Esper Enterprise Edition - Server

Version 5.4.0

by *EsperTech Inc.* [http://www.espertech.com]

Preface

This document describes the Esper Enterprise Edition server.

The *Chapter 1, Introduction* chapter is a brief high-level, overview of the Esper Enterprise Edition server.

The *Chapter 2, Getting Started* chapter provides information pointers, Enterprise Edition component communication overview and deployment options.

The *Chapter 4, Clustering, Scalability and High-Availability* chapter explains how Enterprise Edition servers can work together in a cluster to achieve goals such as horizontal scalability, fail-over and disaster recovery.

The *Chapter 10, Examples* chapter overviews the examples that come with the server and how to install the examples of an Esper distribution.

Chapter 1. Introduction

This document is an introduction to the concepts and terminology behind the Esper Enterprise server. Throughout the docs, references to files and folders are relative to the installation directory.

Esper Enterprise Edition contains the following important files in the installation directory:

1. `index.html` - Overview, documentation portal and quick start page.
2. `LICENSE.txt` - License.
3. `RUNNING.txt` - Instructions to run.
4. `license_3rdparties.txt` - License information pertaining to 3rd-party software packaged with Esper Enterprise Edition.
5. `changelog.txt` - Release notes with version change description and known issues.

The Esper Enterprise Edition folders in the installation directory are as follows:

1. `bin` - Startup, shutdown, classpath and other scripts. The *.sh files (for Unix systems) are functional duplicates of the *.bat files (for Windows systems).
2. `conf` - Configuration files and related XSDs. The most important file in here is `esper-default.xml`. It is the main configuration file for the server.
3. `data` - Files describing example continuous displays are in the `hqsvc` subfolder, and default JMS broker binary files in the `jmsbroker-endpoint-default` subfolder.
4. `doc` - Documentation folder.
5. `examples-*` - Examples folders.
6. `lib-*` - Lib folder for distribution jar files, war files and their dependencies. The `lib-security` folder can be used if securing EsperHQ using LDAP, Spring framework and/or Spring security (see EsperHQ documentation for security).
7. `logs` - By default logging is to the terminal. You may change the logging configuration file `conf/log4j.xml` to place log files in the folder.
8. `webapps` - Contains client application web application.
9. `hotdeploy` - For hot-deploying and un-deploying CEP applications with EPL modules.
10. `temp` - Folder for temporary files that may be created during deployment.

The information in the configuration files is read at startup, meaning that any change to the files necessitates a restart of the server.

Chapter 2. Getting Started

2.1. Information Pointers

If you are new to Complex Event Processing or the Esper CEP engine, please review the *index page* [http://www.espertech.com/esper] and *quick start* [http://espertech.com/esper/quickstart.php]. You need to be familiar with the concept of events, event processing language (EPL), statement and event stream.

Enterprise Edition adds enterprise features including:

- A small and very lightweight server that loads a given Esper configuration.
- Hot deployment of text file EPL modules as well as packaged CEP applications with application code.
- Web application archives for deployment to a servlet container of your choice.

The server includes startup and shutdown scripts as outlined in `RUNNING.txt` in the root folder of the distribution. The server reads configuration files as described in the next chapter.

Esper Enterprise Edition provides the same Esper CEP engine component as the community version on the *Esper project site* [http://www.espertech.com/esper].

The `index.html` file in the top folder of the distribution overviews Esper Enterprise Edition and contains all documentation links.

Related documentation is:

1. Please see the Esper reference manual for EPL language and CEP engine considerations.
2. The Esper HQ web application client manual discusses continuously-updating displays, engine management GUI functions, JavaScript and HTML 5, jQuery plugin and faceless launcher.
3. The Esper HQ web application server manual outlines client deployment and configuration options.
4. The CEP Push Services manual instructs how the CEP engine data push and subscription management works.

2.2. License Information

The file `espertech.license` contains your license information and keys. This file must not be edited as it is signed electronically. Existence, integrity and validity checks of the license information are done at runtime.

EsperHQ and Enterprise Edition attempt to find the license file in your classpath under the name `espertech.license`.

The license key file is placed in the product installation `conf` folder.

2.3. Server Command Line Parameters

Esper Enterprise Edition comes with a custom, very small server. The scripts and command line options are described here.

The `esperee` (Windows) and `esperee.sh` (Linux) commands print the start options.

Additional command line parameters may be passed on the server command line. These are printed via the `esperee run -?` (Windows) and `esperee.sh run -?` (Linux) command line parameters.

To run from a configuration file other than the default `esper-default.xml` configuration, use the following command:

```
esperee run -config conf/myconfiguration.xml  (Windows)
esperee.sh run -config conf/myconfiguration.xml  (Linux)
```

The `-name` parameter is followed by an identifier and can be used to provide an engine URI name, which is `default` if none is specified.

The `-config` parameter is followed by a file name and loads an Esper configuration. You may in addition load an EsperHA configuration.

The `-haconfig` parameter is followed by a file name and loads an EsperHA configuration. You may in addition load an Esper configuration.

The `-home` parameter is followed by a directory and overrides the Esper EE base directory.

The `-cluster` parameter if a flag to indicate clustered operation.

The `-cluster-config` parameter is followed by the cluster configuration file name, for use with the `-cluster` parameter.

The `-cluster-jmxport` parameter is followed by a JMX port number, for use with the `-cluster` parameter.

The `-cluster-name` parameter is followed by a name for the cluster, for use with the `-cluster` parameter.

2.3.1. Starting Additional CEP Engines

By default the command line parameters instruct the server to start a single CEP engine with URI `default`. To start additional CEP engine you specify the `-more` parameter followed by a list of

engine URIs, Esper and EsperHA configuration files. Alternatively you may dynamically start and destroy additional CEP engines and related CEP application(s) via hot deployment.

The `-more` parameter is followed by a semicolon-separated list of engine URI, engine configuration file name and EsperHA configuration file name in the format:

```
-more engine_1_uri[,engine_1_config_file [,engine_1_HA_config_file]]
 [;engine_2_uri,...[;...]]
```

If the Esper configuration file name is not provided, the default configuration applies. If the EsperHA configuration file name is not provided, no EsperHA configuration is loaded. Configuration file names are relative to the installation root directory.

The bootstrap initializes any additional CEP engines before the default CEP engine. Note that since CEP Push Services, JDBC and JMX do not need to be started for additional CEP engines (all recognize multiple engines automatically), remove them from the configuration file for any additional engines.

The following example command line initializes the `default` CEP engine and in addition initializes a second CEP engine by URI `cepone` using the default configuration:

```
esperee run -more cepone
```

In the next sample command line the bootstrap initializes the second CEP engine by URI `cepone` and also loads its configuration from file `conf/cepone.cfg.xml`:

```
esperee run -more cepone,conf/cepone.cfg.xml
```

In this third sample command line the bootstrap initializes the second CEP engine by URI `cepone` and also loads its configuration from file `conf/cepone.cfg.xml` and EsperHA configuration from file `conf/cepone.hacfg.xml`:

```
esperee run -more cepone,conf/cepone.cfg.xml,conf/cepone.hacfg.xml
```

The following example command line initializes the `default` CEP engine and in addition initializes two more CEP engines by URI `cepone` and `ceptwo`:

```
esperee run -more cepone,conf/cepone.cfg.xml;ceptwo,conf/ceptwo.cfg.xml,conf/
ceptwo.hacfg.xml
```

2.4. Configuring EsperHA (High-Availability, not included)

EsperHA for high-availability is provided by a separate distribution. Please read the Getting Started section in the EsperHA documentation set to become familiar with the EsperHA configuration file and binaries.

Please copy the jar files that came with your EsperHA distribution to the empty `lib/esperha` folder of the Enterprise Edition distribution.

In the classpath scripts `setclasspath.sh` (Linux) or `setclasspath.bat` (Windows) in the `bin` folder of the distribution find the section that adds the EsperHA jar files. Remove the comments thus adding EsperHA jar files to the classpath.

Pick the relevant EsperHA default configuration file and drop the file into the `conf` folder of the installation.

The start command that loads the EsperHA configuration file is:

```
esperee run -haconfig conf/esperha-default.xml (Windows)

esperee.sh run -haconfig conf/esperha-default.xml  (Linux)
```

The EsperHA license file `espertech.license` should be placed into the `conf` folder under the installation root.

If using the default store with EsperHA, create a directory by name `esperha-default-store` under the `conf` folder of the installation root to hold the store files.

If using another provider for EsperHA store, please ensure that any additional jar files that may be required are added to the classpath scripts.

2.4.1. High-Availability and Hot Deployment

EsperHA persists statement information as well as event type information for statements marked as durable or resilient (or for all statements if your EsperHA configuration sets the default to resilient). EsperHA also persists information about which modules have been deployed including the module name. The module name and statement name are both relevant information used to distinguish which modules to deploy or which statements to create and which modules or statements already exist.

When the server process ends and is restarted, EsperHA recovers module information, event types and statements from the current store. To prevent recovery, you may set an EsperHA configuration flag to truncate the store when opened.

During startup, the hot deploy service picks up all deployment files and undertakes a deployment process for each JAR, EPL and WAR file. The hot deploy service follows the steps discussed next to deploy modules and statements.

If your deployed EPL files (text and WAR files) specify a module name and a module by that name has already been deployed earlier, then the hot deploy service skips deployment for that module.

If your deployed EPL files do not specify a module name, then the hot deploy service deploys and creates new statements even if the statement may already exist from a prior deployment. To prevent duplicate statement creation, you may assign a statement name to each statement via the `@Name` annotation and set the following setting in the EsperHA configuration:

```
<esperha-settings ignore-duplicate-statements="true"/>
```

2.4.2. High-Availability and WAR File Deployment

EsperHA event type and statement recovery requires that any classes referenced as event types, library methods, annotation classes or other extensions are available in the classpath at CEP engine initialization time. When using EsperHA with WAR file deployment, it is currently a requirement that such classes are added to the server classpath and not to the WAR file.

2.5. Configuring Multiple CEP Engines

Your CEP application(s) design may include multiple separate CEP engines. One reason to use multiple CEP engines is for separating unrelated applications. A second reason is when CEP engines require different configurations such as when one engine runs providing external time and a second CEP engine runs using system time.

Enterprise Edition allows running any number of CEP engines side-by-side in the same Java VM, each identified by a unique engine URI. When starting Enterprise Edition server in the default configuration, the server starts a single CEP engine with the URI `default`. When hot deploying EPL and WAR files the default configuration deploys to that `default` engine.

You may initialize multiple CEP engines via two mechanisms: First, you may use the `-more` parameter on the command line to have the bootstrap start additional CEP engines as part of server startup and destroy each CEP engine when the server is shut down. Second, you may use hot-deploy engine jar files to start and destroy CEP engines including related EPL modules and Java code optionally with annotated classes, as described in *Section 5.3, “CEP Engine Jar Hot Deployment”*.

Each CEP engine must be assigned an engine URI, which is a unique identifier that can be any name. Each individual CEP engine can run with or without EsperHA configuration, independently of other CEP engines in the same JVM process.

The CEP Push Services, JDBC and JMX adapters automatically recognize the multiple CEP engine in the single JVM process and should only be configured and started once per JVM process.

In Esper HQ client you can see multiple CEP engine URIs in the URI drop-down dialog box on the upper-right hand corner. You may set the CEP engine that the GUI currently operates on by changing the value of the drop down dialog box followed by selecting an operation. Eventlets also support receiving data from multiple CEP engines.

