynchronous and is always available.

When you execute the `addLockStateChangedCallback()` action you specify the callback action you want to be executed. The `addLockStateChangedCallback()` action returns a unique integer reference Id that you can use to remove the callback at a later date if required.

To remove a previously-added lock state changed callback, execute `RiskFirewall.removeLockStateChangedCallback()` and specify the integer reference Id that was returned when you added the callback. To remove all previously-added lock state changed callbacks, execute the `RiskFirewall.clearLockStateChangedCallback()` action.

Configuring risk firewall behavior for rejected orders

When an order comes into a risk firewall, each rule class instance is queried to determine whether the order complies with that rule class instance. But even when the query responses indicate that an order would be rejected, it is the setting of the rejection mode configuration parameter that determines risk firewall behavior.

There are several ways in which the rule instance query responses can indicate whether the order should be approved or rejected.

- If the `CONFIG_FAST_FAIL_MODE` configuration parameter is enabled, an order is considered rejected as soon as there is a single query response that indicates that the order does not comply with a rule class instance. Subsequent rule class instances are not evaluated. This is the default behavior.
- If the `CONFIG_FAST_FAIL_MODE` configuration parameter is not enabled, an order is still considered rejected as soon as there is a single query response that indicates that the order does not comply with a rule class instance. However, subsequent rule class instances are still evaluated for their query responses. This is less efficient, but provides a comprehensive response across all rule class instance queries.
- If the `CONFIG_FAST_FAIL_MODE` configuration parameter is not enabled then the risk firewall must receive a response from all rule class instances before the time specified by the `CONFIG_TIMEOUT_DURATION` configuration parameter has elapsed. The default time period is 5 seconds. This ensures that the state of an order is not blocked by unresponsive rule class instances. If the timeout period is reached and not all query responses have been received then the order would be rejected as a result of the query request timing out.

If an order is considered to be rejected, what the risk firewall does next depends on its rejection mode setting. The configuration parameter `CONFIG_REJECTION_MODE` can have one of the following values:

`CONFIG_REJECTION_MODE` Setting	**Behavior when an order would be rejected**
`CONFIG_REJECTION_MODE_HARD`	The order is rejected. The risk firewall sends an `OrderUpdate` event that contains the rejection details back to the risk firewall order sender that submitted the order. The risk firewall passes only approved orders to a risk firewall order receiver. This is the default behavior.

CONFIG_REJECTION_MODE Setting	Behavior when an order would be rejected
CONFIG_REJECTION_MODE_SOFT	The order is pended. The risk firewall sends an OrderUpdate event that contains the rejection details back to the risk firewall order sender that submitted the order. The application can allow a dealer to do any one of the following: ■ Amend the order to be correct and re-submit it to the risk firewall for evaluation. Note that your cannot send a new order with the same order Id. You must always amend the existing, pending order. ■ Cancel the order. ■ Leave the order as is and allow it to bypass the risk firewall. Again, the RiskFirewall.overrideSoftReject() action does this. Soft rejection mode uses a configurable timeout (CONFIG_SOFT_REJECTION_DURATION) to ensure that orders are not pended for indefinite periods of time. Once this timeout period has expired the order is rejected. The default timeout period is 60 seconds.
CONFIG_REJECTION_MODE_MONITOR	The order is passed forward to a risk firewall order receiver instance as though it had been approved. Set this mode when you want to monitor the orders being evaluated by the risk firewall. When MONITOR mode is set, you should also execute RiskFirewall.addQueryResponseCallback() to add a callback that will provide the rejection information.

When the CONFIG_REJECT_BY_DEFAULT configuration parameter is enabled (the default) the risk firewall rejects an order if it does not match any rule class instances. The risk firewall treats this rejection according to its rejection mode setting. If soft rejection is enabled then the order is pended. However, amendments to the order would also be rejected until the application adds a suitable rule class instance or corrects the original order details.

Note:
By default, approval from the risk firewall requires an order to match at least one instance of every registered rule class.

Setting risk firewall parameters

When you create a risk firewall instance it has the same configuration parameter settings as the risk firewall factory you use to create it. Each risk firewall configuration parameter has a default

setting, which is appropriate for the most common risk firewall use cases. You only need to set risk firewall parameters if the default settings do not meet your application requirements.

There are many configuration parameters that you can set to specify the behavior of a risk firewall. The constant values used to specify these configuration parameters are defined in `com.apama.firewall.Consts`. See also "Default settings for risk firewall configuration parameters" on page 112.

To set a configuration parameter for a particular risk firewall instance, execute the `com.apama.firewall.RiskFirewall.setParams()` action on the risk firewall instance. Alternatively, you can set the configuration parameter on a risk firewall factory instance and then use that factory instance to create the risk firewall. See "Setting risk firewall factory parameters" on page 104.

To set parameters on a risk firewall instance:

1. Use a risk firewall factory to create a risk firewall instance. For example:

   ```
   com.apama.firewall.RiskFirewall rfw :=
      (new com.apama.firewall.RiskFirewallFactory).
         create(mainContext, "myRiskFirewall");
   ```

2. Create a `com.apama.utils.Params` object.
3. Add a parameter name/value pair to the parameters object you created.
4. Repeat the previous step for each parameter you want to set.
5. Execute the `setParams()` action on the risk firewall instance and pass it the `Params` object you created.

If the risk firewall instance is locked, the new parameters will be in effect when you unlock it. If the risk firewall instance is unlocked, the new settings affect subsequent inbound orders.

To obtain the values for any configuration parameters that have been explicitly set, either on the factory that created the risk firewall instance or on the instance itself, execute the `RiskFirewall.getParams()` action. This action does not return values for parameters that have the default setting.

If you set parameters on an unlocked risk firewall it might affect only subsequent orders that enter the risk firewall.

The following sample code sets a parameter on a risk firewall instance.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;
using com.apama.firewall.Consts;
using com.apama.utils.Params;

// This sample demonstrates how to create and configure a risk firewall.

monitor RiskFirewallExample9 {

   // Store the main context.
   context mainContext := context.current();

   action onload() {
```

```
      // Create the risk firewall in the main context, and
      // call it "MyFirewall".
      RiskFirewall rfw := (
         new RiskFirewallFactory).create( mainContext, "MyFirewall" );

      // Modify the risk firewall configuration to use
      // soft rejection mode. This provides the opportunity
      // to amend orders and resubmit them.
      Params rfwParams := new Params;
      rfwParams.addParam(
         Consts.CONFIG_REJECTION_MODE, Consts.CONFIG_REJECTION_MODE_SOFT );
      rfw.setParams( rfwParams );
   }
}
```

Note:
The `CONFIG_ENABLE_PERSISTENCE` and `CONFIGSTORE_PATH_KEY` parameters need to be set using the Service Framework, see "Descriptions of risk firewall configuration parameters" on page 113 for details.

Default settings for risk firewall configuration parameters

There are many parameters that affect risk firewall behavior. All parameters have default settings that provide the most commonly desired behavior. If these settings provide the behavior you want then you do not need to set risk firewall configuration parameters.

However, if you do want to set the value of a risk firewall configuration parameter, see "Setting risk firewall parameters" on page 110 and "Descriptions of risk firewall configuration parameters" on page 113.

The default settings for risk firewall parameters are as follows. If it is blank where the default setting should be then the parameter does not have a default value.

Parameter	Default Setting
`CONFIG_ADD_QUERY_RESPONSE_FAIL`	Enabled
`CONFIG_ADD_QUERY_RESPONSE_PASS`	Disabled
`CONFIG_ADD_QUERY_RESPONSE_WARN`	Enabled
`CONFIG_BUSTED_FILL_DURATION`	5 seconds
`CONFIG_DEFAULT_LATENCY_TIMESTAMPID_BASE`	7000
`CONFIG_ENABLE_LEGACY_MODE`	Disabled
`CONFIG_ENABLE_PERSISTENCE`	Enabled
`CONFIG_FAST_FAIL_MODE`	Enabled
`CONFIG_LOCKED_ON_CREATE`	Enabled
`CONFIG_LOG_AMEND_CANCEL_LATENCY_TIMESTAMPS`	Disabled

Parameter	Default Setting
CONFIG_LOG_INBOUND_LATENCY_TIMESTAMPS	Disabled
CONFIG_LOG_OUTBOUND_LATENCY_TIMESTAMPS	Disabled
CONFIG_RECOVERED_RULE_INSTANCE_ID	
CONFIG_REJECT_BY_DEFAULT	Enabled
CONFIG_REJECTION_MODE	Hard
CONFIG_SOFT_REJECTION_DURATION	60 seconds
CONFIG_TIMEOUT_DURATION	5 seconds
CONFIGSTORE_PATH_DEFAULT	"RiskFirewall.sqlite"
ORDER_OPERATION_CACHE_MAX_ROWS	0
ORDER_OPERATION_CACHE_PATH_DEFAULT	"OrderOperationCache.sqlite"
ORDER_OPERATION_CACHE_PATH_KEY	"OrderOperationCacheStorePath"
ORDER_OPERATION_CACHE_PURGE_INTERVAL	0
ORDER_OPERATION_CACHE_PURGE_TIME	00:00:00
ORDER_OPERATION_CACHE_TIMEOUT	0
RULE_CLASS_PRIORITY_HIGH	0
RULE_CLASS_PRIORITY_LOW	100
RULE_CLASS_PRIORITY_MEDIUM	50
RULE_CLASS_STATE_PARAM_INSTANCE_COUNT	

Descriptions of risk firewall configuration parameters

Constant values defined in `com.apama.firewall.Consts` define the names of the configuration parameters that control risk firewall behavior. These configuration parameters are described here in logical functional groups. For alphabetical reference information, see the ApamaDoc for `com.apama.firewall.Consts`. See also "Setting risk firewall parameters" on page 110 and "Default settings for risk firewall configuration parameters" on page 112.

The following sections describe risk firewall parameters.

Risk firewall factory and instance parameters

`CONFIG_LOCKED_ON_CREATE`

- This constant value defines the name of the configuration parameter that indicates whether a risk firewall is created in a locked state.

- The default is that a risk firewall is created in a locked state. That is, an application must explicitly unlock a risk firewall. See "Unlocking and locking risk firewalls" on page 157.
- To change the setting of this parameter, use the `CONFIG_LOCKED_ON_CREATE` key and set it to the `boolean` value you need.

`CONFIG_TIMEOUT_DURATION`

- This constant value defines the name of the configuration parameter that indicates the timeout duration for event communication.
- The default is 5 seconds. This setting is used when
  - Connecting to a risk firewall
  - Registering a rule class
  - Waiting for queries related to a single order to all be processed

  See also "CONFIG_SOFT_REJECT_DURATION" on page 122 and "CONFIG_BUSTED_FILL_DURATION" on page 119.
- To change the setting of this parameter, use the `CONFIG_TIMEOUT_DURATION` key and set it to the `float` value you need.

`CONFIG_ENABLE_PERSISTENCE`

- This constant value defines the name of the configuration parameter that indicates whether persistence is enabled for the risk firewall. Note that risk firewall persistence is separate from correlator persistence. When risk firewall persistence is enabled then rule class instances are reloaded upon correlator re-start regardless of whether persistence is enabled for the correlator.
- The default is that persistence is enabled.
- This parameter needs to be set using the Service Framework as it is the same for all firewalls created. To change it, you must send in a `ServiceParameters` event during startup:

  ```
  com.apama.service.framework.ServiceParameters("RiskFirewall", "RiskFirewall",
  com.apama.firewall.Consts.CONFIG_ENABLE_PERSISTENCE, "true")
  ```

`CONFIGSTORE_PATH_DEFAULT`

- This constant value defines the default file and path for persisting risk firewall data in configuration store tables.
- The default is `"RiskFirewall.sqlite"`.
- To change the setting of this parameter, use the `CONFIGSTORE_PATH_KEY` parameter.

`CONFIGSTORE_PATH_KEY`

- This constant value defines the configuration key value for setting the risk firewall default file and path for its configuration store tables. Use this parameter when you need to override the default storage location used by the risk firewall. If you change the value of this parameter you must have read/write access to the location you specify.

- The default is `"ConfigStorePath"`
- This parameter needs to be set using the Service Framework as it is the same for all firewalls created. To change it, you must send in a `ServiceParameters` event during startup:

```
com.apama.service.framework.ServiceParameters("RiskFirewall", "RiskFirewall",
com.apama.firewall.Consts.CONFIGSTORE_PATH_KEY, "/path/to/configstore")
```

`RISK_FIREWALL_SERVICE_TYPE`

- This constant value defines the name of the risk firewall service that is registered with the CMF Service Framework.
- The default is `"RiskFirewall"`.
- You cannot modify the value of this parameter. However, you can use this parameter to query the CMF Service Framework directly for all services of this type. Also, the value of this parameter is the prefix of the name of the Apama MemoryStore table that is used to store information about the rule class instances in the risk firewall.

Order operation cache parameters

If you want to set a value for one of these parameters, you must do so in a risk firewall factory before you create a risk firewall. After you create a risk firewall, you cannot change the value of any of these parameters for that risk firewall instance.

`ORDER_OPERATION_CACHE_TIMEOUT`

- This constant value defines the name of the configuration parameter that specifies the timeout duration for automatic purging of rows in the order operation cache. The value of this parameter specifies a number of seconds. It indicates how long a row is stored before it is eligible for automatic purging the next time that automatic purging is done.
- The default is `0`, which means that rows are not automatically purged.
- Use this parameter with the `ORDER_OPERATION_CACHE_PURGE_INTERVAL` and `ORDER_OPERATION_CACHE_PURGE_TIME` parameters.

`ORDER_OPERATION_CACHE_PURGE_INTERVAL`

- This constant value defines the name of the configuration parameter that specifies the purging interval, in seconds, for automatic purging of rows in the order operation cache.
- The default is `0`, which means that rows are not automatically purged.
- For example, if you set this parameter to `60` then the automatic purging algorithm runs every 60 seconds.

`ORDER_OPERATION_CACHE_PURGE_TIME`

- This constant value defines the name of the configuration parameter that specifies the purge time for the order operation cache.
- The format for the purge time is *`hh`*`:`*`mm`*`:ss`. The default is that no purge time is set.

- Use this parameter to set a single time of day to purge the cache.

`ORDER_OPERATION_CACHE_MAX_ROWS`

- This constant value defines the name of the configuration parameter that specifies the maximum number of rows that can be automatically purged.
- The default is `0`, which means that rows are not automatically purged.

`ORDER_OPERATION_CACHE_PATH_DEFAULT`

- This constant value defines the name of the configuration parameter that specifies the default file and path for the tables in the order operation store.
- The default is `OrderOperationCache.sqlite`.

`ORDER_OPERATION_CACHE_PATH_KEY`

- This constant value defines the name of the configuration parameter that specifies the configuration key value for setting the risk firewall default file and path for the tables in the order operation store.
- The default is `OrderOperationCacheStorePath`.
- Use this parameter to override the default storage location used by risk firewalls.

Rule class parameters

`RULE_CLASS_PRIORITY_HIGH`

- This constant value defines the high priority value for a registered rule class.
- The default is 0.
- A rule class with this priority is evaluated first.

`RULE_CLASS_PRIORITY_LOW`

- This constant value defines the low priority value for a registered rule class.
- The default is 100.
- A rule class with this priority is evaluated last unless you have defined your own priority values that are greater than this value.

`RULE_CLASS_PRIORITY_MEDIUM`

- This constant defines the medium priority value for a registered rule class.
- The default is 50.
- This is the default priority assigned to each rule class upon registration.

`RULE_CLASS_STATE_PARAM_INSTANCE_COUNT`

- This constant value defines the name of the parameter that contains the number of rule class instances that have been added for a particular rule class. The risk firewall provides this count when it returns information for registered rule classes as a result of a call to `RiskFirewall.getAllRuleClassInstanceInfo()` or `RiskFirewall.getRuleClassInfo()`.
- There is no default.

`CONFIG_RECOVERED_RULE_INSTANCE_ID`

- When a rule class instance is recovered after correlator re-start, the risk firewall adds this configuration key to the configuration parameters associated with the rule class instance. This parameter defines the rule class instance Id that was used previously in the correlator. This parameter can be useful, for example, if you have a custom rule class implementation that performs recovery of custom state information from persistence.
- The default is `"RECOVERED_RULE_INSTANCE_ID"`.

Rule class instance parameters

`SLICE_EXCHANGEID`

- This constant value defines the name of the rule class instance configuration parameter that can specify a list of exchange Ids. If this parameter has a value, then for this rule class instance to evaluate an incoming order, the incoming order must have one of the specified exchange Ids.
- The default is that this parameter has no value. If you specify a value for this parameter it must be a sequence of strings.
- The default is that the exchange Id slice parameter is a wildcard, that is, it matches all exchange Ids. When this parameter has a value the rule class instance is evaluated for any orders that have one of the specified exchange Ids.

`SLICE_EXTRAPARAM`

- This constant value defines the name of the rule class instance configuration parameter that can specify a list of extra parameters. If this parameter has a value, then for this rule class instance to evaluate an incoming order, the incoming order must have a field that matches one of the keys in the specified dictionary of extra parameters, and the value of the field in the incoming event must be one of the values in the sequence for that key in the specified dictionary.
- The default is that this parameter has no value. If you assign a value to this parameter, it must be `dictionary<string, sequence<string> >`. This provides a set of extra parameter keys, and a set of possible values for each key.
- The default is that the extra parameters slice parameter is a wildcard, that is, it matches all extra parameters. When this parameter has a value the rule class instance is evaluated for any orders that have one of the specified extra parameters.

`SLICE_MARKETID`

- This constant value defines the name of the rule class instance configuration parameter that can specify a list of market Ids. If this parameter has a value, then for this rule class instance to evaluate an incoming order, the incoming order must have one of the specified market Ids.
- The default is that this parameter has no value. If you specify a value for this parameter it must be a sequence of strings.
- The default is that the market Id slice parameter is a wildcard, that is, it matches all market Ids. When this parameter has a value the rule class instance is evaluated for any orders that have one of the specified market Ids.

`SLICE_SERVICEID`

- This constant value defines the name of the rule class instance configuration parameter that can specify a list of service Ids. If this parameter has a value, then for this rule class instance to evaluate an incoming order, the incoming order must have one of the specified service Ids.
- The default is that this parameter has no value. If you specify a value for this parameter it must be a sequence of strings.
- The default is that the service Id slice parameter is a wildcard, that is, it matches all service Ids. When this parameter has a value the rule class instance is evaluated for any orders that have one of the specified service Ids.

`SLICE_SYMBOL`

- This constant value defines the name of the rule class instance configuration parameter that can specify a list of symbols. If this parameter has a value, then for this rule class instance to evaluate an incoming order, the incoming order must have one of the specified symbols.
- The default is that this parameter has no value. If you specify a value for this parameter it must be a sequence of strings.
- The default is that the symbol slice parameter is a wildcard, that is, it matches all symbols. When this parameter has a value the rule class instance is evaluated for any orders that have one of the specified symbols.

`SLICE_TRADERID`

- This constant value defines the name of the rule class instance configuration parameter that can specify a list of trader Ids. If this parameter has a value, then for this rule class instance to evaluate an incoming order, the incoming order must have one of the specified trader Ids.
- The default is that this parameter has no value. If you specify a value for this parameter it must be a sequence of strings.
- The default is that the trader Id slice parameter is a wildcard, that is, it matches all trader Ids. When this parameter has a value the rule class instance is evaluated for any orders that have one of the specified trader Ids.

Risk firewall query parameters

`CONFIG_ADD_QUERY_RESPONSE_FAIL`

- This constant value defines the name of the configuration parameter that indicates whether failure responses from risk firewall query requests should be added to the query response object.
- The default is that the failure responses are added.
- To change the setting of this parameter, use the `CONFIG_ADD_QUERY_RESPONSE_FAIL` key and set it to the `boolean` value you need.

`CONFIG_ADD_QUERY_RESPONSE_PASS`

- This constant value defines the name of the configuration parameter that indicates whether pass responses from risk firewall query requests should be added to the query response object.
- The default is that the pass responses are not added
- To change the setting of this parameter, use the `CONFIG_ADD_QUERY_RESPONSE_PASS` key and set it to the `boolean` value you need.

`CONFIG_ADD_QUERY_RESPONSE_WARN`

- This constant value defines the name of the configuration parameter that indicates whether warning responses from risk firewall query requests should be added to the query response object.
- The default is that the failure responses are added
- To change the setting of this parameter, use the `CONFIG_ADD_QUERY_RESPONSE_WARN` key and set it to the `boolean` value you need.

Order Management Parameters

`CONFIG_BUSTED_FILL_DURATION`

- This constant value defines the name of the configuration parameter that indicates the maximum duration that the risk firewall waits for updates to orders after they have been completed. If an order update arrives within this duration the risk firewall forwards it to the order sender that originated the order and the application determines what happens.
- The default is 5 seconds.
- To change the setting of this parameter, use the `CONFIG_BUSTED_FILL_DURATION` key and set it to the `float` value you need.

`CONFIG_CHANGE_LATENCY_TIMESTAMPID_BASE`

- This string constant defines the name of the configuration parameter used to alter the base number for the latency timestamp identifiers for inbound and outbound order management events. This may be useful if, for example, more than one risk firewall is used in the order management chain.
- The default is `"CHANGE_INBOUND_NEWORDER_TIMESTAMPID"`.

`CONFIG_DEFAULT_LATENCY_TIMESTAMPID_BASE`

- This integer constant defines the default identifier that all inbound/outbound timestamps for the risk firewall will use as a base identifier. All risk firewall timestamp identifiers are offset from this base identifier.
- The default is 7000.
- Timestamp identifiers are used as dictionary keys. If you have multiple risk firewalls in the chain of components that an order must pass through, the timestamp identifier would be overwritten by successive risk firewalls. In that situation, you can change the value of the timestamp identifier base so that each risk firewall uses a different base value. You might also want to change the value of the base identifier if you have another component that is adding timestamps with an identifier key of 7000. For more information about timestamp identifiers, see "Measuring order handling performance in the risk firewall" on page 161.

`CONFIG_INBOUND_AMENDORDER_TIMESTAMPID_OFFSET`

- This constant defines the default identifier for inbound `AmendOrder` timestamps.
- The default is 2.

`CONFIG_INBOUND_CANCELORDER_TIMESTAMPID_OFFSET`

- This constant defines the default identifier for inbound `CancelOrder` timestamps.
- The default is 4.

`CONFIG_INBOUND_NEWORDER_TIMESTAMPID_OFFSET`

- This constant defines the default identifier for inbound `NewOrder` timestamps.
- The default is 0.

`CONFIG_INBOUND_ORDERUPDATE_TIMESTAMPID_OFFSET`

- This constant defines the default identifier for inbound `OrderUpdate` timestamps.
- The default is 6.

`CONFIG_OUTBOUND_AMENDORDER_TIMESTAMPID_OFFSET`

- This constant defines the default identifier for outbound `AmendOrder` timestamps.
- The default is 3.

`CONFIG_OUTBOUND_CANCELORDER_TIMESTAMPID_OFFSET`

- This constant defines the default identifier for outbound `CancelOrder` timestamps.
- The default is 5.

`CONFIG_OUTBOUND_NEWORDER_TIMESTAMPID_OFFSET`

- This constant defines the default identifier for outbound `NewOrder` timestamps.
- The default is 1.

`CONFIG_OUTBOUND_ORDERUPDATE_TIMESTAMPID_OFFSET`

- This constant defines the default identifier for outbound `OrderUpdate` timestamps.
- The default is 7.

`CONFIG_LOG_AMEND_CANCEL_LATENCY_TIMESTAMPS`

- This constant value defines the name of the configuration parameter that indicates whether to log latency timestamps for performance measurement for `AmendOrder` and `CancelOrder` requests passed into the risk firewall.
- The default is that latency timestamps for *`AmendOrder`* and `CancelOrder` requests are logged.
- To disable logging of `AmendOrder` and `CancelOrder` requests, use the `CONFIG_LOG_AMEND_CANCEL_LATENCY_TIMESTAMPS` key and set it to the `boolean` value you need.

`CONFIG_LOG_INBOUND_LATENCY_TIMESTAMPS`

- This constant value defines the name of the configuration parameter that indicates whether to log latency timestamps for performance measurement for order management requests passed into the risk firewall.
- The default is that latency timestamps for inbound order management requests are not logged.
- To enable logging of inbound order management requests, use the `CONFIG_LOG_INBOUND_LATENCY_TIMESTAMPS` configuration key and set it to the `boolean` value you need.

`CONFIG_LOG_OUTBOUND_LATENCY_TIMESTAMPS`

- This constant value defines the name of the configuration parameter that indicates whether to log latency timestamps for performance measurement for requests passed out of the risk firewall.
- The default is that latency timestamps for outbound order management requests are not logged.
- To enable logging of outbound order management requests, use the `CONFIG_LOG_OUTBOUND_LATENCY_TIMESTAMPS` configuration key and set it to the `boolean` value you need.

Order Rejection Parameters

`CONFIG_FAST_FAIL_MODE`

- This constant value defines the name of the configuration parameter that indicates whether the risk firewall runs in fast fail mode. Fast fail mode rejects an order upon the first failure response to a rule query. The combined query response contains only the single failure.
- The default is that fast fail mode is enabled.
- To disable fast fail mode, use the `CONFIG_FAST_FAIL_MODE` key and set it to the `boolean` value you need. If you disable fast fail mode all query failure/warning responses are collated into the combined query response.

`CONFIG_REJECT_BY_DEFAULT`

- This constant value defines the name of the configuration parameter that indicates whether the risk firewall rejects an order that does not match any slice criteria defined for any rule class instance that has been added to the risk firewall.
- The default is that reject by default is enabled.
- To disable this rejection mode , use the `CONFIG_REJECT_BY_DEFAULT` key and set it to the `boolean` value you need. If you disable reject by default mode then an order that does not match any slice criteria defined in the risk firewall is automatically approved.

`CONFIG_SOFT_REJECT_DURATION`

- This constant value defines the name of the configuration parameter that indicates the maximum period the risk firewall pends orders while waiting for a rejection override when running in soft rejection mode.
- The default is 60 seconds.
- To change the setting of this configuration parameter, use the `CONFIG_SOFT_REJECT_DURATION` key and set it to the `float` value you need.

`CONFIG_REJECTION_MODE`

- This constant value defines the name of the configuration parameter that indicates the rejection mode being used by the risk firewall. For details about each rejection mode, see "Configuring risk firewall behavior for rejected orders" on page 109.
- The default is that the risk firewall uses hard rejection mode.
- To change the setting of this configuration parameter, set `CONFIG_REJECTION_MODE` to one of the following configuration parameters.
  - `CONFIG_REJECTION_MODE_HARD`
  - `CONFIG_REJECTION_MODE_MONITOR`
  - `CONFIG_REJECTION_MODE_SOFT`

Legacy risk firewall parameters

`CONFIG_ENABLE_LEGACY_MODE`

- This constant value defines the name of the configuration parameter that indicates whether the risk firewall is operating in legacy mode. In legacy mode, the risk firewall listens for and routes approved OMS events as described here: "Configuring a risk firewall to support the legacy order event interface firewall" on page 175.
- The default is that this parameter is disabled.
- To enable legacy mode, set `CONFIG_ENABLE_LEGACY_MODE`, which is a `boolean`, to `true`.

`LEGACY_FIREWALL_SERVICEID`

- This constant value defines the service Id that was used to send OMS events through the legacy risk firewall. This is defined for use by the risk firewall when used in legacy mode.

- The value of this constant is `"__ObjectionBasedFirewallControllerExternal"`.

`CONFIG_TARGET_SERVICE_EXTRA_PARAM`

- This constant value defines the key name in the OMS event's extra parameters field whose value indicates the service Id to send OMS events to. A risk firewall that is operating in legacy mode uses this setting.
- The default is `"Firewall.TargetService"`.

Overriding default error handling for risk firewalls

The default error handler is invoked if there is an error related to a risk firewall instance. For example, if any of the following happen:

- Registration of a rule class fails.
- Addition, modification, or removal of a rule class instance fails.
- There is an attempt to execute an action that is not available on a risk firewall that is connected to a remote risk firewall instance.
- There is an attempt to update, amend, or cancel an order for which the specified order Id is unknown.
- You try to remove an error callback and specify an incorrect error callback reference identifier.
- You try to set a parameter and specify an invalid parameter name.

The default error handler sends a message to the correlator log file at the `ERROR` level. To change this behavior, execute `com.apama.firewall.RiskFirewall.addErrorCallback()`, which adds the specified callback to the set of callbacks executed if there is an error related to that risk firewall instance.

You can execute the `addErrorCallback()` action multiple times on the same risk firewall instance to implement multiple error handling callbacks for that risk firewall instance. If you add one or more error callbacks to a risk firewall instance then the default error callback is not executed for that risk firewall instance. However, you can call the default error callback from your own error callback by using the `RiskFirewall.defaultErrorCallback()` action and providing the parameters. The default error callback will send a message to the correlator log file at the `ERROR` level.

The parameters of a user-defined error callback include the risk firewall instance and also a `com.apama.utils.Error` event. An `Error` event has fields for a message, a dictionary of parameters, and an error type code. The `addErrorCallback()` action adds the specified callback to the set of callbacks executed if there is an error in the operation of the specified risk firewall instance.

The `com.apama.firewall.ErrorConstants` event defines the error type codes that can apply to risk firewall instances. See the ApamaDoc for details.

When you add an error callback the return value is an integer reference ID that you can specify if you execute `RiskFirewall.removeErrorCallback()` to discontinue execution of that error callback. To remove all error callbacks, execute the `RiskFirewall.clearErrorCallbacks()` action. If you

remove all previously set error callbacks then error handling behavior reverts to calling the default error callback.

Descriptions of DataViews exposed by risk firewalls

A risk firewall exposes the following DataViews:

- `RiskFirewallDataview` — For each risk firewall that has been created in this correlator, this DataView contains
  - The name of the risk firewall
  - A string representation of the risk firewall parameters object
  - An indication of whether the risk firewall is locked
- `FirewallRuleDefinitionDataView` — For all rules registered with all risk firewalls in this correlator, this DataView contains
  - The name of the risk firewall
  - The name of the rule class
  - The description of the rule class
  - A string representation of the parameter schema (`com.apama.utils.ParamsSchema`) for this rule class
  - The priority of the rule class
- `AllRuleClassInstances` — This DataView is exposed by the MemoryStore table that contains entries for all rule class instances for all risk firewalls in this correlator. This DataView contains
  - The name of the risk firewall
  - The name of the rule class
  - The Id of the rule instance
  - A string representation of rule instance's configuration
- `OrderOperationCache` — This DataView is exposed by the MemoryStore table that contains entries for the order operation cache, which contains information for each failed order operation that has gone through any risk firewall in this correlator. This DataView contains extensive information for each failed order operation, including risk firewall name, rule class name, query request Id, rule instance Id, order operation type, and much much more. The entries in the order operation cache, and hence in the MemoryStore table, are recovered upon correlator re-start and automatically purged according to the settings of these parameters:
  - `ORDER_OPERATION_CACHE_TIMEOUT`
  - `ORDER_OPERATION_CACHE_PURGE_INTERVAL`
  - `ORDER_OPERATION_CACHE_PURGE_TIME`

- `ORDER_OPERATION_CACHE_MAX_ROWS`
- `ORDER_OPERATION_CACHE_PATH_DEFAULT`
- `ORDER_OPERATION_CACHE_PATH_KEY`

If you set values for these parameters, you must do so in the risk firewall factory before you create the risk firewall. For a risk firewall instance, you cannot change the value for one of these parameters after you create the instance. For details, see "Descriptions of risk firewall configuration parameters" on page 113.

The order operation cache can get very large. If rows are not deleted either through automatic purging or by invoking `RiskFirewall.clearObjection()` then the cache contains all failed order operations that have entered a risk firewall in this correlator. Order operation cache content is persisted between correlator re-starts.

By default, there is no auto-purging of the order operation cache. An alternative to auto-purging is to manually remove information for failed operations by invoking the `RiskFirewall.clearObjection()` action. This action takes two parameters, the name of the rule class that generated the objection and the request Id of the objected-to operation. You can obtain the request Id from the `OrderOperationCache` DataView.

Setting up risk firewall evaluation rules

For each risk firewall instance you create, you must register at least one rule class. For each default rule class that you register, you must also add at least one rule class instance. For each custom rule class that you register, you might or might not need to add at least one rule class instance. This depends on how you implemented your custom rule class. See "Implementing custom risk firewall rule classes" on page 168.

For example, consider `com.apama.firewall.rules.OrderPriceLimitRiskFirewallRule`. This rule class checks the order price of any new order, or amendment to an order, and rejects that order if the price exceeds an amount specified in a rule class instance. For example, you might have added a rule class instance that specifies that the price limit is 20.0 when the symbol is `SOW`. The risk firewall would reject an order for `SOW` when the price is more than 20.0.

For the risk firewall to approve an order, the order must comply with (it must match) at least one instance of each registered rule class. If your custom rule class does not require the addition of a rule class instance, then an order must match the rule class for the risk firewall to approve the order.

Registering rule classes with a risk firewall

A risk firewall instance provides two actions for registering a rule class:

- `RiskFirewall.registerRuleClass()` takes one argument, which is the `com.apama.firewall.RuleClass` implementation to register. This action registers that rule class with the risk firewall instance it is called on.

- `RiskFirewall.registerRuleClassCb()` takes two arguments. The first argument is the `com.apama.firewall.RuleClass` implementation to register. The second argument is a callback that gets executed if registration is successful.

Both actions are asynchronous, their availability is pending until the risk firewall is created, and they are not available from a risk firewall that is connected to a remote risk firewall instance. It is an error if the time period specified by the `CONFIG_TIMEOUT_DURATION` parameter elapses before the risk firewall is created. This prevents the risk firewall from being blocked while waiting to register a rule class implementation that has possibly entered an error state.

A number of rule classes are provided with the risk firewall. See "Descriptions of default rule classes" on page 126. If these rule classes do not meet your application requirements, see "Implementing custom risk firewall rule classes" on page 168.

Descriptions of default rule classes

The CMF provides the following rule classes, which you can immediately instantiate and register:

Rule Class	Description
Client credit limit	Ensures that the net cash open and pending position accumulated based on a specified slice has not been breached. Currency normalization is not supported.
Order operation ratio	Ensures that the number of amend order operations and/or cancel order operations in a specified time window do not exceed the allowed number of amend/cancel operations as defined by the specified ratio between amend/cancel order operations and new orders.
Order price limit	Ensures that the price of an order does not fall outside a specified price range.
Order quantity limit	Ensures that the quantity of an order does not fall outside a specified range.
Order throttle limit	Prevents more than a specified number of new orders, amendments and/or cancellations from being placed within a specified time period.
Order to trade ratio	Ensures that the number of order operations (new, amend, cancel) in a specified time window do not exceed the allowed number of operations as defined by the specified ratio between order operations and trades.
Order value limit	Ensures that the value (price * quantity) of an order does not fall outside a specified range.
Position limit	Ensures that the net quantity open and pending position of an order does not fall outside a specified quantity range.
Reservation enforcer	Ensures that orders placed against reservation contracts do not exceed the minimum/maximum quantities set for reserved positions.

The default rule classes are in the `com.apama.firewall.rules` package. Each default rule class has an associated event that defines the configuration parameters for that rule class. See "Adding rule

instances to rule classes" on page 146. In addition, for each default rule class, you can specify the following configuration parameters for each rule class instance that you add to a risk firewall.

Parameter	Type	Defines a set of ... that will be used as a slice for the specific rule class instance being added or modified
SLICE_SYMBOL	sequence <string>	Symbols
SLICE_SERVICEID	sequence <string>	Service identifiers
SLICE_MARKETID	sequence <string>	Market identifiers
SLICE_EXCHANGEID	sequence <string>	Exchange identifiers
SLICE_TRADERID	sequence <string>	Trader identifiers
SLICE_EXTRAPARAM	dictionary<string, sequence<string> >	Extra parameters

See "Examples of slice filters for rule class instances" on page 148.

The source code for the default rule classes is provided as samples that you can use to develop custom rule classes that are based on the default rule classes. You can find the source code in the `samples\Risk Firewall Rules Sample` folder of your CMF installation folder. Note that in the sample, the package names have been changed so that they do not conflict with the actual default implementation.

About the ClientCreditLimitRiskFirewallRule

The `ClientCreditLimitRiskFirewallRule` checks the total value of all orders that are being placed by a specific trader, or set of traders, to ensure that the total value does not breach the maximum credit limit that a trader is allowed.

Note:
The difference between the client credit limit rule class and the position limit rule class is that the client credit limit rule class acts on the cash position, which is the cumulative value of all orders, where as the position limit rule class acts on the cumulative quantity position.

When adding a client credit limit rule class instance, set either individual minimum and maximum limits, or a symmetric limit. For example, 100 would set a positive limit to +100 and a negative limit to -100. Setting both symmetric and asymmetric configuration parameters for the same limit (warning or objection) results in an error, and the rule class instance is not added.

This rule implementation relies on the order price being available when an order is placed against it, so that the rule can determine the order value. Therefore, this rule will not operate as expected for market orders being placed with a zero price value. In this case, the initial order may be allowed through the risk firewall. When the order receives a fill, then the risk firewall rule instance will be updated to reflect the current client credit position. At that point, any new orders, including market orders, will be rejected.

This risk firewall rule implementation is not designed for foreign exchange (FX) or mixed markets, as this implementation does not normalize the order values to a base currency value. It will use the value in the native currency for that order.

This risk firewall rule checks both the open and pending positions of matching orders.

When you use the client credit limit rule class (or the position limit rule class) there are callbacks that you must add. Specifically, you must add at least one of each type of callback to the order receiver that will handle orders approved by these rule classes:

- `OrderReceiver.addAcceptedOrderCallback()`
- `OrderReceiver.addAcceptedAmendOrderCallback()`
- `OrderReceiver.addAcceptedCancelOrderCallback()`

Each callback must route or enqueue the order event (`com.apama.oms.NewOrder`, `AmendOrder`, `CancelOrder`) to the context that contains the position tracker that the application is using. This is required for the rule class to operate in the expected way.

The client credit limit rule class uses the CMF position service to calculate the cumulative cash position being tracked. This requires the application to have created both an open position tracker and a pending position tracker (see "Using default position trackers" on page 193). You then pass the names of these position tracker instances into the rule class's `create()` action. This allows the rule class to track the positions of orders from other areas of your application.

The CMF's position service requires the order management (`com.apama.oms`) events to be sent (either routed or enqueued) to the position tracker instances that you created. This means that your application code must ensure that OMS events are sent to the position trackers being used after the orders have been approved by the risk firewall. Failure to do so may result in the position being calculated incorrectly, and subsequently the rule class may approve orders that would breach the defined limits.

CAUTION:
Where possible, the position trackers used should be in the same context as the risk firewall. This can help avoid a situation in which the client credit limit rule class approves orders that breach the defined limits. Such a situation might happen because the risk firewall can accept orders synchronously by means of callback actions, but the CMF position service is asynchronous. The position service listens for OMS events, and routes or enqueues position updates. Orders that are sent into the risk firewall should allow time for the correlator input queue to process the position update. The code sample at the end of this topic demonstrates what you must do to correctly implement the client credit limit rule class. Alternatively, you may implement a custom rule class implementation (see "Implementing custom risk firewall rule classes" on page 168) that can operate safely in a synchronous architecture.

The `com.apama.firewall.rules.ClientCreditLimitRiskFirewallConsts` object defines the constants for the configuration settings for the client credit limit rule class, as well as their default values. To configure a rule class instance, you should use these constants rather than specific values in order to ensure compatibility. The following table provides information about client credit limit rule class configuration parameters.

Constant Name of Input Parameter	Description
RULE_CLASS_NAME	The rule class name, which is "ClientCreditLimitRiskFirewallRule".
CLIENT_CREDIT_LIMIT_PARAMETER	Symmetric objection limits. This is the symmetric cash position up to which orders may accrue before the rule class rejects subsequent orders.
CLIENT_CREDIT_LIMIT_DEFAULT	Symmetric objection limits default, which is 0.0 by default.
CLIENT_CREDIT_WARNING_PARAMETER	Symmetric warning limits. This is the symmetric cash position up to which orders may accrue before the rule class issues a warning for subsequent orders.
CLIENT_CREDIT_WARNING_DEFAULT	Symmetric warning limits default, which is 0.0 by default.
MIN_CLIENT_CREDIT_LIMIT_PARAMETER	Minimum objection limit. This is the minimum cash position up to which orders may accrue before the rule class rejects subsequent orders.
MIN_CLIENT_CREDIT_LIMIT_DEFAULT	Minimum objection limit default value, which is 0.0.
MIN_CLIENT_CREDIT_WARNING_PARAMETER	Minimum warning limit. This is the minimum cash position up to which orders may accrue before the rule class issues a warning for subsequent orders.
MIN_CLIENT_CREDIT_WARNING_DEFAULT	Minimum warning limit default value, which is 0.0 by default.
MAX_CLIENT_CREDIT_LIMIT_PARAMETER	Maximum objection limit. This is the maximum cash position up to which orders may accrue before the rule class rejects subsequent orders.
MAX_CLIENT_CREDIT_LIMIT_DEFAULT	Maximum objection limit default value, which is 0.0 by default.
MAX_CLIENT_CREDIT_WARNING_PARAMETER	Maximum warning limit. This is the maximum cash position up to which orders may accrue before the rule class issues a warning for subsequent orders.
MIN_CLIENT_CREDIT_WARNING_DEFAULT	Maximum warning limit default value, which is 0.0 by default.

Constant Name of Input Parameter	Description
CHECK_CANCEL_PARAMETER	Indicates whether this risk firewall rule instance should apply to `CancelOrder` objects.
CHECK_CANCEL_DEFAULT	Default value for whether this risk firewall rule instance should apply to `CancelOrder` objects. By default, this is `true`, which means that cancel order requests are counted.
SLICE_POSITION_SERVICEID	The set of service IDs that the position service will use for this instance if it needs to be different from those provided for the instance slice. The format is a stringified `sequence<string>`. This must be set when running in legacy mode. You may want to check against the position of orders that have been approved by a risk firewall in legacy mode.

The following table provides information about client credit limit rule class state information parameters. The risk firewall uses these constants as part of the state returned in the `com.apama.firewall.InstanceInfo` object when you call the `com.apama.firewall.RiskFirewall.getRule`*`xxx`*`Info()` set of actions.

Constant Name of Output Parameter	Description
CURRENT_OPEN_POSITION	Float value for the current open cash position for a rule class instance.
CURRENT_PENDING_MIN_POSITION	Minimum pending cash position for a rule instance.
CURRENT_PENDING_MAX_POSITION	Maximum pending cash position for a rule instance.
CURRENT_TOTAL_MIN_POSITION	Current total minimum (open + minimum pending) cash position for a rule class instance.
CURRENT_TOTAL_MAX_POSITION	Current total maximum (open + maximum pending) cash position for a rule class instance.

The following code provides an example of using the client credit limit rule class.