To use the hot-deployer with multiple CEP engine to target WAR or EPL deployment artifacts to a given CEP engine, please follow the hot deploy service configuration instructions below.

2.6. Configuring Cluster Operation

Use the `-cluster` parameter on the command line to have the server process coordinate cluster membership. Cluster operation automates fail-over by assigning primary and standby roles between server processes. We use the term `primary` to describe a single cluster member that is bootstrapped and active. We use the term `standby` to describe one or more cluster members that are waiting to become primary.

In clustered operation, each server joins a single named cluster before continuing the bootstrap. After the server joins the cluster the server is assigned either the primary or a standby role. Only after the server is elected to become primary within the cluster does the server continue the bootstrap process. When the server that was elected primary ends, one other server in the same cluster (a standby) is elected primary and that server continues its bootstrap process.

As clusters are identified by a cluster name, with the default cluster name being `esperee-cluster`, all server processes that participate in the same cluster must have the same cluster name assigned. In the default configuration, the failure detection protocol for cluster members is based on TCP sockets.

To assign a unique JMX management port to each server process, provide a value for the `cluster-jmxport` parameter on the command line.

When using EsperHA with clustered operation, each cluster member should have the same Esper and EsperHA configuration file. If using EsperHA with a shared file system, cluster members should have access to shared file system files to ensure that the current primary upon bootstrap and initialization can recover engine state.

Enterprise Edition server uses the open source JGroups (*index page* [http://www.jgroups.org/]) toolkit for reliable group membership management. Enterprise Edition utilizes JGroups' TCP stack as configured by `espereeserver-jgroups-tcp.xml`, by default. Each host or server and port requires a customized copy of the `espereeserver-jgroups-tcp.xml` configuration.

You may optionally configure the JGroups communication and failure detection protocol by adding the `-cluster-config` parameter followed by the configuration file name to the command line. Please consult the JGroups documentation for additional information.

In the `bind_port` setting enter the port number for cluster heartbeats. When running multiple servers on the same host, please assign a unique port number to each server.

In the `bind_addr` setting enter the host name or IP address of the host.

In the `initial_hosts` setting enter a list of host and port combinations that send heartbeats to this server. The format of the entry is *host*[*port*] for each such entry.

At the time of server startup, the primary server in a cluster displays startup messages similar to below:

```
Clustered operation, cluster name 'esperee-cluster'   configuration 'conf/
espereeserver-jgroups-tcp__myhost.xml'
-------------------------------------------------------------------
GMS: address=myhost-num, cluster=esperee-cluster, physical address=x.y.z.a:7800
-------------------------------------------------------------------
Clustered operation elected the process as primary, continuing bootstrap.
```

Servers that are entering standby to the primary server display similar startup messages:

```
Clustered operation, cluster name 'esperee-cluster'   configuration 'conf/
espereeserver-jgroups-tcp__mystandby.xml'
-------------------------------------------------------------------
GMS:       address=mystandby-num,       cluster=esperee-cluster,       physical
 address=x.y.z.a:7800
-------------------------------------------------------------------
Clustered operation elected the process as standby, waiting to become primary.
```

2.7. Configuring the EsperJDBC JDBC Server

EsperJDBC is part of Enterprise Edition and is useful for querying the CEP engine via the inward-facing JDBC driver. EsperJDBC includes a server component that accepts incoming client JDBC connection over the network. Please see the EsperJDBC documentation for more information.

In the default configuration the JDBC server component is not listening to a port. To enable the JDBC server, please change the default configuration file `esper-default.xml` in the `conf` folder and remove the comments around the JDBC plug-in loader section.

If using multiple CEP engines per process, please start the JDBC server only once. It is not necessary to start the JDBC server for each CEP engine separately and the server recognizes each CEP engine in the same process.

2.8. Configuring EsperIO Adapters

EsperIO adapters are part of the Enterprise Edition distribution. For the purpose of brevity, the distribution package does not include all 3rd-party dependencies jar files for all EsperIO adapters.

To activate a particular EsperIO adapter in Enterprise Edition server, please follow these steps.

First, consult the EsperIO documentation for the `plugin-loader` adapter configuration. Add the configuration section to the configuration file `esper-default.xml` in the `conf` folder.

Second, please copy any jar files required by the adapter to the `esper-io` folder in the `lib` folder of the distribution.

Last, in the classpath scripts `setclasspath.sh` (Linux) or `setclasspath.bat` (Windows) in the `bin` folder of the distribution add the required EsperIO and dependent jar files to the classpath.

2.9. Configuring Same Host Multiple Servers

When configuring multiple servers for the same host, please follow the instructions herein to configure unique ports. All configuration files can be found in the `conf` folder under the installation root.

We recommend using a separate directory and installation per server.

The port number of the web application container can be found in the Esper configuration file `esper_default.xml`. Locate the settings under the name `Webapp_Service` and change the `port` value to a unique port number.

The port number of the JMX RMI registry is also in the Esper configuration file. Locate the settings under the name `EsperJMX` and change the `rmi-registry-port` value to a unique port number.

The port number of the embedded JMS broker is also in the Esper configuration file. Locate the settings under the name `JMS_Provider_Bootstrap` and change the `connectorURL` value to a unique port number. Also modify the CEP Push Services configuration file `ceppushsvc-default.xml` and assign the same URL to the setting `java.naming.provider.url`. Also modify the EsperHQ web application configuration file `esperhq-default.xml` and assign the same URL to the setting `java.naming.provider.url`.

The default embedded ActiveMQ instance starts a JMX registery port 1099. If you are using the default embedded instance, for multiple JMS servers on the same host, please disable ActiveMQ JMX by adding `<init-arg name="useJmx" value="false" />` to the `JMS_Provider_Bootstrap` configuration.

2.10. Running Post-Startup Scripts

It can sometimes be useful to run a script after server startup completed and all deployments, including those in the `hotdeploy` folder, are done.

The server can be configured to run one or multiple Groovy scripts. Groovy is a scripting language very close to Java and is further described in *Groovy Home* [http://www.groovy-lang.org/]. A sample script is provided under the installation root folder in file `bin/post_startup_script.groovy`.

By default, the engine does not execute any scripts. You must first enable the plugin for the script engine. Please edit `conf/esper-default.xml` under the installation and comment-in `PostStartupScript` provided by the class `GroovyExecutorPlugin`.

Specify the script names by adding an `init` parameter for each script, appending the number of the script to the name.

The sample entry below executes the post-startup script:

```
<plugin-loader name="PostStartupScript"
    class-name="com.espertech.esper.server.groovy.GroovyExecutorPlugin">
  <init-arg name="script.0"
    value="${ESPEREE_BASE}/bin/post_startup_script.groovy"/>
</plugin-loader>
```

The server executes `plugin-loader` entries in the order they are listed. Therefore, for executing post-startup scripts, we recommend placing `PostStartupScript` as the last entry in the configuration file.

The sample Groovy script provided with the distribution looks up a dataflow declaration and instantiates a dataflow.

2.11. Running Scripts At Runtime

Enterprise Edition provides a dataflow operator to execute Groovy scripts at runtime. This is further explained in *Chapter 8, Runtime Script Execution*. Groovy is a scripting language very close to Java and is further described in *Groovy Home* [http://www.groovy-lang.org/].

Scripts can be useful to dynamically interact with the CEP engine instance(s) such as for managing statement listeners or dataflow instances. Scripts can also execute when an event arrives: This allows interacting with an external resource or triggering an external process based on event data.

Script execution takes place within a dataflow instance. Therefore you can use the EsperHQ GUI or REST services to parameterize and manage script execution.

Chapter 3. Architecture Overview

3.1. Component Overview

You may use Enterprise Edition server to run all components in a single JVM or you may run multiple Enterprise Edition server or other JVMs with select components. Communication utilizes REST and JMS standards thereby allowing distribution of components.

The table shown below discusses the logical tiers and associated components.

Table 3.1. Logical Tiers

Name	Description
Client Tier	Web browser capable of running JavaScript, HTML 5, web sockets (client side), Ajax. Runs EsperHQ Rich Client GUI web browser application, locally.
Web Server Tier	Java 7 server process providing the web application container for the EsperHQ Rich Client GUI application, for REST web services and for web sockets (server side). Runs EsperHQ WAR file and HQ Services REST web services.
CEP Server Tier	Java 7 server process running Esper CEP and serving REST service requests. Runs Esper, CEP Management REST web services and CEP Push REST web services.

3.1.1. Client Tier: EsperHQ Rich Client GUI

The EsperHQ rich client is a client application that runs solely within the web browser of the client hardware, and utilizes the client hardware CPU and memory.

The client application is a JavaScript and HTML 5 application. It utilizes web sockets to receive streaming data from its originating server

The client application communicates with REST web services deployed on server hardware using HTTP or HTTPS, Ajax and JSON standards. It solely requires and communicates with HQ Services provided by its originating server. It does not directly communicate with the CEP server tier.

Part of the EsperHQ JavaScript applications are also jQuery extensions to activate eventlet streaming displays.

3.1.2. Web Server Tier: EsperHQ Web Application

The Esper HQ web application is the WAR file deployment unit that contains the web application and provides certain configurations.

3.1.3. Web Server Tier: HQ Services REST Web Services

HQ Services provides REST web services for the EsperHQ Rich Client GUI.

HQ Services communicates to one or more servers hosting CEP engine(s) via CEP Management Services (REST) and CEP Push Services (REST).

HQ Services manages the communication with one or more CEP Push Services by utilizing an Java Message Service (JMS) bus architecture. Each web application instance is responsible for managing data subscriptions for multiple EsperHQ client applications.

When receiving streaming data from CEP server(s), HQ Services requires and connects to a JMS provider.

3.1.4. CEP Server Tier: CEP Management REST Web Services

CEP Management utilizes Esper APIs and provides REST web services for HQ Services.

CEP Management web services perform functions such as EPL statement, module deployment and dataflow instance management, browsing of engine information and other design-time and runtime functionality.

3.1.5. CEP Server Tier: CEP Push Services REST web services

CEP Push Services provides REST web services for HQ Services.

CEP Push Services web services manage subscriptions and data exchange between CEP servers and web layer servers.

CEP Push Services requires a JMS provider.

If your application requires CEP Push Services for streaming data to EsperHQ web applications, Esper and a CEP Push Services must be present in the same process (JVM).

3.2. Push Communication

Enterprise Edition components work together to provide a multi-tier fan-out push-based architecture aimed at minimizing message traffic in terms of number of messages and message size.

All eventlet data streams are push-based all the way from the CEP server to the web browser client.

Pull-based communication is used during eventlet activation only, in the default eventlet configuration, and can be disabled.

3.2.1. CEP Server Tier Data Push

The CEP server tier pushes streaming data to the web server tier. Within the CEP server tier the component CEP Push Services is responsible for managing subscriptions and pushing data out. Within the web server tier the component HQ Services is responsible for subscribing, heartbeats, receiving pushed data and fanning out to client tier web browsers.

A CEP Push Services endpoint pushes CEP engine output to one or more JMS topics using non-persistent messaging and setting a short-lived message expiry. CEP Push Services is responsible for packaging multiple logical messages together into JMS messages according to defined service levels.

HQ Services instances deployed in web application servers receive JMS topic messages and perform a transformation and routing step to individual web browser clients that we describe below under client tier fan-out.

A CEP Push Service endpoint keeps track of all EsperHQ web application instances (web server tier) and their subscription interests. If subscription interests overlap, CEP Push Services ensures that logical messages are published only once and not duplicated.

A CEP Push Service endpoint is not aware by any means of EsperHQ clients, web browsers or activated eventlets. It is only indirectly aware of activated eventlets by the resources consumed by an eventlet as managed by EsperHQ web application instances that are clients to CEP Push Services endpoints.

3.2.2. Web Server Tier Data Push

The web server tier pushes streaming data to web browser clients. Within the web server tier the component HQ Services is responsible for managing client activations, web sockets and pushing data out. Within the client tier the EsperHQ rich client JavaScript application is responsible for managing activations and its web socket connection, receiving pushed data and fanning out within the eventlets on the same web page.

HQ Services receives JMS topic messages from servers in the CEP server tier. HQ Services performs a message transformation and subscription lookup step and then a routing step to individual web browser clients.

Chapter 4. Clustering, Scalability and High-Availability

We use the term clustering to mean multiple Enterprise Edition servers working together to achieve certain goals. For scalability, clustering technologies can be used to add processing power to event processing engines by adding servers that share the load. For high availability, a server cluster can be used to maximize uptime of event processing engines.

While Enterprise Edition components have clustering considerations integrated at the product design level, there is no single approach to clustering that is optimal for all use cases and implementation sites. Implementing a cluster is a matter of assembling components in different configurations.

We provide a list of related information and an overview of common clustering patterns next.

4.1. Related Information

Please see the Enterprise Edition server documentation (this document) for configuring server failover.

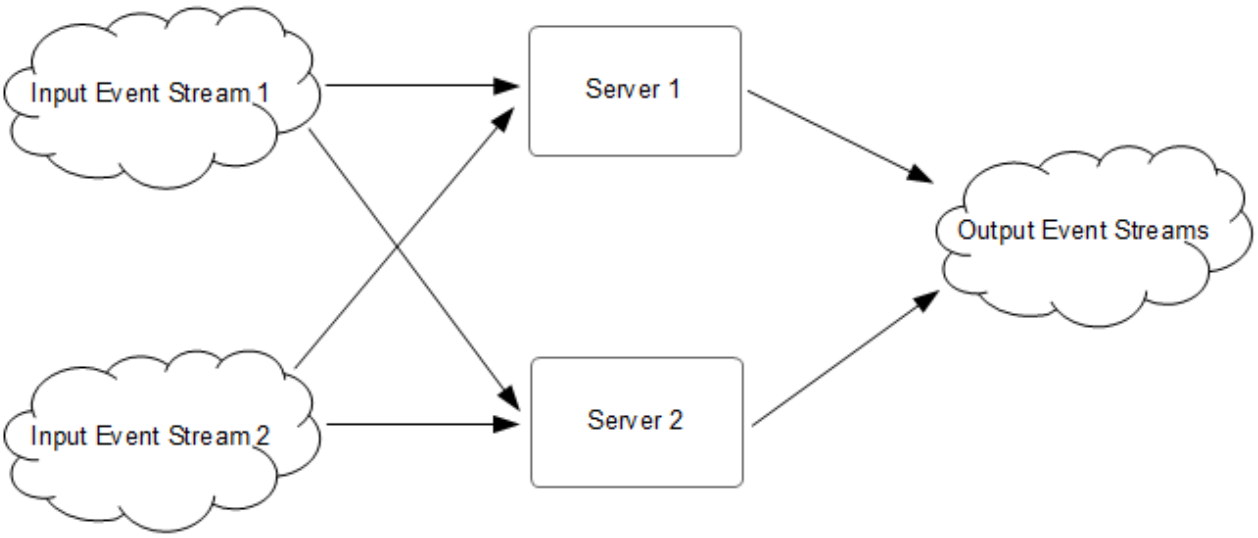
Enterprise Edition includes a TCP socket-based transport for server-to-server communication. Please find more information in *Chapter 7, Socket Transport*.

Please see the EsperIO documentation for configuring common server-to-server transports such as AMQP or JMS.

4.2. Mirrored Hot-Hot Pattern

In this pattern there are two or more cluster members that both (or each) receive all input.

The following diagram illustrates this pattern:



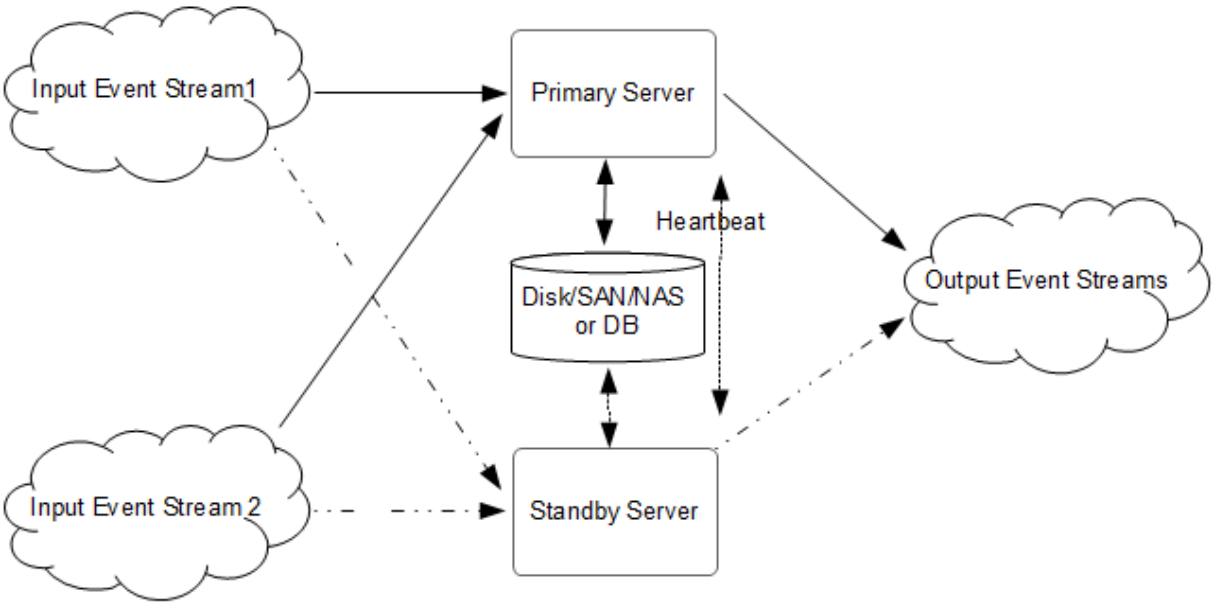
The characteristics of this pattern are:

- There are two servers per cluster and each server has identical EPL statements.
- Each server receives all input events and performs all calculations.
- All servers in the cluster are primary servers.
- The downstream system receives output events from each server (two of everything).

4.3. Hot-Standby with Shared Storage Pattern

In this pattern there are two cluster members that are in a hot-standby relationship. The EsperHA store (file system or db) is accessible to both servers. Only the primary server receives all input. The standby secondary server waits to take over from the primary server.

The following diagram illustrates this pattern:



The characteristics of this pattern are:

- Critical event processing application state is written to a shared storage. This is functionality provided by EsperHA. The shared storage can be a disk, a disk on a storage area network (SAN), a disk on a network-attached storage (NAS) or can be a relational database.
- The primary server receives all input events and performs all calculations.
- When the primary server fails the secondary server takes over. This is detected by means of heartbeat or operationally. The secondary server recovers state from shared storage including event processing continuous queries.

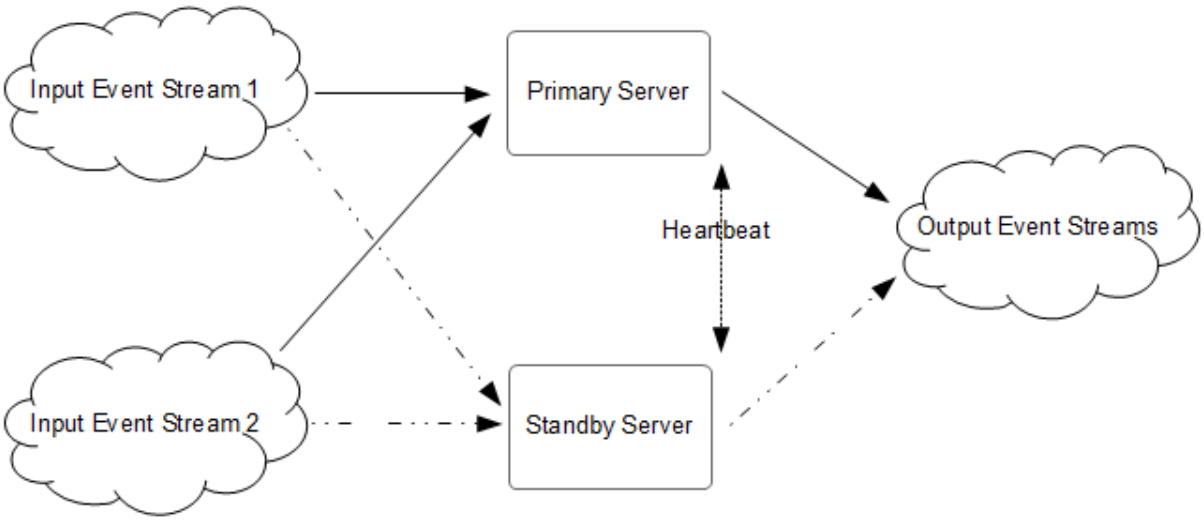
4.3.1. Disaster Recovery Scenario

To implement a disaster recover scenario, an offsite implementation can combine the hot-standby with shared storage pattern. The disaster recovery site can run an identical deployment by means of shared storage implemented over a network connection using either SAN or relational database and replication.

4.4. Hot-Standby Stateless Pattern

This pattern is easy to set up and most suitable for entirely stateless or nearly stateless event processing. In this pattern there are two cluster members that are in a hot-standby relationship. The two servers do not share storage. Only the primary server receives all input. The standby secondary server waits to take over from the primary server.

The following diagram illustrates this pattern:



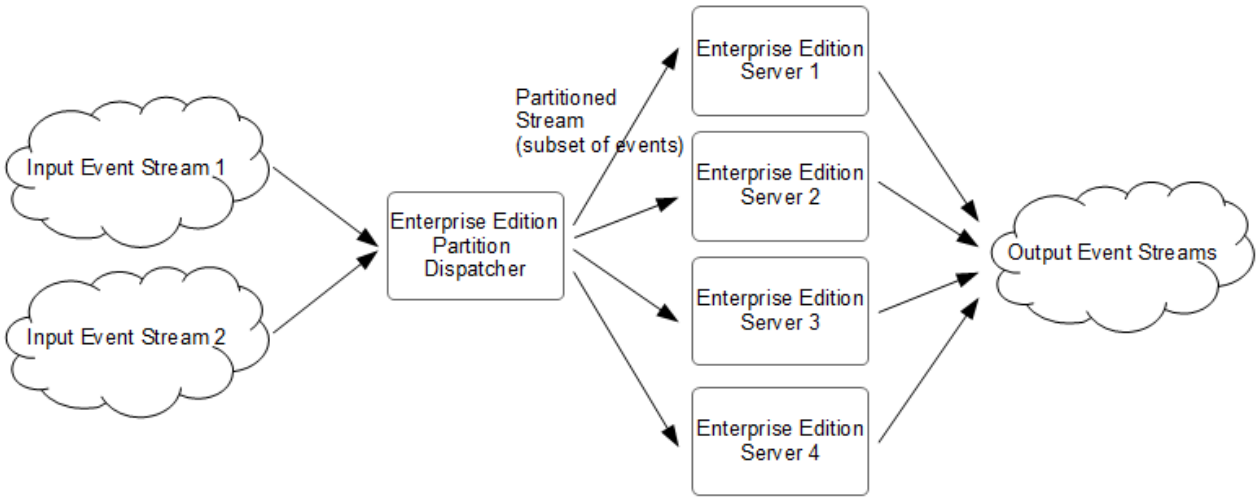
The characteristics of this pattern are:

- The primary server receives all input events and performs all calculations.
- When the primary server fails the secondary server takes over. This is detected by means of heartbeat or operationally.