```
using com.apama.position.tracker.OpenPositionTrackerFactory;
using com.apama.position.tracker.PendingPostionTrackerFactory;

using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;
using com.apama.firewall.RuleClassFactory;
using com.apama.firewall.rules.ClientCreditLimitRiskFirewallRule;
using com.apama.firewall.rules.ClientCreditLimitRiskFirewallRuleConsts;

using com.apama.utils.Params;
using com.apama.oms.NewOrder;
```

```
using com.apama.oms.AmendOrder;
using com.apama.oms.CancelOrder;
using com.apama.oms.UpdateOrder;

monitor ClientCreditLimitExample {
   context mainContext := context.current();
   integer trackersCreated := 0;

   action onload() {

      // Create an instance of the open position tracker in the main context.
      (new OpenPositionTrackerFactory).create(
         mainContext, "MyOpenPositionTracker", cbCreatedTracker );
      // Create an instance of the pending position tracker in the main context.
      (new PendingPositionTrackerFactory).create(
         mainContext, "MyPendingPositionTracker", cbCreatedTracker );
   }

   // This action is called after the position tracker instance
   // has been created.
   action cbCreatedTracker( boolean success, string msg ) {
      log "Position Tracker has been created: "+success.toString()+" : "+msg;

      trackersCreated := trackersCreated +1;

      // After all trackers have been created, then start the test.
      if( trackersCreated = 2 ) then {
         startTest();
      }
   }

   action startTest() {
      // Create the risk firewall.
      com.RiskFirewall rfw := (new RiskFirewallFactory).createCb(
         mainContext, "MyFirewall", cbOnRiskFirewallCreated );

      // Create and register the client credit limit rule class.
      RuleClass ruleClass := (new ClientCreditLimitRiskFirewallRule).create(
         mainContext, "MyOpenPositionTracker", "MyPendingPositionTracker" );
      rfw.registerRuleClass( ruleClass );

      // Add rule instances.
      Params instanceConfig := new Params;
      instanceConfig.addFloatParam(
         ClientCreditLimitRiskFirewallRuleConsts.
            CLIENT_CREDIT_LIMIT_PARAMETER, 500.0 );
      instanceConfig.addFloatParam(
         ClientCreditLimitRiskFirewallRuleConsts.
            CLIENT_CREDIT_WARNING_PARAMETER,450.0 );

      integer instanceId := rfw.addRuleInstance(
         ruleClass.getRuleClassName(), instanceConfig );

      // Add handlers for approved OMS events from the risk firewall.
      integer refId1 := rfw.getOrderReceiver().addAcceptedOrderCallback(
         cbOnApprovedNewOrders );
      integer refId2 := rfw.getOrderReceiver().addAcceptedAmendCallback(
         cbOnApprovedAmendOrders );
      integer refId3 := rfw.getOrderReceiver().addAcceptedCancelCallback(
         cbOnApprovedCancelOrders );
```

```
      // Add a callback for obtaining OrderUpdate events from the risk firewall.
      integer refId4 := rfw.getOrderSender().addOrderUpdateCallback(
         cbOnOrderUpdates );

      // The unlock will take effect immediately after the create() action.
      rfw.unlock();
   }

   action cbOnRiskFirewallCreated( RiskFirewall rfw ) {

      // Send a new order to the risk firewall.
      rfw.getOrderSender().sendOrder( com.apama.oms.NewOrder(
         "orderId_1","APMA.L", 9.0, "BUY", "LIMIT", 10,
         "TargetService","","", "TargetMarket","", "TraderA",
         new dictionary<string,string> ) );
}

   // Set up a callback handler for NewOrder events approved by the
   // risk firewall.
   action cbOnApprovedNewOrders( NewOrder order ) {
      // Send the NewOrder event to the context that the position trackers
      // were created in.
      route order;

      // The application can now perform any other actions
      // on this NewOrder event.
      ...
   }

   // Set up a callback handler for AmendOrder events approved by the
   // risk firewall.
   action cbOnApprovedAmendOrders( AmendOrder amend ) {
      // Send the AmendOrder event to the context that the position trackers
      // were created in.
      route amend;

      // The application can now perform any other actions
      // on this AmendOrder event.
      ...
   }

   // Set up a callback handler for CancelOrder events approved by the
   // risk firewall.
   action cbOnApprovedCancelOrders( CancelOrder cancel ) {
      // Send the CancelOrder event to the context that the position trackers
      // were created in.
      route cancel;

      // The application can now perform any other actions
      // on this CancelOrder event.
      ...
   }

   // This action will be called for any updates to the
   // orders the risk firewall is handling.
   action cbOnOrderUpdates( OrderUpdate update ) {
      // Send the OrderUpdate event to the context that the position trackers
      // were created in
      route update;
```

```
      // The application can now perform any other actions
      // on this OrderUpdate event.
      ...
   }
}
```

About the OrderOperationRatioRiskFirewallRule

The `OrderOperationRatioRiskFirewallRule` class ensures that the number of amend order operations and/or cancel order operations in a specified time window (30 seconds by default) do not exceed the allowed number of amend/cancel operations as defined by the specified ratio between amend/cancel order operations and new order operations. An amend or cancel order operation that would exceed the allowed number of amend/cancel operations is rejected by this rule. By default, both amend and cancel order operations are checked against new order operations.

The `com.apama.firewall.rules.OrderOperationRatioRiskFirewallConsts` object defines the constants for the configuration settings for the order operation ratio rule class, as well as their default values. You should use these constants rather than specific values in order to ensure compatibility.

Constant Name of Input Parameter	Description
`RULE_CLASS_NAME`	Rule class name, which is `"OrderOperationRatioRiskFirewallRule"`.
`OBJECTION_LIMIT_PARAMETER`	Specifies the allowed ratio of amend/cancel orders to new orders, which determines the number of order operations (amend or cancel) that can be requested before the rule rejects the operation request. You must set a value for this parameter. For example, suppose you set this parameter to `5`. If there are 3 new orders in a 30 second window then the ratio is 15:3. This rule would allow up to 15 amend/cancel operations to those 3 new orders until another new order is received or the time window for a new order expires. Note that, in the example, each new order is not limited to only 5 amend/cancel operations. Each of the 15 allowed operations can apply to any of the new orders.
`OBJECTION_LIMIT_DEFAULT`	Default value for the objection limit, which is `0` by default.
`WARNING_LIMIT_PARAMETER`	Specifies the allowed ratio of amend/cancel orders to new orders before the risk firewall issues a warning when it allows the operation.

Constant Name of Input Parameter	Description
WARNING_LIMIT_DEFAULT	Default value for the warning limit, which is 0 by default.
TIME_WINDOW_PARAMETER	Number of seconds in the rolling time window.
TIME_WINDOW_DEFAULT	Default number of seconds in the rolling time window, which is 30.0 by default.
CHECK_AMEND_PARAMETER	Indicates whether this risk firewall rule instance should apply to AmendOrder objects.
CHECK_AMEND_DEFAULT	Default value for whether this risk firewall rule instance should apply to AmendOrder objects. By default, this is false, which means that amend order requests are not counted as part of the total number of requests allowed within the specified time window.
CHECK_CANCEL_PARAMETER	Indicates whether this risk firewall rule instance should apply to CancelOrder objects.
CHECK_CANCEL_DEFAULT	Default value for whether this risk firewall rule instance should apply to CancelOrder objects. By default, this is false, which means that cancel order requests are not counted as part of the total number of requests allowed within the specified time window.

The following table provides information about order operation ratio rule class state information parameters. The risk firewall uses these constants as part of the state returned in the com.apama.firewall.InstanceInfo object when you call the com.apama.firewall.RiskFirewall.getRule*xxx*Info() set of actions.

Constant Name of Output Parameter	Description
NEW_ORDER_PER_WIN_OUTPUT	Number of new orders in the current time window.
ORDER_OPS_PER_WIN_OUTPUT	Number of order operations (amend or cancel) in the current time window.

About the OrderPriceLimitRiskFirewallRule

The OrderPriceLimitRiskFirewallRule class checks the order price of any order or amendment to an order and approves that order if the price is equal to or less than the price specified in at least one rule instance configuration.

Note:
This risk firewall rule implementation is not appropriate for market orders. This is because market orders inherently have a price of zero when they are placed and this rule class implementation relies on the order price being available when an order is placed against it.

Therefore, this rule does not operate as expected for market orders since it approves any market order.

The `com.apama.firewall.rules.OrderPriceLimitRiskFirewallConsts` object defines the constants for the configuration settings for the order price limit rule class, as well as their default values. You should use these constants rather than specific values in order to ensure compatibility. Set either individual minimum and maximum limits, or set a symmetric limit. For example, for a symmetric parameter, 100 would set a maximum limit to +100 and a minimum limit to -100. Setting both symmetric and asymmetric configuration parameters for the same limit (objection or warning) results in an error, and the rule class instance is not added.

Constant Name of Input Parameter	Description
`RULE_CLASS_NAME`	Rule class name, which is `"OrderPriceLimitRiskFirewallRule"`.
`PRICE_LIMIT_PARAMETER`	Symmetric objection limits. The minimum and maximum price an order may request before the rule class rejects the order.
`PRICE_LIMIT_DEFAULT`	Symmetric objection limits default, which is `0.0` by default.
`PRICE_WARNING_PARAMETER`	Symmetric warning limits. The minimum and maximum price an order may request before the rule class issues a warning for the order.
`PRICE_WARNING_DEFAULT`	Symmetric warning limits default, which is `0.0` by default.
`MIN_PRICE_LIMIT_PARAMETER`	Minimum objection limit. The minimum price an order may request before the rule class rejects the order.
`MIN_PRICE_LIMIT_DEFAULT`	Minimum objection limit default value, which is `0.0`.
`MIN_PRICE_WARNING_PARAMETER`	Minimum warning limit. The minimum price an order may request before the rule class issues a warning for the order.
`MIN_PRICE_WARNING_DEFAULT`	Minimum warning limit default value, which is `0.0` by default.
`MAX_PRICE_LIMIT_PARAMETER`	Maximum objection limit. The maximum price an order may request before the rule class rejects the order.
`MAX_PRICE_LIMIT_DEFAULT`	Maximum objection limit default value, which is `0.0` by default.
`MAX_PRICE_WARNING_PARAMETER`	Maximum warning limit. The maximum price an order may request before the rule class issues a warning for the order.

Constant Name of Input Parameter	Description
MIN_PRICE_WARNING_DEFAULT	Maximum warning limit default value, which is 0.0 by default.

About the OrderQuantityLimitRiskFirewallRule

The OrderQuantityLimitRiskFirewallRule checks the order quantity of any new order or amendment to an order and approves that order if the quantity is equal to or less than the amount provided in the configuration parameters of at least one instance. This rule also rejects an order amendment that would reduce the quantity to less than the currently executed quantity.

The com.apama.firewall.rules.OrderQuantityLimitRiskFirewallConsts object defines the constants for the configuration settings for the order quantity limit rule class, as well as their default values. You should use these constants rather than specific values in order to ensure compatibility. Set either individual minimum and maximum limits, or set a symmetric limit. For example, for a symmetric parameter, 100 would set a maximum limit to +100 and a minimum limit to -100. Setting both symmetric and asymmetric configuration parameters for the same limit (objection or warning) results in an error, and the rule class instance is not added.

Constant Name of Input Parameter	Description
RULE_CLASS_NAME	The name of the rule class, which is "OrderQuantityLimitRiskFirewallRule".
QUANTITY_LIMIT_PARAMETER	Objection limits. The quantity up to which an order may request before the rule class rejects the order.
QUANTITY_LIMIT_DEFAULT	Objection limits default value, which is 0 by default.
QUANTITY_WARNING_PARAMETER	Warning limits. The quantity up to which an order may be placed with before the rule class issues a warning for the order.
QUANTITY_WARNING_DEFAULT	Warning limits default value, which is -1 by default. This indicates that the warning limit is disabled.
LIMIT_IS_REMAINING_PARAMETER	Indicates whether the limits are inclusive of the quantity already executed.
LIMIT_IS_REMAINING_DEFAULT	Default for whether the limits are inclusive of the quantity already executed, which is false by default.

About the OrderThrottleLimitRiskFirewallRule

The OrderThrottleLimitRiskFirewallRule stops more than a specified number of order operations (new, amend, cancel) within a specified rolling time window (30 seconds by default).

Any order operations that exceed the specified number allowed will be rejected by this risk firewall rule. The order operation types to be checked are configurable when the risk firewall rule instance

is created. By default, all order operations (new, amend, and cancel) are counted, but each of these types can be included/excluded in the throttle count.

The `com.apama.firewall.rules.OrderThrottleLimitRiskFirewallConsts` object defines the constants for the configuration settings for the order throttle limit rule class, as well as their default values. You should use these constants rather than specific values in order to ensure compatibility.

Constant Name of Input Parameter	Description
`RULE_CLASS_NAME`	The name of the rule class, which is `"OrderThrottleLimitRiskFirewallRule"`.
`OBJECTION_LIMIT_PARAMETER`	Objection limits. The maximum number of order management requests (which may include amend order and cancel order requests) that may be requested over the specified time window, after which the rule class rejects order management requests.
`OBJECTION_LIMIT_DEFAULT`	Default value for objection limits, which is `0` by default.
`WARNING_LIMIT_PARAMETER`	Warning limits. The maximum number of order management requests (which may include amend order and cancel order requests) that may be requested over the specified time window, after which the rule class issues a warning for subsequent order management requests.
`WARNING_LIMIT_DEFAULT`	Default value for warning limits, which `0` by default.
`TIME_WINDOW_PARAMETER_`	Time window to be used (in seconds).
`TIME_WINDOW_DEFAULT`	Default time window to be used, which is `30.0` seconds by default.
`CHECK_NEW_PARAMETER`	Indicates whether this risk firewall rule instance should apply to `NewOrder` objects.
`CHECK_NEW_DEFAULT`	Default value for whether this risk firewall rule instance should apply to `NewOrder` objects. By default, new order requests are counted as part of the total number of requests allowed within the specified time window.
`CHECK_AMEND_PARAMETER`	Indicates whether this risk firewall rule instance should apply to `AmendOrder` objects.
`CHECK_AMEND_DEFAULT`	Default value for whether this risk firewall rule instance should apply to `AmendOrder` objects. By default, this is `false`, which means that amend order requests are not counted as part of the total number of requests allowed within the specified time window.

Constant Name of Input Parameter	Description
CHECK_CANCEL_PARAMETER	Indicates whether this risk firewall rule instance should apply to CancelOrder objects.
CHECK_CANCEL_DEFAULT	Default value for whether this risk firewall rule instance should apply to CancelOrder objects. By default, this is false, which means that cancel order requests are not counted as part of the total number of requests allowed within the specified time window.
REJECT_NEW_PARAMETER	Indicates whether this risk firewall rule instance should reject NewOrder objects if the throttle limit has been reached.
REJECT_NEW_DEFAULT	Default value for whether this risk firewall rule instance should reject NewOrder objects if the throttle limit has been reached. By default, this is true, which means that new order requests are rejected if the throttle limit is reached.
REJECT_AMEND_PARAMETER	Indicates whether this risk firewall rule instance should reject AmendOrder objects if the throttle limit has been reached.
REJECT_AMEND_DEFAULT	Default value for whether this risk firewall rule instance should reject AmendOrder objects if the throttle limit has been reached. By default, this is false, which means that amend order requests are allowed even if the throttle limit has been reached.
REJECT_CANCEL_PARAMETER	Indicates whether this risk firewall rule instance should reject CancelOrder objects if the throttle limit has been reached.
REJECT_CANCEL_DEFAULT	Default value for whether this risk firewall rule instance should reject CancelOrder objects if the throttle limit has been reached. By default, this is false, which means that cancel order requests are allowed even if the throttle limit has been reached.
ORDER_OPS_PER_WIN_OUTPUT	Output parameter that defines the current number of order operations within the current time window.

About the OrderToTradeRatioLimitRiskFirewallRule

The OrderToTradeRatioLimitRiskFirewallRule ensures that the number of order operations (new, amend, cancel) in a specified time window (30 seconds by default) does not exceed the allowed number of operations as defined by the specified ratio between order operations and trades. Order operation types to be checked are configurable when the firewall rule instance is created. By default, all order operations are checked.

The `com.apama.firewall.rules.OrderToTradeRatioLimitRiskFirewallConsts` object defines the constants for the configuration settings for the order to trade ratio limit rule class, as well as their default values. You should use these constants rather than specific values in order to ensure compatibility.

Constant Name of Input Parameter	Description
`RULE_CLASS_NAME`	Rule class name, which is `"OrderToTradeRatioLimitRiskFirewallRule"`.
`OBJECTION_LIMIT_PARAMETER`	Specifies the allowed ratio of new/amend/cancel orders to order executions, which determines the number of order operations (new/amend/cancel) that can be requested before the rule rejects the operation request. You must set a value for this parameter. For example, suppose you set this parameter to `5`. If there are 3 order executions in a 30 second window then the ratio is 15:3. This rule would allow up to 15 new/amend/cancel operations until another order execution is received or the time window expires.
`OBJECTION_LIMIT_DEFAULT`	Default value for the objection limit, which is `0` by default.
`WARNING_LIMIT_PARAMETER`	Specifies the allowed ratio of new/amend/cancel orders to order executions before the risk firewall issues a warning when it allows a new/amend/cancel operation.
`WARNING_LIMIT_DEFAULT`	Default value for the warning limit, which is `0` by default.
`TIME_WINDOW_PARAMETER`	Number of seconds in the rolling time window.
`TIME_WINDOW_DEFAULT`	Default number of seconds in the rolling time window, which is `30.0` by default.
`CHECK_NEW_PARAMETER`	Indicates whether this risk firewall rule instance should apply to `NewOrder` objects.
`CHECK_NEW_DEFAULT`	Default value for whether this risk firewall rule instance should apply to `NewOrder` objects. By default, new order requests are counted as part of the total number of requests allowed within the specified time window.
`CHECK_AMEND_PARAMETER`	Indicates whether this risk firewall rule instance should apply to `AmendOrder` objects.
`CHECK_AMEND_DEFAULT`	Default value for whether this risk firewall rule instance should apply to `AmendOrder` objects. By default, amend order requests are counted as part of the total number of requests allowed within the specified time window.

Constant Name of Input Parameter	Description
CHECK_CANCEL_PARAMETER	Indicates whether this risk firewall rule instance should apply to CancelOrder objects.
CHECK_CANCEL_DEFAULT	Default value for whether this risk firewall rule instance should apply to CancelOrder objects. By default, cancel order requests are counted as part of the total number of requests allowed within the specified time window.

The following table provides information about order to trade ratio limit rule class state information parameters. The risk firewall uses these constants as part of the state returned in the com.apama.firewall.InstanceInfo object when you call the com.apama.firewall.RiskFirewall.getRule*xxx*Info() set of actions.

Constant Name of Output Parameter	Description
TRADE_OPS_PER_WIN_OUTPUT	Number of trade operations in the current time window.
ORDER_OPS_PER_WIN_OUTPUT	Number of order operations in the current time window.

About the OrderValueLimitRiskFirewallRule

The OrderValueLimitRiskFirewallRule checks the order value (price * quantity) of any new order, or amendment to an order, and approves that order if the value is equal to or less than the price provided in the rule instance input parameters.

Note:
This risk firewall rule implementation is not appropriate for market orders. This is because market orders inherently have a price of zero when they are placed and this rule class implementation relies on the order price being available when an order is placed against it. Therefore, this rule does not operate as expected for market orders since it approves any market order.

The com.apama.firewall.rules.OrderValueLimitRiskFirewallConsts object defines the constants for the configuration settings for the order value limit rule class, as well as their default values. You should use these constants rather than specific values in order to ensure compatibility.

Constant Name of Input Parameter	Description
RULE_CLASS_NAME	The name of the rule class, which is "OrderValueLimitRiskFirewallRule".
VALUE_LIMIT_PARAMETER	Objection limit. The maximum value (price * quantity) an order may request before the rule class rejects the order.
VALUE_LIMIT_DEFAULT	Objection limit default value, which is 0 by default.

Constant Name of Input Parameter	Description
VALUE_WARNING_PARAMETER	Warning limit. The maximum value (price * quantity) an order may request before the rule class issues a warning for the order.
VALUE_WARNING_DEFAULT	Warning limit default value.

About the PositionLimitRiskFirewallRule

The PositionLimitRiskFirewallRule class checks the current overall position of a particular order, or set of orders, matching the slices configured, and will reject that order if the position exceeds the limits provided in the input parameters. This risk firewall rule checks both the open and pending positions of matching orders.

Note:
The difference between the client credit limit rule class and the position limit rule class is that the client credit limit rule class acts on the cash position, which is the cumulative value of all orders, where as the position limit rule class acts on the cumulative quantity position.

When adding a client credit limit rule class instance, set either individual minimum and maximum limits, or a symmetric limit. For example, 100 would set a positive limit to +100 and a negative limit to -100. Setting both symmetric and asymmetric configuration parameters for the same limit (warning or objection) results in an error, and the rule class instance is not added.

When you use the position limit rule class (or the client credit limit rule class) there are callbacks that you must add. Specifically, you must add at least one of each type of callback to the order receiver that will handle orders approved by these rule classes:

- OrderReceiver.addAcceptedOrderCallback()
- OrderReceiver.addAcceptedAmendOrderCallback()
- OrderReceiver.addAcceptedCancelOrderCallback()

Each callback must route or enqueue the order event (com.apama.oms.NewOrder, AmendOrder, CancelOrder) to the context that contains the position tracker that the application is using. This is required for the rule class to operate in the expected way.

The position limit rule class uses the CMF position service to calculate the overall position being tracked. This requires the application to have created both an open position tracker and a pending position tracker (see "Using default position trackers" on page 193). You then pass the names of these position tracker instances into the rule class's create() action. This allows the rule class to track the positions of orders from other areas of your application.

The CMF's position service requires the order management (com.apama.oms) events to be sent (either routed or enqueued) to the position tracker instances that you created. This means that your application code must ensure that OMS events are sent to the position trackers being used after the orders have been approved by the risk firewall. Failure to do so may result in the position being calculated incorrectly, and subsequently the rule class may approve orders that would breach the defined limits.

CAUTION:
Where possible, the position trackers used should be in the same context as the risk firewall. This can help avoid a situation in which the position limit rule class approves orders that breach the defined limits. Such a situation might happen because the risk firewall can accept orders synchronously by means of callback actions, but the CMF position service is asynchronous. The position service listens for OMS events, and routes or enqueues position updates. Orders that are sent into the risk firewall should allow time for the correlator input queue to process the position update. The code sample at the end of this topic demonstrates what you must do to correctly implement the position limit rule class. Alternatively, you may implement a custom rule class implementation (see "Implementing custom risk firewall rule classes" on page 168) that can operate safely in a synchronous architecture.

The `com.apama.firewall.rules.PositionLimitRiskFirewallConsts` object defines the constants for the configuration settings for the position limit rule class, as well as their default values. You should use these constants rather than specific values in order to ensure compatibility. The following table provides information about position limit rule class parameters.

Constant Name of Input Parameter	Description
`RULE_CLASS_NAME`	The name of the rule class, which is `"PositionLimitRiskFirewallRule"`.
`POSITION_LIMIT_PARAMETER`	Symmetric objection limits. This is the symmetric quantity position up to which orders may accrue before the rule class rejects subsequent orders.
`POSITION_LIMIT_DEFAULT`	Symmetric objection limits default, which is `0` by default.
`POSITION_WARNING_PARAMETER`	Symmetric warning limits. This is the symmetric quantity position up to which orders may accrue before the rule class issues a warning for subsequent orders.
`POSITION_WARNING_DEFAULT`	Symmetric warning limits default, which is `0` by default.
`MIN_POSITION_LIMIT_PARAMETER`	Minimum objection limit. This is the minimum quantity position up to which orders may accrue before the rule class rejects subsequent orders.
`MIN_POSITION_LIMIT_DEFAULT`	Minimum objection limit default.
`MIN_POSITION_WARNING_PARAMETER`	Minimum warning limit. This is the minimum quantity position up to which orders may accrue before the rule class issues a warning for subsequent orders.
`MIN_POSITION_LIMIT_DEFAULT`	Minimum warning limit default.

Constant Name of Input Parameter	Description
MAX_POSITION_LIMIT_PARAMETER	Maximum objection limit. This is the maximum quantity position up to which orders may accrue before the rule class rejects subsequent orders.
MAX_POSITION_LIMIT_DEFAULT	Maximum objection limit default value, which is 0 by default.
MAX_POSITION_WARNING_PARAMETER	Maximum warning limit. This is the maximum quantity position up to which orders may accrue before the rule class issues a warning for subsequent orders.
MAX_POSITION_WARNING_DEFAULT	Maximum warning limit default.
CHECK_CANCEL_PARAMETER	Indicates whether this risk firewall rule instance should apply to CancelOrder objects.
CHECK_CANCEL_DEFAULT	Default value for whether this risk firewall rule instance should apply to CancelOrder objects. By default, this is true, which means that cancel order requests are counted.
SLICE_POSITION_SERVICEID	The set of service IDs that the position service will use for this instance if it needs to be different from those provided for the instance slice. The format is a stringified sequence<string>. This must be set when running in legacy mode. You may want to check against the position of orders that have been approved by a risk firewall in legacy mode.

The following table provides information about position limit rule class state information parameters. The risk firewall uses these constants as part of the state returned in the com.apama.firewall.InstanceInfo object when you call the com.apama.firewall.RiskFirewall.getRule*xxx*Info() set of actions.

Constant Name of Output Parameter	Description
CURRENT_OPEN_POSITION	Integer value for the current open quantity position.
CURRENT_PENDING_MIN_POSITION	Current minimum pending quantity position.
CURRENT_PENDING_MAX_POSITION	Current maximum pending quantity position.
CURRENT_TOTAL_MIN_POSITION	Current total minimum (open + minimum pending) quantity position for a rule instance.
CURRENT_TOTAL_MAX_POSITION	Current total maximum (open + maximum pending) quantity position for a rule instance.

The following code provides an example of using the position limit rule class.

```
using com.apama.position.tracker.OpenPositionTrackerFactory;
using com.apama.position.tracker.PendingPositionTrackerFactory;

using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;
using com.apama.firewall.rules.PositionLimitRiskFirewallRule;
using com.apama.firewall.rules.PositionLimitRiskFirewallRuleConsts;

using com.apama.utils.Params;
using com.apama.oms.NewOrder;
using com.apama.oms.AmendOrder;
using com.apama.oms.CancelOrder;
using com.apama.oms.UpdateOrder;

monitor PositonLimitRuleExample {
 context mainContext := context.current();
 integer trackersCreated := 0;

 action onload() {

  // Create an instance of the open position tracker in the main context.
  (new OpenPositionTrackerFactory).create(
         mainContext, "MyOpenPositionTracker", cbCreatedTracker );
  // Create an instance of the pending position tracker in the main context.
  (new PendingPositionTrackerFactory).create(
         mainContext, "MyPendingPositionTracker", cbCreatedTracker );
 }

 // This action is called after the position tracker instance
 // has been created.
 action cbCreatedTracker( boolean success, string msg ) {
  log "Position Tracker has been created: "+success.toString()+" : "+msg;

  trackersCreated := trackersCreated +1;

  // After all trackers are created, start the test.
  if( trackersCreated = 2 ) then {
   startTest();
  }
 }

 action startTest() {
  // Create the risk firewall.
  RiskFirewall rfw := (new RiskFirewallFactory).createCb(
         mainContext, "MyFirewall", cbOnRiskFirewallCreated );

  // Create and register the position limit rule class.
  RuleClass ruleClass := (new PositionLimitRiskFirewallRule).create(
     mainContext, "MyOpenPositionTracker", "MyPendingPositionTracker" );
  rfw.registerRuleClass( ruleClass );

  // Add some rule instances.
  Params instanceConfig := new Params;
  instanceConfig.addIntegerParam(
         PositionLimitRiskFirewallRuleConsts.POSITION_LIMIT_PARAMETER, 20 );
  instanceConfig.addIntegerParam(
         PositionLimitRiskFirewallRuleConsts.POSITION_WARNING_PARAMETER, 15 );

  integer instanceId := rfw.addRuleInstance(
         ruleClass.getRuleClassName(), instanceConfig );
```

```
 // Add handlers for all the approved OMS events from the risk firewall.
 integer refId1 := rfw.getOrderReceiver().
        addAcceptedOrderCallback(  cbOnApprovedNewOrders );
 integer refId2 := rfw.getOrderReceiver().
        addAcceptedAmendCallback(  cbOnApprovedAmendOrders );
 integer refId3 := rfw.getOrderReceiver().
        addAcceptedCancelCallback( cbOnApprovedCancelOrders );

 // Add a callback for all OrderUpdate events from the risk firewall.
 integer refId4 := rfw.getOrderSender().
        addOrderUpdateCallback( cbOnOrderUpdates );

 // The unlock will take effect immediately after the create() action.
 rfw.unlock();
}

action cbOnRiskFirewallCreated( RiskFirewall rfw ) {

 // Send a new order to the risk firewall.
 rfw.getOrderSender().sendOrder( NewOrder("orderId_1","APMA.L",
                                     9.0, "BUY", "LIMIT", 10,
                                     "TargetService","","",
                                     "TargetMarket","", "TraderA",
                                     new dictionary<string,string> ) );
}

// Set up a callback handler for NewOrder events that the
// risk firewall has approved.
action cbOnApprovedNewOrders( NewOrder order ) {
 // Send the NewOrder event to the context that the position trackers
 // were created in.
 route order;

 // The application can now perform any other actions
 // on this NewOrder event.
     ...
}

// Set up a callback handler for AmendOrder events that the
// risk firewall has approved.
action cbOnApprovedAmendOrders( AmendOrder amend ) {
 // Send the AmendOrder event to the context that the position trackers
 // were created in.
 route amend;

 // The application can now perform any other actions
 // on this AmendOrder event.
     ...
}

// Set up a callback handler for CancelOrder events that the
// risk firewall has approved.
action cbOnApprovedCancelOrders( CancelOrder cancel ) {
 // Send the CancelOrder event to the context that the position trackers
 // were created in.
 route cancel;

 // The application can now perform any other actions
 // on this CancelOrder event.
```

```
      ...
 }

 // This action will be called for any updates to the
 // orders the risk firewall is handling.
 action cbOnOrderUpdates( OrderUpdate update ) {
  // Send the OrderUpdate event to the context that
  // the position trackers were created in.
  route update;

  // The application can now perform any other actions
  // on this OrderUpdate event.
      ...
 }
}
```

About the ReservationEnforcerRiskFirewallRule

The `ReservationEnforcerRiskFirewallRule` class ensures that orders placed against reservation contracts do not exceed the minimum/maximum quantities set for reserved positions. This rule class implementation tracks both the reservation orders and the orders that are submitted against reservation orders.

Reservation requests and changes to these reservations are implemented as order operations (that is `com.apama.oms.New/Amend/CancelOrder` requests). Any changes to the size of the reservation are sent as order updates.

This rule class implementation requires access to the risk firewall that it was registered with so that it can use a risk firewall order receiver to

- Monitor reservation orders that are approved by the rest of the risk firewall rule classes.
- Send order updates to the reservation order state back through the risk firewall.

When an order is placed against a reservation the order quantity is subtracted from the reserved position and added to the pending position. If the order subsequently fills then the filled quantity moves from the pending position to the open position. However if the order is canceled or amended downwards then the quantity is moved from the pending position back to the reserved position. For example, suppose a trader has unfilled buy orders with a type of `RESERVATION` in the market with a total quantity of `1000`. The reserved `maxQtyPosition` is `1000`. This is the maximum that the trader's position can change if all reserved buy orders were filled. Conversely, if a trader has unfilled sell orders with a type of `RESERVATION` in the market with a total quantity of `1000` then there the reserved `minQtyPosition` is `-1000`. The `ReservationEnforcerRiskFirewallRule` class ensures that the reserved `maxQtyPosition` and `minQtyPosition` are observed.

Adding rule instances to rule classes

After you register a rule class with a risk firewall, you add one or more instances of that rule class to that risk firewall. Note that for custom rule classes, the addition of at least one rule class instance is not always a requirement. See "Implementing custom risk firewall rule classes" on page 168.

A rule class instance specifies a set of configuration parameter values to use when evaluating an order to determine if it complies with the rule class. For example, for the `OrderPriceLimitRiskFirewallRule`, you might add an instance that specifies that

- `SLICE_SYMBOL` = `"SOW"`
- `PRICE_LIMIT_PARAMETER` = 20.0
- `PRICE_WARNING_PARAMETER` = 15.0

Orders for `SOW` with prices set at 15.0 or less would be approved. Orders with prices that are more than 15.0 but not more than 20.0 would be approved with a warning while orders with prices that are more than 20.0 would be rejected.

Before you can add a rule class instance to a risk firewall, you must create a `com.apama.utils.Params` object that contains the parameter settings for the rule class instance. When you add the rule class instance to the risk firewall, you specify this `Params` object. For example, the steps for adding an instance of the default `OrderPriceLimitRiskFirewallRule` are as follows:

1. Create a `com.apama.firewall.rules.OrderPriceLimitRiskFirewallConsts` object.
2. Create a `com.apama.utils.Params` object.
3. Add the rule class instance configuration parameters to the `Params` object.
4. Specify the `Params` object when you add the rule class instance to the risk firewall.

To add a rule class instance to a risk firewall, execute one of the following actions:

- `com.apama.firewall.RiskFirewall.addRuleInstance()` — adds a rule class instance to a risk firewall.
- `com.apama.firewall.RiskFirewall.addRuleInstanceCb()` — adds a rule class instance to a risk firewall and then executes the specified callback if addition of the rule class instance is successful.

These actions are asynchronous and their availability is pending until the risk firewall is created and the relevant rule class is registered. It is an error if the time period specified by the `CONFIG_TIMEOUT_DURATION` parameter elapses before the risk firewall is created or before the relevant rule class is registered. These actions are not available when you are connected to a remote risk firewall instance. Both actions take these two parameters:

- *`ruleClassName`* — The name of a rule class. For each default rule class, its name is defined in its associated *`xxx`*`Consts` event, which defines the configuration parameters for that rule class. Alternatively, you can call `getRuleClassName()` on the `RuleClass` instance.
- *`config`* — A `com.apama.utils.Params` object that contains configuration parameter settings for this rule class instance. The risk firewall uses these parameter settings to evaluate an incoming order. For each default rule class, there is an associated event that specifies the configuration parameters for that rule class. The following events are provided in the `com.apama.firewall.rules` package:

```
ClientCreditLimitRiskFirewallRule
ClientCreditLimitRiskFirewallRuleConsts
```

```
OrderOperationRatioRiskFirewallRule
OrderOperationRatioRiskFirewallRuleConsts

OrderPriceLimitRiskFirewallRule
OrderPriceLimitRiskFirewallRuleConsts

OrderQuantityLimitRiskFirewallRule
OrderQuantityLimitRiskFirewallRuleConsts

OrderThrottleLimitRiskFirewallRule
OrderThrottleLimitRiskFirewallRuleConsts

OrderToTradeRatioRiskFirewallRule
OrderToTradeRatioRiskFirewallRuleConsts

OrderValueLimitRiskFirewallRule
OrderValueLimitRiskFirewallRuleConsts

PositionLimitRiskFirewallRule
PositionLimitRiskFirewallRuleConsts

ReservationEnforcerRiskFirewallRule
ReservationEnforcerRiskFirewallRuleConsts
```

In addition, the `addRuleInstanceCb()` action takes a callback as its third parameter.

Both actions return a unique, integer `instanceId`, which you can use later if you need to modify or remove this rule class instance.

The following code provides an example of adding a rule class instance:

```
// Add a rule class instance.
com.apama.firewall.rules.OrderPriceLimitRiskFirewallRuleConsts priceLimitConsts
   := new com.apama.firewall.rules.OrderPriceLimitRiskFirewallRuleConsts;

// Set configuration parameters to issue a
// warning when an order price is more than 15.0 and to
// reject an order with a price more than 20.0.
coma.apama.utils.Params ruleParams := new com.apama.utils.Params;
ruleParams.addFloatParam( priceLimitConsts.PRICE_LIMIT_PARAMETER,   20.0 );
ruleParams.addFloatParam( priceLimitConsts.PRICE_WARNING_PARAMETER, 15.0 );
// Set a symbol parameter.
sequence<string> symbol := ["SOW"];
ruleParams.addParam(com.apama.firewall.Consts.SLICE_SYMBOL, symbol);

// Add the rule class instance to a risk firewall represented by rfw.
integer instanceId := rfw.addRuleInstance(
   priceLimitConsts.RULE_CLASS_NAME, ruleParams );
```

Examples of slice filters for rule class instances

By default, a slice parameter is a wildcard that matches any value. You can specify a single value or a sequence of values for any number of slice parameters for each rule class instance.

Suppose you set one slice parameter for a rule class instance. The risk firewall evaluates each order that contains at least one value that you specified in that slice parameter. In other words, an order with any one of the values specified in the sequence causes the order to be evaluated by that rule

class instance. For example, for a particular rule class instance, you set the SLICE_SYMBOL parameter to a sequence that contains "A", "B" and "C". That rule class instance evaluates an order if its symbol is A or B or C.

Now suppose you set more that one slice parameter for a rule class instance. The risk firewall evaluates an order only when there is a match for each slice parameter. That is, the risk firewall applies AND logic when you specify more than one slice parameter. For example, suppose you set the following slice parameters:

SLICE_SYMBOL is ("A", "B", "C")

SLICE_SERVICEID is ("1", "2", "3")

SLICE_TRADERID is ("TM", "TN", "TO")

For this rule class instance to evaluate an order, the order's symbol must be A or B or C, and the order's service Id must be 1 or 2 or 3, and the order's trader Id must be TM or TN or TO.

If you do not specify a particular slice parameter for a rule class instance it has the effect of a wildcard. For example, suppose you do not specify the SLICE_TRADERID parameter but for the SLICE_SYMBOL parameter you specify a sequence that contains "EUR/USD". This causes the rule class to match against orders for "EUR/USD" for any trader. In addition, because the client credit limit and position limit rule classes calculate cumulative values, instances of those rule classes will calculate a cumulative value/position for all traders. Note that specifying an empty sequence has the same effect as not defining the particular parameter.

Suppose you set the SLICE_SYMBOL parameter to ("A", "B") for an instance of the position limit default rule class. This rule class instance tracks the combined position for both A and B. If you want to track A orders against the A position, and you also want to track B orders against the B position, then you must add two rule class instances — one rule class instance would have A as the value of the SLICE_SYMBOL parameter and the other rule class instance would have B as the value of the SLICE_SYMBOL parameter.

The following table provides examples of slice filters you might specify along with their results.

Slice Filter	Matches Orders With
SLICE_SYMBOL="SOW"	Symbol value of SOW regardless of which adapter the order is going to. That is, the values of the service Id and market Id do not matter. For a position limit rule class instance, the open and pending positions would be the total of all orders for SOW on all adapters.
SLICE_SYMBOL="SOW" SLICE_SERVICEID="MyService"	Symbol value of SOW that are going to the adapter identified by the MyService Id. For example, for an the order price limit rule instance, this filter ensures that no trader sends an order to the MyService adapter with a price that is above a specified amount for the specified symbol.

Slice Filter	Matches Orders With
`SLICE_SERVICEID="MyService"`	Service Id of `MyService`. For a position limit rule class instance, the open and pending positions would be the total of all orders that are going to the adapter identified by the `MyService` Id, regardless of the symbol or any other information in the order. Suppose you want to limit the maximum position for each symbol that is traded on a particular service. This requires a different rule class instance for each symbol. The slice filter for each rule class instance would specify a particular, different symbol and the service Id.

In the case of the client credit limit rule class, you might be trading foreign exchange currencies or assets over different traded currencies, where calculating a cumulative cash position does not make sense if the traded currencies are different. For example, you do not want to add USD to Yen totals without first converting them to a normalized currency.

Most of the default rule classes are per-order checks. Consequently, defining groups is usually what you want to do.

Suppose you want to check that the currency is one of `GBP`, `EUR` or `USD`, and you also want to check that another parameter exists (for example, a client reference). This other parameter does not need to have a specific value, that is, you want to specify it as a wildcard. The following code accomplishes this:

```
dictionary< string, sequence<string> > sliceEPs
            := new dictionary< string, sequence<string> >;
sliceEPs.getOrAddDefault( "Currency" ).append( "GBP" );
sliceEPs.getOrAddDefault( "Currency" ).append( "EUR" );
sliceEPs.getOrAddDefault( "Currency" ).append( "USD" );
sliceEPs.getOrAddDefault( "ClientRef" ).append( "" );   // Wildcard client ref

com.apama.utils.Params params := new com.apama.utils.Params;
params.addParam( com.apama.firewall.Consts.SLICE_EXTRAPARAM,
      sliceEPs.toString() );
rfwIface.addRuleInstance( "MyRuleClass", params );
```

Sample code for registering rule class and adding rule class instances

The following sample code registers the default `OrderPriceLimitRiskFirewallRule` class and does not specify a callback. This code then shows how to add rule class instances for the registered rule class.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;
using com.apama.firewall.rules.OrderPriceLimitRiskFirewallRule;
using com.apama.firewall.rules.OrderPriceLimitRiskFirewallRuleConsts;
using com.apama.utils.Params;
using com.apama.oms.NewOrder;
```

```
monitor RiskFirewallExample10 {

  context mainContext := context.current();

  action onload() {
    // Create a risk firewall in the main context.
    RiskFirewall rfw := (new RiskFirewallFactory).
       create( mainContext, "MyFirewall" );

    // Register one of the default rule classes.
    rfw.registerRuleClass( (new OrderPriceLimitRiskFirewallRule).create() );

    // Add a rule class instance.
    OrderPriceLimitRiskFirewallRuleConsts priceLimitConsts :=
       new OrderPriceLimitRiskFirewallRuleConsts;

    // Set configuration parameters to issue a
    // warning when an order price is more than 15.0 and to
    // reject an order with a price more than 20.0.
    Params ruleParams := new Params;
    ruleParams.addFloatParam( priceLimitConsts.PRICE_LIMIT_PARAMETER,   20.0 );
    ruleParams.addFloatParam( priceLimitConsts.PRICE_WARNING_PARAMETER, 15.0 );

    // Add the rule class instance to MyFirewall.
    integer instanceId := rfw.addRuleInstance(
       priceLimitConsts.RULE_CLASS_NAME, ruleParams );

    // Unlock the risk firewall and send orders in.
       ...

    // This order will be approved:
    rfw.getOrderSender().sendOrder( NewOrder( "orderId_1","APMA",
       10.0, "BUY", "LIMIT", 10,
       "TargetService","","",
       "TargetMarket","", "TraderA",
       new dictionary<string,string> ) );

    // This order will issue a warning as it breaches the price
    // warning level that has been set, but is under the price limit
    // level.
    rfw.getOrderSender().sendOrder( NewOrder( "orderId_2","APMA",
       "TargetService","","",
       "TargetMarket","", "TraderA",
       new dictionary<string,string> ) );

    // This order will be rejected as it breaches the price limit
    // that has been set.
    rfw.getOrderSender().sendOrder( NewOrder( "orderId_3","APMA",
       21.0, "BUY", "LIMIT", 10,
       "TargetService","","",
       "TargetMarket","", "TraderA",
       new dictionary<string,string> ) );
 }

 ...
   }
}
```

Sample code for registering rule class and specifying callback

The sample code below shows registration of a rule class along with specification of a callback.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFactory;
using com.apama.firewall.rules.OrderPriceLimitRiskFirewallRule;

monitor RiskFirewallExample11 {
   action onload() {
      // Create a new risk firewall instance.
      RiskFirewall rfw := (new RiskFirewallFactory).
         create( context.current(), "MyFirewall" );

      // Register a risk firewall rule class with the
      // new risk firewall instance and specify a callback.
      rfw.registerRuleClassCb( (new OrderPriceLimitRiskFirewallRule).
         create(), cbOnRuleClassRegistered );
   }

   // This action is called upon successful registration.
   action cbOnRuleClassRegistered(
      RiskFirewall rfw, string ruleClassName ) {
         log "Successfully registered RuleClass: "+ruleClassName;
    }
}
```

Setting rule class priority

Rule class priority is the order in which the risk firewall queries the rule class instances that match an order. It applies to a registered rule class and so instances of the same class that have been added to a risk firewall always have the same priority. The default behavior is that all registered rule classes have the same priority (a value of `50`). If there are multiple matching rule class instances for an order then the evaluation order is unpredictable.

The `RiskFirewall.setRuleClassPriority()` action lets you set the order in which the risk firewall queries rule classes to evaluate orders. For example, this is useful if you want to perform a basic set of checks on an order before trying to evaluate a much more computationally expensive operation, such as breaching a specific currency limit. The `setRuleClassPriority()` action takes two arguments:

- *`ruleClassName`* is the name of the registered rule class you are assigning the priority to.
- *`ruleClassPriority`* is an integer that indicates the priority level. A value of zero is the highest priority and `MAX_INT` is the lowest priority. For convenience, three integer constants are provided (in `com.apama.firewall.Consts`) that make it easy to set high, medium, and low priority ranges:
  - `RULE_CLASS_PRIORITY_HIGH` has a value of `0`.
  - `RULE_CLASS_PRIORITY_MEDIUM` has a value of `50`.
  - `RULE_CLASS_PRIORITY_LOW` has a value of `100`.

For example:

```
using com.apama.firewall;
// This is an important rule class so check it first.
rfwIface.setRuleClassPriority(“MyImportantRuleClass”,
   Consts.RULE_CLASS_PRIORITY_HIGH );
// This rule class a lot of processing so check it last.
rfwIface.setRuleClassPriority( “MyComplexRuleClass”,
   RULE_CLASS_PRIORITY_LOW );
// This rule class is of medium-high importance.
rfwIface.setRuleClassPriority(“MyMiddleRuleClass”, 25 );
```

Obtaining information about registered rule classes and rule instances

A risk firewall provides actions to obtain information for registered rule classes and added rule class instances. You can call these actions on a local risk firewall or a remotely connected risk firewall. For each action, you specify (at least) a callback that returns the requested information.

You can use these interfaces to get the state of a rule class or rule class instance. For example, if you register the position limit rule class provided with the CMF, you can query the current positions accrued by a position limit rule class instance. This can be useful to monitor the current state of the rule class, and perhaps publish the information to a user interface.

You can obtain information for

- A registered rule class
- All registered rule classes
- An added rule class instance
- All added instances for a particular rule class
- All added instances for all registered rule class

To obtain information about registered rule classes:

- `RiskFirewall.getRuleClassInfo()`

  Specify the name of the rule class you want information for and specify a callback.

  The callback returns a `com.apama.firewall.RuleClassInfo` object, which contains the following information for the specified rule class:

  - Name
  - Description
  - Evaluation priority setting
  - Configuration schema associated with this rule class. The content of the configuration schema depends on the rule class implementation.
  - Any state information, if this rule class publishes state information.

- `RiskFirewall.getAllRuleClassInfo()`

  Specify a callback.

The callback returns the information listed above for each rule class that is registered with this risk firewall. The information is in a dictionary of `RuleClassInfo` objects, one for each registered rule class.

To obtain information about added rule class instances:

- `RiskFirewall.getRuleInstanceInfo()`

  Specify the instance Id of the rule class instance you want information for and specify a callback. The rule class instance Id is the unique, integer identifier that was returned when the rule class instance was added to the risk firewall.

  The callback returns a `com.apama.firewall.RuleInstanceInfo` object, which contains the following information for the specified rule class instance:

  - Rule class name that this object is an instance of
  - Rule class instance Id
  - A `com.apama.firewall.InstanceInfo` object, which contains two `com.apama.utils.Params` objects:
    - Configuration schema that was used to create this rule class instance. The content of the configuration schema depends on the rule class implementation.
    - Any state information for this rule class instance, if its rule class publishes state information.

- `RiskFirewall.getAllRuleInstanceInfo()`

  Specify the name of a rule class to obtain information for each instance that has been added. Also specify a callback.

  The callback returns a `com.apama.firewall.AllRuleInstanceInfo` object, which contains the following information for the specified rule class:

  - Name of the rule class this information is for.
  - A dictionary of `com.apama.firewall.InstanceInfo` objects. There is an `InstanceInfo` object for each rule class instance. Each `InstanceInfo` object contains two `com.apama.utils.Params` objects:
    - Configuration schema that was used to create this rule class instance. The content of the configuration schema depends on the rule class implementation.
    - Any state information for this rule class instance, if this rule class publishes state information.

- `RiskFirewall.getAllRuleClassInstanceInfo()`

  Specify a callback.

  The callback returns a `com.apama.firewall.AllRuleClassInstanceInfo` object, which is a dictionary:

- The key field values are the names of the registered rule classes.
- The data field values are also dictionaries. There is a dictionary for each registered rule class. In this dictionary, the key field values are the instance Ids of the added rule class instances. The data field values are `com.apama.firewall.InstanceInfo` objects, which contain the schema and state for that rule class instance.

Modifying rule instances

You can modify a rule class instance that you previously added to a risk firewall. A risk firewall instance provides two actions for modifying a rule class instance. You can call these actions on a local risk firewall but not on a connected remote risk firewall.

- `RiskFirewall.modifyRuleInstance()` takes two arguments:
  - `instanceId` identifies the rule class instance you want to modify. Specify the integer identifier that was returned when you added the rule class instance to the risk firewall.
  - `config` is a `com.apama.utils.Params` object that specifies all the configuration parameter(s) for the rule class instance you are modifying. This includes the parameters you are changing and the parameters that remain the same.
- `RiskFirewall.modifyRuleInstanceCb()` takes the same two arguments and also a third argument that specifies a callback. The risk firewall executes this callback after successfully modifying the rule class instance. The callback takes three arguments: the risk firewall object, the rule class name, and the rule class instance identifier.

  Call this action when your application requires notification that the rule class instance has been changed.

If there is an error while trying to modify the rule class instance, the risk firewall calls any error callbacks you added or it calls the default error callback if you did not add any error callbacks.

The risk firewall uses the modified rule class instance for all subsequently received orders. You modify a rule class instance in the same way that you add a rule class instance. See "Sample code for registering rule class and adding rule class instances" on page 150.

Removing rule instances

You can remove a rule class instance that you previously added to a risk firewall. A risk firewall instance provides two actions for removing a rule class instance. You can call these actions on a local risk firewall but not on a connected remote risk firewall.

- `RiskFirewall.removeRuleInstance()` takes one argument, `instanceId`, which identifies the rule class instance you want to remove. Specify the integer identifier that was returned when you added the rule class instance to the risk firewall.
- `RiskFirewall.removeRuleInstanceCb()` takes two arguments:
  - `instanceId` identifies the rule class instance you want to remove. Specify the integer identifier that was returned when you added the rule class instance to the risk firewall.

- `cbCompleted` is a callback that the risk firewall executes after successfully removing the rule class instance. The callback takes three arguments: the risk firewall object, the rule class name, and the rule class instance identifier.

  Call this action when your application requires notification that the rule class instance has been removed.

If there is an error while trying to remove the rule class instance, the risk firewall calls any error callbacks you added or it calls the default error callback if you did not add any error callbacks.

Persisting rule class instances

A risk firewall provides persistence for rule class instances that have been added to it. You do not need to do anything to persist rule class instances; this is the default behavior.

Note:
Risk firewall persistence is completely separate from correlator persistence. That is, rule class instances added to a risk firewall are persistent regardless of whether persistence has been enabled for the correlator.

A risk firewall itself is not persistent, nor are registered rule classes. Upon correlator restart, you must always re-create the risk firewall and re-register rule classes. After that, the risk firewall restores its rule class instances.

The following information is persistent for each rule class instance that has been added to a risk firewall:

- The rule class instance identifier, which was returned when the rule class instance was added.
- The name of the registered rule class that the instance was created from.
- The configuration parameters for the rule class instance.

If a registered rule class requires additional state information to be persistent, for example, the current position accrued, your application must restore that information before re-registering rule classes. You can use the CMF's "Configuration Service" on page 237 to help you persist rule class data.

A risk firewall also persists and restores the set of objections/warnings that it has issued for evaluated orders. The persistent information includes:

- The query order request type (New/Amend/Cancel Order) that this objection/warning was issued against
- The query request, which contains information about the original order (orderId, symbol name, service/market/exchange Ids)
- The query response, which is the result of the query (the objection/warning, a string that contains the result message)

If an application must be restarted and recovered, it is up to the application to ensure that

- Orders are not left in an unknown state. In particular, the following are not persistent:

  - Queued orders
  - In-process orders, (submitted but not fully filled and not cancelled)
  - Pended orders for a risk firewall that is configured for soft rejection
- Adapters cancel outstanding orders.
- Incoming client orders are cancelled prior to termination, or on recovery if the application was terminated unexpectedly.

If you want to change default behavior and not persist rule class instances, you can change the setting of the `CONFIG_ENABLE_PERSISTENCE` configuration parameter. This constant value (defined in `com.apama.firewall.Consts` defines the name of the configuration parameter that indicates whether persistence is enabled for the risk firewall. To change the setting of this parameter, use the `CONFIG_ENABLE_PERSISTENCE` key and set it to the `boolean` value you need.

Unlocking and locking risk firewalls

A risk firewall can be locked and unlocked to ensure that it does not process orders before an application is ready. The locking mechanism can also be helpful if it is necessary to stop risk firewall operation for some reason, for example:

- There have been an excessive number of rejections.
- An excessive number of orders have been placed.
- There is a rogue trading algorithm.

The default behavior is that a risk firewall is created in a locked state. This is because there are a number of things that must be done before the risk firewall is ready to accept orders:

- Risk firewall rule classes must be registered.
- Risk firewall rule class instances must be added.
- A risk firewall order receiver must be created. The order receiver handles orders that have been approved by the risk firewall.

After an application determines that the tasks listed above have been done, the application can call one of the following actions to unlock the risk firewall. These actions are available on a local risk firewall but not on a remotely connected risk firewall.

- `RiskFirewall.unlock()` does not take any arguments nor does it return anything.
- `RiskFirewall.unlockCb()` takes a single argument, which is a callback. Upon successfully unlocking the risk firewall, this callback is executed. Call this action when your application requires notification that the risk firewall has been unlocked.

If there are any pending rule class registrations or rule class instance additions, they are completed before the risk firewall is unlocked. If the risk firewall was processing orders when it was locked and there are queued orders then they are processed in the order in which they were received.

To lock a risk firewall, call the `RiskFirewall.lock()` action. This circuit-breaker style lock prevents all orders from being allowed through this risk firewall. For orders that are in the risk firewall, what happens depends on the setting of the risk firewall's rejection mode. With hard rejection mode, in-process orders are rejected. With soft rejection mode, in-process orders are queued until the risk firewall is unlocked or until manual intervention by a dealer. The `lock()` action is available on a local risk firewall and not on a remotely connected risk firewall.

To change the default lock state upon risk firewall creation, use the `CONFIG_LOCKED_ON_CREATE` risk firewall configuration parameter. This constant value (defined in `com.apama.firewall.Consts`) defines the name of the configuration parameter that indicates whether a risk firewall is created in a locked state. The default is that a risk firewall is created in a locked state. That is, an application must explicitly unlock a risk firewall. To change the setting of this parameter, use the `CONFIG_LOCKED_ON_CREATE` key and set it to the `boolean` value you need.

To find out whether a risk firewall is in a locked state, call the `RiskFirewall.isLocked()` action. This action returns `true` if the risk firewall is locked.

Sending orders into a risk firewall

You use a `com.apama.firewall.OrderSender` object to send orders into a risk firewall. To obtain an order sender object, call the `RiskFirewall.getOrderSender()` action on the risk firewall you want to send orders to. This can be a local risk firewall or a remotely connected risk firewall. The `getOrderSender()` action returns an `OrderSender` object that you can use with only the risk firewall that supplied it. An order sender object provides actions for

- Sending new orders into the risk firewall
- Amending and cancelling orders that the risk firewall is processing
- Adding one or more callbacks, which the risk firewall executes whenever it receives an update to an order that it is handling

The risk firewall is handling a particular order from the time an order sender submits the order to the risk firewall until the timeout duration (set by the `CONFIG_BUSTED_FILL_DURATION` configuration parameter) has elapsed after the order has been completed or rejected. Thus, the period that the risk firewall is handling an order can include, for example, evaluation of the order, the order being sent to an adapter, the filling of the order, an order update being returned to the risk firewall's order receiver, the order update going into the risk firewall, and the order update arriving at the originating risk firewall order sender.

There are two actions for sending a new order into the risk firewall that provided the `OrderSender` object. You can call these actions on a local risk firewall or on a remotely connected risk firewall. The risk firewall must be unlocked.

- `OrderSender.sendOrder()` takes one argument, a `com.apama.oms.NewOrder` object, which is the new order you want the risk firewall to evaluate. This action does not return anything.
- `OrderSender.sendOrderCb()` takes two arguments:
  - A `com.apama.oms.NewOrder` object, which is the new order you want the risk firewall to evaluate.

- A *cbCompleted* callback that the risk firewall executes whenever it receives an update for this order. The callback itself takes one argument, which is a `com.apama.oms.OrderUpdate` object. This action does not return anything.

Typically, you need only one order sender. However, you can obtain multiple `OrderSender` instances from a risk firewall. For example, you might have multiple trading strategies and use a different order sender for each one. All `OrderSender` objects receive all `OrderUpdate` events from the risk firewall.

If you spawn to a new context you cannot use the order sender in the new context.

See "Sample code for using OrderSender" on page 159.

Sample code for using OrderSender

The following sample code show the tasks that must be accomplished before an application can use an order sender to submit orders to a risk firewall. This example uses an order sender to submit a hardcoded new order to the risk firewall. In a real application, the `sendOrder()` action would likely be called when a user-interface event occurs, such as a user clicking on a **Buy** button.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;
using com.apama.oms.NewOrder;

monitor RiskFirewallExample12 {

   action onload() {
      // Use a risk firewall factory to create a new
      // risk firewall instance named MyFirewall. This example
      // uses the default configuration parameter settings.
      RiskFirewall rfw :=
         (new RiskFirewallFactory).create(context.current(), "MyFirewall");

      // Register a rule class and add an instance of that class.
      ...

      // Create a receiver for the orders coming out of the
      // risk firewall and add a callback to be executed
      // upon approval of new orders.
      integer refId := rfw.getOrderReceiver().
         addAcceptedOrderCallback( cbOnApprovedNewOrders );

      // Unlock the risk firewall.
      rfw.unlockCb( cbOnFirewallUnlocked );
    }

   // This action is called after the risk firewall is unlocked.
   // It indicates that the risk firewall is ready to receive orders.
   // Typically, orders are sent at a later time and a callback that
   // indicates when orders can be sent is not needed.
   action cbOnFirewallUnlocked ( RiskFirewall rfw ) {

      // Send a new order to the risk firewall.
      rfw.getOrderSender().sendOrder(
         NewOrder("orderId_1","APMA.L", 1.0, "BUY", "MARKET", 10,
         "TargetService", "","","TargetMarket","", "TraderA",
         new dictionary<string,string> ) );
```

```
    }

    // Set up a callback handler for new orders that the
    // risk firewall has approved.
    action cbOnApprovedNewOrders( NewOrder order ) {
       log "Firewall has approved the NewOrder: "+order.toString();
    }
}
```

Setting order update callbacks

You might want to do something when there is an update to an order that the risk firewall is currently handling. To do this, you can add an order update callback to an order sender object. The risk firewall executes any added order update callbacks as follows:

- Each time it receives a `com.apama.oms.OrderUpdate` object for an order that it is handling. The update can come from only an `OrderReceiver`.
- Each time it rejects an order. Typically, this is due to an order failing the evaluation of the registered rule classes. However, the risk firewall can also reject new/amend/cancel orders for other reasons, such as the risk firewall being locked, or if the amend/cancel order is for an unknown order Id.

To set an order update callback, execute `com.apama.firewall.OrderSender.addOrderUpdateCallback()`. This action takes one argument, which is the callback to be executed. The callback itself takes one argument, which is a `com.apama.oms.OrderUpdate` object. The `addOrderUpdateCallback()` action registers a callback with the risk firewall instance it is executed on. The registered callback is executed whenever the risk firewall receives or generates an `OrderUpdate` for an order that it is currently handling. You can add more than one order update callback.

The `addOrderUpdateCallback()` action returns a unique integer reference Id that you can use to remove the callback at a later date if required.

To remove a previously-added order update callback, execute `OrderSender.removeOrderUpdateCallback()` and specify the integer reference Id that was returned when you added the callback. To remove all previously-added order update callbacks, execute the `OrderSender.clearOrderUpdateCallbacks()` action.

Modifying orders

After you submit an order to a risk firewall, you can change the order while the risk firewall is processing it by calling the `OrderSender.sendAmend()` action. This action sends an amendment to an existing order to the risk firewall associated with this `OrderSender` object.

The `OrderSender.sendAmend()` action takes one argument, which is a `com.apama.oms.AmendOrder` object. The order Id in this object must match the order Id of an order that the risk firewall is handling. If there is a match, the risk firewall queries all registered rule classes and added rule class instances to determine whether to approve the amended order. If the amended order is approved then it is forwarded to any order receivers that are associated with this risk firewall.

The risk firewall rejects the amended order if its order Id does not match the order Id of an order that the risk firewall is handling. The risk firewall might also reject the order based on the query results. If the risk firewall rejects the amended order, the risk firewall generates an `OrderUpdate` event that contains information about the rejection. If you added any order update callbacks to the originating `OrderSender`, the risk firewall executes them, which sends the `OrderUpdate` to the originating `OrderSender`.

The risk firewall then executes any query response callbacks that you added to the risk firewall.

Cancelling orders

After you submit an order to a risk firewall, you can cancel the order while the risk firewall is processing it by calling the `OrderSender.sendCancel()` action. This action sends a cancellation request for an existing order to the risk firewall associated with this `OrderSender` object.

The `OrderSender.sendCancel()` action takes one argument, which is a `com.apama.oms.CancelOrder` object. The order Id in this object must match the order Id of an order that the risk firewall is handling. If there is a match, the risk firewall determines whether to allow the cancellation. The criteria used to determine whether to allow the cancellation is up to the rule class implementation. The default rule classes always approve cancellation requests. If the cancellation request is approved then it is forwarded to any order receivers that are associated with this risk firewall.

The risk firewall rejects a cancellation request if its order Id does not match the order Id of an order that the risk firewall is handling. The risk firewall might also reject the cancellation request for some other reason that is application-specific. If the risk firewall rejects the cancellation request, the risk firewall generates an `OrderUpdate` event that contains information about the rejection. If you added any order update callbacks to the originating `OrderSender`, the risk firewall executes them, which sends the `OrderUpdate` to the originating `OrderSender`.

The risk firewall then executes any query response callbacks that you added to the risk firewall.

Measuring order handling performance in the risk firewall

To help you measure performance, the risk firewall can add timestamps to order events. If added, the timestamps are in the "`__timestamps`" dictionary, which is a field in the extra parameters dictionary of each order event:

- `com.apama.oms.NewOrder`
- `com.apama.oms.AmendOrder`
- `com.apama.oms.CancelOrder`
- `com.apama.oms.OrderUpdate`

An inbound timestamp indicates when the order is placed against the risk firewall and an outbound timestamp indicates when the order is about to go to the order receiver callbacks. Each timestamp is based on the current UTC time.

Adding timestamps adds latency to order handling in the risk firewall. Consequently, the default behavior is that the risk firewall does not add timestamps. If you want the risk firewall to add

timestamps, change the setting of one or more of the following risk firewall configuration parameters from `false` to `true`.

- `CONFIG_LOG_INBOUND_LATENCY_TIMESTAMPS` indicates whether timestamps are added for inbound new orders and inbound order updates.
- `CONFIG_LOG_OUTBOUND_LATENCY_TIMESTAMPS` indicates whether timestamps are added for outbound new orders and outbound order updates.
- `CONFIG_LOG_AMEND_CANCEL_LATENCY_TIMESTAMPS` indicates whether timestamps are added for amend orders and cancel orders, both inbound and outbound.

For example, if you enable timestamp identifiers for inbound and outbound new orders and update orders, you can measure performance by determining the time that the order spent in the risk firewall:

OutboundTimestamp − *InboundTimestamp* = *TimeOrderSpentInRiskFirewall*

When the risk firewall adds timestamps, the addition includes an identifier for each timestamp. The timestamp identifier is the key in the "`__timestamps`" dictionary contained in the extra parameters dictionary. A timestamp identifier indicates the order type and whether the order was inbound or outbound. To determine the timestamp identifier for an event, the risk firewall uses the value of the `CONFIG_LATENCY_TIMESTAMPID_BASE` configuration parameter (the default is `7000`) with the offsets defined by the following constants:

- `CONFIG_INBOUND_NEWORDER_TIMESTAMPID_OFFSET` is `0`
- `CONFIG_INBOUND_AMENDORDER_TIMESTAMPID_OFFSET` is `2`
- `CONFIG_INBOUND_CANCELORDER_TIMESTAMPID_OFFSET` is `4`
- `CONFIG_INBOUND_ORDERUPDATE_TIMESTAMPID_OFFSET` is `6`
- `CONFIG_OUTBOUND_NEWORDER_TIMESTAMPID_OFFSET` is `1`
- `CONFIG_OUTBOUND_AMENDORDER_TIMESTAMPID_OFFSET` is `3`
- `CONFIG_OUTBOUND_CANCELORDER_TIMESTAMPID_OFFSET` is `5`
- `CONFIG_OUTBOUND_ORDERUPDATE_TIMESTAMPID_OFFSET` is `7`

For example, the timestamp identifier you would use for when a new order entered the risk firewall would be:

```
(CONFIG_LATENCY_TIMESTAMPID_BASE+CONFIG_INBOUND_NEWORDER_TIMESTAMP)
```

Receiving approved orders from a risk firewall

You use a `com.apama.firewall.OrderReceiver` object to receive approved orders from a risk firewall and to set callbacks that handle the approved orders according to your application requirements. Callbacks on order receiver components are the only way to process risk firewall approved orders.

To obtain an order receiver object, call the `RiskFirewall.getOrderReceiver()` action on the risk firewall you want to receive orders from. This can be a local risk firewall or a remotely connected

risk firewall. The `getOrderReceiver()` action returns an `OrderReceiver` object that you can use with only the risk firewall that supplied it.

An order receiver receives `NewOrder`, `AmendOrder`, and `CancelOrder` objects that have been approved by the risk firewall. The callbacks that you add to an order receiver determine what happens to each received event. For example, you can add callbacks that send orders to an adapter, a trading algorithm, or a smart order router.

An order receiver also receives order updates from the external components that fill orders, for example, a particular market provider. The order receiver passes these order updates back into the risk firewall so the registered rule classes can increment any cumulative values and perform any housekeeping. The order update is then passed to the risk firewall order sender that originated the order. This completes the order.

An order receiver object provides actions for

- Adding callbacks that the risk firewall executes after it approves an order. You can add different callbacks according to whether the approved order is new, amended, or cancelled. See:
  - "Setting accepted-order callbacks" on page 163
  - "Setting accepted-amended-order callbacks" on page 164
  - "Setting accepted-cancelled-order callbacks" on page 164
- Sending order updates into the risk firewall. The order updates must be for an order that the risk firewall is currently handling. See "Processing order updates" on page 165.

Typically, you use only one order receiver. However, you can obtain multiple `OrderReceiver` instances from a risk firewall. For example, you might want to create more than one order receiver in the case where you want to monitor approved orders that are going to an exchange. You could have one order receiver connected to your adapter and another order receiver connected to a user interface. All `OrderReceiver` objects receive all approved OMS events from the risk firewall.

If you spawn to a new context you cannot use the order receiver in the new context.

For an example of using an `OrderReceiver` object, see "Sample code for using OrderSender" on page 159, which shows how to use both components.

Setting accepted-order callbacks

After a risk firewall approves an order, it calls any accepted-order callbacks that you have added to the order receiver that receives the approved order. For example, you might want to add an accepted-order callback that forwards approved orders to a particular adapter.

To add an accepted-order callback, call `com.apama.firewall.OrderReceiver.addAcceptedOrderCallback()`. This action takes one argument, which is the callback to be executed. The callback itself takes one argument, which is a `com.apama.oms.NewOrder` object. The action registers a callback with the risk firewall instance it is executed on. The registered callback is executed whenever the risk firewall approves an order. You can add more than one accepted-order callback.

The `addAcceptedOrderCallback()` action returns a unique integer reference Id that you can use to remove the callback at a later date if required.

To remove a previously-added accepted-order callback, call `OrderReceiver.removeAcceptedOrderCallback()` and specify the integer reference Id that was returned when you added the callback. To remove all previously-added accepted-order callbacks, call the `OrderReceiver.clearAcceptedOrderCallbacks()` action.

Setting accepted-amended-order callbacks

After a risk firewall approves an amendment to an order, it calls any accepted-amended-order callbacks that you have added to the order receiver that receives the approved order amendment. For example, you might want to add an accepted-amended-order callback that forwards approved order amendments to a particular adapter.

To add an accepted-amended-order callback, call `com.apama.firewall.OrderReceiver.addAcceptedAmendCallback()`. This action takes one argument, which is the callback to be executed. The callback itself takes one argument, which is a `com.apama.oms.AmendOrder` object. The action registers a callback with the risk firewall instance it is executed on. The registered callback is executed whenever the risk firewall approves an amended order. You can add more than one accepted-amended-order callback.

The `addAcceptedAmendCallback()` action returns a unique integer reference Id that you can use to remove the callback at a later date if required.

To remove a previously-added accepted-amended-order callback, call `OrderReceiver.removeAcceptedAmendCallback()` and specify the integer reference Id that was returned when you added the callback. To remove all previously-added accepted-amended-order callbacks, call the `OrderReceiver.clearAcceptedAmendCallbacks()` action.

Setting accepted-cancelled-order callbacks

After a risk firewall approves cancellation of an order, it calls any accepted-cancelled-order callbacks that you have added to the order receiver that receives the approved cancellation. For example, you might want to add an accepted-cancelled-order callback that forwards approved cancellations to a particular adapter.

To add an accepted-cancelled-order callback, call `com.apama.firewall.OrderReceiver.addAcceptedCancelCallback()`. This action takes one argument, which is the callback to be executed. The callback itself takes one argument, which is a `com.apama.oms.CancelOrder` object. The action registers a callback with the risk firewall instance it is executed on. The registered callback is executed whenever the risk firewall approves cancellation of an order. You can add more than one accepted-cancelled-order callback.

The `addAcceptedCancelCallback()` action returns a unique integer reference Id that you can use to remove the callback at a later date if required.

To remove a previously-added accepted-cancelled-order callback, call `OrderReceiver.removeAcceptedCancelCallback()` and specify the integer reference Id that was returned when you added the callback. To remove all previously-added accepted-cancelled-order callbacks, call the `OrderReceiver.clearAcceptedCancelCallbacks()` action.

Handling orders rejected by a risk firewall

Your application can identify a rejected order in several ways:

- Add a query response callback to a risk firewall. The risk firewall executes this callback, which provides rule class query results, for every order it evaluates. The default behavior is that query results contain only failure and warning responses, and not approval responses. See "Setting risk firewall query response callbacks" on page 107.
- Order rejection is typically apparent in a `com.apama.oms.OrderUpdate` event that is received by an order sender component as the result of an order update callback. Of course, `OrderUpdate` events are also used to indicate that an order has been filled, amended, or cancelled.

Orders that are rejected by the risk firewall rather than an external service have the extra parameter `"__Firewall_Reject"` set to `"true"`.

If the risk firewall is running in soft rejection mode, you can call the `RiskFirewall.overrideSoftReject()` action to allow a previously rejected new, amended, or cancelled order through the Risk Firewall. You can call this action from a local risk firewall or from a remotely connected risk firewall.

Processing order updates

After an external component, such as an adapter or OMS handler, processes an order, it sends a `com.apama.oms.OrderUpdate` event to the risk firewall order receiver that forwarded the order. Your application should then call the `OrderReceiver.sendOrderUpdate()` action to pass the order update back into the risk firewall. The `OrderReceiver.sendOrderUpdate()` action takes one argument, which is a `com.apama.oms.OrderUpdate` event for an order that the risk firewall is currently processing.

The risk firewall then executes the `RuleClass.processRuleClassUpdate()` action and `RuleClass.processRuleInstanceUpdate()` action, if they are implemented. This allows the registered rule classes to update any internal state that might be dependent on the information in the order update. For example, a custom rule class implementation might track when an order has been filled and by how much.

After all registered rule classes process the order update, the risk firewall calls the order update callbacks and passes the order update to the order sender that originated the order.

Considerations when implementing multiple firewalls

You can create multiple risk firewall instances in a single context or in multiple contexts. For example, you might choose to create one risk firewall for evaluating client orders and another risk firewall for evaluating internal orders.

You must ensure that if multiple risk firewalls send orders to the same venue, they do not use conflicting rule class configurations. You want to avoid a situation where one risk firewall would reject an order that another risk firewall would approve.

After you create a risk firewall, if you spawn to a new monitor or different context you cannot use that risk firewall from the new monitor or context. The main reason for this is that risk firewall names must be unique within an application. If you need to use the risk firewall from the spawned monitor or context, you can connect from the spawned monitor or context to risk firewall in the source monitor.

Sample code for creating risk firewalls in multiple monitors

The following example shows the creation of risk firewalls in three different monitors.

The first monitor creates a risk firewall instance, registers a rule class, and adds a rule class instance. After the risk firewall is unlocked, the monitor tries to submit an order to it by using the risk firewall order sender. The set of events after creating the risk firewall are asynchronous and are pended until after the CMF Service Framework is activated, the risk firewall is created, the rule class is registered, and the rule class instances are added. OMS events are not pended by the risk firewall. Consequently, there is no attempt to send the new order immediately after the call to unlock the risk firewall.

The second monitor connects to the risk firewall created in the first monitor. The second monitor uses the risk firewall's order receiver to register a callback. This callback receives any new orders that the risk firewall approves and logs them.

The third monitor also connects to the risk firewall created in the first monitor. It queries the remotely connected risk firewall and logs any failures.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;
using com.apama.firewall.QueryRequest;
using com.apama.firewall.CombinedQueryResponse;
using com.apama.firewall.QueryConstants;
using com.apama.utils.Params;
using com.apama.oms.NewOrder;

monitor RiskFirewallExample13 {
   context mainContext := context.current();

   action onload() {
      // Use a risk firewall factory to create a new risk firewall
      // instance called "MyFirewall" in the current context.
      // Use the default configuration parameter settings.
      RiskFirewall rfw :=
         (new RiskFirewallFactory).create( mainContext, "MyFirewall" );

      // Register a risk firewall rule class.
      rfw.registerRuleClass( (new MyRiskFirewallRuleClass).create() );

      // Define the configuration for the new rule class instance.
      Params config := new Params;
      config.addParam( "SYMBOL", "APMA.L" );
      config.addParam( "PRICE_LIMIT", (22.2).toString() );

      // Add the rule class instance to the risk firewall.
      integer instanceId := rfw.addRuleInstance(
         "MyRiskFirewallRuleClass", config );
```

```
      // Unlock the risk firewall to allow orders to be processed.
      rfw.unlockCb( cbOnFirewallUnlocked );
   }

   // This action is called when the risk firewall is unlocked.
   action cbOnFirewallUnlocked ( RiskFirewall rfw ) {
      // Send a new order to the risk firewall.
      rfw.getOrderSender().sendOrder( NewOrder(
         "orderId_1", "APMA.L", 1.0, "BUY", "MARKET", 10,
         "TargetService", "", "", "TargetMarket", "",  "TraderA",
         new dictionary<string,string> ) );
    }
}

// This monitor receives any approved OMS events from the
// risk firewall and logs them.
monitor RiskFirewallExample14 {
   context mainContext := context.current();

   action onload() {
      // Using a risk firewall factory to connect to the
      // risk firewall created in the previous monitor.
      RiskFirewall rfwRemote :=
         (new RiskFirewallFactory).connect( mainContext, "MyFirewall" );

      // Add a receiver for orders approved by the risk firewall.
      // Add a callback to be executed after an order is approved.

      integer refId := rfwRemote.getOrderReceiver().
         addAcceptedOrderCallback( cbOnApprovedNewOrders );
    }

   // Set up a callback handler for new orders that the
   // risk firewall has approved.
   action cbOnApprovedNewOrders ( NewOrder order ) {
      log "Firewall has allowed the NewOrder: "+order.toString();
   }
}

// This monitor checks for any queries that have failed and logs them
// This is similar to how an auditor part of an application might work.
monitor RiskFirewallExample15 {
   context mainContext := context.current();

   action onload() {
      // Use a risk firewall factory to connect to the risk firewall
      // that was created in the first monitor, "MyFirewall".
      RiskFirewall rfwRemote :=
          (new RiskFirewallFactory).connect( mainContext, "MyFirewall" );

      // Add a query response callback to monitor the state of all queries
      // in the risk firewall.
      integer refId := rfwRemote.addQueryResponseCallback(
         cbQueryResponseHandler );
   }

   // Set up a callback handler for queries that the risk firewall
   // has processed.
   action cbQueryResponseHandler(
      RiskFirewall rfwRemote,
```

```
         QueryRequest query,
         CombinedQueryResponse response ) {
      // Report any errors.
      if( response.overallResponse =  QueryConstants.QUERYRESPONSE_FAIL )
         then {
            log "The risk firewall rejected the following query:
               "+query.toString() at ERROR;
      }
   }
}
```

Implementing custom risk firewall rule classes

If the default rule classes provided with the risk firewall do not meet your needs you can implement a custom rule class. Implementation of a custom rule class is accomplished by using a `com.apama.firewall.RuleClassFactory` to create a `com.apama.firewall.RuleClass` object. You then override the default action implementations in your `RuleClass` object with actions that provide the rule class logic you need. After you create a custom rule class, you register it with a risk firewall. You can then add instances of your custom rule class just as you would add instances of a default rule class.

Introduction to custom rule class implementation

A `com.apama.firewall.RuleClass` object defines the following default actions. As with default rule classes, the risk firewall calls these actions on your custom rule class as needed.

- `addRuleInstance()` — Adds an instance of your custom rule class.
- `modifyRuleInstance()` — Modifies an instance of your custom rule class.
- `removeRuleInstance()` — Removes an instance of your custom rule class.
- `evaluateRuleClassQuery()` — Queries the global state of your custom rule class.
- `evaluateRuleInstanceQuery()` — Queries each added instance of your custom rule class.
- `processRuleClassUpdate` - Processes an order update against the global state of the rule class.
- `processRuleInstanceUpdate()` — Processes an order update once for each added instance of your custom rule class.
- `getRuleInstanceState()` — Obtains information/state for an instance of your custom rule class.
- `getRuleClassState()` — Obtains information/state for your custom rule class.
- `getRuleClassName()` — Obtains the name of your custom rule class.
- `getRuleClassDescription()` — Obtains the description of your custom rule class.
- `getSchema()` — Obtains the configuration schema for your custom rule class.

At a minimum, a custom rule class must override either

`evaluateRuleClassQuery()`

or

`evaluateRuleInstanceQuery()` and `addRuleInstance()`

If you override only `evaluateRuleClassQuery()` then you can use your rule class to check all OMS events that are passed into the risk firewall. This can be useful if you want to sanity check (global state check) every order for some specific criteria. However, you will not be able to add any rule class instances. If you choose to implement all of the actions then your rule class can perform a sanity check on every order and also evaluate every order against any rule class instances that have been added.

The `addRuleInstance()`, `modifyRuleInstance()` and `removeRuleInstance()` actions each provide two callbacks. One callback indicates to the risk firewall that the operation was successful. The other callback indicates an error. This lets you implement these operations in an asynchronous way. After your custom rule class has completed the operation, it must execute one of the callbacks.

If there is an error while adding, modifying, or removing a custom rule class instance, your implementation should create a `com.apama.utils.Error` object that contains useful, human-readable, details about the error. This error object provides an `errorType` parameter, which is always one of

- `FAILED_TO_ADD_RULE_INSTANCE`
- `FAILED_TO_MODIFY_RULE_INSTANCE`
- `FAILED_TO_REMOVE_RULE_INSTANCE`

These constants are defined in `com.apama.firewall.ErrorConstants`. If an error callback has been set on the risk firewall then the error type lets the caller programmatically identify the cause of the error. However, if your custom rule class implementation provides another value for `errorType`, it is stored in the `Error` object's parameters as `ERROR_SUBTYPE`.

The risk firewall `RuleClass` interface stores instance configurations internally. However, if your custom rule class maintains any internal (or live) state information then your custom rule class must also cache and (if necessary) persist this state information. An application's custom rule class implementation can use the CMF Configuration Service to persist/recover any internal state information. See "Configuration Service" on page 237.

The risk firewall uses the CMFs Configuration Store to store custom rule class instance configurations. This enables queries by other parts of the application from any context. Also, DataViews are automatically published for risk firewall `RuleClass` instance tables. You can use these DataViews to publish risk firewall `RuleClass` information and instances to a user interface.

When the risk firewall needs to evaluate a query against a custom rule class implementation, it will first attempt to call the `evaluateRuleClassQuery()` action. If the query passed against this action, the success callback action allows the application to specify whether the `evaluateRuleInstanceQuery()` action will be called for each individual custom rule class instance.

If required, the `evaluateRuleInstanceQuery()` action will be called for each custom rule class instance that was added, and will provide both the instance Id and the configuration that was defined for this instance, as well as the `QueryRequest` object. As the configuration parameters for the `RuleClass` instances are stored internally to the risk firewall, this means that your `RuleClass`

implementation does not have to cache information about its instances unless it needs to (either because of efficiency or any other reason).

Execution of `evaluateRuleClassQuery()` or `evaluateRuleInstanceQuery()` results in approval or rejection of an order. Execution of `ProcessRuleClassUpdate()` and `ProcessRuleInstanceUpdate()` cannot reject the order update.

General steps for implementing custom rule class

The general steps for implementing a custom risk firewall rule class are as follows:

1. Use `com.apama.firewall.RuleClassFactory` to create a `com.apama.firewall.RuleClass` object, which defines a number of actions.
2. Write implementations that override the default `RuleClass` actions. You are required to override either the `evaluateRuleClassQuery()` action or both the `addRuleInstance()` and `evaluateRuleInstanceQuery()` actions. You must also write implementations that override any other `RuleClass` actions your custom rule class requires.
3. If you override the `RuleClass.addRuleInstance()` action, you should
   - Check that the configuration of an instance of your custom rule class that an application is trying to add is valid.
   - Provide informative error messages if an instance of your custom rule class is missing a required parameter setting or has an invalid parameter setting.
4. If you override the `RuleClass.evaluateRuleClassQuery()` action or the `RuleClass.evaluateRuleInstanceQuery()` action, you should
   - Perform the custom logic that you created the rule for. You are provided with both the query request and the original configuration for the rule instance being evaluated.
   - Based on the results of your custom evaluation, construct a query response object that uses the query request identifier.
   - Call the provided completion callback with the query response.
5. Define the configuration schema for your custom rule class. The configuration schema specifies the parameters for evaluating an order against your custom rule. For each parameter, you can specify the parameter name, parameter type, default value, and a description.
6. Use the rule class object to register your custom rule class with one or more risk firewalls.
7. Add one or more instances of your custom rule class to each risk firewall with which you registered your custom rule class.

See "Sample code for implementing custom risk firewall rule class" on page 171.

Best practice

The following are the recommended practices for implementing a custom rule class.

- Encapsulate the logic of your custom rule class in an event.
- In this event, define an `action` type field named `create()` that returns a `RuleClass` object.
- In your `create()` action:
  - Instantiate the `com.apama.firewall.RuleClassFactory`, and call `create()` on this factory to obtain a `com.apama.firewall.RuleClass` object.
  - Create and override each `RuleClass` action required by your custom rule class.

The default rule class implementations follow these best practices. See also "Sample code for implementing custom risk firewall rule class" on page 171.

Sample code for implementing custom rule class

The following code is an example of defining a custom rule class. It shows a very basic rule class implementation that checks new order prices to ensure they do not exceed a defined maximum value. In this example, a new rule class instance can be defined for a single symbol name, and a single price limit. This is only to reduce the complexity of the example code. The rule class is implemented as an event object to show the best practice implementation pattern.

The risk firewall default rule classes follow a similar implementation. See the risk firewall samples in the `samples` directory of your CMF installation directory.

```
using com.apama.firewall.RuleClassFactory;
using com.apama.firewall.RuleClass;
using com.apama.firewall.QueryRequest;
using com.apama.firewall.QueryResponse;
using com.apama.utils.Error;
using com.apama.utils.Params;
using com.apama.utils.ParamsSchema;

event MyRiskFirewallRuleClass {

   action create() returns RuleClass {

      // Construct an instance of the risk firewall rule class interface
      // that overrides provided actions to implement custom functionality.
      // This example does not override the modifyRule() action because the
      // implementation does not support modifying rule instances. This
      // example also does not override the deleteRuleInstance() action
      // because this implementation does not do any clean up.
      RuleClass ruleClassIface :=
         (new RuleClassFactory).create( "MyRiskFirewallRuleClass" );
      ruleClassIface.addRuleInstance           := addRuleInstance;
      ruleClassIface.getSchema                 := getSchema;
      ruleClassIface.evaluateRuleInstanceQuery := evaluateRuleInstanceQuery;

      return ruleClassIface;
   }

   action getSchema() returns ParamsSchema {

      // Define the configuration schema for this new rule class.
      ParamsSchema configSchema := new ParamsSchema;
      configSchema.addItemMinimal(
```

```
      "SLICE_SYMBOL","string","","Symbol set this price limit is for");
   configSchema.addItemMinimal(
      "PRICE_LIMIT","float","0.0","Maximum price allowed for orders");

   return configSchema;
}

// This action is called whenever the Risk Firewall Service needs to
// create a new instance of this custom rule class implementation.
action addRuleInstance( integer instanceId, Params config,
                        action<> cbOnSuccess,
                        action< Error > cbOnError ) {
   boolean success  := false;
   Error error := new Error;

   // Check that the configuration is valid.
   if( config.hasParam( "SLICE_SYMBOL" ) ) then {
      if( config.hasParam( "PRICE_LIMIT" ) ) then {
         // Required parameters are provided.
         success := true;
      } else {
         // Add some information about missing parameter.
         error.params.addParam("MISSING_PARAM", "PRICE_LIMIT");
         error.message := "Failed to add Risk Firewall Rule Instance :
            "+instanceId.toString()+" :
               Missing 'PRICE_LIMIT' configuration value.";
      }
   } else {
      // Add some information about missing parameter.
      error.params.addParam("MISSING_PARAM", "SLICE_SYMBOL");
      error.message := "Failed to add Risk Firewall Rule Instance :
         "+instanceId.toString()+" :
         Missing 'SLICE_SYMBOL' configuration value.";
   }

   // Use a callback to indicate to the risk firewall whether the
   // operation was successful.
   if( success ) then {
       cbOnSuccess();
   } else {
       cbOnError( error );
   }
}

// This action is called whenever the Risk Firewall Service needs to
// query this custom rule class implementation.
action evaluateRuleInstanceQuery(
   integer instanceId,
   Params config,
   QueryRequest query,
   action< integer/*instanceId*/ QueryResponse > cbCompleted ) {

      // Construct the query result object, default to pass.
      QueryResponse queryResponse := new QueryResponse;

      // Set the requestId on the response to match it to the request.
      queryResponse.requestId := query.requestId;

      // Check that this symbol is in the specified slice.
      string symbol := query.params.getSymbol();
```

```
         <string> symbolSlice := sequence<string>.parse(
            config.getParam( "SYMBOL_SLICE" ) );

         if( symbolSlice.indexOf( symbol ) > 0 ) then {

            // Check if the order price exceeds the price defined in
            // this rule. If it does, return the query as invalid.
            float  orderPrice := query.params.getPrice();
            float  priceLimit := config.getFloatParamOr( "PRICE_LIMIT", 0.0 );

            if( orderPrice > priceLimit ) then {
               queryResponse.response := QueryConstants.QUERYRESPONSE_FAIL;
               queryResponse.message  :=
                  "Price Order Limit exceeded for "+symbol+" - Limit is:
                     "+priceLimit.toString();
            }
         }

      // Execute callback to indicate to the Risk Firewall Service
      // whether the query passed or failed.
      cbCompleted( instanceId, queryResponse );
   }
}
```

The following code shows how this custom `RuleClass` implementation would be registered.

```
using com.apama.firewall.RiskFirewallFactory;
using com.apama.firewall.RiskFirewall;

monitor RiskFirewallCustomRuleClassExample {
   context mainContext := context.current();

   action onload() {
      // Use the risk firewall factory to create a new risk
      // firewall instance called "MyFirewall". In this case,
      // use the default configuration parameter settings.
      RiskFirewall rfw :=
         (new RiskFirewallFactory).create( mainContext, "MyFirewall" );

      // Register custom rule class with this risk firewall.
      myFirewall.registerRuleClass( (new MyRiskFirewallRuleClass).create() );

      // Alternatively, for notification of when registration of the
      // rule class is complete, call the following action.
      rfw.registerRuleClassCb(
         (new MyRiskFirewallRuleClass).create(), cbOnRuleClassRegistered );
    }

   // This action is called after the rule class is successfully registered.
   action cbOnRuleClassRegistered(
      RiskFirewall rfw, string ruleClassName ) {
         log "Successfully registered RuleClass: "+ruleClassName;
    }
}
```

How a risk firewall queries rule classes

The following background information may be helpful for implementing custom rule classes. Keep in mind that for a risk firewall to approve an order, that order must comply with at least one instance of each rule class that is registered with that risk firewall. If a custom rule class does not require instances to be added, then approval of an order requires that the order comply with the custom rule class itself.

When a risk firewall receives an order it creates a `com.apama.firewall.QueryRequest` object, which contains a request Id, a request type (new, amend, or cancel), and a `com.apama.oms.OrderState` object. The risk firewall sends the query request to rule classes and their instances and receives a `com.apama.firewall.QueryResponse` object back.

The risk firewall first sends the query to the registered rule classes according to any priority ordering that has been set. If more than one rule class has the same priority, the query goes to those rule classes in the order in which they were registered. If no priority ordering has been set, then the query goes to the registered rule classes in the order in which they were registered. The risk firewall queries the rule class implementation before querying any rule class instances. This allows validation against any global, state information, and it can also allow your custom rule implementation to iterate over its own instances.

If the rule class query is successful then the risk firewall begins to query each instance of each rule class.

By default, if a rule class instance returns a failure in the query response object, then no further query requests are sent to the other instances for that rule class, or any other rule classes that have not been queried. As the mechanism is asynchronous, there may be outstanding queries being calculated. These will be ignored when they return. On a failed query response, the associated OMS event is rejected.

This early termination mechanism is configurable, so that it can be disabled if responses from all registered rule classes are required. However, this has a performance impact as the response is pended until all registered rule classes and their instances have been queried.

An application can set one or more callbacks with the risk firewall by using the `RiskFirewall.addQueryResponseCallback()` action to monitor query requests. A query response callback provides a `com.apama.firewall.CombinedQueryResponse` object that contains the set of query responses from all of the queried rule classes. This might be a subset of those registered depending on the query and the failure mode in operation. In a query response callback, the application cannot override the response that has been returned from the risk firewall. The application can, for example, use the information provided in the query response to determine whether the failure is critical enough to prevent further orders from being processed by the risk firewall, or to simply monitor orders being processed (such as an auditing system).

The risk firewall uses a configurable timeout, set by the `CONFIG_TIMEOUT_DURATION` configuration parameter, on all the query requests. This ensures that the state of an order is not blocked by unresponsive rule classes. If the timeout period is reached (the default setting is 5 seconds), the default behavior is that the order will be rejected as a result of the request timing out.

On receipt of the query response objects, the risk firewall determines whether the query result was pass or fail, and takes the appropriate action. For a failed query result, the appropriate action is determined by the rejection mode that the risk firewall is using. See "Configuring risk firewall behavior for rejected orders" on page 109.

Configuring a risk firewall to support the legacy order event interface firewall

The risk firewall's `OrderSender` and `OrderReceiver` objects also provide built-in backward compatibility support for applications that were using the legacy Firewall Service. The `CONFIG_ENABLE_LEGACY_MODE` risk firewall configuration parameter enables a set of listeners on the risk firewall to listen for `com.apama.oms` events with a `ServiceId` of `"__ObjectionBasedFirewallControllerExternal"`. These orders must also contain the service Id that this OMS event is destined for as an extra parameter with the key `"Firewall.TargetService"`.

These steps are automatically done for the application if it is using the CMF Order State Containers when using the *xxx*`ViaFirewall()` actions. This mode also routes any approved OMS events from the risk firewall, using the service Id that was stored in the `"Firewall.TargetService"` extra parameter.

Smart Order Router

The Smart Order Router (SOR) framework is part of the Order Management components of the Capital Market Foundation, which include order management, exchange simulation, and firewall functionality.The SOR framework provides a mechanism by which an order can be created and executed with a user-defined strategies. The original parent order may be executed by creating one or more child orders. The results of the execution of child orders will be reflected in the parent order object.

For example, a strategy might split a parent order into two separate child orders, each containing half of the parent order quantity. These child orders will then be placed on two separate venues. When one child order is filled, the parent orders quantity executed field will reflect that child fill. When both child orders fully fill, the parent order will also indicate that it has been fully filled. See the SOR sample for more information.

The user API provides functionality sufficient for the majority of applications. The API includes a SOR strategy factory object which can be used to create strategy instances. Each strategy instance allows you to define a set of callback actions at each stage of handling the SOR order.

If datasources run in the main context, processing of the orders in the same context could create a performance bottleneck. Therefore, the interfaces have been designed such that the SOR framework can be created in a different context. The SOR framework uses the CMF's Order Management Support component to subscribe to `OrderUpdate` events that are received in the main context, and reflect them to the context that the SOR is running in. As a result, the main context is only used for reflecting the received order management events to the appropriate context for processing.

Creating a strategy instance

To create a strategy instance, obtain a object from the factory, which can then be used to query and create a specific strategy.

```
// Create a new StrategyHelperInterface
com.apama.sor.strategy.StrategyHelperInterface strategyHelperIface :=
```

```
   (new com.apama.sor.strategy.StrategyHelperFactory).createInterface();
```

Once the `StrategyHelperInterface` has been created, you can create a new strategy instance and implement the full set of the strategy callbacks required.

Implementable callbacks include:

- `strategyError()` – Called if there has been an error in processing the new order (for example, if the order was rejected, or an attempt was made to modify/cancel an order with an invalid order ID).
- `setChildOrderHandler()` – Called after the Strategy has been created, and allows for overriding the handling of child orders before they are placed on the venue.

The following code extract shows how you might create a new strategy with callback actions:

```
// Define the variables to be used to create the Strategy
com.apama.sor.utils.Auditor auditor;
com.apama.sor.strategy.StrategyHelperCallbacks userDefinedCallbacks :=
   com.apama.sor.strategy.StrategyHelperCallbacks(myStrategyErrorCallback,
   myStrategyChildOrderHandlerCallback );
// Create a new Strategy Helper Interface
com.apama.sor.strategy.StrategyInterface strategyIface :=
   strategyHelperIface.createStrategy(“MY_STRATEGY”, context.current(),
   auditor, userDefinedCallbacks );
```

In some cases, you will not want to do anything specific for some of these callback actions. So to reduce the amount of code that you have to write, which in turn reduces the complexity, you can alternatively use the default implementations of some of the callback actions. The `StrategyHelperInterface`, contains default callback implementations. You can override only the appropriate actions.

This example shows how to override the "`strategyError`" callback action and leave the default behaviors for the other callbacks:

```
// Define the variables to be used to create the Strategy
com.apama.sor.utils.Auditor auditor;
com.apama.sor.strategy.StrategyHelperCallbacks userDefinedCallbacks :=
   strategyHelperIface.getDefaultHelperCallbacks();
userDefinedCallbacks.strategyError := myStrategyErrorCallback;
// Create a new Strategy Helper Interface
com.apama.sor.strategy.StrategyInterface strategyIface :=
   strategyHelperIface.createStrategy(“MY_STRATEGY”, context.current(),
   auditor, userDefinedCallbacks );
```

Creating orders for the strategy

After the strategy instance has been created, use the `StrategyInterface` to create new orders and to handle existing orders.

The `StrategyInterface` supports the following methods:

- `create()`– Creates a new order for the strategy instance.
- `amend()`– Amends an order currently being handled by the strategy instance.

- `cancel()`– Cancels an order currently being handled by the strategy instance.
- `stopStrategy()`– Stops the strategy instance. If child orders exist in the market these will be pulled before the strategy is stopped.
- `getStrategyName()`– Returns the name of the strategy.
- `getParentOrder()`– Returns the parent order from the strategy for the `OrderId` provided.

Once the new order has been received and validated by the application, the `create()` action is called to create new order with the Strategy. This action takes a success and a failure callback action. If there was an error creating the order with the Strategy, then the failure callback will be called with the OrderId and an appropriate error message. If the order was successful, the success callback is called with the `ParentOrderContainer` object.

There are various actions that can be performed on the `ParentOrderContainer` to assist the user in creating an algorithm to execute the order. These include:

- `makeChildOrder()`– This is the primary function that is used to create the underlying child orders.
- `amendChildOrder()`– Amends a specific child order given a specific `OrderId`.
- `cancelChildOrder()`– Cancels a specific child order given a specific `OrderId`.
- `cancelChildOrders()`– Cancels child orders given a list of `OrderIds`
- `cancelAllChildOrdersFromDestination()`– Cancels all child orders made to a given destination
- `cancelAllChildOrders()` – Cancels all active child orders
- Other actions are available to get to get details about the order itself such as the quantity, price, side, type of order, quantity remaining, quantity working, etc.

Overriding the processing of child orders

As different datasources potentially require specific information to be provided when placing orders with them, a mechanism is provided to allow users to process the child orders before they are placed, amended, or cancelled.

In order to override the default behavior, the user must define a callback action for the `setChildOrderHandler()` action on the `StrategyHelperInterface`. This action will be called from the strategy when a new order is created, and needs to return a `ChildOrderNormaliserInterface`.

As with the other interfaces, the user can choose which action behavior they want to override. The following code extract shows how a new `ChildOrderNormaliserInterface` can be created, overriding the behavior for creating a new instance of a `ChildOrderNormaliser`, and for processing new child orders.

```
// Create a ChildOrderNormaliser Interface
com.apama.sor.oms.ChildOrderNormaliserInterface normaliser :=
   (new com.apama.sor.oms.ChildOrderNormaliserFactory).createInterface();
normaliser.create := myNormaliserCreate;
normaliser.processNewOrder := myNormaliserNewOrderHandler;
```

Amending or cancelling the parent order

When the user wants to amend or cancel an order currently being handled by the framework, they can call the `amend()` or `cancel()` action on the `StrategyInterface`. If the validation of the amend/cancel was successful, the user will be called back on the `success()` callback that was passed in as a parameter of amend or cancel action.

If validation was unsuccessful, and there was a problem with the orders amend/cancel, then the user will be called back on the `failed()` callback that was passed in as a parameter of the amend or cancel action with an appropriate error response.

Stopping the strategy

After the strategy has finished processing the new order, the user must call the `stopStrategy()` action on the `StrategyInterface` object that was returned when the strategy was initially created.

The point at which the strategy should be stopped is completely user defined. Generally, this action is called after the order has been fully filled, or to terminate the strategy gracefully in the event of an error.

Order bridging services

The order bridging service is analogous to the adapter status and market data bridging services, but for order management events. It provides bridging of `com.apama.oms.*` events between two correlators, so that an application in the client correlator can submit orders to an adapter or other order destination connected to the server correlator. An order management bridge instance is configured by sending configuration events to the client and server correlators.

On the client side, send a `com.apama.oms.ConfigureClientSideBridge` event, and on the server side, send a `com.apama.oms.ConfigureServerSideBridge` event. See the ApamaDoc for these events for details about their fields. In most cases, the fields of the client and server configuration events for a given bridge instance are identical. However, the `instanceName` field must be unique across all order management bridge services used by the application.

In addition to configuring the order management bridge itself, an application should also configure monitoring of the fake adapter connection between the two correlators, using an instance of the `com.apama.connection.IAFStatusFaker` and `com.apama.adapters.IAFStatusToStatusConvertor` services on both sides of the connection. If this is not done, the inter-correlator connection will appear to be down to the order management bridge service and it will not forward any events in either direction. The generic bridging service described above handles this automatically; it is recommended that applications use this service wherever possible. Otherwise, see the implementation of the `com.apama.adapter.bridge.ConfigBridge` service in the Adapter Bridging Utils bundle to see how bridge connection monitoring is implemented.

Once the order management bridge service is configured, `com.apama.oms.NewOrder`, `AmendOrder` and `CancelOrders` events routed by the client-side application for the bridged service will be forwarded to the server correlator, where they should be handled by the order routing adapter service as normal. Any `OrderUpdate`events generated by the order routing adapter service will in

turn be forwarded back to the client correlator where they can be handled by application listeners in the usual way.

The order management bridge deals with multiple clients bridging to the same server-side order routing adapter service in the following ways:

- Each client correlator should configure a separate connection to an order management bridge service with a unique `instanceName` field and channel.
- Multiple order management bridge service clients within the same correlator should ensure that the order identifiers used on new orders are unique within that client correlator.

Trade reporting services

The Trade Services bundle is a small collection of services for handling trade streams. A trade stream is an event-based representation of all the trades that an application – or the organization running the application – was involved in. This contrasts with a tick stream, which is a representation of all the trades occurring on a particular venue between all participants. Trade events are typically generated by order routing – as opposed to market data – adapters and will contain a much richer set of fields than the corresponding tick events.

Trade gateway

The trade gateway is analogous to the position service and market data gateway services - it enforces transactional behavior on the events used by the trade services. As with the other gateway, applications do not need to do anything to use the trade gateway.

Trade extractor service

The trade extractor service monitors the order flow within an application and generates trade events when trades occur on monitored orders. Similar to the position services, the trade extractor service uses a subscription-based model allowing applications to subscribe to a stream of trade events. Again, the stream of interest is identified by a set of "slicing" parameters (see the description of slicing in "Defining slices that identify positions to be tracked" on page 186), but unlike the position services only a single value can be specified for each slice dimension. Otherwise the interface to the trade extractor is very similar to any of the position services, as shown in this example:

```
com.apama.trade.OrderMonitoringTradeExtractorConstants const:=
   new com.apama.trade.OrderMonitoringTradeExtractorConstants;
com.apama.trade.Subscribe sub := new com.apama.trade.Subscribe;
sub.serviceName := const.SERVICE_NAME();
sub.subscriptionId := integer.getUnique();
sub.symbol := "EUR/USD";
sub.service := "FIX";
sub.market := "CNX";
sub.trader := "A";
route sub;
com.apama.trade.SubscriptionResponse subr;
on com.apama.trade.SubscriptionResponse(
   serviceName = const.SERVICE_NAME(), subscriptionId =
```

```
sub.subscriptionId):subr {
   if subr.success then {
      log "Success!";
   }
   else {
      log "Failure!";
   }
}
com.apama.trade.Trade trade;
on all com.apama.trade.Trade(serviceName = const.SERVICE_NAME(),
tradeStreamId = subr.tradeStreamId):trade {
   log "Trade: "+trade.toString;
}
```

5 Analytics

- Using the position service framework 182
- Creating and configuring a currency converter 216

This section describes analytic components.

Using the position service framework

Applications can use the position service framework to calculate positions in different ways. The framework provides several default position tracker implementations as well as EPL interfaces that let you build custom position tracker implementations.

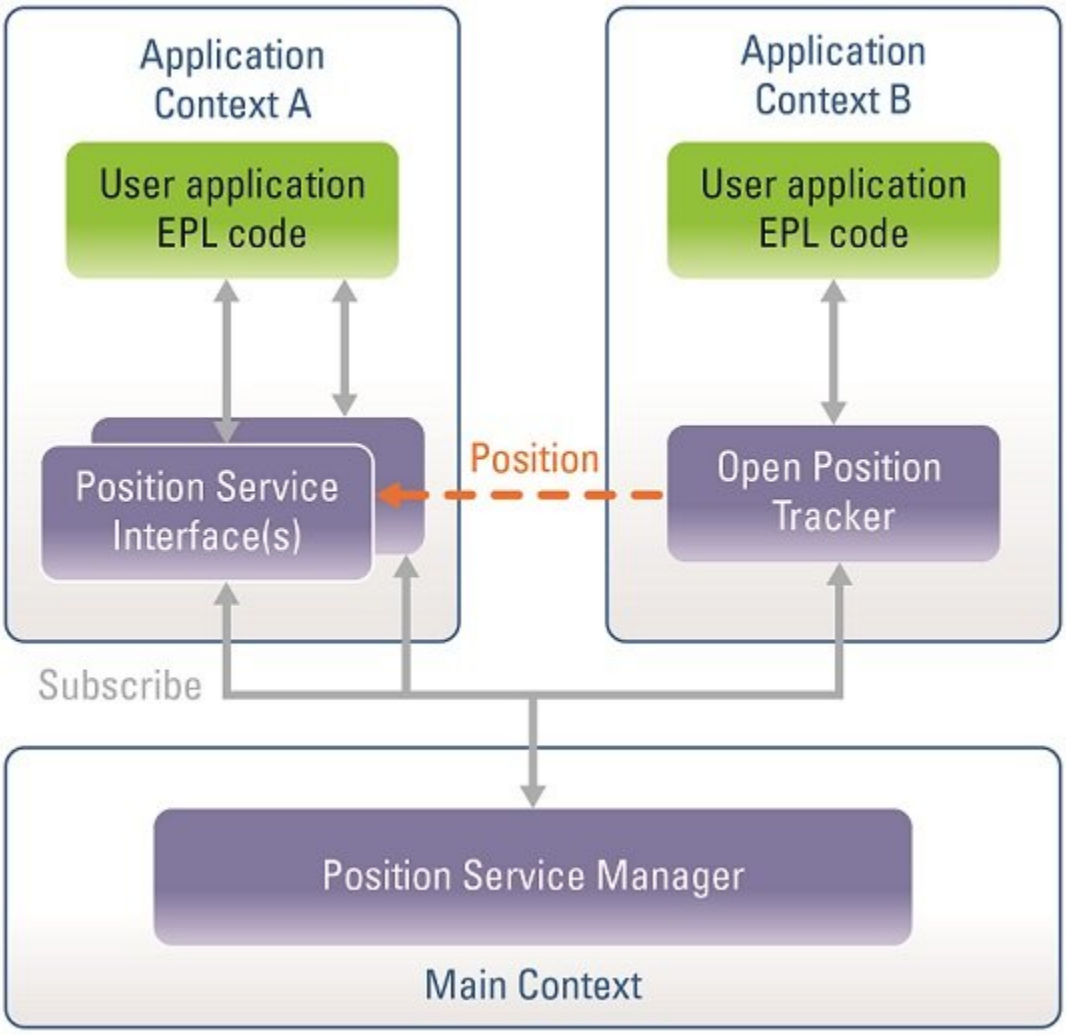
The position service framework uses position tracker names that are consistent with standard finance terminology. The position service framework provides support for your applications to track:

- Open positions - actual long and short positions that result from buy or sell order executions.
- Pending positions - potential long and short positions that could occur if currently open buy or sell orders are filled.
- Reserved positions - potential future positions that have been reserved.
- Realized profit and loss - actual profit and loss resulting from order executions and normalized to a specified currency.

For reference information, see the ApamaDoc.

Position service framework architecture

The following figure shows the typical architecture of the position service framework.



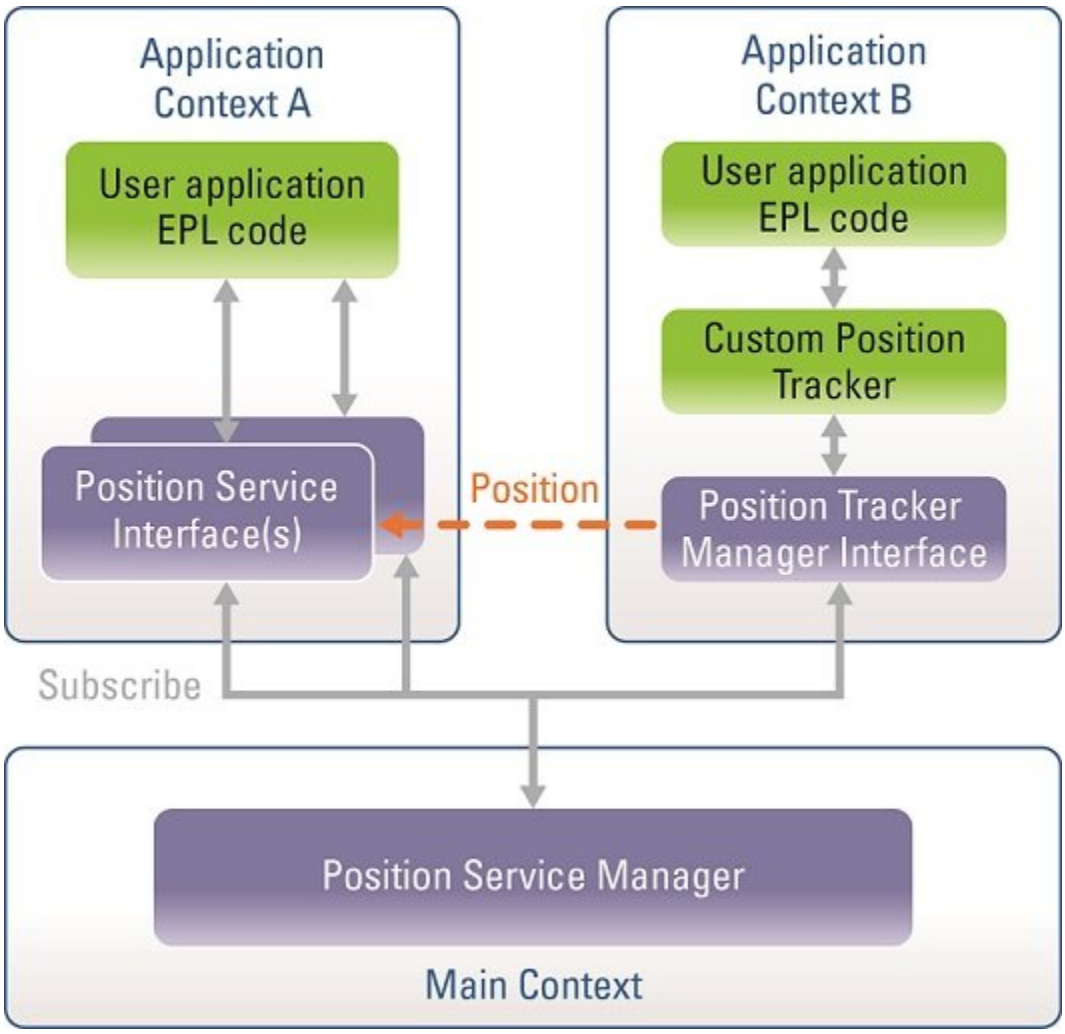
The “position service interface” provides the main interface for your application to subscribe to position trackers and monitor position updates based on the configuration provided. You can create a position service interface in any context and use it to subscribe to one or more previously-created position trackers.

The correlator automatically creates the “position service manager” in the main context of your application. This happens upon injection because the Position Management Service bundle is in your application. The position service manager handles all position tracker subscriptions in the application and automatically takes care of their persistence and recovery. See “Position service framework and persistence” on page 215. Each subscription has a unique subscription ID. An application must use the position service interface to interact with the position service manager; an application cannot interact directly with the position service manager.

The position service framework supports multiple contexts. You can create position service interfaces in any context, and you can set up as many position trackers as desired in as many contexts as you need.

The following figure includes the “position tracker manager interface” in the position service architecture. The position tracker manager interface lets you create custom position tracker

implementations that you can use in the position service architecture. The default position tracker implementations use the position tracker manager interface.



Overview of using the position service framework

To use the position service framework:

- Add the correct bundle(s) to your application.
- Create position trackers.
- Subscribe to position trackers

Adding bundles

You must add the Position Management Service bundle to your application in order to use the position service framework. You also need to add the Position Trackers bundle to your application. This bundle contains implementations for the default open, pending, reserved and realized P&L

position trackers. If you implement your own position trackers and use them in place of the default implementations then you do not need the Position Trackers bundle.

Creating position trackers

With the correct bundles in your application, set up one or more position trackers in the context that is receiving the orders that you want to track the position for.

To Set Up This Tracker Type	Call This Event's create() Action
Open position tracker	`com.apama.position.tracker.OpenPositionTrackerFactory`
Pending position tracker	`com.apama.position.tracker.PendingPositionTrackerFactory`
Reserved position tracker	`com.apama.position.tracker.ReservedPositionTrackerFactory`
Realized P&L tracker	`com.apama.position.tracker.RealizedPnLTrackerFactory`

When you call the `create()` action you specify a reference to the main context, which contains the position service manager. You also specify a user-defined tracker name that is unique in your application, and the user-defined action to call when the tracker has been set up. For examples, see "Creating and subscribing to the open position tracker" on page 196.

Subscribing to position trackers

In each context in which you want to monitor position updates from the position trackers you created, perform the following steps. While it is permissible to create the position tracker and perform the following steps all in the same context, using the same context is not required. That is, you can create the position tracker in one context and perform the following steps in as many other contexts as you need.

1. Instantiate a position service configuration object (`com.apama.position.PositionConfigParams`), which you specify in the next step. This configuration object is for future use; it contains no information in this release.

2. Create a position service interface, which your application will use to manage subscriptions to position trackers from the current context. You can have any number of instances of the position service interface in any number of contexts. To create a position service interface, call the `create()` action defined in the `com.apama.position.PSFactory` event. The `create()` action parameters specify a reference to the main context, a position service configuration object, and the user-defined action to execute upon creation of the interface object. An instance of

`com.apama.position.PSInterface` is returned in the callback. See "Creating and using position service interfaces" on page 189.

3. Subscribe to a position tracker that you previously created by executing the `subscribe()` or `subscribeAndMonitor()` action defined in the `com.apama.position.PSInterface` interface object. When you subscribe you specify:
   - The unique name of the position tracker that you want to subscribe to. This is the same name you provided when you created the position tracker.
   - A position service configuration object, which lets you provide details for the particular position you want to track. See "Defining slices that identify positions to be tracked" on page 186.
   - A user-defined action to execute in response to subscribing.
4. Define callback actions that will be executed in response to successful subscriptions. For examples, see "Creating and subscribing to the open position tracker" on page 196.

Defining slices that identify positions to be tracked

When you subscribe to a default position tracker, one of the parameters is a configuration object that can contain slice details for the position you want to track. Slices are a set of criteria that are used to match against orders coming into the system. If a subscription has no slice information configured it will match all orders.

Slicing lets you split a stream of events into substreams along several dimensions. Position service slices can split on the following dimensions:

- Symbol
- Service identifier
- Exchange identifier
- Market identifier
- Trader (owner) identifier

A slice can match a set of values in each dimension, or all values. For example:

- Symbol EUR/USD, for example, to track the organization-wide position in this instrument.
- Service identifier FIX and market identifier CNX, for example, to track the net position across all symbols and traders on a single market.
- Symbol EUR/USD, Service identifier FIX, market identifier CNX and trader identifier A or B, for example, to track the net position of this group of two traders for the EUR/USD instrument on a single market.

For details about the configuration parameters that let you specify slices , see "Using default position trackers" on page 193.

The following code extract constructs a configuration object that specifies a slice to monitor the net position of MSFT and APMA instrument by trader Gordon Gekko.

```
// Create a set of slice details to track the positions for.
// In this example, slice on symbols and traders:
sequence<string> symbolSlice := ["MSFT","APMA" ];
sequence<string> traderSlice := ["GGekko"];

com.apama.position.PositionConfigParams config :=
    new com.apama.position.PositionConfigParams;
config.addParam( openPosConsts.getGenericConsts().
    TRACKER_CONFIG_SYMBOL_SLICE, symbolSlice.toString() );
config.addParam( openPosConsts.getGenericConsts().
    TRACKER_CONFIG_TRADERID_SLICE, traderSlice.toString() );
```

Working with Position events

The `com.apama.position.Position` event is a general purpose position object that the default position trackers publish. The `Position` event contains the following fields:

```
event Position {
   integer minQtyPosition;
   float minCashPosition;
   integer maxQtyPosition;
   float maxCashPosition;
   dictionary<string, string> extraParams;
   action setFullPosition
      (integer minQty, float minCashPosition,
       integer maxQty, float maxCashPosition);
   action setMaxPosition(integer qty, float cashPosition);
   action setMinPosition(integer qty, float cashPosition);
   action setPosition(integer qty, float cashPosition);
}
```

The table below describes how the default position tracker implementations use the fields in the `Position` event. However, a custom position tracker implementation can use the data fields in the `Position` event in any way, to store any values relevant to your application.

Field	Description
`minQtyPosition` `maxQtyPosition`	These fields contain a cumulative quantity for all orders being tracked. The specific meaning of the value depends on which position tracker is publishing the event. See: ■ "Creating and subscribing to the open position tracker" on page 196 ■ "Creating and subscribing to the pending position tracker" on page 200 ■ "Creating and subscribing to the reserved position tracker" on page 202

Field	Description
	■ "Creating and subscribing to the realized profit and loss tracker" on page 203
`minCashPosition` `maxCashPosition`	These fields contain a cumulative cash position for all orders being tracked. For each order, the quantity is multiplied by the price and then the results for all orders are added together. The specific meaning of the value depends on which position tracker is publishing the event. See: ■ "Creating and subscribing to the open position tracker" on page 196 ■ "Creating and subscribing to the pending position tracker" on page 200 ■ "Creating and subscribing to the reserved position tracker" on page 202 ■ "Creating and subscribing to the realized profit and loss tracker" on page 203
`extraParams`	Any extra information associated with the position being tracked can be stored in this dictionary. Typically, dictionary content is specific to the position tracker that publishes this event.
`setX()` actions	These actions are for setting quantity and cash position values in custom position tracker implementations. Also, you might use these actions if you make a trade outside the system and you want to adjust a position to include that trade. ■ `setFullPosition()` - Sets values for `minQtyPosition`, `maxQtyPosition`, `minCashPosition` and `maxCashPosition`. ■ `setMaxPosition()` - Sets values for `maxQtyPosition` and `maxCashPosition`. ■ `setMinPosition()` - Sets values for `minQtyPosition` and `minCashPosition`. ■ `setPosition()` - Sets `minQtyPosition` and `maxQtyPosition` to the same value. Also sets `minCashPosition` and `maxCashPosition` to the same value.

It is important to define slice details correctly so that a subscription tracks what you want it to track. For example, suppose you specify multiple instruments (perhaps `APMA`, `MSFT`, `TNBT`) in the symbol slice. Even if you use the same currency to trade all of them, the value of the `minQtyPosition` and `maxQtyPosition` fields will be a mix of quantities for three different symbols, which might be meaningless as a value on its own. If you want to know the number of shares traded for just one of these symbols, you must define the slice accordingly with a single symbol.

When you are using the open, pending, or reserved default position trackers, the `minCashPosition` and `maxCashPosition` fields are meaningful when all instruments are traded on a single common currency. If multiple instruments are specified in the symbol slice, and different cash currencies are used for different instruments then the cash position values are not meaningful. For order executions, you can obtain a cash position in a single normalized currency by using the realized profit and loss default position tracker. For pending or reserved orders, you would need to apply custom normalization functions on the position updates to obtain a cash position in a single normalized currency.

Creating and using position service interfaces

To use the position service framework, your application must create an instance of the `com.apama.position.PSInterface` event. This event provides actions that let your application interact with the position service framework.

To create a position service interface object, execute the `create()` action defined in the `com.apama.position.PSFactory` event. The `create()` action parameters specify a reference to the main context, a position service configuration object, and the user-defined action to execute upon creation of the interface object. An instance of `com.apama.position.PSInterface` is returned in the callback.

Note:
In this release, the position service configuration object is not used. You must still specify it but it is reserved for future use.

You use the `PSInterface` object to execute actions for

- "Managing subscriptions" on page 189
- "Managing position trackers" on page 191
- "Managing the position service framework" on page 193

See the ApamaDoc for reference information for these actions.

Code that provides an example of setting up the position service framework is in "Creating and subscribing to the open position tracker" on page 196.

Managing subscriptions

The `PSInterface` object provides actions that let your application manage subscriptions:

- `subscribeAndMonitor()` - This action lets you subscribe to a position tracker and receive updates for the specified position. This is the functional equivalent of the `subscribe()` and the `addUpdateCallback()` actions. This action requires you to specify two callback actions. One callback action is executed when the subscription request is successful. The other callback action is executed when there is a position update.

  By default, this action waits until the tracker is registered and available, or until a configurable timeout duration has expired. Or, you can configure this action to return immediately if the specified position tracker has not been created. For additional information, see the ApamaDoc

for `com.apama.position.Constants`, which defines the constants that indicate the timeout duration and whether to return immediately. The default timeout duration is five seconds.

- `subscribe()` - This action lets you subscribe to a position tracker. You must specify a callback action to be executed when the subscription request is successful.

  By default, this action waits until the tracker is registered and available, or until a configurable timeout duration has expired. Or, you can configure this action to return immediately if the specified position tracker has not been created. For additional information, see the ApamaDoc for `com.apama.position.Constants`, which defines the constants that indicate the timeout duration and whether to return immediately. The default timeout duration is five seconds.

- `unsubscribe()` - This action sends a request to unsubscribe from an existing subscription held by the position service manager. You must specify a callback action to be executed when the unsubscription request is successful.

- `getAllSubscriptionDetails()` - This action allows applications to query all subscriptions made by this position service interface.

- `getSubscriptionDetails()` – This helper action allows applications to retrieve details for a subscription that has been made by this position service interface for the specified subscriptionId.

Code that provides an example of subscribing to a default position tracker is in "Creating and subscribing to the open position tracker" on page 196.

Obtaining subscription information

You can use the position service interface object to obtain information about subscriptions and position trackers. The following code extract provides examples of doing this.

```
action queryDemonstration() {

   // Use the interface object to obtain information for all subscriptions:
   dictionary<integer/*subscriptionId*/,
      com.apama.position.SubscriptionDetails > subscriptions;
   subscriptions := psIface.getAllSubscriptionDetails();

   // Get the details for a specific subscription. You must specify
   // the Id for the subscription. The Id is returned from a successful
   // subscription request.
   integer subscriptionId := 1;
   com.apama.position.SubscriptionDetails details :=
      psIface.getSubscriptionDetails( subscriptionId );

   // Get the current position of a subscription:
   com.apama.position.Position position :=
      psInterface.getCurrentPosition( subscriptionId );

   // Get details about a specific position tracker, and provide a
   // boolean that indicates whether the asynchronous call should
   // wait to respond if the position tracker has not yet been registered.
   psIface.getTrackerInfo( "MyPositionTrackerName",
      com.apama.position.tracker.OpenPositionTrackerConstants.TRACKER_TYPE,
      true, cbTrackerDetails );
    }
```

```
// This action is called when information about the position tracker
// is returned:
action cbTrackerDetails( com.apama.position.TrackerInfo trackerInfo ) {
   log "TRACKER DETAILS: Name="+trackerInfo.name+" :
      Type="+trackerInfo.type+" :
      Context it was created in: "+trackerInfo.ctx.toString();
   log "TRACKER CONFIG SCHEMA: "+trackerInfo.configSchema.toString();
}
```

Managing position trackers

The `PSInterface` object provides actions that let your application manage position trackers:

- `addUpdateCallback()` – This action registers a user-defined callback action to be called every time that a specified subscription's position changes. You can define more than one update callback per subscription.
- `addSymbolSliceUpdateCallback()` — This action registers a user-defined callback action to be called every time that the position changes for a specified symbol slice in a specified subscription. You can define more than one update callback per subscription.
- `removeUpdateCallback()` – This action removes the update callback previously registered for the specified reference Id, which was returned when the callback was added.
- `getCurrentPosition()` - This action returns the current position for the specified subscription
- `setPosition()` – This action allows the application to manually set the specified subscription's position in the tracker to the specified value. You must specify a callback action to be executed when the position update has been made, and to indicate whether or not the operation was successful. This action also returns the new position. Any matching subscriptions receive position updates.
- `setPositionWithConfig()` – This action is a duplicate of the `setPosition()` action but it allows an extra configuration parameter to be specified. For position adjustments, you can specify the `TRACKER_CONFIG_UPDATE_SLICE` configuration parameter. The value of this configuration parameter is of the type `com.apama.position.tracker.generic.SliceKey`.
- `adjustRelativePosition()` - This action allows the application to manually adjust (by a relative amount) the subscription's position in the tracker. This action uses the position object you specify to adjust the current position. For each of the four data fields, `minQtyPosition`, `masQtyPosition`, `minCashPosition`, and `maxCashPosition`, this action adds the value in the specified position object's field to the corresponding current value held by the tracker. If you want to decrease the value of a position, specify a negative number in the appropriate field of the postion object you specify. You must specify a callback action to be executed when the position adjustment has been made, and to indicate whether or not the operation was successful. This action returns the new position. Any matching subscriptions receive position updates.
- `adjustRelativePositionWithConfig()` – This action is a duplicate of the `adjustRelativePosition()` action, but it allows an extra configuration parameter to be specified. For position adjustments, you can specify the `TRACKER_CONFIG_UPDATE_SLICE` configuration parameter. The value of this configuration parameter is of the type `com.apama.position.tracker.generic.SliceKey`.

- `resetPosition()` – This action allows the application to manually set the specified subscription's position to zero in the tracker. This is the equivalent to using the `setPosition()` action with a zero value. A parameter to this action is a callback action that will be executed when the position adjustment has been made, and will indicate whether or not the operation was successful. Any matching subscriptions receive position updates.
- `getTrackerInfo()` - This action returns information about a specific position tracker implementation, including the context that it has been created in. This action can also be configured to wait until the tracker is registered and available before returning.

Updating positions being tracked

In your application, you can change an existing position that is being tracked by a particular subscription. You can set the position to any of the following:

- A value that you specify, including zero
- A value relative to a position you provide

The following code extract provides examples of changing the value of a position. It is assumed that the position service interface object was already created.

```
action modifyPositionInformation( ) {

    // After a subscription has been made, adjust the existing position.
    // First create a new Position object that will be used to
    // change the existing position.
    com.apama.position.Position newPosition := new com.apama.position.Position;

    // To set a single "net" position such as what the open position tracker
    // provides, execute the following action:
    newPosition.setPosition( 10, 1.354 );

    // Or, to set separate min and max values such as what the pending position
    // tracker provides, execute the following action:
    newPosition.setFullPosition( -10, -1.354, 20, 2.641 );

    // Or, to set just the min or max values separately:
    newPosition.setMinPosition( 10, 1.354 );
    newPosition.setMaxPosition( 20, 2.641 );

    // Use the Position object to set a subscription's
    // position to an absolute new position:
    psIface.setPosition( subscriptionId, newPosition, cbPositionAdjusted );

    // Or, adjust the subscription's existing position relative to
    // the position provided:
    psIface.adjustRelativePosition(
       subscriptionId, newPosition, cbPositionAdjusted );

    // Or, set the subscription's existing position to zero:
    psIface.resetPosition( subscriptionId, cbPositionAdjusted );

    // For custom position trackers, or more advanced position changes,
    // you may want to provide to the position tracker more information about
    // the change. For example, why the modification has been made. You
    // can use the following actions to do this:
```

```
    com.apama.position.PositionConfigParams config :=
       new com.apama.position.PositionConfigParams;
    // Add any extra configuration specific to the position tracker:
    psIface.setPositionWithConfig( subscriptionId, newPosition,
       config, cbPositionAdjusted );
    psIface.adjustRelativePositionWithConfig( subscriptionId, newPosition,
       config, cbPositionAdjusted );
}

action cbPositionAdjusted( integer subscriptionId,
   com.apama.position.Position newPosition, boolean success, string msg ) {
    // Check whether the modification was made successfully or not.
    // You are provided with the new updated position.
    ...
}
```

Managing the position service framework

The `PSInterface` object provides actions that let your application obtain the Id for the position service interface you are using and delete position service interface instances:

- `getSubMgrId()` - This action returns the unique identifier for the position service interface object you are using.
- `deleteInterface()` - This action removes any subscriptions and deletes the position service interface object. You must specify a callback action to be executed when the delete request is complete.

The following code extract provides an example of deleting a position service interface object:

```
// This action is called to clean up:
action cleanup() {

    // Alternatively, delete the old one, and create a new one.
    psIface.deleteInterface( cbDeleted );
}

// This action is called in response to the deleteInterface() request:
action cbDeleted( boolean success, string msg ) {
    // Check if the delete was successful, or log an error:
    if( not success ) then {
        log "Delete Failed!!  "+msg to ERROR;
    }
}
```

Using default position trackers

The CMF provides the following default implementations of position trackers:

- The open position tracker tracks actual long and short positions that result from buy or sell order executions.
- The pending position tracker tracks potential long and short positions that could occur if currently open buy and sell orders are filled. This excludes reserved orders.

- The reserved position tracker has the same behavior as the pending position tracker but the reserved position tracker tracks only orders of type `RESERVATION`.
- The realized profit and loss tracker tracks actual profit and loss for order executions and normalizes the values to a base currency.

To use a position tracker, you create it in the context that is receiving orders in your application and then subscribe to it by using the position service interface, which can be in the same context or in any other context. When you create a position tracker you specify a name for it and this name must be unique within your application. A single position tracker can accept any number of subscriptions. One or more position trackers can be in the same context. You can set up as many position trackers as you need as long as each position tracker has a unique name within your application. For example, you might have 50 position trackers with each one in a different context and with the position service interface object in some other context.

Each default position tracker you create listens for orders (using Order Management Service events) in the context in which the tracker was created. The default behavior is as follows:

Tracker Instance Type	Matches On All
Open position	Order executions
Pending position	Unfilled orders (not including reserved orders)
Reserved position	Reserved orders
Realized profit and loss	Order executions

To specify a set of criteria to match orders against, add slice information to the configuration you specify when you subscribe to a tracker.

Configuring default tracker subscriptions

When you subscribe to a default tracker implementation, you specify a configuration object, `com.apama.position.PositionConfigParams`, which is a dictionary of parameter names and values.

Configuration options that are common to all default position trackers are defined by the constants in `com.apama.position.tracker.GenericPositionTrackerConstants`. Configuration options that are specific to a default position tracker are available in `com.apama.position.tracker.`*`xxx`*`TrackerConstants`. The `com.apama.position.tracker.`*`xxx`*`TrackerConstants` events each define the `getGenericConsts()` action, which gets the set of configuration constants, including the common constants, for the corresponding type of default position tracker.

To modify a default position tracker's configuration, call the `addParam()` action on the tracker's `com.apama.position.PositionConfigParam` object.

The set of common configuration parameters includes, but is not limited to:

Configuration Parameter	Description
"SYMBOL_SLICE"	Match against a sequence of symbols that you specify. If you do not specify this slice then all symbols are matched.
"SERVICEID_SLICE"	Match against a sequence of service Ids that you specify. If you do not specify this slice then all service Ids are matched.
"MARKETID_SLICE"	Match against a sequence of market Ids that you specify. If you do not specify this slice then all market Ids are matched.
"EXCHANGEID_SLICE"	Match against a sequence of exchange Ids that you specify. If you do not specify this slice then all exchange Ids are matched.
"TRADERID_SLICE"	Match against a sequence of trader Ids/owner Ids that you specify. If you do not specify this slice then all trader Ids/owner Ids are matched.
"USE_FIREWALL"	Set this parameter to `true` to enable matching of orders being placed through the CMF risk firewall. This parameter is set to `false` by default.
"ADD_ORDER_DETAILS"	Set this parameter to `true` to add information about the last trade to the `extraParams` dictionary field in the `Position` event published for this subscription. This parameter is set to `false` by default.
"SUBSCRIPTION_NAME"	This configuration parameter lets you specify a unique name (within your application) for a subscription. The position tracker then handles that subscription independently of any other subscriptions that have the same configuration. For example, you might want to monitor a trading algorithm's position separately from a firewall rule position. See "Implementing independent default position trackers" on page 207.
"TRACK_SEPARATE_CURRENCIES"	Indicates that you want to also track FX currency positions as separate currencies. For example, a symbol slice of `EUR/USD` and `GBP/USD` will also track the positions of `EUR`, `USD`, and `GBP` separately.
"CURRENCY_SEPARATOR"	This configuration parameter lets you specify the currency separator to use when splitting symbols. The default, `/`, is specified by the "DEFAULT_CURRENCY_SEPARATOR" parameter.

Creating and subscribing to the open position tracker

The default implementation of the open position tracker tracks the quantity and cash position of order executions. To use the open position tracker, you create an instance of it in the context in which your application receives orders. You then use the position service interface object to subscribe to the open position tracker you created.

To create an instance of an open position tracker, execute the `create()` action defined in `com.apama.position.tracker.OpenPositionTrackerFactory`. This action takes three parameters:

Parameter	Description
`mainContext`	A reference to the main context. When the Position Management Service bundle is in your application the correlator automatically sets up the position service manager in the main context.
`trackerName`	A string that specifies a name for this tracker. This name must be unique in your application.
`cbCreated`	This callback action returns a `boolean` parameter that indicates whether creation of the tracker was successful, followed by a `string` parameter that can hold a message that can be logged.

After an open position tracker is set up, subscribe to it by calling the `subscribeAndMonitor()` action or the `subscribe()` action, which are both defined in `com.apama.position.PSInterface`. Both actions subscribe to the specified tracker. The `subscribeAndMonitor()` action also registers an update callback each time there is an update to the position being tracked. A tracker can accept any number of subscriptions. The `subscribeAndMonitor()` action takes the following parameters:

- `trackerName` is the name of the open position tracker to subscribe to. This is the unique name that was specified when the tracker was created.
- `trackerType` indicates the type of tracker you are subscribing to. When subscribing to an instance of the default open position tracker, specify the tracker type as the constant `TRACKER_TYPE` string defined in the `com.apama.position.tracker.OpenPositionTrackerConstants` event.
- `config` is a `com.apama.position.PositionConfigParams` object that contains any configuration for this subscription, such as the slice criteria you want to match orders against.
- `updateCallback` is a user-defined update callback action that the tracker will call when there is an update to the position subscribed to.
- `subscribedCb` is a user-defined callback action that the tracker calls in response to the subscription request.

When there is a change in the position being tracked, the update callback (or callbacks) that have been registered for the subscription will be called. This callback action contains a `com.apama.position.Position` event object that defines the current position for that subscription. By default, the open position tracker calculates net values for the combined long and short quantity traded and cash position:

Field	Description
`minQtyPosition` `maxQtyPosition`	Each of these fields contain the same value. The value indicates the net quantity for all long and short executed orders that match the subscription configuration. A negative number indicates a short position.
`minCashPosition` `maxCashPosition`	Each of these fields contain the same value. The value indicates the net cash position, which is the sum of (quantity times price) for each executed order.

For example, if you bought 12 and sold 5 the `minQtyPosition` and `maxQtyPosition` fields each contain `7` and so you are long 7. If you bought 5 and sold 12 the `minQtyPosition` and `maxQtyPosition` fields each contain `-7` and so you are short 7.

If you want to separately track open long and short positions specify the `"TRACKER_CONFIG_TRACK_SEPARATE_LONG_SHORT"` constant in the `PositionConfigParams` object you specify when you call the `subscribeAndMonitor()` action or the `subscribe()` action. If this parameter is set, then the `Position` event field values indicate the following:

Field	Description
`minQtyPosition`	Sum of the quantities of all executed short orders. This is always a negative number.
`maxQtyPosition`	Sum of the quantities of all executed long orders. This is always a positive number.
`minCashPosition`	Sum of (quantity times price) for all executed short orders.
`maxCashPosition`	Sum of (quantity times price) for all executed long orders.

The following code shows an example of creating two open position trackers in two different contexts.