4.5. Partitioned Stream Scalability Pattern

This pattern partitions the stream to scale an Enterprise Edition cluster from one to multiple servers. A stream is partitioned among the nodes in the cluster. Each server receives as input events a partition (a subset) of the stream. Operations applied to a stream are applied by each server in parallel across each partition.

The following diagram illustrates this pattern:



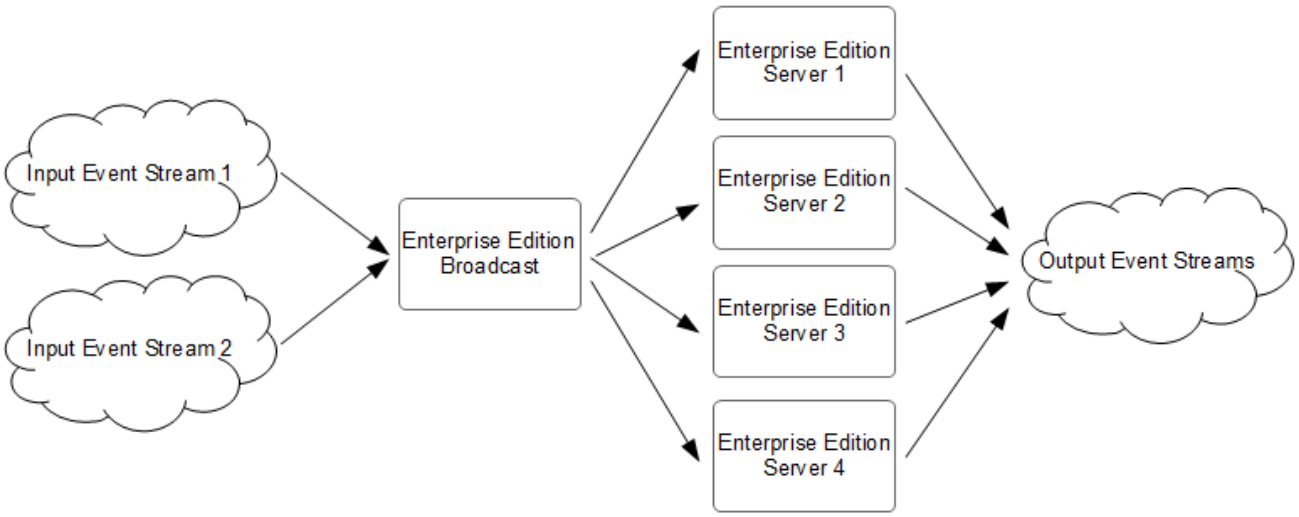
The characteristics of this pattern are:

- Consistent hashing is a common means to partitioning the stream. A second option is assigning individual keys or key ranges to servers.
- An Esper Enterprise Edition server acts as a dispatcher of input stream events, wherein each input stream event is assigned to a single partition (substreams) and dispatched to servers.
- Each server executes identical EPL statements on a subset of input stream events.

4.6. Partition by Use Case Scalability Pattern

This pattern assigns use cases to servers to scale an Enterprise Edition cluster from one to multiple servers. Each server receives all input stream events and executes a subset of EPL statements.

The following diagram illustrates this pattern:



The characteristics of this pattern are:

- Each server receives all input events and performs a subset of EPL statements specific to the use cases assigned to the server.
- All servers in the cluster are primary servers.
- The downstream system receives output events from each server (per use case).

Chapter 5. Hot Deployment of CEP Applications and CEP Engines

Enterprise Edition offers runtime loading and starting as well as stopping and unloading of CEP applications and/or CEP engines.

Enterprise Edition supports hot (runtime, dynamic) deployment of CEP applications. To summarize, you may hot deploy files:

- You may simply create an EPL module text file with the `.epl` extension and drop the file into the `hotdeploy` directory.
- By packaging your CEP application into a J2EE-standard WAR file format you can deploy and undeploy EPL modules together with related Java classes at runtime without requiring a restart.
- By packaging your CEP application into a JAR file format including Esper configuration files, EPL modules and related Java classes with class annotations you may deploy a new CEP engine without requiring a restart.

Your CEP application(s) design may include multiple separate CEP engines. Enterprise Edition allows running any number of CEP engines side-by-side in the same Java VM, each identified by a engine URI. When starting Enterprise Edition server in the default configuration, the server starts a single CEP engine with the URI `default`. When hot deploying EPL and WAR files the default configuration deploys to the `default` engine.

One situation where you may want to use multiple CEP engines is when you require different and incompatible CEP engine configurations. For example, you can start one CEP engine with high-availability configuration and a second engine CEP engine without high-availability.

When starting Enterprise Edition server in the default configuration, the server starts a CEP engine with the URI `default` and monitors the `hotdeploy` folder. The server registers any new files, changes to files or removed files and reacts by performing a new deployment for new files, an undeployment for deleted files or a redeployment (undeploy and deploy) for changed files.

If your CEP application(s) consist of statements only and there is no Java application code or all application code is already present in the container, you may simply copy an EPL module text file with the extension `.epl` to the `hotdeploy` folder or delete the file to undeploy or change the file to redeploy.

If your CEP application(s) consist of EPL statements and related Java application code, you can follow the web application archive (WAR) packaging standard to take advantage of hot deploy, or place the related application code into the classpath and deploy EPL module `.epl` files.

If your CEP application(s) consist of Esper configuration files, EPL modules and custom Java classes with Java class method-level annotations you may package all into a JAR file and deploy to a dedicated CEP engine. We use the name *CEP Engine Jar* for such JAR files.

At time of server startup the server inspects the hot deployment directory for JAR, EPL and WAR files. The server startup processes all JAR files first.

After processing any JAR files, at time of server startup the server inspects all EPL text files and all WAR files including their packaged EPL files. The server analyzes dependencies as declared via `uses`-clause between each EPL module. The server then deploys EPL text files and WAR files in the order as required by all `uses`-clauses, if any. If no `uses`-clauses are specified, the server deploys all EPL text files first and deploys any WAR files next.

When hot deploying to a web application server such as Apache Tomcat instead of Esper Enterprise Edition server, you deployment unit is WAR as your application server will not understand `.epl` EPL module files or `.jar` engine JAR files.

In the default configuration the hot deploy service deploys all files (WAR, EPL) to the default engine. If you have initialized and are running multiple CEP engines in the same JVM process, please see *Section 6.7.1, "Hot Deployer CEP Engine Targeting"* for information on how to target certain files to a given CEP engine.

5.1. EPL Module Hot Deployment

The EPL module file format is described in detail in the Esper documentation under packaging and deployment. The examples in the distribution all have *example.epl* files that contain all EPL statements for each example.

To deploy a new EPL module to Enterprise Edition, create a new text file in the `hotdeploy` folder with the extension `.epl` or copy a file to the folder. As you save or copy the file the Enterprise Edition server deploys the EPL module file.

To undeploy simply remove the `.epl` text file. To redeploy change the `.epl` text file in a text editor or copy a new file over the existing file.

You will see log entries on the console for deployment changes.

5.2. CEP Application WAR Hot Deployment

When your CEP applications consists of EPL modules and related application code or only application code and dynamically registered EPL statements, use the web application archive (WAR) for hot deployment. Please consult J2EE web application standards for a description of the WAR standard.

All Enterprise Edition examples are packaged as a WAR file and get deployed during startup, until you remove the example war files from the `hotdeploy` directory.

To deploy a new CEP application to Enterprise Edition, create a WAR file in the `hotdeploy` folder with the extension `.war` or copy a file to the folder. After you save or copy the file the Enterprise Edition server deploys the WAR file.

To undeploy simply remove the `.war` file. To redeploy copy a new file over the existing file.

When packaging into your WAR file any EPL module files (text files with EPL statements that carry the `.epl` extension), you have the following options:

1. Enterprise Edition server can automatically deploy all `.epl` files found in your WAR file. For this purpose declare the `EsperPrepackagedServletContextListener` in the `web.xml` descriptor.
2. Enterprise Edition server can deploy only the named `.epl` files from your WAR file. Declare the `EsperParameterizedServletContextListener` in your `web.xml` descriptor.
3. Load and start any EPL modules or EPL statements in your application code.

5.2.1. Auto-Deploy of Packaged EPL Modules

Enterprise Edition server can automatically deploy all `.epl` EPL modules that are part of the WAR file.

Declare the following listener in the `web.xml` deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
...
  <!--
    Listener to deploy all EPL files (.epl) packaged with the WAR file.
  -->
  <listener>
                                                                    <listener-
class>com.espertech.esper.server.webapp.EsperPrepackagedServletContextListener</
listener-class>
  </listener>
...
</web-app>
```

There are no further context parameters needed. The deployment takes place to the configured default engine URI.

Use this option when deploying to Enterprise Edition server or use below options. Do not use this option when deploying to Apache Tomcat or another web application server.

5.2.2. Configured Deploy of Packaged EPL Modules

Enterprise Edition server can deploy the `.epl` EPL modules that are part of the WAR file and that are explicitly listed as context parameters.

Use this option when deploying to Apache Tomcat or another web application server. We provide the servlet listener in the `lib` folder in jar `esper-server-`*version*`.jar`.

Declare the following listener and context parameters in the `web.xml` deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
...
  <listener>
                                                                    <listener-
class>com.espertech.esper.server.webapp.EsperParameterizedServletContextListener</
listener-class>
  </listener>
  <context-param>
    <param-name>com.espertech.esper.context-param.uri</param-name>
    <param-value>default</param-value>   <!-- The engine URI to use. -->
  </context-param>
  <context-param>
    <param-name>com.espertech.esper.context-param.modules</param-name>
    <param-value>sample.epl</param-value>
  </context-param>
...
</web-app>
```

The servlet context listener attempts to resolve the EPL file from the classpath and the servlet context real path (deployment directory) or via URL. Once successfully resolved the listener parses and deploys the EPL module statements. A comma-separate list of EPL files may be provided instead.

5.3. CEP Engine Jar Hot Deployment

You may package CEP application(s) into a JAR file. The hot deploy service initializes a separate CEP engine for each JAR file. When you remove the JAR file, the hot deploy service destroys the associated CEP engine. The hot deploy service inspects Java classes packaged into the JAR file and acts upon annotations it finds.

Enterprise Edition ships with an example named *EngineJar* that builds a JAR assembly in the format described here and that demonstrates the use of Java class annotations.