```
monitor TrackerMonitor {

    context mainContext := context.current();

    // Spawning to a new context to show that the position
    // service framework can be used in multiple contexts.
    action onload() {
        // Create an instance of the open position tracker and
        // give it a unique name to be used when subscribing to it.
        (new com.apama.position.tracker.OpenPositionTrackerFactory).create
           ( mainContext, "MyFirstOpenPositionTracker", cbCreatedTracker );

        context newCtx := context("TESTCTX", false);
        spawn startTest() to newCtx;
    }

    action startTest() {
        // Create another instance of the open position tracker and
        // give it a unique name to be used when subscribing to it.
```

```
        (new com.apama.position.tracker.OpenPositionTrackerFactory).create
            (mainContext, "MySecondOpenPositionTracker", cbCreatedTracker );
    }

    // This action is called after the position tracker instance
    // has been created
    action cbCreatedTracker( boolean success, string msg ) {
        log "OPEN POSITION TRACKER CREATED: "+success.toString()+" : "+msg;
    }
}
```

The following code shows an example of creating a position service interface and then subscribing to the open position trackers created in the previous sample.

```
monitor PositionServicesDemo {
    context mainContext := context.current();

    // To set up the position service framework, you create an
    // instance of PSInterface, which provides the user interface.
    com.apama.position.PSInterface psIface;

     // A constant string that will be used:
    com.apama.position.tracker.OpenPositionTrackerConstants openPosConsts;

    action onload() {

        // Create a position service configuration object.
        // In this case, with no specific configuration.
        // Then create an instance of PSInterface:
        com.apama.position.PositionConfigParams config :=
           new com.apama.position.PositionConfigParams;
        (new com.apama.position.PSFactory).create
           ( mainContext, config, cbCreated );
    }

    action cbCreated( com.apama.position.PSInterface psInterface,
           boolean success, string msg ) {
        psIface := psInterface;

        // After creating the interface object, subscribe to trackers.
        // Create a set of slice details to track the positions for.
        // In this case, slice on symbols and traders:
        sequence<string> symbolSlice := ["APMA","MSFT" ];
        sequence<string> traderSlice := ["ggekko"];
        com.apama.position.PositionConfigParams config :=
           new com.apama.position.PositionConfigParams;
        config.addParam(openPosConsts.getGenericConsts().
           TRACKER_CONFIG_SYMBOL_SLICE,   symbolSlice.toString() );
        config.addParam(openPosConsts.getGenericConsts().
           TRACKER_CONFIG_TRADERID_SLICE, traderSlice.toString() );

        // Subscribe to the open position tracker you previously created:
        psIface.subscribeAndMonitor( "MyFirstOpenPositionTracker",
           openPosConsts.TRACKER_TYPE, config, positionChanged,
           cbSubscribedWithCallback );

        // OR, if you do not want to register an update callback,
        // call subscribe().
        psIface.subscribe( "MySecondOpenPositionTracker",
           openPosConsts.TRACKER_TYPE, config, cbSubscribed );
```

```
    }

    // This callback action confirms a subscription has been made.
    // It provides the details for the original subscription,
    // a boolean indicating success/failure and an optional message
    // field (primarily used for error strings). Also, this is
    // the callback action registered for the subscribeAndMonitor()
    // action. You do not need to write an update callback action.
    // This action contains the same parameters as the
    // cbSubscribed() callback action, but with an extra reference
    // Id so you can remove an update callback if required later.
    action cbSubscribedWithCallback(
        integer subscriptionId,
        com.apama.position.SubscriptionDetails subscriptionDetails,
        integer updateCallbackRefId, boolean success, string msg ) {
        ...
    }

    // This callback action is called whenever a position changes.
    // It provides the subscriptionID that has been updated, and
    // the new position for that subscription.
    action positionChanged(
        integer subscriptionId, com.apama.position.Position newPosition ) {
        // Do something now that the position has been updated
        ...
    }

    // This callback action confirms a subscription has been made.
    // It provides the details for the original subscription,
    // a boolean indicating success/failure and an optional message
    // field (primarily used for error strings).
    action cbSubscribed( integer subscriptionId,
              com.apama.position.SubscriptionDetails subscriptionDetails,
              boolean success, string msg ) {
        // A user-defined update callback can be added to obtain updates for
        // position changes. Typically, this needs to be done for custom
        // trackers. The action returns a unique ID so that this specific
        // update callback can be removed at a later date if required.
        integer cbRefId :=
           psIface.addUpdateCallback( subscriptionId, positionChanged );
    }
}
```

The following code provides an example of separately tracking currencies.

```
using com.apama.position.PSInterface;
using com.apama.position.PositionConfigParams;
using com.apama.position.SubscriptionDetails;

monitor TrackCurrenciesSeparately {

   ...
   action cbCreated(PSInterface psInterface, boolean success, string msg )  {
      PositionConfigParams trackerConfig := new PositionConfigParams;
      sequence<string> symbolSlice := [ "EUR/USD", "EUR/JPY" ];
      trackerConfig.addParam( openPosConsts.getGenericConsts()
         .TRACKER_CONFIG_SYMBOL_SLICE, symbolSlice.toString() );
      trackerConfig.addParam( openPosConsts.getGenericConsts()
         .TRACKER_CONFIG_TRACK_SEPARATE_CURRENCIES, true );
```

```
      // Subscribe to the open position tracker.
      // This will also track EUR, USD and JPY separately.
   psInterface.subscribe( "MyOpenPositionTracker", openPosConsts.TRACKER_TYPE,
      trackerConfig, cbSubscribed );
   }

   action cbSubscribed( integer subId, SubscriptionDetails subDetails,
      boolean success, string msg ) {
      // Subscribe just to the EUR slice updates
      psInterface.addSymbolSliceUpdateCallback(subId, "EUR", callback);
   }
   ...
}
```

Creating and subscribing to the pending position tracker

The default implementation of the pending position tracker tracks the cumulative quantity and cash position of unfilled orders that are not of the reserved type. These unfilled orders represent how much your open position for the same slice information could change if those orders complete. To use the pending position tracker, you create an instance of it and then you subscribe to it by using a position service interface object (`com.apama.position.PSInterface`).

To create an instance of a pending position tracker, execute the `create()` action defined in `com.apama.position.tracker.PendingPositionTrackerFactory`. The `create()` action takes three parameters:

Parameter	Description
`mainContext`	A reference to the main context. When the Position Management Service bundle is in your application the correlator automatically sets up the position service manager in the main context.
`trackerName`	A string that specifies a name for this tracker. This name must be unique in your application.
`cbCreated`	This callback action returns a `boolean` parameter that indicates whether creation of the tracker was successful, followed by a `string` parameter that can hold a message that can be logged.

After a pending position tracker is set up, subscribe to it by calling the `subscribeAndMonitor()` action or the `subscribe()` action, which are both defined in `com.apama.position.PSInterface`. Both actions subscribe to the specified tracker. The `subscribeAndMonitor()` action also registers an update callback each time there is an update to the position being tracked. A tracker can accept any number of subscriptions. The `subscribeAndMonitor()` action takes the following parameters:

- `trackerName` is the name of the pending position tracker to subscribe to. This is the unique name that was specified when the tracker was created.
- `trackerType` indicates the type of tracker you are subscribing to. When subscribing to an instance of the default pending position tracker, specify the tracker type as the constant `TRACKER_TYPE` string defined in the `com.apama.position.tracker.PendingPositionTrackerConstants` event.

- `config` is a `com.apama.position.PositionConfigParams` object that contains any configuration for this subscription, such as the slice criteria you want to match orders against.
- `updateCallback` is a user-defined update callback action that the tracker will call when there is an update to the position subscribed to.
- `subscribedCb` is a user-defined callback action that the tracker calls in response to the subscription request.

Code for creating a pending position tracker and then subscribing to it is almost the same as the code for creating and subscribing to an open position tracker. The differences are that you use the `PendingPositionTrackerFactory.create()` action and when subscribing you use the `TRACKER_TYPE` string defined in the `PendingPositionTrackerConstants` event. See the sample code in "Creating and subscribing to the open position tracker" on page 196.

When there is a change in the position being tracked, the update callback (or callbacks) that have been registered for the subscription will be called. This callback action contains a `com.apama.position.Position` event object that defines the current position for that subscription. The `Position` event field values indicate the following:

Field	Description
`minQtyPosition`	Sum of the quantities of all unfilled short orders. This is the quantity that would be sold if all unfilled sell orders for this position were executed. This value indicates how much shorter the position would become if all orders filled. This is always a negative number.
`maxQtyPosition`	Sum of the quantities of all unfilled long orders. This is the quantity that would be bought if all unfilled buy orders for this position were executed. This value indicates how much longer the position would become if all orders filled. This is always a positive number.
`minCashPosition`	Sum of (quantity times price) for all unfilled short orders. This is the amount you would be short if all unfilled sell orders for this position were executed. This value indicates how much shorter the cash position would become if all orders filled. The pending position tracker calculates cash position based on the most recent filled order for the position being tracked.
`maxCashPosition`	Sum of (quantity times price) for all unfilled long orders. This is the amount you would be long if all unfilled buy orders for this position were executed. This value indicates how much longer the cash position would become if all orders filled. The pending position tracker calculates cash position based on the most recent filled order for the position being tracked.

A `Position` event's `min` and `max` fields let you separately track short positions and long positions. If you are interested in a net value instead of separate long and short values add the appropriate `min` and `max` event field values together.

The following table shows how the quantity for corresponding open and pending positions might change.

Order State	Open Position	Pending `minQtyPosition`	Pending `maxQtyPosition`
Start	`+5`	`0`	`0`
Buy `6` order placed	`+5`	`0`	`6`
Sell `15` order placed	`+5`	`-15`	`6`
Buy `6` order filled	`+11`	`-15`	`0`
Sell `15` order filled	`-4`	`0`	`0`

Creating and subscribing to the reserved position tracker

The default implementation of the reserved position tracker tracks the cumulative quantity and cash position for reserved orders for a particular slice. A reserved order lets an application request the right to take a position up to a specified quantity. Reservations are designed but not limited to be used with the CMF's risk firewall. Reservations limit the total order exposure while still allowing an application's strategy the freedom to trade with fewer concerns about reaching risk limits. The reserved position tracker does not grant or enforce reservations. It only lets applications monitor changes to the reserved position for a particular slice. You should use reserved position trackers with the Reservation Enforcer Risk Firewall rule.

The behavior of the reserved position tracker is the same as the behavior of the pending position tracker except that the reserved position tracker tracks only orders of the type `RESERVATION`. Reserved orders represent how much the cumulative quantity and cash position of your open position for the same slice could change.

To use the reserved position tracker, you create an instance of it and then you subscribe to it.

To create an instance of a reserved position tracker, execute the `com.apama.position.tracker.ReservedPositionTrackerFactory.create()` action. For a description of the parameters you specify, see "Creating and subscribing to the pending position tracker" on page 200.

After a reserved position tracker is set up, subscribe to it by executing `com.apama.position.PSInterface.subscribeAndMonitor()`. This action subscribes to the specified tracker and registers an update callback each time there is an update to the position being tracked. A tracker can accept any number of subscriptions.

When an order is placed against a reservation, the order quantity is subtracted from the reserved position and added to the pending position. If the order subsequently fills, the filled quantity moves from the pending position to the open position. If the order is canceled, or amended downwards, the quantity is moved from the pending position back to the reserved position. For example, if a trader has unfilled buy orders with a type of `RESERVATION` in the market with a total quantity of 1000, then there will be a reserved `maxQtyPosition` of `1000`. This is the maximum that the trader's position can change if all reserved buy orders were filled. Conversely, if a trader has

unfilled sell orders with a type of `RESERVATION` in the market with a total quantity of 1000, then there will be a reserved `minQtyPosition` of `-1000`.

Code for creating a reserved position tracker and then subscribing to it is almost the same as the code for creating and subscribing to an open position tracker. The differences are that you use the `ReservedPositionTrackerFactory.create()` action and when subscribing you use the `TRACKER_TYPE` string defined in the `com.apama.position.tracker.ReservedPositionTrackerConstants` event. See the sample code in "Creating and subscribing to the open position tracker" on page 196.

Creating and subscribing to the realized profit and loss tracker

The default implementation of the realized profit and loss tracker tracks actual profit and loss resulting from order executions and normalized to a specified currency. The order executions match the slice criteria provided in the subscription configuration. The profit and loss values are normalized to a currency that is specified when the tracker is created.

To use the realized profit and loss tracker, you create an instance of it in the context in which your application receives orders. You then use the position service interface object to subscribe to the realized profit and loss tracker you created.

To create an instance of a realized profit and loss tracker, execute the `create()` action defined in `com.apama.position.tracker.RealizedPnLTrackerFactory`. This action takes the following parameters:

Parameter	Description
`mainContext`	A reference to the main context. When the Position Management Service bundle is in your application the correlator automatically sets up the position service manager in the main context.
`trackerName`	A string that specifies a name for this tracker. This name must be unique in your application.
`currencyConverter`	The currency converter interface to use to obtain values needed to normalize profit and loss values. See "Creating and configuring a currency converter" on page 216.
`cbCreated`	This callback action returns a `boolean` parameter that indicates whether creation of the tracker was successful, followed by a `string` parameter that can hold a message that can be logged.

After a realized profit and loss tracker is set up, subscribe to it by calling the `subscribeAndMonitor()` action or the `subscribe()` action, which are both defined in `com.apama.position.PSInterface`. Both actions subscribe to the specified tracker. The `subscribeAndMonitor()` action also registers an update callback each time there is an update to the position being tracked. A tracker can accept any number of subscriptions. The `subscribeAndMonitor()` action takes the following parameters:

- `trackerName` is the name of the realized profit and loss tracker to subscribe to. This is the unique name that was specified when the tracker was created.

- `trackerType` indicates the type of tracker you are subscribing to. When subscribing to an instance of the default realized profit and loss tracker, specify the tracker type as the constant `TRACKER_TYPE` string defined in the `com.apama.position.tracker.RealizedPnLTrackerConstants` event.
- `config` is a `com.apama.position.PositionConfigParams` object that contains the configuration for this subscription, such as the slice criteria you want to match orders against.

  You must specify a value for the `TRACKER_CONFIG_SLICE_CURRENCY` parameter, which is defined in `RealizedPnLTrackerConstants`. The `TRACKER_CONFIG_SLICE_CURRENCY` parameter specifies the currency this slice is traded in. The realized profit and loss tracker will convert values in the specified currency to U. S. dollars. To convert values to a different currency, specify a value for the `TRACKER_CONFIG_NORMALIZE_CURRENCY` parameter.

  If you do not set a value for the `TRACKER_CONFIG_SLICE_CURRENCY` parameter then the behavior of the realized profit and loss tracker is the same as it would be for the default open position tracker.
- `updateCallback` is a user-defined update callback action that the tracker will call when there is an update to the position subscribed to.
- `subscribedCb` is a user-defined callback action that the tracker calls in response to the subscription request.

When there is a change in the position being tracked, the update callback (or callbacks) that have been registered for the subscription will be called. This callback action contains a `com.apama.position.Position` event object that defines the current position for that subscription. By default, the realized profit and loss tracker calculates net values for the combined long and short quantity traded and cash position:

Field	Description
`minQtyPosition` `maxQtyPosition`	Each of these fields contain the same value. The value indicates the net quantity for all long and short executed orders that match the subscription configuration. A negative number indicates a short position.
`minCashPosition` `maxCashPosition`	Each of these fields contain the same value. The value indicates the net cash position in the specified normalized currency. In other words, this is the sum of (quantity times price) for each executed order normalized with the exchange rate at the time of execution.

For example, if you bought 12 and sold 5 the `minQtyPosition` and `maxQtyPosition` fields each contain `7` and so you are long 7. If you bought 5 and sold 12 the `minQtyPosition` and `maxQtyPosition` fields each contain `-7` and so you are short 7.

If you want to separately track open long and short positions specify the `"TRACKER_CONFIG_TRACK_SEPARATE_LONG_SHORT"` constant in the `PositionConfigParams` object you specify when you call the `subscribeAndMonitor()` action or the `subscribe()` action. If this parameter is set then the `Position` event field values indicate the following:

Field	Description
`minQtyPosition`	Sum of the quantities of all executed short orders. This is always a negative number.
`maxQtyPosition`	Sum of the quantities of all executed long orders. This is always a positive number.
`minCashPosition`	Sum of (quantity times price) for all executed short orders in the specified normalized currency.
`maxCashPosition`	Sum of (quantity times price) for all executed long orders in the specified normalized currency.

The following code shows an example of creating two realized profit and loss trackers in two different contexts.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.ccyconverter.CurrencyConverterFactory;
using com.apama.ccyconverter.CurrencyConverter;
using com.apama.cmf.sample.BBAMidPriceExtension;
using.com.apama.position.tracker.RealizedPnLTrackerFactory;
using com.apama.cmf.sample.BBAMidPriceExtension;
monitor TrackerMonitor {
   context mainContext := context.current();

   // Spawning to a new context to show that the position
   // service framework can be used in multiple contexts.
   action onload() {

      // Create a session handler for the currency converter to use.
      SessionHandler sessionHandler := (new SessionHandlerFactory).connect(
         mainContext, "MySession", "MyTransport" );

      // Create a currency converter connected to the session handler,
      // for the realized profit and loss tracker to get FX rates from.
      CurrencyConverter currencyConverter :=
         (new CurrencyConverterFactory).create( mainContext,
         "MyCurrencyConverter",
         (new BBAMidPriceExtension).create(sessionHandler) );

      // Create an instance of the realized profit and loss tracker,
      // give it a unique name to be used when subscribing to it,
      // and specify the currency converter to get FX rates from.
      (new RealizedPnLTrackerFactory).create(mainContext,
         "MyFirstRealizedPnLTracker", currencyConverter, cbCreatedTracker );

      context newCtx := context("TESTCTX", false);
      spawn startTest() to newCtx;
   }

   action startTest() {

      // Connect remotely to the currency converter created in onload()
      CurrencyConverter currencyConverter := (CurrencyConverterFactory).connect(
         mainContext, "MyCurrencyConverter" );
```

```
      // Create another instance of the realized profit and loss tracker,
      // give it a unique name to be used when subscribing to it,
      // and specify the currency converter to get FX rates from.
      (new RealizedPnLTrackerFactory).create(mainContext,
         "MySecondRealizedPnLTracker", cbCreatedTracker, currencyConverter );
   }

   // This action is called after the position tracker instance
   // has been created.
   action cbCreatedTracker( boolean success, string msg ) {
      log "REALIZED PnL TRACKER CREATED: "+success.toString()+" : "+msg;
   }
}
```

The following code shows an example of creating a position service interface and then subscribing to the realized profit and loss trackers created in the previous sample.

```
using com.apama.position.PSInterface;
using com.apama.position.tracker.RealizedPnLTrackerConstants;
using com.apama.position.PositionConfigParams;
using com.apama.position.PSFactory;
using com.apama.position.Position;
using com.apama.position.SubscriptionDetails;
monitor PositionServicesDemo {
   context mainContext := context.current();

   // To set up the position service framework, you create an
   // instance of PSInterface, which provides the user interface.
   PSInterface psIface;

   // A constant string that will be used:
   RealizedPnLTrackerConstants realizedPnLConsts;

   action onload() {

      // Create a position service configuration object.
      // In this case, with no specific configuration.
      // Then create an instance of PSInterface:
      PositionConfigParams config := new PositionConfigParams;
      (new PSFactory).create( mainContext, config, cbCreated );
   }

   action cbCreated( PSInterface psInterface, boolean success, string msg ) {
      psIface := psInterface;

      // After creating the interface object, subscribe to trackers.
      // Create a set of slice details to track the positions for.
      // In this case, slice on symbols and traders:
      sequence>string> symbolSlice := ["APMA", "MSFT" ];
      sequence>string> traderSlice := ["ggekko"];

      PositionConfigParams config := new PositionConfigParams;
      config.addParam(realizedPnLConsts.getGenericConsts().
         TRACKER_CONFIG_SYMBOL_SLICE, symbolSlice.toString() );
      config.addParam(realizedPnLConsts.getGenericConsts().
         TRACKER_CONFIG_TRADERID_SLICE, traderSlice.toString() );

      // Specify that the slice currency is US Dollars:
      config.addParam(realizedPnLConsts.getGenericConsts().
```

```
         TRACKER_CONFIG_SLICE_CURRENCY, "USD" );

      // And that the cash positions should to be normalized to Euros:
      config.addParam(realizedPnLConsts.getGenericConsts().
         TRACKER_CONFIG_NORMALIZE_CURRENCY, "EUR" );

      // Subscribe to the realized profit and loss tracker you previously
      // created:
      psIface.subscribeAndMonitor( "MyFirstRealizedPnLTracker",
         realizedPnLConsts.TRACKER_TYPE, config, positionChanged,
         cbSubscribedWithCallback );

      // OR, if you do not want to register an update callback,
      // call subscribe().
      psIface.subscribe( "MySecondRealizedPnLTracker",
         realizedPnLConsts.TRACKER_TYPE, config, cbSubscribed );
   }

   // This callback action confirms a subscription has been made.
   // It provides the details for the original subscription,
   // a boolean indicating success/failure and an optional message
   // field (primarily used for error strings). Also, this is
   // the callback action registered for the subscribeAndMonitor()
   // action. You do not need to write an update callback action.
   // This action contains the same parameters as the
   // cbSubscribed() callback action, but with an extra reference
   // Id so you can remove an update callback if required later.
   action cbSubscribedWithCallback(
      integer subscriptionId,
      SubscriptionDetails subscriptionDetails,
      integer updateCallbackRefId, boolean success, string msg ) {
      ...
   }

   // This callback action is called whenever a position changes.
   // It provides the subscriptionID that has been updated, and
   // the new position for that subscription.
   action positionChanged(
      integer subscriptionId, Position newPosition ) {
      // Do something now that the position has been updated
      ...
   }
   // This callback action confirms a subscription has been made.
   // It provides the details for the original subscription,
   // a boolean indicating success/failure and an optional message
   // field (primarily used for error strings).
   action cbSubscribed( integer subscriptionId,
      SubscriptionDetails subscriptionDetails, boolean success, string msg ) {

      // A user-defined update callback can be added to obtain updates for
      // position changes. Typically, this needs to be done for custom
      // trackers. The action returns a unique ID so that this specific
      // update callback can be removed at a later date if required.
      integer cbRefId :=
         psIface.addUpdateCallback( subscriptionId, positionChanged );
   }
}
```

Implementing independent default position trackers

In your application, it is possible to have duplicate subscriptions. That is, there can be multiple subscriptions to the same named default position tracker with the same configuration details. For example, you might have 10 duplicate subscriptions in 10 different contexts. By default, the position service framework uses a single position tracker instance for duplicate subscriptions.

However, different parts of an application might require independent position trackers. For example, you might need an open position tracker for client orders, and another open position tracker for internal orders. Or you might be writing a trading algorithm and you want to reset or modify a position independently of a position that is using the risk firewall and that you settle at zero at the end of each day.

To implement an independent default position tracker, specify the `com.apama.position.tracker.GenericPositionTrackerConstants.TRACKER_CONFIG_SUBSCRIPTION_NAME` constant value in the configuration object you specify when you subscribe to the position tracker. This configuration parameter lets you specify a unique name (within your application) for a subscription. The position service framework creates a tracker instance to handle that subscription independently of any other subscriptions that have the same configuration. For example:

```
com.apama.position.tracker.GenericPositionTrackerConstants consts :=
   new com.apama.position.tracker.GenericPositionTrackerConstants;

   com.apama.position.PositionConfigParams config :=
      new com.apama.position.PositionConfigParams;

   config.addParam( consts.TRACKER_CONFIG_SUBSCRIPTION_NAME,
      "MyIndependentSubscription" );
```

Now if you specify `config` in a subscription, you get a new instance of the position tracker with a new subscription based on the unique configuration. If you subscribe again using the same configuration, which includes the same subscription name, the configurations match. So that same position tracker is used for this second, matching, subscription. To set up an independent position tracker for a single subscription you must specify a unique name for each subscription.

Implementing custom position trackers

For applications that have requirements that are not met by the default implementations of the position trackers, you can implement custom position trackers. The position service framework event interfaces you use to create a custom position tracker are defined in the `com.apama.position.tracker` package and described here:

- `PSTrackerManagerInterface` defines actions that a custom position tracker uses to update and publish positions it is tracking. There are also actions for providing information about the tracker instance and its subscriptions.
- `PSTrackerFactory` defines actions that a custom position tracker uses to register itself with the position service manager.
- `PSTrackerInterface` defines actions that every custom position tracker must implement.

A custom position tracker can publish a custom event instead of or in addition to the standard `Position` event. If a custom position tracker publishes the standard `Position` event, it can use the quantity and cash fields for any values needed to be stored. For example, you can use the

`minQtyPosition` and `minCashPosition` fields to store a single net position, and ignore the `maxQtyPosition` and `maxCashPosition` fields. In fact, you can use the quantity and cash fields to store something else entirely. There are simply two `integer` and two `float` fields available to use.

To implement a custom position tracker, the Position Management Service bundle must be in your application.

For reference information, see the ApamaDoc.

Steps for implementing a custom position tracker

To implement a custom position tracker, the basic set of operations that an application must perform are as follows:

- Create an instance of `PSTrackerInterface`. In your code, use this instance to implement required actions: `createInstance()`, `deleteAllInstances()`, `deleteInstance()`, `positionExternallyReset()`, `positionExternallyUpdated()`. At runtime, the position service manager will use this interface object to communicate with your custom position tracker. See "Actions custom trackers must implement" on page 209.
- Use `PSTrackerFactory` to register your custom tracker with the position service manager. When you register your custom position tracker one of the arguments you specify is the `PSInterface` object that you used to implement the required actions. Registering your custom position tracker returns an instance of `PSTrackerManagerInterface`. Your custom position tracker uses this object to communicate with the posistion service manager. See "Registering custom position trackers" on page 210.
- Create an instance of `PSTrackerManagerInterface`. In your code, use this instance to update and publish positions being tracked. See "Actions custom trackers use to manage positions" on page 211.

After you successfully register a custom position tracker, your application can subscribe to it by creating an instance of `PSInterface` and calling one of the subscribe actions in the same way as when using a default position tracker. When a new subscription to your custom tracker is requested the position service manager calls the `createInstance()` action that you defined in the `PSTrackerInterface` object that you specified when you registered your custom tracker.

With a subscription to your custom tracker in place, your application can execute the same `PSInterface` actions it can execute when it uses one of the default position trackers.

See also "Sample code for implementing a custom position tracker" on page 212.

Actions custom trackers must implement

In your custom position tracker, you must create an instance of `PSTrackerInterface` and use it to implement the following actions. When you register your custom position tracker with the position service manager, you pass this interface object as one of the arguments. The position service manager will use this object to communicate with your custom position tracker. See ApamaDoc for details about the parameters required by each of these actions.

The actions your custom tracker must implement are:

- `createInstance()` – The position service manager calls this action for every new subscription request. Your custom position tracker must determine whether this subscription is the same as an existing subscription. If there is no matching subscription then your application must create a new instance of your custom tracker and return the subscription Id for the new tracker instance. If there is a matching subscription then your custom tracker can use the existing instance of the tracker and so return the Id for the existing matching subscription. This lets your tracker use the same tracker instance for multiple subscriptions. After creating a new tracker instance or determining that a suitable one already exists, your custom tracker must call the completion callback action, which provides the subscription Id and an indication of success or failure. The mechanism and criteria to determine whether a requested subscription is a duplicate of an existing subscription is user-defined.
- `deleteInstance()` – The position service manager calls this action when it receives an unsubscribe request and there is only one subscription remaining for that tracker instance. Your custom tracker must delete the specified custom tracker instance and then call the completion callback action, which indicates success or failure.
- `deleteAllInstances()` – The position service manager calls this action when deletion of all custom tracker instances is required. For example, your application is shutting down and you want to clean up state information held by each custom tracker instance. Your custom tracker must delete all tracker instances and then call the completion callback action, which indicates success or failure.
- `positionExternallyReset()` - The position service manager calls this action when there is a request to manually set the position being tracked to zero. Your custom tracker must set the specified subscription's position to zero and call the completion callback action, which indicates success or failure.
- `positionExternallyUpdated()` - The position service manager calls this action when there is a request to manually adjust the position being tracked by a relative or absolute amount. Parameters to this action include the amount of the position change and whether the position should be set to this amount (absolute) or this amount should be added to the current position (relative). There is also a parameter that can contain any additional configuration required to adjust the position. Your custom tracker must update the particular subscription's position as specified and call the completion callback action, which indicates success or failure.

See also "Sample code for implementing a custom position tracker" on page 212.

Registering custom position trackers

In your custom tracker, use the `com.apama.position.tracker.PSTrackerFactory` event to register your custom tracker with the position service manager. There are two actions available for registering your tracker: `registerTracker()` and `registerTrackerWithConfig()`.

Each action

- Registers your custom tracker with the position service manager. The position service manager will use the provided `PSTrackerInterface` object to communicate with your custom tracker.

- Creates a new position tracker manager interface object (`com.apama.position.tracker.PSTrackerManagerInterface`) and returns it to your custom tracker. Your custom tracker uses this object to communicate with the position service manager.

In addition, the `registerTrackerWithConfig()` action specifies an additional callback action that you can use when your custom tracker requires initial configuration. For example, if your custom tracker has its own persistence mechanism. The position service manager calls this action before it finishes registration of your custom tracker.

The arguments to the registration actions are as follows:

Argument	Description
`psTrackerName`	A name for your custom tracker. The name must be unique in your application.
`psTrackerType`	The type of your tracker. This is application-dependent and must be unique in your application.
`mainContext`	A reference to the main context.
`configSchema`	An instance of `com.apama.position.PositionConfigSchema`, which defines the configuration parameters that can be set when subscribing to this tracker.
`trackerCbIface`	An instance of `com.apama.position.tracker.PSTrackerInterface` that defines the set of callback actions the custom position tracker is registering. These are the actions that your custom tracker is required to implement plus any other actions required by your application. See "Actions custom trackers must implement" on page 209.
`cbOnConfig`	A callback action that will be called before registration is complete. Use this action to perform any initial configuration, for example, if you want to configure a particular persistence mechanism. You specify this argument only for the `registerTrackerWithConfig()` action.
`cbRegistered`	A callback action that will be called after registration of a custom tracker. This action returns an instance of `PSTrackerManagerInterface`, a boolean value that indicates success or failure, and a `string` message that you define and that is provided back to you upon registration. A typical use of this string is to provide information if the registration failed, for example, if you try to register a tracker with the same `trackerType` as a tracker already registered.

See also "Sample code for implementing a custom position tracker" on page 212.

Actions custom trackers use to manage positions

After the position service manager registers your custom tracker, it returns an instance of `PSTrackerManagerInterface`. This object provides actions your custom tracker calls to do the following:

- Manually adjust a subscription's position.
  - `updateAbsolutePosition()` – Updates the current position to an absolute value and publishes the position to all subscribers.
  - `updateRelativePosition()` – Updates the current position by a relative value and publishes the position to all subscribers.
  - `updateAbsolutePositionWithoutPublication()` – Updates the current position to an absolute value, but does not publish the position update.
  - `updateRelativePositionWithoutPublication()` – Updates the current position by a relative value, but does not publish the position update.
- Publish a subscription's position to all subscribers: `publishPosition()`.

  This action lets your custom position tracker publish position updates as needed. Typically, this action is used with the `update`*`Xxx`*`PositionWithoutPublication()` action. For example, you might want to incrementatlly adjust a position before informing all subscribers of the new position.
- Query what contexts are being published to: `getPublicationContexts()`.

  This action enables a custom tracker to potentially communicate directly with its subscribers. For example, a custom tracker can publish its own custom position object.
- Obtain information about custom tracker instances and subscribed positions.
  - `getTrackerId()` - Returns the unique reference Id for this custom tracker implementation. This is the Id of the `PSTrackerInterface` that was passed to the position service manager when the custom tracker was registered.
  - `getConfigDetails()` - Returns the configuration details that were used to create a specified instance of the custom position tracker.
  - `getAllConfigDetails()` - Returns all the configuration details for all instances of the custom position tracker that have been created.
  - `getCurrentPosition()` - Returns the current position for a specified instance of the custom position tracker.
  - `getAllCurrentPositions()` - Returns all the current positions for all instances of the custom position tracker.
  - `deregisterTracker()` - Sends a request to the position service manager to remove the custom position tracker from the list of registered position trackers.

See also "Sample code for implementing a custom position tracker" on page 212.

Sample code for implementing a custom position tracker

In the following sample code, a custom tracker increments a position based on `NewOrders` being placed, and filters based on a symbol name slice.

```
package com.apama.cmf.demo;

monitor PSCustomTrackerDemo {

    context mainContext := context.current();
    com.apama.position.tracker.PSTrackerManagerInterface trackerIface;
    dictionary<integer/*instanceId*/,listener> trackerListeners;
    action<> completedSetup;

    action onload() {
       com.apama.position.tracker.PSTrackerInterface trackerCbIface :=
           new com.apama.position.tracker.PSTrackerInterface;
       trackerCbIface.createInstance := createInstance;
       trackerCbIface.deleteInstance := deleteInstance;
       trackerCbIface.positionExternallyUpdated := positionExternallyUpdated;

       // Use the position service tracker factory to create an instance
       // of a position service tracker interface.
       // This call is asynchronous so it requires a callback.
       com.apama.position.PositionConfigSchema configSchema :=
          new com.apama.position.PositionConfigSchema;
       configSchema.addItemMinimal(
          "TRACKER_CONFIG_SYMBOL_SLICE", "sequence<string>", "",
          "Allows a set of symbols to match against to be defined");

       (new com.apama.position.tracker.PSTrackerFactory).registerTracker(
          "MyDemoPositionTracker",
          "MyDemoPositionTrackerType",
          mainContext,
          configSchema,
          trackerCbIface,
          trackerRegistered );
    }

    action trackerRegistered(
       com.apama.position.tracker.PSTrackerManagerInterface psTrackerIface,
       boolean success, string msg ) {
        // Inform the CMF Service Framework that this is fully created:
       completedSetup();
    }

    // Example implementation of the createInstance() action that
    // in this case creates a new subscription for each request
    // and creates a listener for new orders matching the configuration
    // provided. A unique requestId value is required to allow the
    // subscription manager service to match the asynchronous response
    // to the right request.
    action createInstance(
       integer requestId, integer instanceId,
       com.apama.position.PositionConfigParams config,
       action<integer/*requestId*/,integer/*subscriptionId*/,
          boolean/*success*/, string/*msg*/> cbCreated ) {

        // This sample tracker supports filtering on only a symbol name.
        sequence<string> symbols := [];
        if( config.hasParam( "TRACKER_CONFIG_SYMBOL_SLICE" ) ) then {
            string param := config.getParam( "TRACKER_CONFIG_SYMBOL_SLICE" );
```

```
            if( sequence<string>.canParse( param ) ) then {
                symbols := sequence<string>.parse( param );
            }
        }

        string symbol;
        for symbol in symbols {
            com.apama.oms.NewOrder no;
            on all com.apama.oms.NewOrder( symbol=symbol ):no {

                // Based on the new order's quantity/price...
                com.apama.position.Position newPosition :=
                   new com.apama.position.Position;
                newPosition.setPosition( no.quantity, no.price );

                // Adjust the position relative to the quantity/price provided
                // and ask the position tracker interface to publish it:
                trackerIface.updateRelativePosition(instanceId, newPosition);
            }
        }

        // Inform the subscription manager that a new instance
        // has been created:
        cbCreated( requestId, instanceId, true, "" );
    }

    // Example implementation of the deleteInstance() action that
    // cleans up all state for the tracker instance that is being
    // deleted (in this case the listener that was created).
    // A unique requestId value is required to allow the
    // subscription manager service to match the asynchronous
    // response to the right request.
    action deleteInstance( integer requestId, integer instanceId,
        action<integer/*requestId*/,integer/*subscriptionId*/,
            boolean/*success*/, string/*msg*/> cbDeleted ) {
        if( trackerListeners.hasKey( instanceId ) ) then {
            trackerListeners[ instanceId ].quit();
            cbDeleted( requestId, instanceId, true, "" );
        } else {
            cbDeleted( requestId, instanceId, false,
               "Tracker instance not found!" );
        }
    }

    // Example implementation of the positionExternallyUpdated()
    // action that in this case logs the update and sends back a
    // true value to indicate that the position update should be
    // published immediately.
    action positionExternallyUpdated(
        integer requestId, integer subscriptionId,
        com.apama.position.Position positionAdjustment,
        boolean isRelative, com.apama.position.PositionConfigParams config,
        action<integer/*requestId*/, integer/*subscriptionId*/,
            boolean/*updateTable*/,
            boolean/*success*/,string/*msg*/> cbAdjustedPosition ) {
            log "External Position Update:
                isRelative="+isRelative.toString()+" :
                Position="+positionAdjustment.toString() at DEBUG;

        cbAdjustedPosition(
```

```
            requestId, subscriptionId, true, true, "Update Successful");
    }
}
```

Position service framework and persistence

By default, the position service framework persists subscriptions and their position information for the default open position tracker. All subscriptions to the default open position tracker that are active in an application that shuts down will be recovered when the application restarts.

Suppose you subscribe to a default open position tracker, then unsubscribe from that tracker, and then the application shuts down. Restarting the application does not cause recovery of that subscription.

By default, the position service framework does not persist subscriptions to the default pending and reserved position trackers. These trackers monitor orders that are active in the market. If persistence is enabled for these trackers and you shut down the correlator before these orders are filled you would recover a stale position.

If an application does not require recovery of persisted position information, it can disable persistence through the CMF Service Framework (which is the same for all CMF components that support persistence). The following code extract shows how to do this:

```
package com.apama.cmf.demo;

monitor DisablePersistence {
    context mainContext := context.current();
    constant string TRACKER_NAME := "MyTracker";
    string TRACKER_TYPE :=
       com.apama.position.tracker.OpenPositionTrackerConstants.TRACKER_TYPE;

    action onload() {
        // The Service Framework ServiceParameters() event must be sent to
        // the main context prior to creating the position tracker instance.
        dictionary<string,string> serviceParams :=
           {com.apama.position.Constants.ENABLE_PERSISTENCE_KEY:"false"};
        route com.apama.service.framework.ServiceParameters(
           "MyPositionTracker",
           com.apama.position.tracker.OpenPositionTrackerConstants.TRACKER_TYPE,
           serviceParams );
        (new com.apama.position.tracker.OpenPositionTrackerFactory).create(
           mainContext, "MyPositionTracker", cbCreatedTracker );
    }

    action cbCreatedTracker( boolean success, string msg ) {
        log "Open Position Tracker created without Persistence:
           "+success.toString();
    }
}
```

Creating and configuring a currency converter

You can use interfaces in the `com.apama.ccyconverter` package to define a currency converter. A currency converter is used to normalize positions, values and prices from their trading currency to a base currency that you specify.

When you create a currency converter you are provided with an interface to that currency converter. For example, when you create a realized profit and loss default position tracker you must specify a currency converter. The tracker uses the currency converter's interface to obtain data required to normalize values.

This section provides information and instructions for creating and configuring a currency converter. Sample code is in the `samples\Currency Converter Sample` directory in your CMF installation directory. The code examples in this section are taken from that sample.

For details about the realized profit and loss position tracker, see "Creating and subscribing to the realized profit and loss tracker" on page 203.

Overview of using a currency converter

The steps for using a currency converter are as follows:

1. Create a currency calculation extension.

   The extension subscribes to a datasource to obtain the current values for currency exchange rates for specified symbols. The CMF provides two samples of currency calculation extensions in the `samples\Currency Converter Sample\eventdefinitions` directory of your CMF installation directory. See also "Creating a currency calculation extension" on page 216.

2. Create a currency converter that uses the currency calculation extension you created.

   The currency converter regularly receives updated values from the extension. When you create a currency converter you provide an interface to that currency converter. See "Creating a currency converter" on page 218.

3. Write code that uses the currency converter.

   For example, you can create a realized profit and loss default position tracker, which requires specification of a currency converter. The tracker uses the values it receives from the currency converter to calculate actual profit and loss for executed orders normalized to a base currency. See "Creating and subscribing to the realized profit and loss tracker" on page 203.

Creating a currency calculation extension

A currency converter uses a currency calculation extension to obtain the latest values for normalizing to a base currency. An extension connects to a datasource to get its price data. The CMF provides two sample currency calculation extensions in the `samples\Currency Converter Sample\eventdefinitions` directory of your CMF installation directory:

- `BBAMidPriceExtension` connects to a Best Bid Ask datasource to obtain values.

- `DepthPriceExtension` connects to a Depth Market Data datasource to obtain values.

The CMF does not supply any supported extensions. It is expected that you will modify one of the provided extensions or create your own new extension. To create an extension, call the `create()` action on a `com.apama.ccyconverter.CalculationExtensionFactory` object. This action takes no parameters and returns a default implementation of a `CalculationExtension` object. Override the `CalculationExtension` actions as needed to define your custom extension.

To use the extension, specify it when you create a currency converter. See "Creating a currency converter" on page 218.

The currency converter supplies the extension with the symbols for which the currency converter needs data from the extension. The extension connects to a datasource and subscribes to receive data for the specified symbols. The extension obtains the latest values at time intervals specified by the setting of the currency converter's `CONFIG_UPDATE_TIMEOUT` parameter.

To obtain data from the calculation extension, a currency converter calls the following `CalculationExtension` actions. See the ApamaDoc for details.

Action	Description
`getSchema()`	During startup, the currency converter calls this action to obtain the sequence of field names that the currency converter needs to store in its cache. These are the values that the currency converter uses to normalize all symbols to a base currency. The default is that this action returns a sequence that contains the single entry `"VALUE"`. If the extension is to supply more than one value then you must override this action to return all required field names. Also, you must call the currency converter `setCustomValue()` action for each additional value you need from the extension. See "Getting values from the currency converter" on page 230.
`onRegistered()`	When startup is complete the currency converter calls this action, which provides the calculation extension with the currency converter interface that the extension uses to provide values.
`refresh()`	The currency converter calls this action each time it wants the latest values from the calculation extension. The setting of the currency converter's `CONFIG_UPDATE_TIMEOUT` configuration parameter determines how often the currency converter calls this action. The default is `0.5`, which means that the extension provides the latest data every half second. This happens automatically and the currency converter does not need to explicitly invoke the `refresh()` action. If the setting of the currency converter's `CONFIG_UPDATE_TIMEOUT` parameter is `0.0` then `refresh()` is not automatically called and the extension is expected to provide values when it can, either constantly or on its own timer.
`addSymbol()`	The currency converter calls this action after startup for all possible combinations of symbols that the currency converter has been configured to supply values for. It is not expected that the extension

Action	Description
	can provide values for all configured symbols but it is expected to do its best. For example, if the currency converter has been configured to supply a value to convert from `GBP` to `EUR`, `addSymbol()` will be called for both `"GBP/EUR"` and `"EUR/GBP"`, but the extension might only be able to supply values for `"GBP/EUR"`. The `addSymbol()` action can also be called any time the parameters for the currency converter have been updated and a new symbol is required.
`removeSymbol()`	The currency converter calls this action after startup if it no longer needs data for a particular symbol.

Creating a currency converter

When you create a currency converter the new currency converter does the following:

- Registers itself with the Service Framework.
- Uses the specified parameters to construct all possible symbols it should obtain values for. For example, suppose the `CONFIG_SYMBOLS` parameter specifies `"GBP/EUR"`. The currency converter would subscribe to receive information for converting from `GBP` to `EUR` and also for converting from `EUR` to `GBP`. If you specify a currency pair for which there is no direct exchange rate, the currency converter automatically subscribes to a third currency that it can use to calculate the conversion.
- Provides its own interface to the specified extension.
- Provides the constructed list of symbols to the extension.
- Regularly notifies the extension to obtain updated values.
- Creates a listener for remote connections.

To create a currency converter, execute either of the following actions:

- `com.apama.ccyconverter.CurrencyConverterFactory.create()` constructs a new currency converter instance in the current context.
- `com.apama.ccyconverter.CurrencyConverterFactory.createCb()` constructs a new currency converter instance in the current context and then executes the specified callback.

Both actions take these parameters:

- *`serviceManagerCtx`* is a reference to the context that the Service Framework manager has been created in. Currently, this is always the main context.
- *`name`* is a string that contains the name for the currency converter. In an application, two currency converters cannot have the same name.
- `extension` is the interface to the currency calculation extension you created. See "Creating a currency calculation extension" on page 216.

In addition, the `createCb()` action takes a callback action as its last parameter.

Creation of a currency converter is asynchronous. If your application requires notification when the currency converter has been constructed, execute the `createCb()` action. After the CMF is activated and the currency converter is constructed, the specified callback will be executed. For example:

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.ccyconverter.CurrencyConverterFactory;
using com.apama.ccyconverter.CurrencyConverter;
using com.apama.cmf.sample.BBAMidPriceExtension;

monitor CurrencyConverterSample {

   context mainContext := context.current();

   action onload() {
      SessionHandler sHandler := (new SessionHandlerFactory).
         connect(mainContext, "MySession", "MyTransport");

      CurrencyConverter cc1 := (new CurrencyConverterFactory).createCb(
         mainContext,
         "MyCurrencyConverter1",
         (new BBAMidPriceExtension).create(sHandler),
         onCurrencyConverterCreated);
      ...
   }

   action onCurrencyConverterCreated(CurrencyConverter iface) {
      // Currency converter created.
   }
}
```

You may perform actions immediately after the call to create the currency converter, for example you can connect to a the new currency converter. In this case, the create callback will be executed after the connections from remote contexts or monitors are set up. After execution of this callback, you can of course connect from additional contexts and monitors.

If any parameters are set on a `CurrencyConverterFactory`, these parameters have the same settings in a currency converter instance created by that factory. If default error handling is changed for a `CurrencyConverterFactory` instance and then you use it to create a currency converter, if there is an error in the creation of the currency converter then the updated error handling behavior is followed. Also, a currency converter instance created by this factory handles errors in the same way as the factory. See "Configuring a currency converter factory" on page 225.

Any parameters you set for a currency converter factory instance and any error handling behavior you update for that factory apply to only subsequently created currency converters.

Connecting to a currency converter

After you create a currency converter, you can use that currency converter instance from other contexts and from other monitors. For example, you might want to create the currency converter in one context and use it for multiple realized profit and loss trackers that are in other contexts or

monitors. When you want to use a currency converter that was created in another context or monitor you connect to that currency converter.

To connect to a currency converter, execute either of the following actions:

- `com.apama.ccyconverter.CurrencyConverterFactory.connect()` connects to a currency converter instance in another context or monitor and returns a `com.apama.ccyconverter.CurrencyConverter` instance for use in the current context.
- `com.apama.ccyconverter.CurrencyConverterFactory.connectCb()` connects to a currency converter instance in another context or monitor, returns a `com.apama.ccyconverter.CurrencyConverter` instance for use in the current context, and executes the specified callback.

Both actions take these two parameters:

- *`serviceManagerCtx`* is a reference to the context in which the Service Framework manager was created. Currently, this is always the main context.
- *`name`* is a string that contains the name of the currency converter you want to connect to. The name of the currency converter must be available in the context or monitor from which you are connecting. Note that in the context that contains the currency converter you want to connect to, to obtain the name of that currency converter, you can call `com.apama.ccyconverter.CurrencyConverter.getCurrencyConverterName()`.

In addition, the `connectCb()` action takes a callback action as its third parameter. This callback is invoked after connection to the remote currency converter.

The `CurrencyConverter` instance returned by the `connect()` or `connectCb()` action allows you to execute all currency converter actions except the `CurrencyConverter.setParams()` action. You can change configuration parameters only in a local currency converter.

To determine whether the currency converter in the current context is connected to a remote currency converter instance, execute the `CurrencyConverter.isRemote()` action.

Connection to a remote currency converter is asynchronous. Consequently, some currency converter actions that are normally available on a currency converter that is connected to a remote currency converter might not be available until the connection is completely in place. If your application requires notification when the connection to the remote currency converter is in place, execute `connectCb()`and specify a callback.

You can execute `connect()` or `connectCb()` to connect to a remote currency converter that has not yet been created. However, a currency converter you connect to must be created before the configurable timeout period has expired, otherwise an error will be returned. The configuration parameter that specifies the timeout period is `CONFIG_TIMEOUT_DURATION` and the default value is `5.0` seconds.

There can be any number of connections to a currency converter instance.

Any error callbacks that were set on the `CurrencyConverterFactory` prior to using it to connect to the remote currency converter are used if there is an error when connecting to the currency converter.

However, any parameters that were set on the `CurrencyConverterFactory` prior to using it to connect to the remote currency converter are ignored. The only exception to this is the `CONFIG_TIMEOUT_DURATION` parameter. You can set this on a factory and its value is used for the connection to the remote currency converter.

A currency converter that is connected to a remote currency converter instance has the same configuration parameter settings as the currency converter instance it is connected to. You cannot set parameters on a currency converter that is connected to a remote currency converter. This protects remote currency converter instances from modifications by currency converter connections. To change the configuration parameters for a currency converter that is connected to a remote currency converter you must change the configuration parameters directly on the remote currency converter instance itself. When you update the configuration parameters for a currency converter instance any currency converters connected to that remote instance inherit the updated settings.

Connecting to a currency converter instance in the same context is useful when you have two or more EPL monitors in the same context. For example, you can create a currency converter instance in `monitorA` and in `monitorB` you can connect to the currency converter in `monitorA`. You can then track realized profit and loss positions from `monitorB`. A connected currency converter takes care of the communication between the two monitors.

Default settings for currency converter parameters

There are many parameters that affect currency converter behavior. All parameters have default settings that provide the most commonly desired behavior. If these settings provide the behavior you want then you do not need to set currency converter configuration parameters.

However, if you do want to set the value of a currency converter configuration parameter, see the following topics:

- "Descriptions of currency converter parameters" on page 222
- "Setting currency converter factory parameters" on page 225
- "Setting currency converter instance parameters" on page 227

The default settings for currency converter parameters are as follows. If it is blank where the default setting should be then the parameter does not have a default value.

Parameter	Default Setting
`CONFIG_BASE_CURRENCY`	`""`
`CONFIG_CROSS_CURRENCY_INFO`	Empty dictionary
`CONFIG_CURRENCIES`	Empty sequence
`CONFIG_CURRENCY_SEPARATOR`	`/`
`CONFIG_ENABLE_PERSISTENCE`	`false`
`CONFIG_SYMBOLS`	Empty sequence

Parameter	Default Setting
CONFIG_TIMEOUT_DURATION	5.0
CONFIG_UPDATE_TIMEOUT	0.5
CONFIGSTORE_PATH_DEFAULT	"CurrencyConverter.sqlite"
CONFIGSTORE_PATH_KEY	"ConfigStorePath"
CURRENCY_CONVERTER_SERVICE_TYPE	"CurrencyConverter"
DEFAULT_FIELD_NAME	"VALUE"

Descriptions of currency converter parameters

Constant values defined in `com.apama.ccyconverter.Consts` define the names of the configuration parameters that control currency converter behavior. These configuration parameters are described below.

Parameter	Description
CONFIG_BASE_CURRENCY	Defines the name of the configuration parameter that specifies the base currency. The base currency is the currency that the currency converter will normalize values to. The currency converter uses this parameter in conjunction with the CONFIG_CURRENCIES parameter to construct the sequence of symbols for which to obtain values.
CONFIG_CROSS_CURRENCY_INFO	Defines the name of the configuration parameter that specifies the cross currency pairs that the currency converter uses to construct symbols. Use this parameter with the CONFIG_SYMBOLS and CONFIG_CURRENCIES parameter. The type of the value of this configuration parameter must be dictionary<string,string>. See "About cross currencies" on page 224.
CONFIG_CURRENCIES	Defines the name of the configuration parameter that specifies the sequence of currencies that the currency converter will use to construct symbols for which to obtain updates. The type of the value of this parameter must be sequence<string>. The currency converter uses this parameter in conjunction with the CONFIG_BASE_CURRENCY parameter to construct the sequence of symbols for which to obtain values.

Parameter	Description
CONFIG_CURRENCY_SEPARATOR	Defines the name of the configuration parameter that specifies the currency separator that the currency converter uses when constructing or splitting symbols. The default is /.
CONFIG_ENABLE_PERSISTENCE	Defines the name of the configuration parameter that specifies whether persistence is enabled for the currency converter. Note that currency converter persistence is separate from correlator persistence. When currency converter persistence is enabled the last stored values in the currency converter's cache are reloaded to be used until the fresh values are obtained from the calculation extension. The default is false.
CONFIG_SYMBOLS	Defines the name of the configuration parameter that specifies the sequence of symbols that the currency converter supplies values for. If you set values for CONFIG_BASE_CURRENCY and CONFIG_CURRENCIES but not CONFIG_SYMBOLS the currency converter can construct the sequence that would have been specified in CONFIG_SYMBOLS itself. If you specify a value for each of these parameters: CONFIG_BASE_CURRENCY, CONFIG_CURRENCIES, and CONFIG_SYMBOLS, the currency converter constructs a superset of each value in CONFIG_SYMBOLS plus each possible combination that results from the specifications of CONFIG_BASE_CURRENCY and CONFIG_CURRENCIES. Specify a value for CONFIG_SYMBOLS when you know the exact symbols you need values for or when you want to normalize to more than one base currency. When you want to normalize to a single base currency you can use CONFIG_BASE_CURRENCY and CONFIG_CURRENCIES without specifying a value for CONFIG_SYMBOLS.
CONFIG_TIMEOUT_DURATION	Defines the name of the configuration parameter that specifies the maximum duration in seconds that any asynchronous communication can take before an error is returned. For example, if you connect to a remote currency converter, the operation waits no more than 5 seconds for the currency converter to be created. The default is 5.0.

Parameter	Description
CONFIG_UPDATE_TIMEOUT	Defines the name of the configuration parameter that specifies the timeout for obtaining currency values from the extension. For example, using the default (0.5), the currency converter would obtain values from the extension every half second. To disable automatic retrieval of values, set this parameter to 0.0. For additional information, see the documentation for the refresh() action in "Creating a currency calculation extension" on page 216.
CONFIGSTORE_PATH_DEFAULT	Defines the name of the configuration parameter that specifies the default file and path for persisting currency converter data in configuration store tables. The default is "CurrencyConverter.sqlite". To change the setting of this parameter, use the CONFIGSTORE_PATH_KEY parameter.
CONFIGSTORE_PATH_KEY	Defines the name of the configuration parameter that specifies the configuration key value for setting the currency converter default file and path for its configuration store tables. Use this parameter when you need to override the default storage location used by the currency converter. If you change the value of this parameter you must have read/write access to the location you specify.
CURRENCY_CONVERTER_SERVICE_TYPE	Defines the name of the configuration parameter that specifies the currency converter service that is registered with the CMF Service Framework. The default is "CurrencyConverter". You cannot modify the value of this parameter. However, you can use this parameter to query the CMF Service Framework directly for all services of this type.
DEFAULT_FIELD_NAME	Defines the name of the configuration parameter that specifies the default field name used by the currency converter and the calculation extension. The default is *"VALUE"*. You cannot modify the value of this parameter.

About cross currencies

For some currencies, there is no direct way to convert it to a particular second currency. Such currencies are referred to as cross currencies because the currency converter must cross it through an intermediary currency to obtain a conversion value. For example, suppose you want to convert

from DKK to USD. Since there is no DKK/USD exchange rate the currency converter converts from DKK to EUR and then converts from EUR to USD.

Use the CONFIG_CROSS_CURRENCY_INFO parameter to specify the currency pairs that you will want to normalize and for which there is no direct exchange rate. The value of this parameter is a dictionary in which the key is the currency that you want to convert to the base currency and the dictionary value is the intermediary currency required for the conversion. The base currency is specified by the CONFIG_BASE_CURRENCY configuration parameter.

Configuring a currency converter factory

You use a com.apama.ccyconverter.CurrencyConverterFactory instance to create new com.apama.ccycurrency.CurrencyConverter instances. Before you use a currency converter factory to create a new currency converter, you can configure the currency converter factory as described in the topics below.

Setting currency converter factory parameters

There are many configuration parameters associated with currency converter instances. If you want, you can set these parameters on a currency converter factory so that subsequent currency converters created from that factory have the same parameter settings.

The constant values used to specify these configuration parameters are defined in com.apama.ccyconverter.Consts.

When you use a currency converter factory to create a new currency converter instance, the created currency converter instance has the same configuration parameter values as the currency converter factory that created it. However, when you use a currency converter factory to connect to a remote currency converter instance, the returned currency converter has the same configuration parameter values as the currency converter instance it is connected to.

If the default configuration parameter settings meet your needs, you do not need to set currency converter factory parameters. See "Default settings for currency converter parameters" on page 221. However, if you want to configure the currency converter instances you create to have one or more common, non-default configuration parameter settings, you can set the appropriate parameter values on a currency converter factory instance. Any currency converters you create from that factory will have the parameter values you previously set on that factory.

To set a parameter on a currency converter factory:

1. Create a currency converter factory instance.
2. Create a com.apama.utils.Params object.
3. Add a parameter name/value pair to the parameters object you created.
4. Repeat the previous step for each parameter you want to set.
5. Execute the setParams() action on the currency converter factory instance and pass it the Params object you created.

After you set one or more configuration parameters on a currency converter factory instance, you can obtain the parameter values you set on that instance by executing `com.apama.ccycurrency.CurrencyConverterFactory.getParams()`. This action returns the factory configuration parameters you explicitly set and not any default settings for configuration parameters.

Setting currency converter factory parameters applies only to subsequently created currency converter instances.

After you create a currency converter instance, you can still change its configuration parameters. You do this by executing `com.apama.ccyconverter.CurrencyConverter.setParams()` on the currency converter instance. See "Setting currency converter instance parameters" on page 227.

The following example shows how to set a configuration parameter on a currency converter factory instance.

```
using com.apama.ccyconverter.CurrencyConverterFactory;
using com.apama.ccyconverter.Consts;
using com.apama.utils.Params;

monitor CurrencyConverterSample2 {

 CurrencyConverterFactory factory;
 Consts consts;
 string baseCurrency := "USD";
 sequence< string > currency := [ "EUR", "DKK" ];
 sequence < string > symbols := [ "GBP/EUR" ];
 dictionary < string, string > crossInfo := { "DKK":"EUR" };

 action onload() {

  Params params := new Params;

  params.addParam(consts.CONFIG_BASE_CURRENCY, baseCurrency);
  params.addStringSequenceParam(consts.CONFIG_CURRENCIES, currency);
  params.addStringSequenceParam(consts.CONFIG_SYMBOLS, symbols);
  params.addStringDictionaryParam(consts.CONFIG_CROSS_CURRENCY_INFO, crossInfo);
  // Frequency in seconds for updating the currency values:
  params.addFloatParam(consts.CONFIG_UPDATE_TIMEOUT, 3.0);
  factory := new CurrencyConverterFactory;
  factory.setParams(params);

  ...
 }
}
```

Overriding default error handling for currency converter factories

The default error handler is invoked if there is an error related to currency converter factories. For example, if any of the following happen:

- Currency converter creation or connection fails.
- You try to remove an error callback and specify an incorrect error callback reference identifier.
- You try to set a parameter and specify an invalid parameter name.

The default error handler sends a message to the correlator log file at the `ERROR` level. To change this behavior, execute `com.apama.ccyconverter.CurrencyConverterFactory.addErrorCallback()`, which adds the specified callback to the set of callbacks executed if there is an error related to that currency converter factory instance.

You can execute the `addErrorCallback()` action multiple times on the same factory instance to implement multiple error handling callbacks for that factory instance. If you add one or more error callbacks to a factory instance then the default error callback is not executed for that factory instance.

The parameters of a user-defined error callback include the currency converter factory instance and also a `com.apama.utils.Error` event. An `Error` event has fields for a message, a dictionary of parameters, and an error type code. The `addErrorCallback()` action adds the specified callback to the set of callbacks executed if there is an error in the operation of the specified currency converter factory instance.

The `com.apama.ccyconverter.ErrorConstants` event defines the error type codes, which can apply to currency converter factory instances. See the ApamaDoc for details.

When you add an error callback the return value is an integer reference ID that you can specify if you execute `CurrencyConverterFactory.removeErrorCallback()` to discontinue execution of that error callback. To remove all error callbacks, execute the `CurrencyConverterFactory.clearErrorCallbacks()` action. If you remove all previously set error callbacks then error handling behavior reverts to calling the default error callback.

Suppose you add one or more error callbacks to a currency converter factory instance. A currency converter instance created by that factory has the same error callbacks as that factory. Also, if you use that factory to connect to a remote currency converter instance, the returned currency converter uses the same error handling as the factory that was used to connect.

Configuring currency converter instances

You can configure currency converter instances as described in the topics below.

Setting currency converter instance parameters

When you create a currency converter instance it has the same configuration parameter settings as the currency converter factory you use to create it. Each currency converter configuration parameter has a default setting, which is appropriate for the most common currency converter use cases. You only need to set currency converter parameters if the default settings do not meet your application requirements.

There are many configuration parameters that you can set to specify the behavior of a currency converter. The constant values used to specify these configuration parameters are defined in `com.apama.ccyconverter.Consts`. See also "Descriptions of currency converter parameters" on page 222.

To set a configuration parameter for a particular currency converter instance, execute the `com.apama.ccyconverter.CurrencyConverter.setParams()` action on the currency converter instance. Alternatively, you can set the configuration parameter on a currency converter factory instance

and then use that factory instance to create the currency converter. See "Setting currency converter factory parameters" on page 225.

To set parameters on a currency converter instance:

1. Use a currency converter factory to create a currency converter instance. For example:

```
com.apama.ccyconverter.CurrencyConverter cc :=
   (new com.apama.ccyconverter.CurrencyConverterFactory).
      create(mainContext, "myCurrencyConverter",
      (new BBAMidPriceExtension).create(sHandler));
```

2. Create a `com.apama.utils.Params` object.
3. Add a parameter name/value pair to the parameters object you created.
4. Repeat the previous step for each parameter you want to set.
5. Execute the `setParams()` action on the currency converter instance and pass it the `Params` object you created.

The new parameter settings affect subsequent requests sent to the currency converter.

To obtain the values for any configuration parameters that have been explicitly set, either on the factory that created the currency converter instance or on the instance itself, execute the `CurrencyConverter.getParams()` action. This action does not return values for parameters that have the default setting.

Adding update callbacks to currency converters

The currency converter configuration parameter `CONFIG_UPDATE_TIMEOUT` specifies how often the currency converter obtains conversion values from its extension. The default is `5` seconds. If you want, you can add one or more callbacks to be executed each time the currency converter obtains values from its extension.

To set an update callback, execute `com.apama.ccyconverter.CurrencyConverter.addUpdateCallback()`. This action registers the specified callback with the currency converter instance it is executed on. The currency converter calls the registered callback each time the `CONFIG_UPDATE_TIMEOUT` is reached, which indicates that the values from the extension might have been updated. The `addUpdateCallback()` action returns a unique integer reference Id that you can use to remove the callback at a later date if required.

To remove a previously-added update callback, execute `CurrencyConverter.removeUpdateCallback()` and specify the integer reference Id that was returned when you added the callback.

The following code sample shows the addition of an update callback as well as the update callback itself.

```
using com.apama.session.SessionHandlerFactory;
using com.apama.session.SessionHandler;
using com.apama.ccyconverter.CurrencyConverterFactory;
using com.apama.ccyconverter.CurrencyConverter;
using com.apama.cmf.sample.BBAMidPriceExtension;

monitor CurrencyConverterSample2 {
```

```
   context mainContext := context.current();

   action onload() {

      SessionHandler sHandler := (new SessionHandlerFactory).connect(
         mainContext, "MySession", "MyTransport");
      CurrencyConverter cc1 := (new CurrencyConverterFactory).createCb(
         mainContext,
         "MyCurrencyConverter1",
         (new BBAMidPriceExtension).create(sHandler),
         onCurrencyConverterCreated);
   }

   action onCurrencyConverterCreated (CurrencyConverter iface) {
      integer callback := iface.addUpdateCallback(cbUpdateCC1);
   }

   action cbUpdateCC1( CurrencyConverter iface ) {
      log "UPDATE: "+ iface.getCurrencyConverterName() +
         " EUR/USD, Mid Price: " + iface.getValue("EUR", "USD" ).toString() +
         " GBP/EUR, Mid Price: " + iface.getValue("GBP", "EUR" ).toString() +
         " DKK/USD, Mid Price: " + iface.getValue("DKK", "USD" ).toString() +
         " in context : " + context.current().getName() ;
   }

}
```

Overriding default error handling for currency converters

The default error handler is invoked if there is an error related to a currency converter instance. For example, if any of the following happen:

- There is an attempt to execute `CurrencyConverter.setParams()` on a currency converter that is connected to a remote currency converter instance.
- You try to remove an error callback or an update callback and specify an incorrect callback reference identifier.
- You try to set a parameter and specify an invalid parameter name.

The default error handler sends a message to the correlator log file at the `ERROR` level. To change this behavior, execute `com.apama.ccyconverter.CurrencyConverter.addErrorCallback()`, which adds the specified callback to the set of callbacks executed if there is an error related to that currency converter instance.

You can execute the `addErrorCallback()` action multiple times on the same currency converter instance to implement multiple error handling callbacks for that currency converter instance. If you add one or more error callbacks to a currency converter instance then the default error callback is not executed for that currency converter instance. However, you can call the default error callback from your own error callback by using the `CurrencyConverter.defaultErrorCallback()` action and providing the parameters. The default error callback will send a message to the correlator log file at the `ERROR` level.

The parameters of a user-defined error callback include the currency converter instance and also a `com.apama.utils.Error` event. An `Error` event has fields for a message, a dictionary of parameters,

and an error type code. The `addErrorCallback()` action adds the specified callback to the set of callbacks executed if there is an error in the operation of the specified currency converter instance.

The `com.apama.ccyconverter.ErrorConstants` event defines the error type codes that can apply to currency converter instances. See the ApamaDoc for details.

When you add an error callback the return value is an integer reference ID that you can specify if you execute `CurrencyConverter.removeErrorCallback()` to discontinue execution of that error callback. To remove all error callbacks, execute the `CurrencyConverter.clearErrorCallbacks()` action. If you remove all previously set error callbacks then error handling behavior reverts to calling the default error callback.

Setting symbol values in the currency converter cache

You can set a default symbol value in the currency converter's cache. For example, you might want to set initial symbol values before a currency converter starts to use its calculation extension. You would do this in the application code that uses the currency converter.

The currency converter provides two actions for setting default symbol values in its cache:

- `CurrencyConverter.setValue(string, float)` — Specify the symbol whose value you want to set and then the value itself. The value you specify becomes the current default value for that symbol. By default, the extension stores the value in a field whose name is `"VALUE"`. If the extension that the currency converter uses does not use the default `"VALUE"` field, it is an error.

  Use the format `"CCY1/CCY2"` to specify the symbol.

- `CurrencyConverter.setCustomValue(string, string, float)` — Specify the symbol whose value you want to set, the name of the field that holds the value for this symbol, and then the value itself. The value you specify becomes the current default value for that symbol. The extension stores the value in the field whose name you specify and does not store it in the default `"VALUE"` field.

  Use the format `"CCY1/CCY2"` to specify the symbol.

Typically, extensions provide a single value for each symbol. But it is possible for an extension to provide more than one value. Use the `setCustomValue()` action when the calculation extension provides a schema and value for fields other than the default `"VALUE"` field. For example, in `DepthPriceExtension.mon`, the extension uses the `"Bid"` and `"Ask"` fields to contain values for the Best Bid and Ask from the Depth market data source.

Getting values from the currency converter

When you need to get values from the currency converter call the `CurrencyConverter.getValue()` or `CurrencyConverter.getCustomValue()` action. Typically, this is in a currency converter update callback. The realized profit and loss position tracker does this to normalize positions to a base currency.

The `getValue()` action takes two parameters, the currency you want to convert from and the currency you want to convert to. The value returned from the currency converter's cache is the

current default value for the specified currencies. You can then use the returned value as a multiplier to convert a value, price, or position from one currency to the other.

If the currency converter cannot find a value for directly converting between the specified currencies it tries to reverse the currencies. If that does not yield a return value then the currency converter checks the cross currency pairs that have been set up to determine whether it can use a third currency to convert between the first two. See "About cross currencies" on page 224.

The `getCustomValue()` action takes three parameters, the currency you want to convert from, the currency you want to convert to, and the field you want to get from the cache. The value returned from the currency converter's cache is the field value you explicitly request for the specified currencies. This field must have been previously specified with the `CurrencyConverter.setCustomValue()` action. You can then use the returned value as a multiplier to convert a value, price or position from one currency to the other. See "Setting symbol values in the currency converter cache" on page 230.

6 Utilities

- Service Framework 234
- Configuration Service 237
- General Util bundle 241
- Status publisher 241
- Transaction components 241
- User session services 241
- Adapter bridging bundle 241
- Adapter status bridge service 243
- DataView manager 243
- Helper events for creating DataViews 245

This section describes CMF utilities.

Service Framework

The Service Framework allows applications to ensure that all services they use, such as the Risk Firewall and the Position Service, are fully configured before the rest of the application is allowed to start. Configuration may entail loading a persisted set of configuration data from an external source, such as loading persisted Risk Firewall rule instances from a database.

Without the Service Framework, the application would have to arbitrarily assume when the application is fully configured, or listen for a complex set of events indicating that configuration is complete. Such events might include those ensuring that the Risk Firewall has loaded its rule instance, the Position Service has been enabled, and the Market Data Architecture has been made available to send market data through the system.

To support multi-context environments, the Service Framework includes two parts:

- Service Framework Manager

  The Service Framework Manager monitors the status of all Service Interfaces that are created in the application, and the context they are created within. It also responsible for communicating with all instances of the Service Interfaces when the application activates the Service Framework. The application should not communicate directly with the Service Framework Manager other than to send the `Activate()` event to the context it resides in. All other communication with it should be through the Service Interface described below.

- Service Interface

  The Service Interface event object (`com.apama.service.framework.ServiceInterface`) should be created by any application EPL monitor instances that:

  - require the two-stage startup process to ensure that the service is initialized/configured before the rest of the application is allowed to continue
  - depend on another service being fully configured (such as relying on the Risk Firewall being available)

To use the Service Framework in an application, inject the Service Framework bundle into the correlator. This ensures that the appropriate EPL code (including any dependencies) is injected in the appropriate order.

The CMF Service framework employs a two-stage startup. The first phase creates and configures any necessary services. The application should activate the second phase by sending a `com.apama.config.Activate()` event after all EPL code has been injected and any external adapters have been started. This effectively informs all users of those services that they are ready. This ensures that the second stage does not start until all services created by the application have been setup and configured.

ServiceInterface examples

The `ServiceInterface` object provides the following actions:

- `initialise()` creates a new Service Interface instance, and will inform the application (using the specified callback) when all services in the system are available. After that callback occurs, the second stage may proceed.
- `initialiseWithConfig()` does the same as `initialise()`, but also allows the application monitor to be called back using the specified callback action when the Service Manager has been activated (it has received the `com.apama.config.Activate()` event). This allows the monitor to perform necessary configuration before services are made available. The callback action also provides a set of default, service-specific, configuration parameters that can be provided prior to the Service Framework activation by using the `com.apama.service.framework.ServiceParameters()` event.
- `getServiceConfig()` returns a sequence of service configuration objects that match the service name and type that were provided. The service configuration object defines the service name and type, the set of parameters that the service was created with, and also defines the context that the service was created on.
- `waitForService()` will call the user-defined callback action when a service that matches the name and type requested service with the name and type provided has been fully initialized, and the service framework has been activated. This callback will also provide details of any configuration that was applied to that service when it was initialized, and also defines the context that the service was created on. This action can be very useful when building applications that rely on other services in the Capital Markets Foundation.

The following example demonstrates use of `initialise()`:

```
monitor Example {

   action onload() {
      integer id := (
         new com.apama.service.framework.ServiceInterface).initialise(
            "serviceType", "serviceName", context.current(),
            allServicesInitialised );
   }

   action allServicesInitialised( integer serviceId ) {

   // Application code

   }
}
```

An alternate example for a multi-context environment:

```
monitor Example {
   context mainContext := context.current();

   action onload() {
      context myContext := context( "MY_EXAMPLE_CONTEXT", false);
      spawn myNewContext() to myContext;
}

   action myNewContext() {
      integer id := (
```

```
         new com.apama.service.framework.ServiceInterface).initialise(
           “serviceType”, “serviceName”,
           mainContext, allServicesInitialised );
   }

   action allServicesInitialised( integer serviceId ) {

      // Application code

   }
}
```

The following example demonstrates use of the `waitForService()` action :

```
event MyEvent {
}

monitor Example {

   action onload() {
      integer id := (
         new com.apama.service.framework.ServiceInterface).waitForService(
            "serviceType", "serviceName", context.current(),
            cbOnDepedentServiceReady );
   }

   action cbOnDepedentServiceReady(
      com.apama.service.framework.ServiceConfig serviceConfig ) {
     // We can now interrogate the ServiceConfig to find out where
   // the service was created so that we can send an event to it.
   enqueue MyEvent() to serviceConfig.serviceCtx;
   }
}
```

The following example demonstrates use of the `initialiseWithConfig()` action :

```
monitor Example {

   action onload() {
      integer id := (
         new com.apama.service.framework.ServiceInterface).initialiseWithConfig(
            "serviceType", "serviceName", context.current(),
            myServiceIntialised, allServicesInitialised );
   }

   action myServiceInitialised( integer serviceId, dictionary
      <string, string> defaultConfig, action<> configCompleted ) {

      // Application code to configure this service

      // Asynchronously inform Service Interface configuration is complete
      configCompleted();
   }

   action allServicesInitialised( integer serviceId ) {

   // Application code
```

```
    }
}
```

The `delete()` action correctly cleans up any resources used by a Service Interface instance that is no longer required.

Configuration Service

Capital markets applications commonly need to manage collections of configuration data and preserve it across restarts. The CMF Configuration Service provides the functionality to persist, modify, and retrieve configuration data. CMF components such as the Risk Firewall and Position Service use the Configuration Service to maintain their state across application restarts.

The Configuration Service manages configuration stores, collections of tables with application-defined schemas, which are persisted to and loaded from a backing store. A single application can have multiple backing store implementations and each implementation can use a different backing store. The Configuration Store bundle includes a backing store implementation that persists data using Apama MemoryStore. The Configuration Service sample in the `\ASB\Samples\` folder illustrates how to use it.

Backing memory stores implement the `ConfigurationStoreInterface` object. Typically, they use the current CMF memory store interface `MemoryStoreConfigurationStore` object. An application must open a configuration store to load the data into memory. A set of in-memory tables make the data available to the application. The `com.apama.config.ConfigurationTableInterface` object provides access to the data in these tables. Applications should use the `ConfigurationTableInterface` object when updating configuration tables to ensure that updates are persisted to the underlying backing store. In addition, use of the Apama `MemoryStore` to cache the configuration table data simplifies creation of dashboards because DataViews are automatically generated for each table.

To use a backing store for persisting configuration information, an application must first perform the following:

- Add the Configuration Store bundle.
- Initialize the Configuration Service.
- Create a backing store and specify the schema (during creation) or find an existing backing store.

After these initial steps, the application can add, modify, and retrieve configuration data.

Configuration service initialization

To start using the configuration service, an application must create and initialize a configuration store using the desired backing store implementation. For example, to create or load a configuration store backed by the on-disk Memory Store database in file `myApp.db`, see the following code. Access to the configuration database happens in parallel with the execution of application code in the Apama correlator, so races can easily occur if the application does not synchronize its activities with the service framework correctly.

Note:
When you use flat-file databases like `myApp.db`, you should be sure that they are backed up on a regular basis.

```
// CMF service framework
com.apama.service.framework.ServiceInterface frmwk;

// Abstract interface to the configuration service
com.apama.config.ConfigurationStoreInterface iface;

action onload() {
   // Initialize the service framework
   integer serviceId := frmwk.initialise("Sample", "ConfigServiceSample",
      context.current(), allServicesInitialised );
}

// Called when the service framework has been initialised
action allServicesInitialised( integer serviceId ) {
   // Initialize the Memory Store configuration service implementation
   iface := (new com.apama.config.MemoryStoreConfigurationStore).getInterface
      (constants.SERVICE_INSTANCE, constants.STORE_PATH);
   iface.init(integer.getUnique(), onSuccess, onError);
}

// Called after store is initialized and all rows are loaded into memory
action onSuccess(integer id) {
   log "Initialized configuration store" at INFO;
}
```

The `getInterface()` action returns a `com.apama.config.ConfigurationStoreInterface` object - an abstract interface to the store that can be used in the same way regardless of the backing store implementation. Calling the `init()` action on this object loads tables from the store and initializes the in-memory cache. Either the `onSuccess()` or `onError()` action will then be called.

Note:
The `onSuccess()` or `onError()` callback actions must be defined. If an application does not want to supply its own callbacks, the `ConfigurationStoreInterface` object provides default callback implementations. The default implementations route events that can be handled by an application-supplied listener.

If the store was successfully initialized, the application can query the available tables and their schemas, create new tables or get an interface to a named table. For example:

```
// Called when the store has been initialized
// and all persisted rows have been loaded into memory
action onSuccess(integer id) {
   log "Initialized configuration store" at INFO;

   // Assume the table already exists
   com.apama.config.ConfigurationTableInterface _table
      := iface.getTable("TestTable");

   // Iterate over all the rows of the table and log the value of a field
   com.apama.memorystore.Iterator it := _table.begin();
   while not it.done() {
      com.apama.memorystore.Row row := it.getRow();
      log row.getString("TestField") at INFO;
```

```
      it.step();
   }
}
```

The Configuration Service includes a comprehensive sample (the Configuration Service Sample project in the `ASB/samples` directory of the CMF installation) that demonstrates how to use the features of the Configuration Service, including a simple dashboard and the use of the `com.apama.config.ConfigurationRowHelper` event to map individual table rows to event instances.

See the ApamaDoc for more details on the following:

- `com.apama.config.ConfigurationStoreInterface`
- `com.apama.config.ConfigurationTableInterface`
- `com.apama.config.MemoryStoreConfigurationStore`

Creating a configuration store

To create a new table in a successfully initialized configuration store, the application must specify the schema for the table. The configuration service uses the standard Apama MemoryStore schema type to specify the field names and types for the table, but also allows the application to specify the set of fields that will be used to construct the keys for each row of the table. The key field(s) for a row form a key that is unique within the table.

The following example shows how to create a table. Note that instead of assuming the table does not exist, the code should first check and then create the table only if it does not exist:

```
// Called when the store has been initialized and all
//persisted rows have been loaded into memory
action onSuccess(integer id) {
 log "Initialized configuration store" at INFO;

 // If the table already exists, jump directly to the next step
 if iface.hasTable(constants.TABLE_NAME) then {
  log "Sample table already exists" at INFO;
  onSuccess2(integer.getUnique(), iface.getTable("TestTable"));
 }

 // Otherwise, create the sample table
 else {
  com.apama.memorystore.Schema mSchema := new com.apama.memorystore.Schema;
  mSchema.fields := ["stringField", "integerField"];
  mSchema.types := ["string", "integer"];

  com.apama.config.Schema schema := new com.apama.config.Schema;
  schema.tableName := "TestTable";
  schema.schema := mSchema;
  schema.indexFields := ["stringField"];
  schema.extraParams := {};
  log "Creating sample table..." at INFO;
  iface.createTable(integer.getUnique(), schema, onSuccess2, onError);
 }
}

// Called when the sample table has been created or retrieved
```

```
action onSuccess2(
  integer id, com.apama.config.ConfigurationTableInterface _table) {

 log "TestTable created or retrieved successfully" at INFO;
}
```

Adding rows to a configuration table

To add rows to a configuration store table, or to update an existing row, the application must first generate a key for the row. The key is generated from the sequence of values for the index fields specified in the table schema. Note that the key values for non-string fields, such as `integer`, `float` and `boolean`, must be converted to strings when generating the key. As with most configuration service operations, adding or updating a row is an asynchronous operation so the application must wait for it to complete before assuming that the row was successfully added:

```
// Called when the sample table has been created or retrieved
action onSuccess2(integer id,
                  com.apama.config.ConfigurationTableInterface _table) {

 log "TestTable created or retrieved successfully" at INFO;

 // Add a new row
 sequence<string> keyseq := ["aString"];
 key := _table.makeKey(keyseq);
 if key.length() < 1 then {
  log "Failed to create key for new row" at ERROR;
  return;
 }
 else {
  log "Key for new row is: "+key at INFO;
 }

 if _table.hasRow(key) then {
  log "New row already exists!" at ERROR;
  return;
 }
 else {
  com.apama.memorystore.Row row := _table.getRow(key);
  row.setString("stringField", "aString");
  row.setInteger("integer", 42);
  _table.updateRow(integer.getUnique(), key, row, onSuccess3, onError);
 }
}
// Called if the new row is successfully created
action onSuccess3(integer id) {

 log "New row created!" at INFO;
}
```

Retrieving rows from a configuration table

The process for retrieving rows from a configuration table is similar to that for adding or updating rows, in that a key must first be generated from a sequence of index field values. The row corresponding to the key can then be retrieved as a standard Apama MemoryStore `Row` event:

```
action onSuccess3(integer id) {
 sequence<string> keyseq := ["aString"];
 key := _table.makeKey(keyseq);
 if key.length() < 1 then {
  log "Failed to create key for existing row" at ERROR;
  return;
 }
 if not _table.hasRow(key) then {
  log "Row does not exist after it has been created" at ERROR;
  return;
 }

 // Make sure the new row has the right fields
 com.apama.memorystore.Row row := itable.getRow(key);
 log "*** row after creation:" at INFO;
 log row.getString("stringField") at INFO;
 log row.getInteger("integerField").toString() at INFO;
}
```

General Util bundle

A number of useful miscellaneous components and services can be found in the General Util bundle. These include wrapper objects for primitive types, string manipulation functions, and an extension to `integer.getUnique()` that provides IDs that are unique across restarts of the correlator.

Status publisher

The status publisher component is part of the Status Utils bundle. It provides a convenient action-based interface for handling `com.apama.statusreport.*` subscriptions and generating `Status` events. This component will generally be used by adapter services but is recommended for any service that needs to publish status reports.

Transaction components

The components in the Transaction bundle provide support for implementing transactional boundaries for services. Transactions ensure that a set of services receiving the same input events all complete processing of a given event before any results of that processing are passed on to other services.

User session services

The services in the User Session bundle implement support for interaction between an individual application user's session and a CMF-based application.

Adapter bridging bundle

The Adapter Bridging Utils bundle contains services for configuring bridging the standard Apama status reporting, legacy market data protocols, order management protocols, and an adapter whose service monitors are running in a non-CMF based correlator. Such configurations are often used

when the adapter needs to be run on a particular host for security or network topology reasons, or to support multiple client applications connected to a single adapter.

All of the bridging services share a common architecture:

- A client bridge service instance that
  - Runs in the application correlator
  - Listens for events that should be handled by the adapter service monitors
  - Forwards these events to the adapter correlator over a specific channel.
- A service bridge instance that
  - Runs in the adapter correlator
  - Listens for events routed by the adapter service monitors that should be delivered to the application
  - Forwards these events to the application correlator over the specified channel.

A general principle of the bridging services is that to the application, the client side of the bridge should behave exactly like the adapter it is bridging to. Likewise, to the adapter the server side of the bridge should look exactly like a client application.

The bridging services operate at the level of individual adapter services, which are identified by the `serviceId` values used in status and legacy market data subscriptions, and also in order submissions. If you configure multiple bridging services to several adapters with the same service identifier (for example, several different FIX adapters); each bridging service will forward all requests for that adapter service. Therefore it is important that the adapter services themselves are able to correctly distinguish requests intended for each instance of the adapter service. These are the `com.apama.statusreport`, `com.apama.marketdata` and `com.apama.oms` packages, defined in the `StatusSupport.mon`, `TickManagerSupport.mon` and `OrderManagerSupport.mon` files in the Apama installation, respectively.

The bridging services rely on connections being made on specific event channels between the application and adapter correlators. The CMF does not provide any support for creating inter-correlator connections. It is the responsibility of the application builder to ensure that these connections are maintained, for example through custom actions in an Software AG Designer run profile.

It should also be noted that the current implementation of the bridging services is not parallel-aware, so these services should only be run in the main context.

Details of the status, legacy market data and order bridging services and their configuration are described elsewhere in this document. However, the Adapter Bridging Utils bundle does provide a generic service for creating a bridge for a named adapter service. The bridge is configured by sending a `com.apama.adapter.bridge.ConfigureClientSide` event to the application correlator and a `com.apama.adapter.bridge.ConfigureServerSide` event to the adapter correlator, specifying the service name (which must be the same on both sides) and the channel for the inter-correlator connection. These events will trigger creation of status, legacy market data and order bridges as well as monitoring of the bridge connection itself in both directions.

Adapter status bridge service

The adapter status bridging service is part of the Status Utils bundle. It provides bridging of `com.apama.statusreport.*` events between two correlators, so that an application in the client correlator can monitor the status of an adapter connected to the server correlator. A bridge instance is configured by sending configuration events to the client and server correlators. On the client side, send a `com.apama.statusreport.ConfigureClientSideBridge` event, and on the server side, send a `com.apama.statusreport.ConfigureServerSideBridge` event. See the ApamaDoc for these events for details about their fields. In most cases, the fields of the client and server configuration events for a given bridge instance are identical. However, the `instanceName` field must be unique across all status bridges used by the application.

In addition to configuring the bridge itself, an application should also configure monitoring of the fake adapter connection between the two correlators. Use an instance of the `com.apama.connection.IAFStatusFaker` and an instance of the `com.apama.adapters.IAFStatusToStatusConvertor` service on each side of the connection. If this is not done, the status bridge will not forward any events in either direction because the inter-correlator connection will appear to be down. The generic adapter bridging service described above handles this automatically; it is recommended that applications use this service wherever possible. This should always be the case, as it is no more than the adapter service monitors would need to do if all of the instances were running in the same correlator. Otherwise, see the implementation of the `com.apama.adapter.bridge.ConfigBridge` service in the Adapter Bridging Utils bundle to see how bridge connection monitoring is implemented.

Once the bridge is configured, `com.apama.statusreport.SubscribeStatus` or `UnsubscribeStatus` events routed by the client-side application for the bridged service will be forwarded to the server correlator, where they should be handled by the adapter service as normal. Any `Status` or `StatusError` events generated by the adapter service will in turn be forwarded back to the client correlator where they can be handled by application listeners in the usual way.

The adapter bridge deals with multiple clients bridging to the same server-side adapter service in the following ways:

- Each client correlator should configure a separate bridge connection with a unique `instanceName` field and channel.
- Multiple bridge clients within the same correlator should set up status subscriptions in the usual way; these subscriptions will be reference counted by the client-side bridge.

Note:
The generic bridging service may not be suitable if the application does not want status, market data and order management services to be bridged, although there is generally no harm in setting up bridging for services that are not going to be used. The generic service will also not work for more complex configurations, such as different service names on the client and server sides of the bridge.

DataView manager

DataViews provide a mechanism for exposing a view of application data to remote clients, such as Apama dashboards. CMF-based applications are free to use the standard DataView support

events supplied with the Apama CEP platform and documented in *Developing Apama Applications*. However, the CMF provides an alternative support framework for generating DataViews, based around an event object that applications interact with by calling event actions rather than by routing events. This approach is likely to be more efficient than the standard one in applications that make heavy use of DataViews.

The DataView Manager component is part of the Data View bundle. Typically, an instance of the `com.apama.dataview.DataViewManager` event object will be embedded within another object to create a specific DataView, such as the market data DataViews described in a later section. To create a custom DataView, a CMF application should create a Manager object and then call its `define()` actions to define the characteristics of the DataView:

```
com.apama.dataview.DataViewManager dvManager :=
   new com.apama.dataview.DataViewManager;
com.apama.dataview.DataViewResult dvResult;

sequence<string> fieldNames := ["symbol", "avgPrice"];
sequence<string> fieldTypes := ["string", "float"];
sequence<string> keyFieldsNames := ["symbol" ];

dvResult := dvManager.define(
   "ExampleDataView",     // dataview name
   "Example DataView",    // display name
   "A sample DataView to demonstrate the CMF DataView Publisher",// description
   fieldNames,            // field names
   fieldTypes,            // field types
   keyFieldsNames,      // key fields
   context.current() );   // main context

// The contents of the DataView can then be manipulated using the
// addDataViewItem, addOrUpdateDataViewItem, addOrDeltaUpdateDataViewItem,
// updateFullDataViewItem and deleteDataViewItemByKeyFields actions:

//  Define field values and add a row to the DataView
sequence<string> row := ["EUR/GBP", "0.89"];
dvResult := dvManager.addDataViewItem("admin", -1.0, row);

//  Add a second row
row := ["GBP/USD", "1.52"];
dvResult := dvManager.addOrUpdateDataViewItem("admin", -1.0, row);

//  Update fields in an existing row
row := ["EUR/GBP", "0.88"];
dvResult := dvManager.updateFullDataViewItem(-1, -1.0, row);

//  Remove some rows
sequence<string> key := ["EUR/GBP"];
dvResult := dvManager.deleteDataViewItemByKeyFields(key);
dvManager.deleteAllDataViewItems();
```

If an application needs to completely remove a DataView definition, for example to ensure that it no longer appears in the Apama Scenario Browser, it should call the `undefine()` action on the Manager object.

Helper events for creating DataViews

The CMF provides helper events to make it easier to create DataViews of `com.apama.utils.Params` events.

The `com.apama.utils.SingleParamsDataView` event lets you create a DataView that contains the contents of a single `com.apama.utils.Params` event. It creates a two-column DataView that contains a row for each key/value pair from the supplied `Params` event. Actions defined in the `SingleParamsDataView` event let you update the contents of the DataView, as well as overwrite or clear the contents, or remove the DataView.

The `SingleParamsDataView` event type definition is as follows:

```
event SingleParamsDataView {
   action create(Params display, context mainContext, Params config);
   action update(Params display);
   action clear();
   action onSpawn();
   action delete();
}
```

The `config` argument to the `create()` action can contain the following parameters:

Parameter	Description	Default	Type
`NAME`	Name of the DataView you want to create.	No default. Required.	`string`
`DESCRIPTION`	Description of the DataView being created.	Value of the `NAME` parameter	`string`
`KEY_COLUMN_TITLE`	Title for the key column.	`"Key"`	`string`
`VALUE_COLUMN_TITLE`	Title for the value column.	`"Value"`	`string`
`OVERWRITE`	Execution of the `update()` action overwrites content.	`true`	`boolean`
`OWNER`	Owner of items in the DataView.	`"*"` (all users)	`string`

The code below provides an example of how to use the `SingleParamsDataView` event to create and update a DataView.