The JAR file contents are all optional and are as follows:

- Place the Esper configuration into the file `conf/esper.xml`.
- Place the EsperHA configuration into the file `conf/esperha.xml`.
- Place any EPL module files in any folder in the JAR file, preferably the root folder of the JAR, as the hot deploy service continues monitoring that folder for changes to EPL files.
- Place any Java classes into the `classes` folder. These are added to the classpath for the CEP engine. The hot deploy service finds all annotated classes and acts upon each annotated method.

- Place any additional jar files into the `lib` folder. These are added to the classpath for the CEP engine.
- Place a `META-INF/MANIFEST.MF` file into the JAR file and add the `EsperEngineURI:` *engine_uri* entry that specifies the engine URI.

For each such JAR file placed into the hot deploy directory the hot deploy service follows these steps:

1. First, the hot deploy service assigns a CEP engine URI to the JAR file as follows: If the `META-INF/MANIFEST.MF` file exists in the JAR file the value of the `EsperEngineURI` entry is the engine URI. If the file does not exist or the `EsperEngineURI` entry is not found the hot deploy service uses the file name of the JAR file as the engine URI. If a CEP engine with the same engine URI has already been initialized the hot deploy service cancels the engine jar deployment.
2. Next, the hot deploy service unpacks the JAR files to a directory on the same level as the hot deploy directory. The hot deploy service also monitors that directory for any changes to EPL files and automatically deploys, redeploys or undeploys any new, changed or removed EPL files (not recursive).
3. Next, the hot deploy service adds any Java classes in the `classes` directory and all JAR files in the directory path to the classpath.
4. Next the hot deploy service creates an engine configuration from the provided configuration files, if any are provided.
5. Next the hot deploy service inspects all Java classes packaged into the JAR file and acts upon annotations as described below. The hot deploy service finds methods marked with the `@SingleRowFunction` annotation in the Java code and adds each to the configuration as a plug-in single-row function.
6. Next, the hot deploy service initializes the CEP engine.
7. Next, the hot deploy service deploys all EPL modules to that CEP engine in the order as indicated by the `uses`-clause, if present.
8. Finally, the hot deploy service finds methods marked with the `@EngineInitializer` annotation in the Java code and invokes each such initializer method.

Consider the example `enginejar` JAR project shipped with the distribution as a starting point. The example contains Maven assembly instructions that creates a suitable JAR file. Please see *Section 10.6, "Hot Deploy Engine-Jar Example"* for more information.

5.3.1. Annotations

The hot deploy service recognizes annotated Java classes provided in the JAR file.

Specify the `@SingleRowFunction` annotation for each method that provides a plug-in single-row EPL function as a public static method. Such a method may accept any number of parameters.

The next example declares a single-row function by name `containsAny` that takes two array parameters and that returns a boolean value:

```
@SingleRowFunction(name = "containsAny")
public static boolean ContainsAny(Object[] lhs, Object[] rhs) {
...
}
```

Specify the `@EngineInitializer` annotation for each method that should be called after the CEP engine initialization completed and all EPL modules are deployed. The purpose of this method typically is to look up statements by statement name and attach listeners or a subscriber. The method must be public static and accept a single String-type parameter that receives the engine URI.

The next example declares a method that takes the engine URI as a parameter and that the hot deploy service invokes after deployment completed:

```
@EngineInitializer
public static void initEngine(String engineName) {
...
}
```

Chapter 6. Server And Deployment Cookbook

6.1. Deployment Options

6.1.1. Out-of-the-Box Deployment

Esper Enterprise Edition comes with a server that you may start and stop via command line using scripts provided in the `bin` folder of the distribution.

Enterprise Edition sets up classpath from `setclasspath` scripts and the server code loads the `conf/esper-default.xml` default configuration file. Change the classpath script to add or remove from classpath as the server does not automatically add all files in the `lib` folder to the classpath.

The default configuration file starts an embedded JMS provider, all REST services, web application server and deploys the examples:

1. The plug-in for CEP Push Services (endpoint config) which handles data push and subscription management from the CEP engine to clients or other consumers.
2. The plug-in that starts a JMS provider. Enterprise Edition starts Apache ActiveMQ as the JMS provider and any other JMS provider is also supported with minor configuration changes.
3. The web application container plug-in, hosting the web client application. Enterprise Edition starts embedded Jetty as the web application container.
4. The Esper JMX plug-in for remote server management and shutdown. You may remove this plug-in when deploying to another servlet container or web application server.
5. The examples are located in the `hotdeploy` folder of the distribution and are packaged as web application archive files.

6.1.2. Other Deployment Options

Most features of Enterprise Edition are also available to applications that embed Esper in their own code. It is therefore not always necessary to run Enterprise Edition server. Enterprise Edition provides a set of REST-style web services that adhere to the JAX-RS standard. This enables your application to expose these services as part of your application. The EsperHQ GUI application can communicate with your application via the services.

Enterprise Edition provides J2EE-standard WAR files that can be deployed to servlet containers or web application servers, running on same or different hosts, with minimal configuration changes.

We test deployment to the Apache Tomcat web container as part of each release. Other servlet containers may have limitations or additional configuration requirements.

The prepackaged WAR files can be found under the installation root in the `lib/webapp-war` folder.

Enterprise Edition provides the following two web application archives:

- The Esper core WAR contains only Esper, CEP Management REST Services and CEP Push REST Services (no client, for the server tier). The file is named `espercorewar.war`.
- The EsperHQ web application contains the web-based client application and HQ Services REST Services (no CEP engine, for the web client tier) and is named `esperhqapp.war`.

Before deploying either of the WAR files, please follow the configuration instructions below.

If using push services, the JMS provider (broker) for communication between EsperHQ web application and Esper+CEP Push Services can be any JMS provider. If you have an existing JMS provider in your architecture, point the configuration files to that provider. We give instructions for configuration in *Section 6.2, "JMS Provider Configuration"*. If you wish to use Apache ActiveMQ that ships with Enterprise Edition then the default configuration files already start the JMS provider for you.

6.1.3. Esper Core WAR

The `espercorewar` web application archive (WAR) packages Esper and configures a Push Services Endpoint. It exposes CEP Management REST services and CEP Push Services REST services. Deploy this WAR file to the servlet container that should act as a CEP server only. This WAR file does not contain a client.

The WAR file packages a `web.xml` configuration, an Esper configuration file by name `esper-default.xml` and an CEP Push Services configuration file by name `ceppushsvc-default.xml`.

In the WAR file `web.xml` you may list one or more engine URIs and configuration file URLs. The default web configuration is shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <listener>
    <listener-class>
      com.espertech.esper.webapp.cepcore.CoreServletContextListener
    </listener-class>
  </listener>
  <context-param>
    <param-name>com.espertech.esper.context-param.uris</param-name>
    <param-value>default=esper-default.xml</param-value>
  </context-param>
</web-app>
```

The servlet context listener reads the `com.espertech.esper.context-param.uris` context parameter and initializes one or multiple Esper engines. The default configuration above initializes an Esper engine for URI `default` and reads the `esper-default.xml` configuration file from classpath (all files are packaged with the WAR).

The synopsis for the servlet context parameter value is:

```
engine_uri[=configuration_file_url][,...]
```

An example parameter value may looks as follows:

```
<context-param>
<param-name>com.espertech.esper.context-param.uris</param-name>
<param-value>engineUriOne=esperee-one.xml,engineUriTwo=esperee-two.xml
,engineUriThree=file://mydir/myconfigfile.xml</param-value>
</context-param>
```

When deploying to Apache Tomcat, copy the WAR file to the `webapps` folder on your Tomcat installation. At WAR initialization time you should see log messages indicating that engines are initialized and during WAR undeployment you should see log messages indicating that engines are destroyed.

Add IO adapters, event types, plug-ins to be loaded or any additional engine configuration to the `esper-default.xml` configuration file.

Follow the JMS provider instructions below and change `ceppushsvc-default.xml` configuration file to point to the JMS provider of your choice. The default configuration expects that an Apache ActiveMQ JMS provider is running and listening at the default port.

For realtime streaming, the EsperHQ client application requires a web container that provides a web socket implementation. EsperHQ currently only supports the Jetty web socket implementation. Therefore when deploying into Tomcat or another web application container the web socked communication is not available.

6.1.4. EsperHQ Web Application WAR

The EsperHQ web application archive (WAR) packages the web client application and HQ Services REST services. Deploy this WAR file to the servlet container that should act as a web client.

Follow the JMS provider instructions below and change `esperhq-default.xml` configuration file to point to the JMS provider of your choice. The default configuration expects that an Apache ActiveMQ JMS provider is running and listening at the default port. We provide detailed information how to configure EsperHQ web application in a separate document in the `docs` folders of the distribution.

The EsperHQ client web application expects the context root of the web application to be `esperhqapp`.

When deploying to Apache Tomcat, copy the WAR file to the `webapps` folder on your Tomcat installation. If deploying a prepackaged WAR file with version number, rename the directory

removing the version number from the directory name so the WAR directory is simply `esperhqapp`, to ensure Tomcat sets the context path to `esperhqapp` without the version number.

6.2. JMS Provider Configuration

6.2.1. Other JMS Provider Configurations

Herein we provide sample configuration entries for two additional JMS providers. For JMS providers not listed below please check with your JMS provider on JNDI context lookup settings.

For Tibco EMS, use the following settings:

```
<context object-name="QueueConnectionFactory">
 <env name="java.naming.factory.initial"
      value="com.tibco.tibjms.naming.TibjmsInitialContextFactory"/>
 <env name="java.naming.provider.url" value="tibjmsnaming://host:port"/>
 <env name="java.naming.security.principal" value="jndiUsername"/>
 <env name="java.naming.security.credentials" value="jndiPassword"/>
</context>
```

For Apache QPid, use the following settings:

```
<context object-name="qpidConnectionfactory">
 <env name="java.naming.factory.initial"
      value="org.apache.qpid.jndi.PropertiesFileInitialContextFactory"/>
 <env name="connectionfactory.qpidConnectionfactory"
               value="amqp://guest:guest@clientid/prod-only?brokerlist='tcp://
hostname:port'"/>
</context>
```

6.2.2. ActiveMQ Stand-alone JMS Provider Configuration

Please follow the step-by-step instructions for using an external ActiveMQ JMS provider as follows:

1. Back up your production configuration files in the `conf` folder before making changes to production.
2. Download the latest stable ActiveMQ distribution version 5.x from `http://activemq.apache.org/download.html`, for your platform.
3. Install the distribution following the steps in `http://activemq.apache.org/getting-started.html`. Start the ActiveMQ JMS broker following the steps in "Starting ActiveMQ".
4. Open the `conf/esper-default.xml` and remove or comment-out the plugin-loader for `JMS_Provider_Bootstrap`. This disables the embedded JMS broker. Start the server.

The `jconsole` tool can help confirm the correct configuration. In `jconsole` you should not see an entry for `com.apache.activemq` when attaching to the server process. The ActiveMQ broker should show as a separate process.

6.3. EPL Module

A good way to develop a CEP application is to externalize EPL statements into an EPL module file (a text file with the `.epl` extension) and use hot deployment to test and deploy the CEP application. The examples provide a template to start from.

The Esper packaging and deployment section describes EPL module files in detail. The Enterprise Edition server `hotdeploy` folder holds the currently deployed CEP applications: plain text files with the `.epl` extension for EPL modules or WAR archives for EPL packaged with application code.

Additional choices for creating EPL statements or CEP applications are:

The first option is to write a Java class that implements the `PluginLoader` interface as described in the Esper reference manual, list the plugin loader in the Esper XML configuration file and use the provided Esper Enterprise Edition server to execute.

The second option is to use any of the Esper provided examples or the benchmark tool and customize the example to handle your application needs.

The third option is write a simple Java main that starts an Esper engine, reads EPL statements from your own source and sends events.

6.4. Configure Event Types

The `create schema` syntax allows registering event types. When using `create schema` the event type is dynamic and gets removed when the EPL module or statement declaring the event type gets destroyed.

To add a predefined event type, the Esper configuration file or by means of the API you may register event types for use by input and output adapters as well as EPL statements.

6.5. Setting up Input and Output Data Feeds

For setting up input and output data feeds, we recommend to check the EsperIO prebuilt dataflow operators and input/output adapters. Declare a dataflow that incorporates the input or output adapter in combination with the `EventBusSource` or `EventBusSink` operators that allow you to send events into the engine. Use a post-startup script as outlined above to have the engine automatically execute dataflow instances on startup.

If no prebuild dataflow operator or adapter can be found, we recommend writing an input or output adapter as a dataflow operator. Thus an adapter dataflow operator can easily be configured and dataflow runtime instances can be managed by the GUI. Please see the Esper dataflow documentation for more information.

Alternatively to using dataflow operators your application could utilize the `PluginLoader` interface, which is designed for this purpose and discussed in the Esper reference manual API section, and the plugin can be listed in the Esper XML configuration file to start at the time of server startup.

There are also a number of examples as part of the Esper distribution, in Esper JIRA or other web sites that may offer examples of an input or output adapter that suits your needs.

The Esper CEP engine produces easy-to-understand Java class, Map-based, XML-formatted or JSON-formatted output events (aka. renderers). You can utilize a number of existing Java components available that can map a Java class to a certain transport or format.

6.6. Integrating into an Existing Java Server Process

If you have an existing Java application and would like to integrate Esper and REST web services for use with the rich GUI client continuous displays then this section describes the steps to follow.

First, determine if you want to use an existing JMS provider (aka. JMS broker) that your technical environment provides. You may use the Esper Enterprise Edition JMS provider as outlined in the CEP Push Services manual.

Second, create an CEP Push Services configuration file to attach to the JMS provider of your choice. Incorporate the required dependent jar files of CEP Push Services (see `lib` folder) into your existing server process.

Third, CEP Push Services can be started either via Esper XML configuration or via its endpoint API as described in the CEP Push Services manual.

6.7. Configuring Web Application Server Settings

Enterprise Edition in the default configuration starts an embedded Jetty web application container. The plugin providing this functionality is `com.espertech.esper.server.webapp.WebAppPlugin`. This chapter outlines configuration options.

The web application container default port is `8400`.

The embedded web application provider acts as a container for the following web applications and servlets:

- EsperHQ web application, a JavaScript and HTML 5 application found in the directory `webapps/esperhqapp` under the installation root.
- REST Services: CEP Management, CEP Push services, HQ services; As further described in *Chapter 9, REST Services*
- Web Socket full-duplex communications channels over a single TCP connection, as standardized by the IETF.

You may customize this configuration and add items or remove items that are not desired depending on your needs, for example to reduce startup time.

The default configuration is:

```
<plugin-loader name="EsperHQ_Webapp_Service"
      class-name="com.espertech.esper.server.webapp.WebAppPlugin">
  <init-arg name="port" value="8400" />
  <init-arg name="webapps" value="esperhqapp@webapps/esperhqapp"/>
  <init-arg name="rest" value=
    "cepmgmtsvc@token=cepmgmtsvctoken;ceppushsvc@token=ceppushsvctoken"/>
  <init-arg name="websocket" value="hqpush" />
</plugin-loader>
```

The `port` parameter value is an integer port number that applies to all web applications, REST services and web sockets that are listed. You can change the port number.

The `webapps` entry is a list of comma-separated entries that name the web applications that the web container deploys. Each entry in the list has the format *context-root@directory*. You can remove this entry to skip deployment of the web application. We require a context root of `esperhqapp` for the EsperHQ application. You can add entries to deploy additional web applications. The `esperhqapp` web application includes HQ Services REST services.

The `rest` entry is a list of comma-separated entries that name the REST JAX-RS-standard services that the web container deploys. Each entry in the list has the format *name@parameters*. You can remove entries to skip deployment of the respective REST services. The special `token` parameter defines a shared secret for the purpose of authenticating HQ Services. Please see the REST services documentation below for more information.

The `websocket` entry lists the web socket endpoint. You can remove this entry to disable web socket communication. We require the name `hqpush`.

You may additionally configure page access log logging by providing the parameter:

```
<init-arg name="log" value="log_file_dir"/>
```

The log file directory must already exist at startup time. Please create the directory when using this setting. When the log file directory is a relative file path, the log file directory must be found in the `logs` directory under the installation root.

You may configure whether hot deployment is enabled, disabled or startup-only. By default hot deployment is enabled.

If hot deployment is disabled, the server does not use the hot deployment directory. Please set as follows:

```
// Disabled means the hotdeploy directory is not used at all
```

```
<init-arg name="hotdeploy" value="disabled"/>
```

If hot deployment is set to startup, the server considers only those files present at server startup time and does not allow file update. Files added, removed or updated are not considered. Please set as follows:

```
// Startup means only files present on server start are considered
<init-arg name="hotdeploy" value="startup"/>
```

You may configure the name of the hot deployment directory:

```
<init-arg name="hotdeploy-dir" value="directory_name"/>
```

You may configure the frequency in seconds for use in scanning the hot deployment directory. By setting this value to zero the server scans the hot deployment directory only once at time of startup and does not scan thereafter. The default frequency is 3 seconds.

An example initialization parameter for the scan interval is:

```
<init-arg name="scanInterval" value="3"/>
```

6.7.1. Hot Deployer CEP Engine Targeting

If you have initialized and are running multiple CEP engines in the same JVM process and intend to target certain files to a given CEP engine, the hot deploy service provides a setting that can be used to associate a file name to a CEP engine URI via regular expression matching on the file name.

The synopsis for the `hotdeploy-targets` parameter value is as follows:

```
engine-uri=filename_regex[;engine-uri=filename_regex[;...]]
```

Provide a semicolon-separated list of pairs of CEP engine URI and file name regular expression to define the CEP engine target URI for a given file name. The hot deploy service inspects each regular expression in the order provided and matches it against the deployment unit file name. If the regular expression matches or no regular expression was provided, the hot deploy service stops at the first match and deploys to the given CEP engine identified by the URI. If no matches are found, the hot deploy service targets the default CEP engine.

The following example assigns all files starting with `cepone_` to deploy to a CEP engine with URI `cepone`, all files starting with `ceptwo_` to deploy to a CEP engine with URI `ceptwo` and all remaining files deploy to the CEP engine identified by the `default` URI:

```
<init-arg name="hotdeploy-targets" value="cepone=cepone_.*;ceptwo=cepone_.*"/>
```

You may disable hot deployment for all or a subset of files by using the special URI `null`. When the engine URI `null` matches the file name the hot deploy service simply logs the deployment file and skips deployment.

The following example assigns all files to CEP engine URI `null`, causing the hot deploy service to skip deploying any files:

```
<init-arg name="hotdeploy-targets" value="null=.*"/>
```

Chapter 7. Socket Transport

Enterprise Edition includes a socket transport for use with scalability and high-availability patterns in which multiple Enterprise Edition servers are configured in a cluster to achieve a common goal.

Socket transport is a point-to-point transport for sending events between server instances, for example when partitioning streams or partitioning use cases between servers.

The default protocol utilized by socket transport is polling, blocking and buffered. We use the abbreviation PBB. The protocol has the receiving server poll the sending server asking for a given maximum number of events. If the last poll returned no messages, the receiving operator sleeps for a short interval. Blocking queues are utilized by the sending and receiving operators to provide backpressure and hand-off.

7.1. Getting Started

Socket transport consists of two dataflow operators, `SocketE2ESink` and `SocketE2ESource`. Use `SocketE2ESink` on the originating server to send events to another server. Use `SocketE2ESource` on the receiving server to receive events sent from another server.

Before using socket transport, it is necessary to import the socket transport data flow operators and io package. No additional configuration is required in order to run socket transport. The jar file for socket transport is `esper-dataflow-sockete2e-`*version* and is already part of the server classpath.

Please add the following two lines to the Esper configuration file `conf/esper-default.xml`:

```
<esper-configuration>
    <auto-import import-name="com.espertech.esper.transport.sockete2e.*"/>
    <auto-import import-name="com.espertech.esper.dataflow.io.*"/>
    ...
</esper-configuration>
```

The examples below declare the originating and the receiving data flows. The next section explains configurations is greater detail. After declaring the data flows, simply instantiate and start the data flow on each server either through the GUI or Esper APIs such as through a post-startup script.

A sample minimal data flow declaration for use on the originating server is shown next:

```
// Deploy on the originating server
create dataflow OriginatingFlow
  EventBusSource -> stream<OrderEvent> {}

  SocketE2ESink(stream) {
```

```
    settings: {
      class: 'SocketE2ESinkSettings_PBB',
      hosts : '10.0.0.1:60001',
      collector: {class:'ObjectToDataOutputCollectorSerializable'}
    }
  }
```

The sample data flow above specifies that any `OrderEvent` events received by Esper (operator `EventBusSource`) are forwarded to the socket transport sink for transmission to another server.

The next data flow declaration is for use on the receiving server:

```
// Deploy on the receiving server
create dataflow ReceivingFlow
  SocketE2ESource -> stream<OrderEvent> {
    settings: {
      class: 'SocketE2ESourceSettings_PBB',
      hostname : '10.0.0.1', // the interface bind address
      port : 60001,
      collector: {class:'DataInputToObjectCollectorSerializable'}
    }
  }

  EventBusSink(stream) {}
```

The sample data flow above specifies that any `OrderEvent` events received by socket transport source are forwarded to the Esper event bus (operator `EventBusSink`) for further processing.

7.2. `SocketE2ESink` Configuration

Specify the `settings` property of the data flow operator to hold an instance of `SocketE2ESinkSettings_PBB`.

Parameters are:

Table 7.1. SocketE2ESinkSettings_PBB Parameters

Name	Description
hosts (required)	A comma-separated list of host names and port numbers.
collector (required)	The object responsible for serializing event objects to `DataOutput`.
socketTimeout	Double-type socket timeout in seconds, defaults to 5 seconds.
connectTimeout	Double-type connect timeout in seconds, defaults to 5 seconds.

Name	Description
queueSize	Int-type queue size, defaults to 10000.

Provide a list of host names and ports to have the sink connect to the first listed and available host.

In the below configuration the operator will attempt to connect to host `10.0.0.1` on port 60001. When the connection fails, the operator attempts to connect to host `10.0.0.2` on port 60001. When that connection fails, the operator attempts to connect to the first listed host and port again.