```
using com.apama.utils.SingleParamDataView;
using com.apama.utils.Params;
...

integer count := 0;
Params display := new Params;
display.addParam("Key", "Value");
display.addParam("Count", count.toString());

Params config := new Params;
```

```
config.addParam((new SingleParamsDataViewConsts).NAME, "Test Params Dataview");

SingleParamDataView dv := new SingleParamDataView;
dv.create(display, mainContext, config);

// Auto update it every 10 seconds
on all wait (10.0) {
   count := count +1;
   display.addParam("Count", count.toString());
   dv.update(display);
}
```

The `com.apama.utils.MultipleParamsDataView` event lets you create a DataView that contains the contents of multiple `com.apama.utils.Params` events that conform to a schema. The `MultipleParamsDataView` event creates a DataView with columns that match a `com.apama.utils.ParamsSchema` event. Each row represents a `Params` events that matches the `ParamsSchema` event. By default, every field in the schema is a key field but you can specify key fields to identify unique rows. You pass `Params` events to `MultipleParamsDataView` events to add, update or delete selected rows.

The `MultipleParamDataView` event type definition is as follows:

```
event MultipleParamsDataView {
   action create(ParamsSchema schema, context mainContext, Params config);
   action addOrUpdate(Params params);
   action remove(Params params);
   action clear();
   action onSpawn();
   action delete();
}
```

The `config` argument to the `create()` action can contain the following parameters:

Parameter	Description	Default	Type
`NAME`	Name of the DataView you want to create.	No default. Required.	`string`
`DESCRIPTION`	Description of the DataView being created.	Value of the `NAME` parameter	`string`
`KEY_FIELDS`	Sequence of fields the DataView uses as the key. Key fields are used to identify unique items/rows in the DataView.	All fields.	`sequence<string>`
`OWNER`	Owner of items in the DataView.	`"*"` (all users)	`string`

The code below provides an example of how to use the `MultipleParamsDataView` event to create and update a DataView.

```
using com.apama.utils.Params;
```

```
using com.apama.utils.ParamsSchema;
using com.apama.utils.MultipleParamsDataView;
using com.apama.utils.MultipleParamsDataViewConsts;
...
ParamsSchema schema := new ParamsSchema;
schema.addItemMinimal("Name","string","","The Name");
schema.addItemMinimal("Count","integer","0","Count");

Params config := new Params;
config.addParam((new MultipleParamsDataViewConsts).NAME,
                "Test Params Dataview");
// Set 'Name' to be the key field:
config.addParam((new MultipleParamsDataViewConsts).KEY_FIELDS, "['Name']");

// Create the DataView
MultiParamDataView dv := new MultiParamDataView;
dv.create(schema, mainContext, config);

// Add a row
integer count := 0;
Params display := new Params;
display.addParam("Name", "Test");
display.addParam("Count", count.toString());

dv.addOrUpdate(display);

// Auto update it every 10 seconds
on all wait (10.0) {
   count := count +1;
   display.addParam("Count", count.toString());
   dv.update(display);
}
```

See the ApamaDoc for details about the `com.apama.utils.SingleParamsDataView` and `com.apama.utils.MultipleParamsDataView` events.

A Samples

- Installing and running samples 250
- Adapter Support Bundle samples 251
- General CMF samples 269

Samples that ship with CMF are organized into two areas of functionality, the Adapter Support Bundle (ASB) and general CMF features.

Installing and running samples

Before using the samples you must have installed the CMF.

To import samples into your workspace

1. If the **Apama Developer** perspective is not open, open it.
2. From the menu bar, select **File > Import**.
3. Select **General > Existing Projects into Workspace** and click **Next**.
4. For the **Select root directory** field, click **Browse** and navigate to one of the CMF sample directories:
   - The `%APAMA_FOUNDATION_HOME\samples` folder contains general CMF samples.
   - The `%APAMA_FOUNDATION_HOME\ASB\samples` folder contains adapter support bundle samples.
5. Select the top level folder and the wizard displays child sample folders. Uncheck any you do not want to import.
6. If you want to modify a sample and leave the original unchanged, click **Copy projects into workspace**.
7. Click **Finish**.

   The samples are ready to run.
8. Run a sample in one of the following ways:
   - From the **Apama Developer** perspective, right-click the project name and choose **Run As > Apama Application**.
   - From the **Apama Workbench** perspective, choose a sample from the drop-down list of projects and click the **Workbench** button on the launch control panel.

   The Basic and Advanced market data samples contain multiple run configurations, which you can select when launching.
9. To stop a sample:
   a. Close the dashboard.
   b. In the **Console** view, click the **Terminate** button (the red square).

A dialog asks whether you want to terminate all processes associated with the launch.

c. Click **Yes**.

Adapter Support Bundle samples

The `ASB\samples` folder in your CMF installation contains the following samples that demonstrate use of the Adapter Support Bundle.

Basic market data sample

This sample demonstrates the use of MDA functionality. It provides an abstract layer for integration with different market data subscriber APIs of CMF and integrating with DataView items for dashboards. By default, the sample uses the CMF Exchange Simulator to create a set of three simulated feeds defined for the set of symbols listed below. You can also run the sample using an example implementation of an MDA adapter. The exchange simulator is implemented in EPL and generates data in the legacy finance interface events, so it uses the MDA MDBridge component to convert to MDA events. The dummy MDA adapter is a process external to the correlator and sends native MDA events.

Regardless of whether you use the simulator or MDA adapter, the dashboards include the following views:

- The **Datastreams** view displays the set of 5 supported MDA datastream types: BBA, Trade, Depth, Orderbook and Quote. Each of these can be used to connect to a specific symbol on a session for that datastream type.
- The **Sessions** view displays a list of the sessions that have been created and registered with the Session Manager. This includes information such as the name, description, which datasource the session is associated with, whether it is connected and started, and a list of the supported datastreams. The two graphs at the bottom of this tab show the average market data event rates and the latency rates for each session.

Running the sample

You can run the sample using either of the following configurations:

- The **Basic Market Data Sample** configuration uses port 15903 and provides three exchange simulator sessions.
- The **Basic Market Data Sample (dummyMDA Adapter)** configuration uses port 16903 and also provides an example implementation of an adapter that uses the MDA protocol.

To run the sample and select a configuration:

1. In the **Project Explorer**, right-click **Basic Market Data Sample**.
2. From the context menu, select **Run As > Apama Application**.
3. Select the configuration you want to run.

4. Click **OK**.

Exercising the sample

When the sample launches, you first need to start instances of symbols and connect to them:

1. Click the **Datastreams** button.
2. Select any of the tabs.
3. Enter one of the supported symbols or symbol sets.
4. Select a **Session** from the drop-down list.
5. Click **Create instance**.
6. In the output pane, select an instance and click **Connect**.

Default symbol set

By default, the `SimulatorSessions.evt` file defines the following symbols. You can modify or extend the symbol set by altering the `SimulatorSessions.evt` file. Only symbols defined in this event file can be used in the samples, otherwise no data will be generated.

- EUR/USD
- GBP/USD
- EUR/GBP
- USD/JPY
- APMA
- AAPL
- GOOG
- MSFT

Order Manager sample

This sample demonstrates the use of order management functionality. It allows the user or application to do the following:

- Submit one or more orders to market service provider monitors that are loaded in the correlator.
- Receive acknowledgment of the orders.
- Amend or cancel the orders.

Description

The user must submit an order before it is possible to amend or cancel that order. The user cannot amend or cancel a completed order. Once an order has been submitted, modified, or canceled, the market service provider monitor must acknowledge or reject that operation before it allows another operation on that same order. However, a user or application can perform consecutive operations on an order without waiting for replies from the market service provider monitor. The Order Manager sample makes this possible by automatically queueing consecutive requests, and sending them when the market service provider monitor allows modifications to the order.

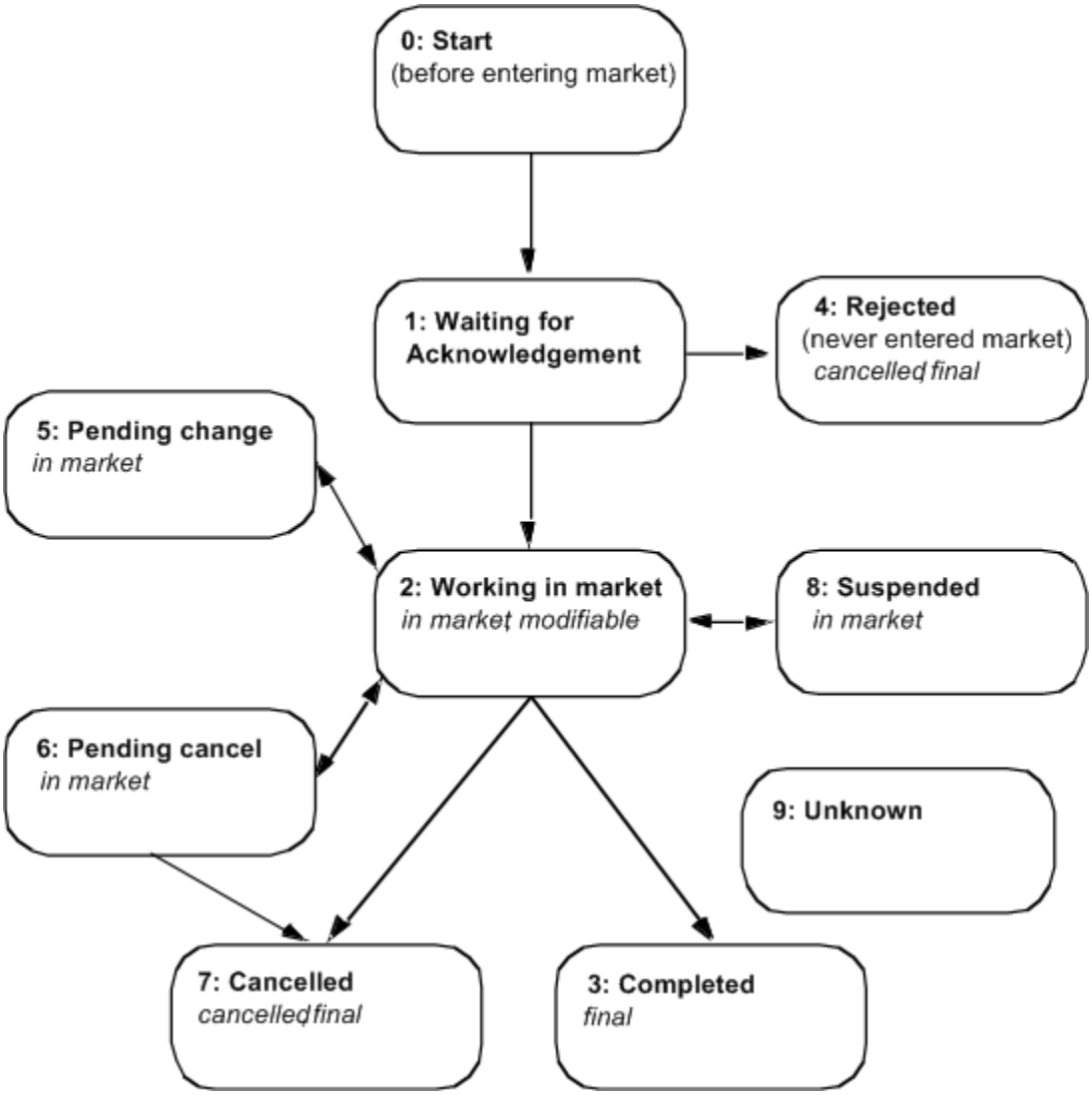
About Order Manager market service provider monitors

Market service provider monitors are outside this sample. The Order Manager sample communicates with these monitors by means of well-known events. These events are defined in the `com.apama.oms` package, which is included in the Legacy Finance Support bundle. For more information, see the ApamaDoc.

During development, a market service provider monitor is a pure EPL simulation market. In a deployed application, a market service provider monitor is part of an adapter that communicates with an IAF process, which communicates with some market.

Order states and status fields

An order is always in one of the following states:



An order can enter the Unknown state from any other state, and can leave the Unknown state and enter any other state. An order is in the Unknown state when the adapter cannot guarantee what state the order is in. This can happen, for example, if the network between the correlator and the market is down.

In the figure, the text in italics shows the status of the order while the order is in that state:

- *in market* — The order is known to the market. Modifications to the order might or might not be allowed.
- *modifiable* — The order is known to the market and modifications to the order are allowed.
- *cancelled* — The trader or the market stopped processing the order before it was fully filled. The order is known to the market.
- *final* — The order is no longer known to the market. Modifications are not allowed.

Each status is a Boolean field in the order status output feed.

An order in the Unknown state can have a status of *in market*, *cancelled*, or *modifiable*.

The order status output feed also supplies Boolean fields for indicating the following for an order:

- *change rejected* — The most recent attempt to modify or cancel the order was rejected by the market. The change might have become impossible because a trade took over the new requested quantity, or it might have been rejected by the market for any other reason.
- *externally modified* — The status message typically gives details for this.
- *is visible* — Some markets consider orders to be known to the market but invisible until a certain condition has been met, for example, stop orders are invisible until the trigger price is reached.

Upon modification or cancellation, the market might modify some fields of an order, including changing fields to values other than those requested. The actual price and quantity that are present in the market are reported in the order status output feed.

Event structures

This sample provides the events listed below in the `com.apama.oms.user` package.

OrderParams

This contains the input values for placement, modification, or cancellation of an order.

Field	Description
`orderIdentifier`	A unique identifier to perform operations on the order.
`serviceIdentifier`	The name of the market service provider monitor to use, or an empty string to use any market service provider monitor.
`brokerIdentifier`	The name of the broker to send the order to. To determine whether you need to specify this field, and obtain more information about the meaning of this field, see the documentation (this is often a `README` file) for the adapter you are using. If the documentation does not mention this field, you do not need to specify it.
`bookIdentifier`	The name of the book the order should be accounted against. To determine whether you need to specify this field, and obtain more information about the meaning of this field, see the documentation for the adapter you are using. If the documentation does not mention this field, you do not need to specify it.
`marketIdentifier`	The name of the market to send the order to. To determine whether you need to specify this field, and obtain more information about the meaning of this field, see the documentation for the adapter you are using. If the

Field	Description
	documentation does not mention this field, you do not need to specify it.
`exchange`	The name of the exchange to send the order to. To determine whether you need to specify this parameter, and obtain more information about the meaning of this parameter, see the documentation for the adapter you are using. If the documentation does not mention this parameter, you do not need to specify it.
`symbol`	The instrument to trade.
`price`	The price to trade at, or `0` for a market order.
`quantity`	The amount to trade. For example, the number of shares to buy or sell.
`side`	`BUY` or `SELL`. Some services also support other values such as `BUY MINUS` or `SELLSHORT`.
`type`	`MARKET` or `LIMIT`. Some services also support other values such as `STOPMARKET` or `STOPLIMIT`. If left blank, the order is placed as a `LIMIT` if a price is specified or a `MARKET` order if no price is specified (or is `0`).
`extraParameters`	Any extra parameters for the service.

OrderStatus

The Order Manager sample generates the order status as an acknowledgment upon placement, modification, or cancellation of an order. This reports all information about each order. Each time there is a change in the order (such as a state or status change), the order status feed provides as a callback.

Field	Description
`orderIdentifier`	A unique identifier to perform operations on the order.
`marketOrderIdentifier`	An identifier supplied by the market, typically unique across all orders in that market.
`symbol`	Identifier for the instrument being traded.
`price`	The price requested either when the order was submitted or the latest modification. A price of `0.0` signifies a market order.
`quantity`	The total number of units, such as shares, to trade, or the number the order has been amended to.

Field	Description
side	The side of the order — BUY, SELL, or some other side supported by the market.
type	The type of the order — MARKET, LIMIT, or some other type supported by the market.
state	The order's state indicated by 0-9. See "Order states and status fields" on page 253.
moneyExecuted	The sum of price * quantity for all fills of this order, or 0.0 if no fills have occurred.
averagePriceExecuted	The volume-weighted average price over all fills, or 0.0 if no fills have occurred. For example, suppose you place an order to buy 100 shares of IBM at up to $10.00 per share. You bought 20 shares at $9.95 and 20 shares at $9.97. The average price executed is $9.96.
lastPriceExecuted	The price obtained per item for the last fill, or 0.0 if no fills have occurred.
lastQuantityExecuted	The number of items traded in the last fill, or 0 if no fills of this order have occurred.
quantityExecuted	The number of items traded so far, or 0 if no fills of this order have occurred.
quantityRemaining	The number of items left to trade in the market as part of this order.
inMarket	true if the order is known to the market.
isVisible	true if the order is visible in the market. Some markets consider orders to be invisible until a certain condition has been met, for example, stop orders are invisible until the trigger price is hit.
modifiable	true if the order can be modified immediately. The Order Manager sample queues any attempts to modify an order when it is not modifiable. The queued modification cannot occur if the order enters a final state before becoming modifiable.
cancelled	true if the order has been rejected or cancelled, possibly before being entered into the market. A cancelled order might have had some quantity traded.
changeRejected	true if the most recent modification or cancellation was rejected by the market. An explanation might be available in the status message field.
externallyModified	true if the order has been modified, for example, by the market or a third party.

Field	Description
`final`	`true` if the quantity specified for the order has been traded, or if the order was cancelled.
`statusMessage`	A message from either the market service provider monitor or the market explaining what has happened. The format and meaning of the message varies from service to service and market to market.
`extraParameters`	Dictionary that specifies additional parameters, which contain more information about the instrument being traded.

OrderManagerInterface

This is an interface which is used to do different order operations. It is created by the `OrderManagerHelperFactory` factory interface.

Operation	Description
`SubmitOrder`	Send the order to the market. The order automatically enters state 1, waiting for acknowledgment.
`ModifyOrder`	Change the order in the market. The order automatically enters state 5, pending change, unless the order is not modifiable at this time.
`CancelOrder`	Cancel the order in the market. The order automatically enters state 6, pending cancel, unless the order is not modifiable at this time. The market can reject a cancellation. Consequently, after an order is in the pending cancel state, it can become cancelled, completed, or, if the cancellation is rejected, working.
`CancelAllOrders`	Cancel all orders. Note that the market might reject some cancellations.
`onOrderStatusAcknowledgement`	This is a callback on order state updates.

Sample code for a factory-to-order operation

```
using com.apama.oms.user.OrderManagerHelperFactory;
using com.apama.oms.user.OrderManagerInterface;
using com.apama.oms.user.OrderStatus;

monitor OMHelperExample {

  context mainContext := context.current();

  action onload() {
    OrderManagerInterface omiFace :=
```

```
      (new OrderManagerHelperFactory).create(mainContext,OrderStatusCb);
    com.apama.oms.user.OrderParams param :=
    com.apama.oms.user.OrderParams("1","EUR/USD",10.1,1000,"BUY","LIMIT",
      "OrderSample","","","OrderManager","","", new dictionary<string,string>);
    omiFace.SubmitOrder(param);
  }

  action OrderStatusCb(OrderStatus orderStatus) {
    log "Received order status for order ID = "+ orderStatus.orderIdentifier +
      " status = " + orderStatus.statusMessage at INFO;
  }

}
```

Multi-Leg Order Manager sample

The Multi-Leg Order Manager sample lets you submit a single order that is actually a container for multiple orders. Each of the contained orders is referred to as a leg.

Description

This sample demonstrates the use of multi-leg order management functionality. It allows the user or application to do the following:

- Submit one or more orders to market service provider monitors that are loaded in the correlator. Each order can contain any number of legs.
- Receive acknowledgment of the orders and/or legs.
- Amend or cancel the orders and/or legs.

About Multi-Leg Order Manager market service provider monitors

Market service provider monitors are outside this sample. The Multi-leg Order Manager sample communicates with EPL from the Legacy Finance Support bundle by using the events defined in the `com.apama.mlom` package. For information about these events, see the ApamaDoc. All events are in the `com.apama.mlom` package.

Event structures

This sample provides the events listed below in the `com.apama.mlom.user` package.

OrderParams

This contains the input values for placement, modification, or cancellation of an order.

Field	Description
`orderId`	A unique identifier to perform operations on the order.

Field	Description
serviceId	The name of the market service provider monitor to use, or an empty string to use any market service provider monitor.
tradeId	The name of the trader.
traderSubId	Additional identification information for the trader.
marketId	The name of the market to send the order to. To determine whether you need to specify this field, and obtain more information about the meaning of this field, see the documentation for the adapter you are using. If the documentation does not mention this field, you do not need to specify it.
marketSubId	Additional identification information about the market to send the order to. To determine whether you need to specify this field, and obtain more information about the meaning of this field, see the documentation for the adapter you are using. If the documentation does not mention this field, you do not need to specify it.
legId	A unique identifier to perform operations on the leg.
symbol	The instrument to trade for the leg identified by the current legId.
price	The price to trade at, or 0 for a market order, for the leg identified by the current legId.
quantity	The amount to trade for the leg identified by the current legId. For example, the number of shares to buy or sell.
side	BUY or SELL. Applies to the leg identified by the current legId. Some services also support other values such as BUY MINUS or SELLSHORT.
type	MARKET or LIMIT. Some services also support other values such as STOPMARKET or STOPLIMIT. If left blank, the order is placed as a LIMIT if a price is specified or a MARKET order if no price is specified (or is 0). This field applies to each leg in the order.
extraParams(dictionary)	Any extra parameters for the service. Applies to each leg in the order.

OrderStatus

The Multi-Leg Order Manager sample updates the order status output as an acknowledgment upon placement, modification, or cancellation of an order. This reports all information about each order, including averages or totals for all legs in a particular order. Each time there is a change in the order (such as a state or status change for any leg), the order status feed provides an update as a callback.

Field	Description
`order_identifier`	A unique identifier to perform operations on the order.
`market_order_identifier`	An identifier supplied by the market, typically unique across all orders in that market.
`symbol`	If all legs in the order are trading the same instrument, this field contains the identifier for the instrument being traded. If all legs are not trading the same instrument, this field is empty.
`price`	The volume weighted average price of all legs in the order. Takes into account any changes to any legs.
`quantity`	The total number of units, such as shares, to trade for all legs in the order. Includes any changes to any legs.
`state`	The order's state indicated by an integer from `0-9`. See "Order states and status fields" on page 253 for a description of what each integer signifies.
`money_executed`	The sum of `price * quantity` for all fills of all legs in this order, or `0.0` if no fills have occurred.
`average_price_executed`	The volume-weighted average price over all fills for all legs, or `0.0` if no fills have occurred.
`last_price_executed`	The price obtained per item for the last fill, or `0.0` if no fills of any legs in this order have occurred.
`quantity_executed`	The number of items traded so far, or `0.0` if no fills of any legs in this order have occurred.
`quantity_remaining`	The number of items left to trade in the market for all legs in this order, or `0.0` if all items have been traded.
`in_market`	Number of legs in the order that are known to the market.
`visible`	Number of legs in the order that are visible in the market. Some markets consider orders to be invisible until a certain condition has been met, for example, stop orders are invisible until the trigger price is hit.
`modifiable`	Number of legs in the order that can be modified immediately.
`cancelled`	Number of legs in the order that have been rejected or cancelled, possibly before being entered into the market. A cancelled leg might have had some quantity traded.
`change_rejected`	Number of legs in the order for which the most recent modification or cancellation was rejected by the market.

Field	Description
	An explanation might be available in the `status message` field.
`externally_modified`	Number of legs in the order that have been modified by anything other than the current sample, for example, the market or a third party.
`final`	Number of legs in the order that have entered the final state. This means the leg was completed, cancelled, or rejected. Note that a leg in the final state can still be updated. A leg that is settled can no longer be updated.
`status_message`	A message from either the market service provider monitor or the market explaining what has happened. The format and meaning of the message varies from service to service and market to market.
`numLegs`	Number of legs in the order.
`numSettled`	Number of settled legs in the order. A settled leg cannot be updated.
`last_quantity_executed`	The number of items traded in the last fill, or `0.0` if no fills of any legs in this order have occurred.
`type`	The type of the order — `MARKET`, `LIMIT`, or some other type supported by the market.
`side`	The side of the order — `BUY`, `SELL`, or some other side supported by the market.
`extraParams`	Dictionary that specifies additional parameters in a *`key`*`,`*`value`* format.

OrderLegStatus

The Multi-Leg Order Manager sample updates the leg status output as an acknowledgment upon placement, modification, or cancellation of a leg. Each time there is a change in a leg (such as a state or status change), the leg status feed provides output.

Field	Description
`leg_identifier`	A unique identifier to perform operations on the order leg.
`market_order_identifier`	An identifier supplied by the market, typically unique across all orders in that market.
`symbol`	Identifier for the instrument this leg is trading.

Field	Description
price	The price requested either when the leg was submitted or the latest modification to the leg. A price of 0.0 signifies a market order.
quantity	The total number of units, such as shares, to trade, or the number the leg has been amended to.
side	The side of the leg — BUY, SELL, or some other side supported by the market.
type	The type of the leg — MARKET, LIMIT, or some other type supported by the market.
state	The order's state indicated by 0-9. See "Order states and status fields" on page 253.
money_executed	The sum of price * quantity for all fills of this leg, or 0.0 if no fills have occurred.
average_price_executed	The volume-weighted average price over all fills of this leg, or 0.0 if no fills have occurred. For example, suppose you submit a leg to buy 100 shares of IBM at up to $10.00 per share. You bought 20 shares at $9.95 and 20 shares at $9.97. The average price executed is $9.96.
last_price_executed	The price obtained per item for the last fill for this leg, or 0.0 if no fills have occurred.
last_quantity_executed	The number of items traded in the last fill for this leg, or 0.0 if no fills of this leg have occurred.
quantity_executed	The number of items traded so far, or 0.0 if no fills of this leg have occurred.
quantity_remaining	The number of items left to trade in the market as part of this leg.
in_market	true if the leg is known to the market.
visible	true if the leg is visible in the market. Some markets consider legs to be invisible until a certain condition has been met, for example, stop orders are invisible until the trigger price is hit.
modifiable	true if the leg can be modified immediately.
cancelled	true if the leg has been rejected or cancelled, possibly before being entered into the market. A cancelled leg might have had some quantity traded.

Field	Description
change_rejected	true if the most recent modification or cancellation of the leg was rejected by the market. An explanation might be available in the status message field.
externally_modified	true if the leg has been modified by anything other than the sample, such as the market or a third party.
final	true if the quantity specified for the leg has been traded, or if the leg was cancelled or rejected. A final leg can still be updated.
status_message	A message from either the market service provider monitor or the market explaining what has happened. The format and meaning of the message varies from service to service and market to market.
settled	true if this leg is final and can no longer be updated.
extraParams	Dictionary that specifies additional parameters, which contain more information about the instrument being traded.

MultilegOrderManagerInterface

The Multi-Leg Order Manager sample provides this interface for different order operations.

Action/Operation	Description
action< OrderParams /*params*/ > StartMultilegOrder	Start submission of a new order. See "Submitting multi-leg orders" on page 265.
action< OrderParams /*params*/ > AddLeg	Add a leg to the current order's set of legs.
action< OrderParams /*params*/ > SubmitMultilegOrder	Send the order to the market. The order automatically enters state 1, waiting for acknowledgment.
action <string /*orderId*/ > StartModifyMultilegOrder	Open order for modifications. See "Modifying multi-leg orders" on page 266.
action< OrderParams /*params*/ > RemoveLeg	Remove the leg identified by the current legId from the order identified by the current orderId.
action< OrderParams /*params*/ > AmendLeg	For the leg identified by the current legId, modify the price, quantity, side, type, or extraParams parameter.

Action/Operation	Description
`action< OrderParams /*params*/ > ModifyMultilegOrder`	Send modified order to the market. The order automatically enters state `5`, pending change, unless the order is not modifiable at this time. See "Modifying multi-leg orders" on page 266.
`action< OrderParams /*params*/ > CancelMultilegOrder`	Cancel the order in the market. The order automatically enters state `6`, pending cancel, unless the order is not modifiable at this time. The market can reject a cancellation. Consequently, after an order is in the pending cancel state, it can become cancelled, completed, or, if the cancellation is rejected, working.
`action< OrderParams /*params*/ > CancelLeg`	Cancel the leg in the market. The leg automatically enters state `6`, pending cancel, unless the leg is not modifiable at this time. The market can reject a cancellation. Consequently, after a leg is in the pending cancel state, it can become cancelled, completed, or, if the cancellation is rejected, working.
`action< OrderStatus /*update*/ > onOrderStatusAcknowledgement`	This is a callback on order state updates.
`action< OrderLegStatus /*legUpdate*/ > onLegStatusAcknowledgement`	This is a callback on leg state updates.

Submitting multi-leg orders

In a multi-leg order, there is data that is common to all legs in the order and there is data that is relevant to each of the legs. To submit an order, you must define the legs of the order.

To submit an order:

1. Set the order's identifier by specifying a value for the `orderId` field of the `OrderParams` event.
2. Call the `StartMultilegOrder` operation.
3. For one leg, add the corresponding value to the field of the `OrderParams` event.
4. Invoke the `AddLeg` operation.
5. Repeat Step 3 and Step 4 for each leg.
6. Invoke the `StartModifyMultilegOrder` operation for the same `orderId`.

After invoking the `StartModifyMultilegOrder` operation, the sample will acknowledge the order status and/or leg status (as a callback) for each order/leg update.

Modifying multi-leg orders

To add a new leg, modify an existing leg, or remove a leg:

1. Set the `orderId` field to the identifier of the order you want to modify.
2. Invoke the `StartModifyMultilegOrder` operation.
3. Follow the appropriate steps for modifying, adding or removing a leg:

   To modify a leg:

   a. Set the `legId` field to the identifier of the leg you want to modify.

   b. Assign updated values to the fields of the `OrderParams` event.

   c. Invoke the `AmendLeg` operation.

   To add a leg:

   a. Specify a unique value for the `legId` field of the `OrderParams` event and specify the appropriate values for the other fields.

   b. Invoke the `AddLeg` operation.

   To remove a leg:

   a. Set the `legId` field to the identifier of the leg you want to remove.

   b. Invoke the `RemoveLeg` operation.
4. Repeat all substeps in Step 3 for each leg you want to modify, add, or remove.
5. Invoke the `ModifyMultilegOrder` operation.

Working with multi-leg orders

Order identifiers and leg identifiers must be unique in the scope of the Multi-Leg Order Manager sample.

At any one point in time, each order has one status and one state. Likewise, each leg has one status and one state. The status is a text description of where the order or leg is in the order process. The state is a number, from 0 through 9, that indicates more precisely the location of the order or leg in the order process. See "Order states and status fields" on page 253 for further information. All information in that topic applies to legs as well as orders.

When the legs in an order are in different states, the state of the order is unknown (`9`). When the legs in an order are all in the same state, the state of the order is the same as the state of the legs.

The Multi-Leg Order Manager sample maintains an order book that contains information about each of the orders it has submitted. The order book indicates the state and status of each order placed. In the order book, an order identifier distinguishes each order from every other order placed from that sample. You always specify the order identifier as a parameter when you amend, cancel, retrieve, or clear an order.

The Multi-Leg Order Manager sample also maintains a leg book for each order. The leg book contains information about each of the legs submitted as part of the order. In the leg book, a leg identifier distinguishes each leg from every other leg in that order. You always specify the leg identifier as a parameter when you amend, cancel, retrieve, or clear a leg.

Updating final orders

When a leg is final, it has been completed, cancelled, or rejected. Sometimes, it is necessary to update a leg that is final. The Multi-Leg Order Manager sample continues to listen for update events for final legs. If an update event occurs, it performs the update.

If you do not want to allow updates to a final leg, you must specify that the leg is settled. To do this, your client must send a `LegUpdate` event to the Multi-Leg Order Manager sample and specify `true` for the `settled` field. `LegUpdate` events are defined in the `com.apama.mlom` package. For additional information, see the ApamaDoc.

Sample code for a multi-leg order operation

```
using com.apama.mlom.user.MultilegOrderManagerFactory;
using com.apama.mlom.user.MultilegOrderManagerInterface;
using com.apama.mlom.user.OrderStatus;
using com.apama.mlom.user.OrderLegStatus;
using com.apama.mlom.user.OrderParams;

monitor MultilegOrderManagerHelperExample {

  context mainContext := context.current();

  action onload() {

    MultilegOrderManagerInterface iface :=
      (new MultilegOrderManagerFactory).create(mainContext, OrderStatusCb,
        OrderLegStatusCb);

    /****** To place an order with Order_1 identifier ******/

    //start multi-leg operation
    OrderParams param := OrderParams("Order_1","MarketSimulator",
      "","","","","","",0.0,0.0,"","","",new dictionary<string,string>);
    iface.StartMultilegOrder(param);

    //add leg
    param := OrderParams("Order_1","MarketSimulator",
      "","","","","Leg_1","INSTRUMENT_1",1.23,700.0,"SELL","MARKET","",
      new dictionary<string,string>);
    iface.AddLeg(param);

    //place an order
    param := com.apama.mlom.user.OrderParams("Order_1","MarketSimulator",
      "","","","","","",0.0,0.0,"","","",new dictionary<string,string>);
    iface.SubmitMultilegOrder(param);

    /****** END ******/

    /****** To modify the order ******/
```

```
    param := OrderParams("Order_1","MarketSimulator",
      "","","","","","",0.0,0.0,"","","",new dictionary<string,string>);

    //Start modify operation -> for Order_1 identifier
    iface.StartModifyMultilegOrder("Order_1");

    //Modify leg -> changed type and quantity
    param := com.apama.mlom.user.OrderParams("Order_1","MarketSimulator",
      "","","","","Leg_1","INSTRUMENT_1",1.23,900.0,"SELL","LIMIT","",
      new dictionary<string,string>);
    iface.AmendLeg(param);

    //Submit modify
    param:= com.apama.mlom.user.OrderParams("Order_1","MarketSimulator",
      "","","","","","",0.0,0.0,"","","",new dictionary<string,string>);
    iface.SubmitMultilegOrder(param);

    /****** END ******/

    /****** To cancel the order leg ******/

    param:= com.apama.mlom.user.OrderParams("Order_1","MarketSimulator",
      "","","","","Leg_1","",0.0,0.0,"","","",
      new dictionary<string,string>);
    iface.CancelLeg(param);

    /****** END ******/

    /****** To cancel the order ******/

    param:= com.apama.mlom.user.OrderParams("Order_1","MarketSimulator",
      "","","","","","",0.0,0.0,"","","",new dictionary<string,string>);
    iface.CancelMultilegOrder(param);

    /****** END ******/

  }

  action OrderStatusCb(OrderStatus orderStatus) {
    log "Received order status for order ID = "
      + orderStatus.order_identifier +" status = "
      + orderStatus.status_message at INFO;
  }

  action OrderLegStatusCb(OrderLegStatus legStatus) {
    log "Received leg status for leg ID = "+ legStatus.leg_identifier
      +" status = " + legStatus.status_message at INFO;
  }

}
```

Configuration Service

This sample demonstrates the most commonly used Configuration Service operations:

1. Creating and opening a configuration store backed by an on-disk Memory Store database.
2. Defining and opening a table in the configuration store.

3. Iterating over the rows of an existing table.
4. Adding, updating and removing configuration table rows.
5. Using a wrapper object with the generic configuration row helper to provide a more natural event-based interface to configuration rows.

Many configuration service operations are asynchronous and their success or failure can be signaled by callback actions or routed events. The sample demonstrates both techniques. The tables in the configuration store are exposed as DataViews to simplify visualization and modification of the configuration store.

The first time you run the sample, an on-disk database will be created with no rows in it. Any rows you add while running the sample will be persisted to the database and re-loaded when you run the sample again. To add a row, enter values in the three text fields, select the capabilities and click **Add > Update**. The sample simply demonstrates how configuration values can be stored and retrieved, the values are not used to actually configure anything.

General CMF samples

Samples in the `samples` folder of the CMF installation include those described below.

Advanced market data sample

The Advanced market data sample demonstrates the use of the Aggregator and Synthetic CrossRate services as well as basic MDA functionality. Most of the functionality is duplicated from the Basic market data sample. The **Synthetic Sessions** button displays a page that can be used to create either an Aggregator Session or a CrossRate Session from the existing selection of Sessions.

Note:
To operate correctly, the Aggregator must be created with three different simulated sessions from the set of sessions in the drop down lists.

Once the Aggregator has been created, you can connect to the Aggregator Session from either the **BBA** or the **Aggregated Book** tabs. The **Aggregated Book** tab can be used with the Aggregator or Synthetic CrossRate sessions. Similarly, the **Synthetic Cross Rate** tab can be used to create a Synthetic Cross Rate Session from one of the underlying Sessions (although in practice, you can use the actual Synthetic Cross Rate service to connect to two seperate underlying Sessions).

For Synthetic CrossRate sessions, different zymbol names should be provided for front and back legs that share a common currency. For example, the default values are GBP/USD and USD/JPY which share USD as a common currency to create the synthetic currency GBP/JPY. The two tables below show the Best Bid/Ask (BBA) for the two underlying legs, and the resulting cross rate BBA that was calculated.

By default, the `SimulatorSessions.evt` file defines the following symbols. You can modify or extend the symbol set by altering the `SimulatorSessions.evt` file. Only symbols defined in this event file can be used in the samples, otherwise no data will be generated.

- EUR/USD

- GBP/USD
- EUR/GBP
- USD/JPY
- APMA
- AAPL
- GOOG
- MSFT

Market Monitor

This sample provides an example of a market monitor dashboard and shows how to use the depth publisher service, which randomly generates depth events. The associated dashboard displays generated data for every subscribed symbol from the depth DataView. This dashboard has only a few objects and no functions, but demonstrates how to build dashboards.

In the **Market** pane, select a symbol to view bid and ask details.

The `init.evt` file in the `events` directory does the following:

- Initializes the CMF.
- Initializes the DataViews for the market monitor.
- Configures symbol simulation.

The market monitor dashboard uses the depth and tick DataView that is automatically generated by the CMF (for all subscribed depth and tick symbols) and can display top-line depth, a detailed break down of depth (top five levels), as well as last traded price and quantity. Select **Help > Command Line Options** from the dashboard menu to see a list of command line options. You can start the project from the command line or enter options in the run configuration and run the sample from Software AG Designer.

Market Simulation

The Market Simulation sample demonstrates how to use the order publisher state container component. This sample uses the CMF exchange simulator to create simulated markets for several symbols. A simple order flow generator drives the simulations, which will generate a stream of price and trade data for each subscribed symbol. As in the Market Monitor sample, the market depth data is displayed on the dashboard.

This sample provides simulated market data for each symbol and the dashboard also shows trades. In the Market pane, select a symbol to view bid and ask details.

Order Blotter

The Order Blotter sample demonstrates how to use the order monitor state container component. This sample extends the Market Simulation sample. The market simulator and an order flow generator are used to provide simulated markets for several symbols. However, in this sample the dashboard is used to visualize the order flow, rather than the market data derived from it. The dashboard also allows you to interact with orders by amending or canceling them. Select an order to enable the modification buttons.

Market Simulation in the Order Blotter sample is the same as the Market Simulation sample. As it generates order into an exchange, this sample will be monitoring the orders, rather than monitoring the depth from the exchange.

The Order Blotter provides a DataView with all of the active orders as well as the last 50 finalized orders. The number is limited to prevent too many DataViews.

Similar to other samples, the `init.evt` file in the `events` directory does the following:

- Initializes the CMF.
- Initializes the DataViews for the market monitor.
- Configures symbol simulation.

The Order Blotter sample demonstrates the following:

- How to monitor orders with a DataView
- How to amend and cancel orders
- How to create an interactive dashboard for amending and cancelling orders with simple validation

OrderState Container

This sample demonstrates the use of the CMF `OrderPublisherStateContainer` and the `OrderReceiverStateContainer` objects. Seperate monitors and associated dashboards are implemented to show how each of these OrderState container objects can be used in your application, and how they can interact with each other.

Simple OMS

The Simple OMS sample demonstrates Market Monitor, Order Blotter, Order Entry, and Order Simulation functionality. This sample shows a very simple order management system - the dashboard supports monitoring market price and trade data and submitting, amending and cancelling orders.

The Simple OMS sample uses a more sophisticated order flow generator that attempts to model the impact of user orders on the market, gradually shifting the market up, down, and sideways. It also allows the simulation to be stopped and started to allow for easier interaction with the simulated market during testing.

To exercise the sample, select a symbol in the **Market** pane.

Simulation and order entry are implemented as follows:

- Order simulator

  Structurally similar to the market simulator used in other samples, the order simulator sends out orders to generate a market using the CMF empty exchange. It has some extra sophistication to mimic market impact, gradually drifting markets (up/down/sideways), and includes a dashboard to start and stop the simulation, allowing for interaction while testing.

- Order entry

  EPL and a DataView allow for the entering orders from the dashboard. This sample uses the Depth DataView (also used by the Market Monitor dashboard) to provide the end user with a selection of symbols for entering orders.

SOR sample

The SOR sample provides an example of building a strategy using the SOR framework. It listens for new orders, spawns a strategy and the spawned monitor dies when the strategy completes the order. This example does not generate orders, create an aggregated book, or initialize a dashboard. The code shows how to use the SOR framework.

Trading Algorithms

The Trading Algorithms sample demonstrates the following algorithms:

- Iceberg

  This algorithm takes in a symbol, price, side (buy/sell), show quantity, and a total quantity. It ensures that the show quantity is placed on the market, in total, at any point in time with the current symbol, price, and side. Extra logic delays the placing the new order after execution to allow for any subsequent executions that are about to occur. The delay handles the circumstance where multiple orders on the market all execute simultaneously.

- PegOrder

  More sophisticated than the Iceberg algorithm, PegOrder has a minimum and maximum show quantity (and randomizes between the two), a timeout, a side, a symbol, and a total quantity as part of its parameters. It will work the required quantity (in one or more orders) at the best price in the market on the correct side. When the best price shifts (more aggressively or less) it will cancel all of the orders and join the new best price appropriately. It uses the timeout to wait for the market to stabilize before it cancels and submits new orders.

You can observe the algorithms by stopping the simulator and entering direct trades or by having instances of the algorithms trade against each other.

To enter a direct trade

1. Select a symbol from the **Symbol** drop down list.

2. Optionally, change the quantity and/or price.
3. Click the appropriate **BUY** or **SELL** button.

To use one of the algorithms to trade

1. Click the **Peg Order** or **Iceberg** button and a secondary dialog displays.
2. Select a symbol from the **Symbol** drop down list.
3. Optionally, change the quantity, price, or parameters for the algorithm.
4. Click the appropriate **BUY** or **SELL** button.

To amend or cancel an order

1. In the **Orders** pane, select a trade.
2. Click **Amend** or **Cancel**.

B Legacy Finance Interfaces

- Understanding event protocols 276
- The actors in com.apama.oms 277
- Market data package overview 288

This chapter describes events that are related to certain finance adapters purchased as separate products from Apama. The events described are in the following packages:

- `com.apama.oms`
- `com.apama.marketdata`
- `com.apama.statusreport`

Understanding event protocols

Apama EPL follows the actor model of asynchronous computation. The actor model is based on humans: Humans are actors and events represent the exchange of words and deeds between them. Similarly, each monitor instance plays the role of actor and events are passed as messages. Thinking about event protocols in this way will help you understand them more intuitively.

For example, suppose we model an election in Apama EPL. The actors would include:

- The nominators of candidates
- The candidates
- The voters
- The vote counters
- The independent election monitors

Necessary messages to pass between these actors might include:

- It is time to nominate candidates
- I nominate candidate X
- I, candidate X, will/will not run as a candidate
- The time for nominations has passed
- The time to vote has begun
- I vote for X
- The time to vote has ended
- The winner is X
- The election was unfair and is void

This immediately gives you a feel for the interaction, but additional rules are required to conform to electoral process:

1. Temporal Rules - When can events be sent?
    - The winner cannot be declared until all the votes have been tallied.

- Votes cannot be cast after the election has closed and the result cannot be determined until the election has closed.
- Candidacy acceptance/rejection must follow a nomination

2. Authority Rules - Who can send and receive which events?
   - The independent monitors cannot themselves vote.
   - Voters cannot pass their votes to anyone other than the counters.
   - Only actors nominated can accept/reject candidacy.
3. Cardinality Rules - How many events of each type will be sent?
   - At most one vote can be cast per voter.
   - At most one winner can be declared.
   - Either one acceptance or rejection of candidacy will be sent in response to each nomination.
4. Value Constraint Rules - What content can be sent in events?
   - You cannot vote for a candidate that was not nominated, or that did not accept the candidacy.
   - The winner must be one of the actors who accepted a nomination.

Especially for Capital Markets applications, an understanding of these four types of rules is fundamental to understanding Apama EPL protocol. It is not enough to have the event definitions and to know the actors.

The actors in com.apama.oms

The `com.apama.oms` package supports order publishers, receivers, and monitors, which are implemented in Order State Containers. See "Order state containers" on page 86 for more information on the implementations.

Order publishers

Apama Order Publishers deal with submitting orders (such as from an external source or an algorithm), to an IAF adapter or to another internal service (such as the Risk Firewall).

Order receivers

Apama order receivers can be one of the following:

- Bridges order operations between the publisher and some downstream system.
- Deals with order operations immediately, such as internal crossing pools.

The order receiver will typically live in an adapter service monitor and will send order operations out to an IAF adapter and receive updates back from it too. In this second mode it is important to

realize that certain race conditions may occur when the publisher sends an amendment, the receiver may send a trade report before the amendment's confirmation or rejections. It also means that multiple operations may be occurring in parallel, such as the publisher may send two amendments, without receiving a response to the first one.

Note:
Please note that Apama finance interfaces do not support recoverability and busted fills.

Order monitors

Order monitors are entirely passive actors: they listen, but they do not pass any of the events involved in the protocol. While they are not described further in the rest of this document, they must follow the rules described for the other two, in order to correctly surmise the state of the order.

The events in com.apama.oms

Order operations, or events, include those necessary to create, amend, and cancel orders. The `OrderManagerSupport.mon` in the Legacy Finance Support Bundle contains `com.apama.oms protocol` definitions for the following events:

- `NewOrder`: sent from a publisher to submit an order to a receiver.
- `AmendOrder`: sent from a publisher to the receiver as a request to amend an order.
- `CancelOrder` sent from a publisher to the receiver as a request that an order be canceled.
- `OrderUpdate` sent from a receiver to the publisher to notify of any change to the state of an order.

Orders contain a composite destination key, which will determine where they are sent. The key includes a `service` identifier, a `market` identifier and an `exchange` identifier. Not all destination details are present in all events as primary fields, so the extra parameters `SERVICE_NAME`, `Market` and `Exchange` must be added to a number of the events in the protocol.

Temporal rules

The temporal rules for the `com.apama.oms` interface include the following:

Order updates for execution changes and for state changes should be temporally separated

To simplify the user's logic for any given update and to clearly separate the two types of change, an `OrderUpdate` event should either convey an execution, or a state change--but not both. If a fill causes an order to become final, two messages can be sent, first the execution, then the `OrderUpdate` with the `inMarket` flag set to `false`. Although by all packaged adapters do not obey this rule, it is a best practice.

No `OrderUpdate` events can be sent for a particular order before the `NewOrder` event

The `com.apama.oms` interface has not been designed to allow unsolicited feeds of order status to be processed. Order updates will only be sent after a new order event has been sent containing that order identifier. Adapters should also enforce this rule.

No `AmendOrder` or `CancelOrder` events can be sent can be sent before a `NewOrder`

The `AmendOrder` and `CancelOrder` events only make sense when they have been preceded by a `NewOrder` event with the same identifier. No publisher should send them before a `NewOrder`.

Amend order events can be sent as soon as a new order has been sent

The receiver must not assume that the publisher will wait for confirmation that the order is in the market. Amendments can be confirmed before the order itself is confirmed as in market. One example is where an adapter restricts the number of orders per second that can be sent to the exchange. In this situation, `NewOrder` events may be queued at the service monitor. Any `AmendOrder` sent corresponding to the order can be applied to the queued `NewOrder` directly. If this happens, the adapter will confirm the amendment by sending an `OrderUpdate` with the new price and quantity, before the user receives an order update with `inMarket` set to `true`.

Order receivers should acknowledge receipt of all order operations

As mentioned above, order receivers come in two types, some actually process and match orders, others, such as adapters, forward the requests on to another receiver downstream. If the receiver is actually processing the order, it can send an update immediately. If it is not, and is only forwarding on the request, the receiver should send an `OrderUpdate` event back to the publisher as an acknowledgment. This results in more informative status history in order blotters.

The following table contrasts the flow with and without acknowledgments:

Correct flow for a receiver	Flow without acknowledgments
order sent	order sent
order acknowledged by adapter	order in market
order in market	order amendment sent
order amendment sent	
order amendment acknowledged by adapter	

If the adapter never sent an acknowledgment, we would not know if it had received it or not. If we never got a response to an amendment, we would not know if the adapter had failed to receive the order amendment, or whether it was a downstream issue.

Another reason for doing this relates to new orders, as described in: "Cardinality rules" on page 281.

These acknowledgments of receipt for amend and cancel requests should be implemented by re-routing the latest `OrderUpdate` events with a change to only the status field. Note this in the case of a `NewOrder` event, an initial `OrderUpdate` with `inMarket` set to `false` and with quantity and price set to those in the `NewOrder` event.

These acknowledgments of receipt of amend and cancel requests should be implemented by re-routing the latest `OrderUpdate` with a change to only the `status` field. Not that in the case of a `NewOrder` an initial `OrderUpdate` with `inMarket` set to `false` and with quantity and price set to those in the `NewOrder` event.

No fills should be sent to the publisher until the order is `inMarket`

Orders are not considered tradeable until they have been marked as in market. This means that a trade should only be conveyed in an `OrderUpdate` event with `inMarket` set to `true`. Note that because updates containing fills and state changes should not be combined, receivers should not send the fill in the first `OrderUpdate` with `inMarket` set to `true`, but publishers should interpret such an `OrderUpdate` as a trade if they do catch it.

No order operations or order updates can be sent after an order is final

Once an order is considered final, the order receiver must send no more order updates and the publisher should send no more amend or cancel operations. This allows both the receiver and the publisher to tear down any listeners they have to free up resources from memory and hypertree entries.

When is an order final?

An order is final in either of the following cases:

- A canceled event has been received
- It was in the market but no longer is

Either of the following `OrderUpdate` events should result in an order being finalized:

- `OrderUpdate(cancelled = true)`
- `OrderUpdate(inMarket = true) -> OrderUpdate(inMarket = false)`

If all other rules are followed, these are the only two conditions that should result in finality. They might occur independently or at the same time. Typically only orders that have been fully filled will not terminate with the `canceled` flag set to `true`, but order publishers should not assume that the absence on a canceled flag means that an order was fully filled. That should only be determined by looking at the `qtyRemaining` field.

There is one more condition for finality. If an order has never been marked in market and has a `qtyExecuted` greater than 0, then the order publisher should interpret this as the order having been filled and canceled. Note that this can only happen if other rules are broken; namely that updates to an order's state and executions on that order should not be conveyed in the same order update and that no fills should be sent until the receiver has been notified that the order is in market.

A quantity remaining being equal to 0 does not mean that an order is final. This is to avoid problems for adapters where execution reports and order status are not synchronized. In such adapters, an amendment that increases the size of an order may be in flight when a trade taking `qtyRemaining` to 0 is received. After this event, the amend event is confirmed, which takes the `qtyRemaining` back up above 0. The order was never final, but the `qtyRemaining` was reported as 0. In addition to this, certain order types, such as reservation orders in the firewall can have their trades reversed.

Authority rules

These are the authority rules for `com.apama.oms`:

- Only publishers should send new orders

  Only a publisher actor should send `NewOrder` events.

- Only the publisher that sent a new order event can amend or cancel that order

  To simplify the implementation of order publishers, only the publisher that sent a particular `NewOrder` event can send `AmendOrder` or `CancelOrder` events with a matching order identifier. The publisher that creates an order owns the order.

- Only receivers update orders

  Order receivers publish the current state of an order. Publishers ask to make changes using new, amend and cancel order events.

- Only one receiver can send updates for any particular order

  Only one order receiver can give the definitive status of an order. This means only one order receiver can send updates for it. The usual way to ensure this is to use the service identifier to determine which receiver should be used. Other actors must not send order updates for an order owned by another receiver.

- Order monitors send no events

  Order monitors listen, but they do not pass any of the events involved in the protocol. While they are not described further in the rest of this document, they must follow the rules described for publishers and receivers, in order to correctly interpret the state of the order.

Cardinality rules

Cardinality rules include the following

- Each new order can have zero or more updates

  The `com.apama.oms` interface does not guarantee a response to `NewOrder` events. New orders contain a service identifier, and if no order receiver is listening to that identifier, the order will be ignored. Due to this limitation, order publishers should setup a timeout when submitting new orders. If they receive no update in that amount of time, they should send a cancel order event. To reduce the risk off false timeouts, receivers should always take care to acknowledge receipt of new order events as soon as they receive them.

- If there is at least one update to an order, the last one must imply the order is final

  When a receiver processes a `NewOrder` event, the last update it sends must imply that an order is final. See the rules for determining when an order is final in "Temporal rules" on page 278. The receiver must send exactly one update telling the publisher the order is final. Then, the receiver must not send any updates for that order. If the receiver does not send an order update implying finality, the publisher will never know that the order is final. This will cause resource leaks. If the receiver sends an update after sending one that implies the order is final, the update will be ignored. Thus, it is vital that the update implying finality is the last update sent for that order.

- For each new order, multiple amend and cancel events can be sent

  As soon as a `NewOrder` event has been sent, that publisher can send as many amendments and cancellation operations as they like. Note that the temporal rules means that none can be sent after an order is final. If any amend or cancel events are sent after an order is final, they will be silently ignored. `AmendOrder` events can be sent after `CancelOrder` events. For example, cancel events might be rejected because some exchanges will not allow you to cancel an order that has been in the market less than a specified time. If an amend event is received after a cancel event, it will only be applied if the cancel event is rejected. The order must be considered amending until one of the following occurs:

  - Cancellation is confirmed
  - Cancellation is rejected, and:
    - Amendment is confirmed, or
    - Amendment is rejected

Value constraint rules

This topic describes `com.apama.oms.NewOrder` event fields. The constraints on the fields in the events are best explained by detailing each field of each event individually.

- `NewOrder`

  the new order event submits an order from a publisher to a receiver. Its fields specify details of the order, such as where it is to be sent, how big it is and whether it is a buy or sell order.

- `orderId`

  The order identifier specified in the new order links all events for a particular order together. All order updates, amends and cancels pertaining to this order must have the same value for `orderId`. The interface does not impose any restrictions on the `orderId` value, other than that it must be a unique value within a correlator instance. Third-party systems to which adapters connect, may have arbitrary restrictions on the length, format and uniqueness of order IDs so adapter implementations may need to map the order ID supplied by an order publisher to a value that will work with a specific third-party system. The publisher should ensure that the identifier is globally unique, preferably across correlator runs. One common technique is to use `integer.getUnique()` combined with `currentTime`, to generate a unique ID at runtime. Receivers should be selective. When they receive a new order event, they should only listen

for amend and cancel events where both the order identifier and service identifier match. This will avoid the complications that can arise if a badly-behaved publisher only ensures identifiers are unique per receiver.

- `symbol`

  This is the symbol for the instrument that the publisher wishes to trade. The type of identifier used will differ from receiver to receiver and no normalization is performed. The publisher must ensure they use the right identifier. Some a adapters use RIC codes, some CUSIPs, some ISINs and others use their own symbology. Publishers should publish symbols with the correct case (upper/lower). Receivers can optionally choose to ignore case if it is safe to do so. The documentation for each receiver implementation should detail whether it is case sensitive or not.

- `price`

  This is a floating point field containing the price for an order. Interpretation of this field is adapter specific and will vary with order type. Commonly `MARKET` type orders will ignore this field and `LIMIT` orders are submitted using this price. Stop orders, where supported, may either use this field for the stop trigger price, or for the limit price of stop limit orders. The exact behavior for stop orders is receiver specific, if supported at all.

- `side`

  The side of the market the order is being submitted against. Publishers must send `BUY` or `SELL`, in upper case. Receivers should accept sides in a case insensitive way, but do not have to. Some adapter's order receivers will accept additional sides, such as `SHORT SELL`. Please note these are not really sides (an order book only has two of those), they are really modifiers for accounting and compliance purposes and should be conveyed as extra parameters. There is no guarantee that any given publisher, monitor or receiver will understand any type other than `BUY` and `SELL`. CMF components such as the position service may not handle such sides correctly.

- `type`

  The type of the order can be any string, but order receivers will only understand a small set of values. These are adapter specific. Almost all receivers support `MARKET` and `LIMIT` order types. Publishers should send these types in upper case and receivers should a accept them in a case insensitive manner. The documentation of a receiver implementation must document all order types supported and any requirements in terms of extra parameters. Common types used include `STOP`, `STOP MARKET`, and `STOP LIMIT`.

- `quantity`

  This integer field indicates how much of an instrument should be executed. The exact meaning differs from receiver to receiver, especially in foreign exchange where some receivers use this field to represent a number of contracts, others use it to represent a cash figure. For example, trading 1M EUR/USD might require a quantity of 1000000, or if in lots of 100,000 it might require 10, or it might simply be 1. Please see the receiver-specific documentation in the adapter readme file for details.

- `serviceId`

This field specifies which order receiver is to process the new order. It is used to construct a composite destination key that determines where an order gets sent. Publishers can publish to any service identifier they like. Receivers must only listen on one identifier and that must never be an empty, , identifier. The publisher has no guarantees that it will receive a response to a new order event. No receiver may be listening on that identifier.

- `brokerId`

  This field is seldom used. Some receivers are not direct connections to exchanges, but are a way of sending orders to a network of brokers. In that situation, the broker identifier can be used to instruct the adapter which broker should handle the order. Receivers that support this field should give further details in their documentation. When an adapter supports the broker id field, it can sometimes be considered part of the composite destination key, for example, in CMF firewall rules. A few older adapters use the `brokerId` to specify the user id to use when connecting to the OMS system. This is an incorrect use of this field. The correct way for adapters to determine the user id for the OMS system is to use the `ownerId` field. This `ownerId` corresponds to the Apama user ID, the dashboard user ID. As it is often impossible for the user IDs in Apama and the OMS to match and because they are not always in a one-to-one mapping, the adapter should provide a means to configure Apama user IDs to OMS system user IDs. See the `ownerId` description for more details.

- `bookId`

  This field is also seldom used. It is a field used for accounting purposes when sending orders through order management systems that support the concept of a book. Adapters that support this field should document details of its use.

- `marketId`

  The market identifier is the part of the composite destination key that determines where an order gets sent. Its original purpose was to allow the user to send orders to different kinds of market, for example, spot, forward and futures markets. Many receivers still use it for this purpose. In other receivers, which allow more than one instance to be run at once, such as the QFIX adapter, the market identifier is used to specify which instance the order is for. The instances themselves are normally configured using a `SessionConfig` event. Details of such events are to be found in the documentation for the relevant receivers.

- `exchange`

  The exchange is part of the composite destination key that determines where an order gets sent. It is used with receivers that connect to technologies that give a uniform interface to many exchanges. For example, it would be used to send orders to the LSE, NYSE, eCBOT or Xetra. Details on how this field is used can be found in the documentation for each receiver.

- `ownerId`

  The owner identifier field contains the user ID that an order should be associated with. Typically this will match a dashboard user name. Order receivers must not expect the user identifiers used by external systems to be in this field. As these two ids will seldom match, the receiver must implement a lookup service to map dashboard user to the downstream user id. This mapping will be configured with receiver specific events. The receiver must document the technique it uses.

- `extraParams`

  The new order event contains this dictionary parameter to allow receivers to be provided with additional details about an order, such as their duration (IOC, FOK, GTD) or parameters for complex order types, such as stop limit orders. Details about supported parameters should be given in the documentation of each receiver, mentioning which are optional and which mandatory.

Connection failure handling in order receivers

Apama has developed a set of best practices for order receivers (typically adapter service monitors) to follow when dealing with disconnections or other errors in the underlying adapter. While these are not strictly part of the `com.apama.oms` API or protocol they should be followed where possible by any order receiver component.

Best practices for handling connection failures include the following:

- The order receiver should use the `IAFStatusManager` to track the state of its connection to the IAF and the IAF's connection to the remote service.
- If the receiver becomes disconnected from the IAF or the IAF becomes disconnected from the remote service, the receiver should:
  - Reject all new orders
  - Send an update with the `unknownState` flag set to true for all orders where the state is undefined while the connection is down. When the connection is restored these orders should be updated again to report the new known state of each order
  - If the state of the order is known, for example, the IAF has lost its connection to the remote service and the service is configured to cancel all orders on disconnect, the known state of the order should be reported as usual
- If an update for an unknown order is received from the adapter, the receiver should log a warning or otherwise alert the user to this unexpected event.

Test cases for order receivers

A minimum recommended set of test cases should be implemented for all order receivers to ensure compliance with the rules listed in this document. The following table describes these test cases:

Test	Details
Filter on `serviceID`	The receiver should always filter on the `serviceID` on new, amend and cancel order events. Just filtering on `serviceID` for the last two is not sufficient. The test should also explicitly check that empty `serviceID` is not matched.
Destination key `extraParams` are required	All order update events must contain the correct values for the `SERVICE_NAME`, `Exchange`, and `Market` parameters. The latter

Test	Details
	two are allowed to be missing if not used by the receiver, `SERVICE_NAME` must always be present.
No order updates should be sent after the order is final	The tests for the adapter should all make sure that the adapter sends no updates after the order is final. For example, after an event that indicates canceled, or after one that indicates in market and then one that indicates not in market.
Adapter handles amend and cancel as soon as it receives a new order	A common mistake for adapters is to start listening for amendments and cancellations after an order has been acknowledged by the market, rather than as soon as the new order is received.
Adapter supports concurrent amendments	A common problem in adapters is not properly supporting concurrent amendments.
Adapter does not change any fields in response to an amend/cancel	A common failing of adapters is to put the details of an amend or cancel into an order update before a response from the market is received. The only field that is allowed to change is the status field.
Adapter cleans up listeners for amend and cancel order events after order is final	This test should confirm that there are no listener leaks.
Adapters that fake amendments must make this transparent	It is a common failing for the quantity field and quantity executed field to represent only the new replacement order.
Adapters must send the correct events on an order that fills immediately	It is a common failing to forget to put the order `inMarket`, before it is marked as not being `inMarket`. If this happens then the publisher will not know that the order is finished. QtyRemaining = 0 does not mean an order is final. See "Cardinality rules" on page 281.
Adapters must listen to all cancel requests until an order is final	A common failing of an adapter is to have an `on CancelOrder` listener, rather than an `on all CancelOrder` listener. Cancellations can fail, so it is important to listen for more than one of them.
Adapters must always mark orders as final eventually	Either an order will be marked downstream because it is finished, or because the adapter connection was lost, or even the correlator was shutdown, but it must happen at some point. This usually requires ondie actions to be implemented.
Adapters must handle disconnection	It is a common failure not to handle adapter disconnection correctly. As mentioned above the order should be marked as `stateUnknown` and canceled.

Test	Details
Adapters must handle updates from downstream systems while amending/canceling	It is a common failing of an adapter to ignore updates from a downstream system while processing an amendment, or cancellation. This is often fill information.
Adapters must handle external amends	If a downstream system allows components other than the adapter to modify orders, the adapter must report these as external amendments.
Adapters should reject all unsupported amendments	Many adapters fail to realize that the user tried to amend the symbol, because they do not support this. The receiver must validate these fields and send an update with `orderChangeRejected` set to `true` and with an appropriate status message.
Adapters must be able to report rejected amendments and cancellations	Some adapters fail to do this.
Adapters must always send updates where `quantity` = `qtyRemaining` + `qtyExecuted`	Any event that does not have this property is invalid.
Adapters must only set `inMarket` when the market has confirmed the order and unset it when the order is final	It is a common mistake to set the `inMarket` flag prematurely and to forget to unset it when the order is final.
Adapters must always reject amendments when they have the modifiable flag set to `false`	It is a common mistake to not remember to set modifiable to `true` and accept amendments even if the flag is not set. Only orders that have been marked as modifiable, or for which no order update has yet been sent, can be modified.
Adapters must correctly implement average price calculation	Not all adapters implement the average price correctly. Either this field must be reported directly from the downstream system, or the cash executed must be tracked by the receiver.
Adapters that require user ids must provide a mapping facility	It is a common mistake implementing an adapter to expect the ownerId field to contain the user ID the downstream user will recognize. The value will be the dashboard user id when sending orders, so the adapter must provide a mapping facility.
Adapters must be fully documented	There are a lot of features in the `com.apama.oms` protocol that are entirely adapter specific. These must be documented. Every adapter needs to detail: ■ Supported order types ■ Supported extra parameters, whether they are mandatory or optional

Test	Details
	■ Which fields and extra parameters can be amended ■ The definition of quantity values in orders (particularly in FX) ■ Supported markets and exchanges ■ Details of broker and book support ■ Details of session configuration (multi instance support) ■ Details of user name mapping

Market data package overview

The market data interfaces defined in `TickManagerSupport.mon` have been designed to facilitate the distribution of depth data and tick data. Depth data gives either the best bid and offer (ask) prices and quantities available, or information for multiple levels of depth. When multiple levels of depth are provided, the depth is aggregated by price. A few adapters will send un-aggregated order book data in the depth event. Tick data is a stream of events detailing trades that occur on an exchange.

Market data comes from markets or matching engines. When orders are sent to a receiver they contain a composite destination key, which will determine where an order is sent. This same key, therefore, is often used to request and identify market data. In some cases where different connectivity is used to obtain market data, than to submit orders, then the details will not match. In this role it is called a composite source key. The key comprises service identifier, market identifier and exchange identifier. Not all the details of the destination are present in all events as primary fields, so the extra parameters `SERVICE_NAME`, `Market`, and `Exchange` must be added to a number of the events in the protocol.

In addition to the normal fields used in a composite source key, depth subscriptions/publishers may also support different types of depth subscription, level 1, 2, and maybe 3. When this happens an additional extra parameter containing the type of depth data is used in the key as well. Further details are given in the value constraints sections of the relevant events.

Market data actors

There are two actors in `com.apama.marketdata`

- The publishers of data

  Typically a matching engine or adapter

- The consumers of data

  These are typically DataViews, or algorithms

One publisher will be sending data to 0 or more consumers at any point in time.

Please note that the CMF contains a number of utilities for dealing with this protocol. There is an object called the `MarketDataKey`, which the consumers of data can use to ensure they get the correct data. There are two others called the `DepthPublisher` and `TickPublisher`, these can be used to reduce the development time of market data publishers, implementing reference counting and ensuring the correct extra parameters are set.

Market data events

There are a number of events in the market data interfaces and many of them are identical for depth and tick data. Where this is the case, the details of the events are omitted:

- `SubscribeDepth` and `SubscribeTick`

  The subscribe events for the tick and depth services are identical, these are sent from the consumers to the publishers.

  There are no explicit acknowledgments of subscriptions, so success is determined by actually receiving the requested data.

- `UnsubscribeDepth` and `UnsubscribeTick`

  The unsubscribe events are also identical for both tick and depth and are sent from consumers to publishers.

- `DepthSubscriptionError` and `TickSubscriptionError`

  Use of these events should be considered deprecated, but as many adapters do use them, details are given. They are symmetric for Tick and Depth.

  The reason that use of these events is cautioned against, is that they do not contain any of the composite source key fields and have no extra parameters field. This means that you cannot work out which publisher actually had an error. This means that in FX aggregation, for example, an error on a symbol on one adapter, will make the system think that ALL adapters have failed for that symbol.

  Instead of using these events, consumers and publishers of market data should use the `com.apama.statusreport` interface, to monitor the availability of market data. Since the exact usage of this protocol is adapter specific, please refer to each adapter's documentation to determine how to subscribe to this data.

- `Depth`

  Depth data represents aggregated depth data.

- `Tick`

  The tick event represents a trade.

Market data publisher error handling

Apama has developed a set of best practices for market data publishers (typically adapter service monitors) to follow when dealing with disconnections or other errors in the underlying adapter.

Whilst these are not strictly part of the `com.apama.marketdata` API or protocol they should be followed where possible by any market data publisher component. Some of these rules have already been mentioned above but it is useful to list them all in one place:

- The market data publisher should use the IAFStatusManager to track the state of its connection to the IAF and the IAF's connection to the remote service
- If the publisher becomes disconnected from the IAF or the IAF becomes disconnected from the remote service, the publisher should:
  - Not assume that all subscriptions have been canceled by the adapter
  - When the connection is restored, explicitly unsubscribe and resubscribe for all subscriptions that were active when the connection failed
- As discussed above the error reporting events in the `com.apama.marketdata` API should not be used. Instead publishers should report connection failures through the `com.apama.statusreport` API.

Adapter status

The `StatusSupport.mon` ships with Apama. The `com.apama.statusreport` interface, defined in `StatusSupport.mon`, is a companion protocol to `com.apama.oms` and `com.apama.marketdata` for detecting and handling connection status issues. Please note that the `com.apama.statusreport` interface is not specific to capital markets applications. This section presents a de-facto standard pattern for using the interface that is expected to be used by all Apama capital markets adapters. Adapters in other application areas may choose to use different status reporting patterns.

One of the main things that this interface is used for is to give users and algorithms access to the IAF connectivity information produced by the `com.apama.adapters` protocol.

Please note that the CMF contains a number of utilities for dealing with this protocol. The `ConnectionMonitor.mon` service can convert `com.apama.adapters` events into `com.apama.statusreport` events. Normally this configuration event would be sent by an adapter's logon manager service. The `StatusPublisher` object can be used to implement other status publishers. It implements reference counting and ensures correctly formatted status messages.

The actors in `com.apama.statusreport` include:

- The publisher
- The consumer

The events in `com.apama.statusreport` include:

- `SubscribeStatus`
- `UnsubscribeStatus`
- `Status`
- `StatusError`

The temporal rules for the status report interface are basically the same as for market data. We have a subscription based service, where consumers must subscribe and unsubscribe. If the reference count of a subscription is greater than 0 the publisher should distribute status events. Unlike the market data interface there are error events that can be associated with a subscription. These should only occur after a subscription event has been sent, and when the reference count is above 0.

Authority rules and cardinality rules are symmetric to the market data protocol.

The status event itself contains a `Boolean` value that specifies whether an object is available for a given connection, service, and sub service. The summaries field contains a selection of adapter specific tags adding additional status information and there is also an `extraParams` field for anything else.

The exact usage of the fields `serviceId`, `object`, `subServiceId`, and `connection` are adapter specific, see the documentation for each to see what values are supported. However, a de-facto standard has developed around the pattern used in the adapters supported by the FX Aggregator solution accelerator and this should be followed, as far as possible, for all new adapters. This pattern expresses a minimum set of requirements; individual adapters can and should expose more detailed status information as long as this is documented clearly.

- The `serviceId` should be the same as the service name used by the adapter for the market data and/or order management interfaces.
- At a minimum, the adapter object should be supported.
- No standard values for `subServiceId` are defined. This field should be left blank unless the adapter documentation specifies supported values.
- The `connection` field should match the `marketId` used by the adapter in market data subscriptions and order submission. As noted above this typically refers to a specific adapter transport instance.

A status subscription for `(SERVICE_NAME, Adapter, "", Market)` should therefore succeed for any adapter following this pattern.

The adapter object represents the overall status of the adapter including the service monitors, the connection from the correlator to the IAF, the codec/transport, the connection to the remote service, any required login to the service and readiness of the service to handle subscriptions/orders/etc. The available flag of status events for the adapter service should only be `true` if the adapter is in a state where any subscription requests or order management events sent in by a client will be accepted by the adapter and passed through to the underlying remote service.

Several standard values have also been defined for the summaries sequence in the status event. These should be used where possible rather than defining new values for the summaries:

- Connected: The adapter transport is connected to the remote service
- Logged On: The adapter has successfully logged on to the remote service
- Open: The remote exchange is open for trading

Adapters are free to define additional objects, `subServiceIds`, and summaries keys to express more fine-grained status if desired (for example, a market data adapter can use the Status interface to indicate subscription errors for specific instruments without using the unreliable `Tick` or `DepthSubscriptionError` events).

Authority rules

Authority rules for the Legacy Market Data API include the following:

- Any consumer may send a subscription to get data
- Only one publisher can send data and errors for a given source key

  To avoid conflicting data, only one publisher should try and transmit data for any given source key.
- Only consumers that have sent a subscription can send unsubscribe requests

  To ensure that the publisher keeps a correct reference count, only consumers that have an active subscription can send unsubscribe requests.

Cardinality rules

Cardinality rules for the Legacy Market Data API include the following:

- Consumers can only have one active subscription per destination key

  This ensures that the reference count the publisher keeps is correct.
- Each subscription must be paired with an unsubscribe

  Any subscription that is sent must be paired with an unsubscription at some point. Note that often this means that a monitor must normally send unsubscribe events from its on die action, if and only if it had active subscriptions.

Temporal rules

Temporal rules for the Legacy Market Data API include the following:

- No depth or tick events are sent unless at least one subscription is active

  Publishing of market data should be lazy to avoid swamping the correlator. This means that the publisher needs to implement a reference count. Data should only be sent if the reference count is greater than 0. Note that the publisher should never let the reference count go negative.
- Cached depth events are sent for every successful subscription

  Depth data may not update for a period of time. Anything that subscribes to the data wants to have access to it as soon as possible. `Com.apama.marketdata` does not have an explicit subscription response event, so instead the publisher should retransmit the latest Depth data for each new subscription. Note that the first subscription may cause a cascade of events that

include sending a subscription request out to an external adapter, for this reason depth will not be immediately available for that subscription.

- No cached tick events will be sent for a successful subscription

  Depth events are value events. That is they represent the latest value of some signal. This makes it safe to retransmit these. Tick events are true events. This means they cannot be retransmitted. For example, an algorithm may be trying to implement a market participation algorithm. Any such algorithm would overtrade if ticks were retransmitted.

- No market data consumer should send an unsubscription before a subscription

  To make sure that the reference count the publisher keeps is correct, consumers must not send an unsubscribe event that does not match a subscription.

Value constraint rules

Value constraint rules for `SubscribeDepth` and `SubscribeTick` are symmetric and include the following:

- `serviceId`: identifies which publisher (adapter) needs to process the request.
- `marketId`: part of the composite datasource key, use of it is the same as for the market identifier in a NewOrder event.
- `symbol`: identifies the instrument that the consumer wants market data for.
- `extraParams`: dictionary field key used like the corresponding field in a new order event. Some publishers will interpret fields to decide if depth data should be level 1 (best only), level 2 (market by price) or level 3 (market by order). Note that where these extra parameters are supported, they must be considered as part of the source key. This means that consumers must filter based on these extra parameters, and the publisher must put them in the extra parameters of depth events. The `MarketDataKey` and `DepthPublisher` objects in the CMF handle this situation. There is no exchange field in a subscribe depth/tick event, so the consumer must send the exchange extra parameter in this event. The publisher should filter based on exchange. No extra parameter set is interpreted as `"Exchange"=""`.

`UnsubscribeDepth` and `UnsubscribeTick` are identical to the subscription events. The same rules apply for all their fields.

Use of `DepthSubscriptionError` and `TickSubscriptionError` events is discouraged, because they do not contain enough fields to identify where the error occurred. Instead the publisher and consumer should communicate depth or tick subscription errors using the `com.apama.statusreport` interface. They contain the following fields:

- `symbol`: identifies the symbol that had the error.
- `status`: a human readable status message, which must describe the error. An empty status string indicates that any previous error condition has now been cleared.
- `fault`: states whether the error message represents a permanent fault. If this field is `true`, the market data consumer should assume that its subscription has been canceled and that it must re-subscribe in order to receive more data. If this field is `false`, the error is a transient one and

consumers can expect to receive more data eventually. After sending an error event with `fault=true`, a publisher should ensure that the reference count for the relevant subscription is reset to zero and that no more data is sent for the subscription until a new consumer subscribes.

The `Depth` event is used to represent the depth in the market. Note that the depth event does not contain any details of the composite source key in its primary fields. The consumer of the data must filter the market data using the extra parameters. The CMF `MarketDataKey` makes doing this much easier. All publishers must ensure that the sequences representing prices are the same length as the sequences for quantities such as, bid price and quantity, ask price and quantity. This will allow the consumer to safely use the same index to access entries in both sequences, without excessive safety tests.

A `Depth` event includes the following fields:

- `symbol`: the symbol to which the market data pertains. It will match the subscription.
- `bidPrices`: a sequence of bid prices. They must be arranged best price to worst, that is, the largest number to the smallest.
- `askPrices`: a sequence of ask prices. They must be arranged best price to worst i.e. smallest number to largest. Occasionally some adapters will convey Market By Order data through what was originally intended to display Market By Price data. When the do this, the same price may be listed more than once. This must be clearly documented when it occurs.
- `midPrices`: this field is effectively deprecated. The field was originally intended to give several levels of mid prices, but because no sensible definition of these is possible and because even one mid price can be calculated in multiple different ways, publishers should leave this field empty and allow the consumer to calculate their own mid price using their method of choice. This also helps keep the size of the event down.
- `bidQuantities`: the depth available in the market. Each quantity maps to the price at the same index in the bidPrices field. For this reason, the two sequences must be the same length. The value is the total contract quantity in the market, not the number of orders in the market.
- `askQuantities`: the depth available in the market. Each quantity maps to the price at the same index in the askPrices field. For this reason, the two sequences must be the same length. The value is the total contract quantity in the market, not the number of orders in the market.
- `extraParams`: can be used by the publisher to transmit any information it likes to the consumer, such as the VWAP of the market. Note that, because there are none of the composite datasource keys in the main event, they must all be provided in the extra parameters. The publisher must provide them and the consumer should use the CMF `MarketDataKey` object to filter depth events based on them. The values should be in `SERVICE_NAME`, `Exchange` and `Market`. Remember too that another extra parameter containing the depth subscription type may need to be present.

Tick events contain the following fields:

- `symbol`: the symbol to which the market data pertains. It will match the subscription.
- `price`: the price of the last trade.
- `quantity`: the quantity of the last trade.

- `extraParams`: can be used by the publisher to transmit any information to the consumer, such as the trade type. Note that, because there are none of the composite datasource keys in the main event, they must all be provided in the extra parameters. The publisher must provide them and the consumer should use the CMF `MarketDataKey` object to filter events based on them. The values should be in `SERVICE_NAME`, `Exchange` and `Market`. Remember too that another extra parameter containing the depth subscription type may need to be present.

Test cases for market data publishers

The rules above suggest a minimum set of test cases for order publishers. They are given in the table below.

Test	Details
The publisher must filter on all composite source keys and publish data with them too	The adapter must filter subscriptions and unsubscriptions based on the service, market and, where used, the exchange. They must also put all of these fields in the Depth and Tick events they produce. Note that the exchange is only provided in extra parameters. Where adapters support multiple types of depth subscriptions the receiver must filter based on the extra parameter containing the subscription type. This parameter is adapter specific. Subscriptions with an empty serviceId should be ignored.
The publisher must start and stop Depth and Tick data based on the reference count	The adapter should lazily provide data based on the reference count being greater than 0.
The publisher should not allow the reference count go below 0	Publishers might accidentally send too many unsubscriptions. When this happens the reference count must never go below 0, so that a single new subscription will re-enable the data.

C Legacy Market Data Management

- Market data gateway service 298
- Market data bridging service 299
- Market data publisher components 300
- Depth and tick DataViews 301

Although previous sections described the Market Data Architecture that should be used, some existing EPL applications and IAF adapters still make use of the legacy market data management events. The legacy market data management events and relevant Market Data utilities are included in the CMF for backwards compatibility purposes, but will be deprecated in a future release.

The components and services in the Market Data Utils bundle provide functionality for market data receiving, publishing, and transferring into, out of and within CMF applications. In the current version of CMF, market data generally refers to the abstractions for market prices and trade reports defined by the `com.apama.marketdata.*` events in the Legacy Finance Support Bundle. MDA provides a newer interface for handling market data.

These components and services are in use in existing applications to:

- Subscribe to market data from an adapter or some other source.
- Publish or distribute market data to other parts of the application, or other systems. This could include implementing a new market data adapter.
- Transparently bridge market data between multiple correlators.

Market data gateway service

The market data gateway service enforces transactional behavior on market data (`Depth` and `Tick`) events processed by the CMF. This ensures that all effects of each market data update within the CMF, for example, changes to tracked positions, are known before the update itself or any events that depend on it are delivered to the application. From the application's perspective, using market data through the gateway is straightforward:

1. The application subscribes to market data in the usual way, by routing a `com.apama.marketdata.SubscribeDepth` or `com.apama.marketdata.SubscribeTick` event with the appropriate parameters for the adapter or other datasource. However, rather than routing the subscription request directly to the adapter service, it is routed to the gateway service instead. For example, to subscribe to market depth for a symbol on the "FIX" service:

   ```
   // Extra parameters for subscription dictionary<string,string>
   extraParams := new dictionary<string,string>;

   // Without the gateway route
   com.apama.marketdata.SubscribeDepth("FIX", "MyFIX", "EUR/USD", extraParams);

   // With the gateway com.apama.marketdata.Constants mdConstants;
   extraParams.add(mdConstants.TARGET_SERVICE_EXTRA_PARAM(), "FIX");
   route com.apama.marketdata.SubscribeDepth(
      mdConstants.SERVICE_NAME_EXTERNAL(),
      "MyFIX", "EUR/USD", extraParams);
   ```

   Notice that the subscription request is routed to the service named by the `SERVICE_NAME_EXTERNAL` constant and that the name of the real service should be the value of the extra parameter key named by the `TARGET_SERVICE_EXTRA_PARAM` constant.

2. The gateway will route a subscription to the target service and set up a listener for `Depth` events arriving from that service.

3. When a `Depth` event from the target service is received by the gateway listener, the gateway re-routes the `Depth` event *within* the CMF correlator, so it can be processed by position-keeping and other analytic components.

4. Once all the CMF's internal processing of the `Depth` event is complete, it will be re-routed with the original (that is, the gateway) service name to be picked up by any listeners created in the application:

```
com.apama.marketdata.Depth depth;
on all com.apama.marketdata.Depth(symbol="EUR/USD"):depth {
   if depth.extraParams[mdConstants.SERVICE_EXTRA_PARAM()] :=
      mdConstants.SERVICE_NAME_EXTERNAL() then {

      // Handle this Depth event
      }
}
```

Note that because the service name is not a top-level field in the `Depth` event, the application should check the value of the extra parameters key named by the `SERVICE_EXTRA_PARAM` constant.

There is a performance implication to using the market data gateway service. Every incoming market data update is re-routed twice: once to be processed by the CMF's internal analytics, then again to be processed by the application. This will have an effect on the maximum throughput and minimum latency that can be achieved when the gateway service is used. However, this impact must be weighed against the application correctness benefits of using the gateway service. Without the transactional guarantees provided by the gateway service, an application might receive a position update before, or after, the market data event that triggered the position change, depending on minor details of the implementation. For some applications, this might not matter, but if it does, the gateway service is a convenient mechanism for ensuring reliable application behavior.

Market data bridging service

The market data bridging service is analogous to the adapter status bridging service described previously, but for market data events. It provides bridging of `com.apama.marketdata.*` events between two correlators, so that an application in the client correlator can subscribe to market data published by an adapter or other source connected to the server correlator. A market data bridge instance is configured by sending configuration events to the client and server correlators. On the client side, send a `com.apama.marketdata.ConfigureClientSideBridge` event, and on the server side, send a `com.apama.marketdata.ConfigureServerSideBridge` event. See the ApamaDoc for these events for details about their parameters. In most cases, the parameters of the client and server configuration events for a given bridge instance are identical. However, the `instanceName` parameter must be unique across all market data bridge services used by the application.

In addition to configuring the market data bridge itself, an application should also configure monitoring of the fake adapter connection between the two correlators, using an instance of the `com.apama.connection.IAFStatusFaker` and `com.apama.adapters.IAFStatusToStatusConvertor` services on both sides of the connection. If this is not done, the inter-correlator connection will appear to be down to the market data bridge service and it will not forward any events in either direction. The generic bridging service described above handles this automatically; it is recommended that applications use this service wherever possible. Otherwise, see the

implementation of the `com.apama.adapter.bridge.ConfigBridge` service in the Adapter Bridging Utils bundle to see how bridge connection monitoring is implemented.

Once the market data bridge service is configured, `com.apama.marketdata.SubscribeDepth`, `UnsubscribeDepth`, `SubscribeTick` and `UnsubscribeTick` events routed by the client-side application for the bridged service will be forwarded to the server correlator, where they should be handled by the market data adapter service as normal. Any `Depth` or `Tick` events generated by the market data adapter service will in turn be forwarded back to the client correlator where they can be handled by application listeners in the usual way.

The market data adapter bridge deals with multiple clients bridging to the same server-side market data adapter service in the following ways:

- Each client correlator should configure a separate connection to a market data bridge service with a unique `instanceName` field and channel.
- Multiple market data bridge service clients within the same correlator should make market data subscriptions in the usual way; these subscriptions will be reference counted by the client-side bridge service.

Market data publisher components

The Depth and Tick publisher components provide a convenient action-based interface for tracking subscriptions for `com.apama.marketdata.Depth` and `Tick` events and for publishing these events to subscribers in response to market data updates. These components will generally be used by adapter services but they are recommended for any service that needs to publish market data to users or other parts of an application. The publisher components handle all of the book-keeping associated with publishing market data to multiple subscribers, such as subscription reference counting and caching of the last market depth to send to new subscribers.

A new `com.apama.marketdata.DepthPublisher` object can be initialized in two different ways:

1. From a `com.apama.marketdata.SubscribeDepth` event. This reflects the most common case, where a service is listening for the *first* subscription request made for each symbol offered by the service, then constructing a depth publisher to handle this and all subsequent subscriptions.
2. From a `com.apama.marketdata.DepthKey` event. Using this method implies that the service knows in advance which symbols applications will be subscribing to, so that the depth publishers can be constructed in advance.

In both cases, the depth publisher will set up listeners for future subscription and unsubscription events for the same key (such as, the same combination of service, exchange and market identifiers, and symbol). The publisher will keep track of the number of active subscriptions and route a `com.apama.marketdata.DepthRefCount0` event when this number falls to zero. The service should listen for this event so it knows when to stop publishing depth updates for a given symbol.

Market depth updates are routed by the publisher when its `setData()` or `setDataWithExtraParams()` actions are called. The publisher also provides actions to update market depth in response to common order management operations:

- New order submitted: `changeDepthOnNewOrder()`

- Order amended: `changeDepthOnAmendOrder()`
- Order canceled: `changeDepthOnCancelOrder()`
- Order filled: `changeDepthOnOrdersFilled()`

After using the `changeDepth*()` actions, the service should call the `routeData()` action to deliver the updated market depth to all subscribers.

The design of the `com.apama.marketdata.TickPublisher` object is very similar to that of the depth publisher. See the ApamaDoc for these objects for full details of the actions they offer.

Depth and tick DataViews

The depth and tick DataView components use the generic CMF DataViewManager component to provide simple table-based representations of market data published by a CMF application.

A new depth DataView is created by constructing a `com.apam.marketdata.DepthDataView` object then calling its `initialise()` action. Each row in the DataView table holds market depth data for a single depth key. To add or update a row, call the depth DataView's `set()` action, passing in a `com.apama.marketdata.DepthKey` to identify the row and a `Depth` event to supply the market data. Rows that are no longer needed can be removed using the `remove()` action.

The `tick DataView` object (`com.apama.marketdata.Tick`) is very similar to the depth DataView. The only significant different is that the `set()` action takes a `com.apama.marketdata.TickKey` event and a `com.apama.marketdata.Tick` event as its arguments.