```
hosts : '10.0.0.1:60001,10.0.0.2:60001'
```

The `collector` parameter is required and must provide the class name of an implementation of the `com.espertech.esper.client.dataflow.io.ObjectToDataOutputCollector` interface. The collector is responsible for serializing the event object to `java.io.DataOutput`.

The `ObjectToDataOutputCollectorSerializable` serializes using Java serialization. For best performance, consider providing a collector implementation.

7.3. `SocketE2ESource` Configuration

Specify the `settings` property of the data flow operator to hold an instance of `SocketE2ESourceSettings_PBB`.

Parameters are:

Table 7.2. SocketE2ESourceSettings_PBB Parameters

Name	Description
hostname (required)	The host name or ip address to use for providing the socket server i.e. bind interface.
port (required)	The port number to use for providing the socket server.
collector (required)	The object responsible for serializing event objects to `DataOutput`.
maxPoll	Int-type parameter setting the maximum number of events to receive in one round trip, defaults to 100.
maxLatency	Int-type parameter setting the wait time between polls when a poll returns no data, defaults to 100 milliseconds.
queueSize	Int-type queue size, defaults to 10000.

The `collector` parameter is required and must provide the class name of an implementation of the `com.espertech.esper.client.dataflow.io.DataInputToObjectCollector` interface. The collector is responsible for de-serializing the event object from `java.io.DataInput`.

The `DataInputToObjectCollectorSerializable` de-serializes using Java serialization. For best performance, consider providing a collector implementation.

7.4. JMX Monitoring

Both source and sink operators register JMX MBeans with the platform MBean server that provide key statistics.

Please enable the platform MBean server access by setting `JAVA_OPTS=-Dcom.sun.management.jmxremote.port=9005 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false` on the command line before starting the server.

The source operator on the receiving side provides:

- The `ServerMsgRecv` object provides count, one-minute, five-minute, 15-minute and mean rates of messages received.
- The `ServerPollNumMsg` object provides count of polls and mean, minimum, maximum and x-percentile message count polled.
- The `ServerPollTime` object provides information on time spent polling for more messages.
- The `ServerQueueDepth` object provides current queue depth.

The sink operator on the sending side provides:

- The `ClientMsgSend` object provides count, one-minute, five-minute, 15-minute and mean rates of messages sent.
- The `ClientPollNumMsg` object provides count of polls and mean, minimum, maximum and x-percentile message counts.
- The `ClientQueueDepth` object provides current queue depth.

Chapter 8. Runtime Script Execution

Runtime script execution is disabled in the default configuration to prevent a possible security concern. Please first enable runtime script execution by following the steps below.

You can use a dataflow operator to execute Groovy scripts at runtime as described herein. Groovy is a scripting language very close to Java and is further described in *Groovy Home* [http://www.groovy-lang.org/].

Script execution takes place within a dataflow instance. The dataflow operators provided are `GroovySource` and `GroovySink`. Each operator takes either the inline script or a script filename as a required parameter. Additional parameters may be passed to the Groovy script.

Since script execution is as part of a dataflow instance, this allows you to use EsperHQ GUI as well as REST services as well as Esper APIs to manage script execution.

Scripts may utilize the complete Esper API as well as any other class in the server classpath. Scripts can also receive parameters via GUI or REST services and from the dataflow declaration.

Scripts can be useful to dynamically interact with the CEP engine instance(s) such as for managing statement listeners and other dynamic behavior. Scripts can also execute when an event arrives: This allows interacting with an external resource or triggering an external process based on event data.

Please review the Esper documentation on dataflows for an overview and additional information. Please refer to Esper JavaDoc documentation for API questions.

Unrelated to what is described here, Esper provides JSR 223 script execution as part of EPL statement processing. Please see the Esper documentation for more information.

Also unrelated to what is described here, post-startup script execution is described in *Section 2.10, "Running Post-Startup Scripts"*.

8.1. Enabling Runtime Script Execution

By default, runtime script execution is disabled. You must edit the classpath script adding the required jar file and add an import to the default configuration file.

The jar file `esper-dataflow-groovy-`*version*`.jar` provides the functionality. Please edit the classpath by editing the file `bin/setclasspath.sh` (Linux) or `bin/setclasspath.bat` (Windows) and comment-in the jar file.

Please add the following line to the Esper configuration file `conf/esper-default.xml`:

```
<esper-configuration>
```

```
    <auto-import import-name="com.espertech.esper.groovy.*"/>
    ...
</esper-configuration>
```

8.2. `GroovySink` - Execute Script Upon Event Arrival

The Groovy script must provide an `onInput` method taking a single `Object` type parameter.

A sample dataflow that provides the script inline and that simply prints the provided event is shown below.

```
create dataflow ProcessEventInGroovyScript
  EventBusSource -> stream<MyEvent> {}
  GroovySink(stream) {
    script: '
      public void onInput(Object event) {
        System.out.println(event);
      }
      '
}
```

Implement the methods provided by the `DataFlowOpLifecycle` interface (see Esper docs) to have the engine invoke the Groovy script on initialization, open and close of the operator.

The next sample shows a minimal inline script that, in addition, provides all lifecycle methods:

```
create dataflow ProcessEventWithLifecycleCallbacks
  EventBusSource -> stream<MyEvent> {}
  GroovySink(stream) {
    script: '
      import com.espertech.esper.dataflow.interfaces.*;
      public void initialize(DataFlowOpInitializateContext initContext) {
        System.out.println(initContext);
      }
      public void open(DataFlowOpOpenContext openContext) {
        System.out.println(openContext);
      }
      public void onInput(Object event) {
        System.out.println(event);
      }
      public void close(DataFlowOpCloseContext closeContext) {
        System.out.println(closeContext);
      }
      '
}
```

8.3. `GroovySource` - Executing a Script Once

You may execute a script once upon dataflow instantiation by using the `GroovySource` operator. The dataflow instance transitions to completed, after the script completed executing.

It is not necessary to provide any particular method in the Groovy script. The engine runs from the start of the Groovy script.

The dataflow declaration shown next executes a script stored in a file only a single time, at the time the dataflow is instantiated.

```
create dataflow RunScriptOnce
  GroovySource {
    file: 'my_script.groovy'
  }
```

8.4. `GroovySource` - Originating Events from Groovy

You may originate events provided by a Groovy script with the `GroovySource` operator.

Your script must provide a method `next` that takes a `EPDataFlowEmitter` as its argument. The script can use the `submit` method of the emitter to provide events to the dataflow.

In this example dataflow, the script is an inline script which instantiates `MyEvent` events. The `EventBusSink` operator makes the new events available to EPL statements for processing.

```
create dataflow GroovyOriginatedEvents
  GroovySource -> mystream<MyEvent> {
    script:'
      import com.espertech.esper.dataflow.interfaces.*;
      import com.mycompany.myevents.MyEvent;
      public void next(EPDataFlowEmitter emitter) throws InterruptedException {
        emitter.submit(new MyEvent("Event"));
        Thread.sleep(1000);
      }
    '}
  EventBusSink(mystream) {}
```

8.5. Operator Parameters

Either the `script` or the `file` property must be provided. All parameters can be provided at time of dataflow instantiation.

Any additional parameters to the dataflow operator are passed through to the Groovy script.

Parameters are:

Table 8.1. GroovySink Parameters

Name	Description
script	The Groovy script, or specify the `file` parameter providing the script instead.
file	The file name of the Groovy script, or specify the `script` parameter providing the actual script instead.

When a `file` is specified, the engine resolves the file as an absolute file path, and if not found then relative to the Enterprise Edition base directory, and if not found from classpath.

All parameters are passed through to the Groovy script. In the Groovy you can simply refer to such parameters as variables.

For example, the following dataflow declaration expects that either the file or script are provided at time of instantiation, and passes a value for `threshold` to the Groovy script:

```
create dataflow RunScriptOnce
  GroovySource {
    threshold: '100'
  }
```

Chapter 9. REST Services

Enterprise Edition ships with sets of RESTful web services utilizing JSON to allow easy integration. Web services are based on the JAX-RS standard for the Representational State Transfer (REST) architectural pattern.

The following table summarizes the REST services:

Table 9.1. REST Services

Name	Description	Subcontext Root
CEP Management Services	Administration of CEP and associated functionality.	`cepmgmtapi/v1`
CEP Push Services	Subscription management and data push.	`ceppushapi/v1`
HQ Services	Addressing multiple CEP engines (aka. endpoints); HQ web client services.	`hqapi/v1`

We provide detailed information about services as part of the REST services documentation set linked to the `index.html` file in the installation root and in the `doc` folder of the installation.

Individual or all REST services can be disabled by EsperHQ configuration.

9.1. Error Handling

A successful service invocation returns HTTP status code 200 OK.

All service invocations that fail return any of the following HTTP status codes:

Table 9.2. HTTP Status Codes

HTTP Status Code	Description
404 Not Found	Returned when a resource was not found.
405 Method Not Allowed	Returned when an operation is not allowed on a resource.
400 Bad Request	Returned when the service fails or the data passed to the service is invalid.
500 Internal Server Error	Returned when there is an internal error. In this case, please inspect the server logs and report to us with as much information as possible so we can reproduce the issue.

All error responses have a response body that contains a JSON-formatted object that has a `message` and a `details` property. The `message` property contains the reason text. The `details`

property is for support and error handling. It contains a list of exceptions as additional information to the error.

A sample response body indicating an error is:

```
{
   "message":".... service message here ....",
   "details":[
      {
         "exceptionClass":"java.lang.SampleExceptionClass",
         "exceptionMessage":"....same or additional message....",
         "exceptionStackTrace":".... stack trace here...."
      }
   ]
}
```

Each object under `details` may contain:

Table 9.3. REST Services

Field	Description
exceptionClass	The exception fully-qualified class name.
exceptionMessage	The exception message.
exceptionStackTrace	The exception stack trace, for support.
line	A line number, where applicable.
expression	Expression causing the exception, where applicable.
fieldName	The name of the JSON property that the exception refers to, where applicable.

9.2. Security

The EsperHQ JavaScript application invokes only HQ Services (REST), and not CEP Management (REST) or CEP Push Services (REST). Please consult the Esper HQ Web Application Archive documentation for information on securing the web application including HQ Services.

HQ Services REST invocations invoke CEP Management REST Services and CEP Push REST Services. This communication is secured by HQ Services passing a hashed token value as a query parameter for every REST service invocation.

Please consult the Esper HQ Web Application Archive documentation for information on configuring the token shared secret that HQ Services passes to CEP Management and CEP Push Services, specific to each endpoint.

Please review the section *Section 6.7, "Configuring Web Application Server Settings"* in this document for information on configuring the token value expected and verified by CEP Management and CEP Push Services. The token strings must match the HQ Services token strings.

9.3. Exposing REST Services When Not Running Enterprise Edition Server

If you have an Esper-embedded application and would like to expose REST-style web services from that application, then this section provides the relevant instructions.

By having your application expose web services, the EsperHQ web client application can then invoke the web services. Thereby you can use EsperHQ to manage, monitor or debug your application.

Please follow these steps:

1. Set up the application classpath.
2. Register the REST JAX-RS application(s) and expose an HTTP server (unless already provided).
3. Configure EsperHQ for the new host and port.

9.3.1. Classpath Requirements

The following table outlines the classpath additional jar files that must be placed in your application classpath.

Table 9.4. Required Jar Files

Jar File	Description
`esper-eeutil-`*version*`.jar`	The Enterprise Edition utility jar for handling REST-style communication essentials.
`esper-cepmgmtsvc-`*version*`.jar`	CEP management services.
`gson-`*version*`.jar`	Library to convert JSON to objects and objects to JSON. We recommend version 2.3.1 or newer.
`javax.ws.rs.jar`	JAX-RS web services apis. Required as our REST services adhere to JAX-RS standards.
`org.restlet.jar`	Restlet framework (see http://restlet.org). We recommend version 2.3.1 or newer.
`org.restlet.ext.jaxrs.jar`	Restlet framework JAX-RS support. We recommend version 2.3.1 or newer.
`slf4j-api-1.7.2.jar`	SLF4J Logging.

Jar File	Description
`slf4j-log4j12-1.7.2.jar`	SLF4J Logging for Log4j.

9.3.2. JAX-RS Container

If you do not have an existing HTTP server or JAX-RS container as part of the application, please use the Restlet framework to expose an HTTP server that hosts the JAX-RS application. We provide an example code snippet below.

```
String host = "localhost";
int port = 8401;

// Restlet framework setup (assumes host and port defined)
Component comp = new Component();
Server server = new Server(Protocol.HTTP, host, port);
comp.getServers().add(server);

// Add JAX-RS application
JaxRsApplication                application                =                new
 CepMgmtJaxRsApplication(comp.getContext().createChildContext());

// This example provides a null-token, disabling token authentication.
application.getJaxRsRestlet().addSingleton(new
 TokenValidationServiceProvider(null));

// Attach the application to the component and start it
String attachPath = RestRoots.MGMT_V1.toAttachString();
comp.getDefaultHost().attach(attachPath, application);
comp.start();
```

If you do not have an existing JAX-RS container you can use `CepMgmtJaxRsApplication` which is a JAX-RS standard `JaxRsApplication` application.

For CEP push services, please register `CepPushJaxRsApplication`.

Your application may provide a string token value to `TokenValidationServiceProvider` to enable token authentication.

9.3.3. EsperHQ Configuration

The EsperHQ web application archive configuration is described in EsperHQ Web Application Archive manual in the documentation set.

Please edit the `conf/esperhq-default.xml` file and change to `<restservice hosts="http://host:port"/>`. Remove the token settings if no token was provided to `TokenValidationServiceProvider`.

9.4. Example Client and Server

The distribution contains a small REST web services client that invokes CEP Push Services web services using Apache Commons HTTP Client. The example is self-contained and does not run against the Enterprise Edition server but instead runs against a server also contained in the example, which also demonstrates how to host REST services.

Please find the example code in `examples/examples-ceppushsvc` under the installation root. Further description on running the example is provided as part of Push Services documentation.

Chapter 10. Examples

Esper Enterprise Edition comes with additional examples. The source for the examples can be found in the `examples/examples-server` folder of the distribution and the jar files are in the `lib` folder.

All Enterprise Edition samples are packaged for hot deployment and un-deployment (runtime deployment) as in the format of web application archive (WAR) and also as regular jar files.

The packaged example CEP applications are found in the `hotdeploy` folder of the distribution. Remove an example WAR file from the folder to un-deploy or copy additional WAR files into the folder to deploy.

Each example is also packaged as a JAR file so it can also be listed in an Esper configuration file and therefore started automatically when the Esper engine instances starts. List the examples as Esper `plugin-loader` configurations in the default configuration file `esper-default.xml` that ships with the distribution. Each example implements the Esper `PluginLoader` interface but can also be started via its public API. Use the following configuration to start each example as plug-in:

```
<plugin-loader name="ExampleForGeoEvents"
                                                                    class-
name="com.espertech.esper.server.example.geoapp.GeoExampleSimulatorPlugin">
  <init-arg name="engineURI" value="default" />
</plugin-loader>
<plugin-loader name="ExampleForOptionTrading"
                                                                    class-
name="com.espertech.esper.server.example.optionstrade.OptionTradingExampleSimulatorPlugin">
  <init-arg name="engineURI" value="default" />
</plugin-loader>
<plugin-loader name="ExampleForOnlineShop"
                                                                    class-
name="com.espertech.esper.server.example.onlineshop.OnlineShopExampleSimulatorPlugin">
  <init-arg name="engineURI" value="default" />
</plugin-loader>
<plugin-loader name="ExampleGeneric"
                                                                    class-
name="com.espertech.esper.server.example.generic.GenericSamplePlugin">
  <init-arg name="engineURI" value="default" />
</plugin-loader>
```

Additionally, you may deploy each Esper distribution example to Esper Enterprise Edition as outlined below.

10.1. EsperHQ Dashboard Examples

The dashboard examples display data generated by the online shop example which is outlined below.

There are two dashboard examples: One example is an application page that is generated by the Dashboard Page Builder that is part of EsperHQ client GUI. The second example demonstrates launching eventlets scattered over an HTML page and uses HTML IFrames.

10.1.1. Example Dashboard Page Generated by Dashboard Page Builder GUI

The file name for this example is `Example Dashboard JQuery Eventlet for Onlineshop.html` in the "examples" folder in the GUI itself, or in the `data/hqsvc/examples` folder of the installation root.

This dashboard example demonstrates the output of the Dashboard Page Builder in EsperHQ client. The Dashboard Page Builder generates a complete HTML page including JavaScript utilizing the eventlet jQuery extension.

Please see the EsperHQ client documentation for more information on the eventlet jQuery extension.

10.1.2. Example Dashboard Page Using HTML IFrames and Faceless Launcher

The HTML for this example is file `Example Dashboard With HTML IFrames And Launcher For Onlineshop.html` in the "examples" folder in the GUI itself, or in the `data/hqsvc/examples` folder of the installation root.

This example demonstrates launching eventlets as part of HTML content using the faceless launcher URL, which can be useful for portal environments or content management systems.

Please see the EsperHQ client documentation for more information on the eventlet faceless launcher.

10.2. Geo Example

The Geo example is a simulator that analyzes movement of persons geographically and detects presence and vicinity.

The simulator sends person location events that have a person id and the current location in latitude and longitude of the person. The Geo example statements detect when persons come near to each other and stay near to each other for a given time interval.

The sources for the Geo example are in the `examples-server/geo` folder, the class is `com.espertech.esper.server.example.geoapp.GeoExampleSimulatorPlugin` in the jar file by name `esper-example-geo.jar`.

10.3. Online Shop Example

The online shop example is an example out of the domain of customer experience management. It analyzes events generated by a web site when web site visitors view items for sale on the site.

The simulator generates user browse events with user id, product information viewed by the user, page-enter and leave time and stock quantities and price displayed. It creates statements to obtain key statistics.

The sources for the online shop example are in the `examples-server/onlineshop` folder, the class is `com.espertech.esper.server.example.onlineshop.OnlineShopExampleSimulatorPlugin` in the jar file by name `esper-example-onlineshop.jar`.

10.4. Option Trade Example

The option trade example correlates a market data feed that contains bid and offer prices for instruments with order execution received from an exchange.

The simulator generates market price events and exchange execution events. It creates statements to compute a position and valuation.

The sources for the option trade example are in the `examples-server/optiontrade` folder, the class is `com.espertech.esper.server.example.optiontrade.OptionTradingExampleSimulatorPlugin` in the jar file by name `esper-example-optiontrade.jar`.

10.5. Installing Esper Distribution Examples

First, copy the example jar file `esper-example-`*`name-version`* to the `lib` directory of the distribution.

Second, copy the `etc` directory of the Esper example to the `lib` directory of the distribution.

For Windows environments, edit the `setclasspath.bat` in the `bin` directory of the distribution and add the following lines to the location of classpath setup:

```
set CLASSPATH=%CLASSPATH%;%LIB%\esper-name-version.jar
set CLASSPATH=%CLASSPATH%;%LIB%\etc
```

For Linux environments, edit the `setclasspath.sh` in the `bin` directory of the distribution and add the following lines to the location of classpath setup:

```
CLASSPATH=$CLASSPATH:$LIB/esper-example-name-version.jar
CLASSPATH=$CLASSPATH:$LIB/etc
```

Finally, edit the default engine configuration file `esper-default.xml` in the `conf` folder of the distribution to start the example as part of the configuration.

For example:

```
<plugin-loader                        name="Example"                        class-
name="com.espertech.esper.example.autoid.AutoIdSamplePlugin">
 <init-arg name="engineURI" value="autoid" />
</plugin-loader>
```

When deploying multiple examples, we would recommend using a different engine URI for each example to avoid conflicting configurations between the examples.

Esper distribution examples send events during startup as well as include a simulator that continuously sends events. Some examples such as the named window fire-and-forget queries don't have a continuous event simulator therefore may not produce an event stream to see in the web client, however you can see the statements or named window in the web client.

The complete list of examples is as follows:

```
esper-example-autoid-version.jar
esper-example-marketdatafeed-version.jar
esper-example-matchmaker-version.jar
esper-example-namedwinquery-version.jar
esper-example-ohlcpluginview-version.jar
esper-example-qossla-version.jar
esper-example-rfidassetzone-version.jar
esper-example-stockticker-version.jar
esper-example-transaction-version.jar
```

The complete list of plug-in configurations is as follows:

```
<plugin-loader                    name="ExampleAutoId"                    class-
name="com.espertech.esper.example.autoid.AutoIdSamplePlugin">
  <init-arg name="engineURI" value="autoid" />
</plugin-loader>
<plugin-loader                name="ExampleMarketDataFeed"                class-
name="com.espertech.esper.example.marketdatafeed.MarketDataFeedSamplePlugin">
  <init-arg name="engineURI" value="marketdatafeed" />
</plugin-loader>
<plugin-loader                  name="ExampleMatchMaker"                  class-
name="com.espertech.esper.example.matchmaker.MatchMakerSamplePlugin">
  <init-arg name="engineURI" value="matchmaker" />
</plugin-loader>
<plugin-loader               name="ExampleNamedWindowQuery"               class-
name="com.espertech.esper.example.namedwinquery.NamedWindowQuerySamplePlugin">
```

```
  <init-arg name="engineURI" value="namedwinquery" />
</plugin-loader>
<plugin-loader                    name="ExampleOHLC"                    class-
name="com.espertech.esper.example.ohlc.OHLCSamplePlugin">
  <init-arg name="engineURI" value="ohlcsample" />
</plugin-loader>
<plugin-loader              name="ExampleQualityOfService"              class-
name="com.espertech.esper.example.qos_sla.QualityOfServiceSamplePlugin">
  <init-arg name="engineURI" value="qos_sla" />
</plugin-loader>
<plugin-loader                name="ExampleRFIDAssetZone"                class-
name="com.espertech.esper.example.rfidassetzone.LRMovingSamplePlugin">
  <init-arg name="engineURI" value="rfidassetzone" />
</plugin-loader>
<plugin-loader                 name="ExampleStockTicker"                 class-
name="com.espertech.esper.example.stockticker.StockTickerSamplePlugin">
  <init-arg name="engineURI" value="stockticker" />
</plugin-loader>
<plugin-loader                 name="ExampleTransaction"                 class-
name="com.espertech.esper.example.transaction.TransactionSamplePlugin">
  <init-arg name="engineURI" value="transactionsample" />
</plugin-loader>
```

10.6. Hot Deploy Engine-Jar Example

The distribution provides the `EngineJar` example to demonstrate how to build and hot deploy a CEP engine jar file. Please see *Section 5.3, "CEP Engine Jar Hot Deployment"* for more information.

The `examples/examples-server/enginejar` directory contains two sub-projects. The `enginejar` sub-project builds a JAR file `example-enginejar-`*version*`-assembly.jar` that you may copy to the hot deploy directory. The second sub-project `embeddedlib` builds a library JAR file that is included in the assembly for demonstrating annotations and classpath inclusion of jar files.