CEP Push Services

Version 5.4.0

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

Preface

This document describes Esper CEP Push Services. The document assumes that the reader has prior knowledge of Esper.

If you are new to Esper, please study the tutorials and case studies available on the public web site at `http://www.espertech.com/esper`, and skim over the reference documentation.

The *Chapter 1, Overview* chapter is the best place to start.

Chapter 1. Overview

Esper Push Services (Push Services for short) is a service and publishing component for use with the Esper CEP engine. It provides an application-layer publish-subscribe multi-tiered data distribution service for efficient and scalable delivery of CEP engine data based on active logical subscriptions. In addition, Push Services provides REST web services for managing subscriptions and obtaining ongoing information.

Push Services is built on the Java Message Service (JMS) standard. Push Services utilizes the control bus enterprise integration pattern for providing management services in a request-response communication style over a JMS queue. Push Services manages any number of JMS topics or queue destinations for data distribution.

Why does one need a Push Service? In the simplest case where there is only a single source of data and type of message that goes to a single consumer and there are no requirements around subscription timeout, stale subscriptions or quality of service then one may not need a Push Services.

The idea of the Push Service component is to handle:

1. A dynamic space of data sources (Esper EPL statements, engine metrics etc).
2. Mapping of a logical data sources to CEP engine data sources. For example, a subscription may ask for the dynamic set of Esper statements tagged "Visit" (via the @Tag annotation) and the published data must dynamically include all current and newly created EPL statements tagged.
3. Quality of Service requirements by flexible policy configuration.
4. Tracking and managing subscriptions to efficiently share server resources where possible, to reduce or eliminating duplicate data on the transport.
5. Subscriber timeout, e.g. when a subscriber unexpectedly disappears.
6. Subscription to stale logical data sources, e.g. when a subscribed-to EPL statement gets destroyed.
7. Subscription history.
8. Mapping of logical data sources to the underlying transport (JMS destination(s)) and effective marshalling of payload.

In the next section, *Chapter 2, Key Concepts*, the document introduces the key concepts.

The *Chapter 3, Getting Started* section outlines the dependent jar files and introduces configurations.

The section *Chapter 4, Configuration* section explains the static and runtime configuration options.

Chapter 2. Key Concepts

Before reading this chapter, it is important to understand key concepts of CEP and the section will use Esper CEP engine terminology.

2.1. REST Web Services and Data Bus

Esper Push Services provides a REST web services and manages a dynamic data bus. REST web services manage subscriptions and sessions. The dynamic data bus is zero or more channels for transmission of data at same or different quality-of-service level and transmission message format.

2.2. Quality of Service

In Esper Push Services, the term Quality of Service refers to the policy by which available data is formatted and published to consumers. It defines the event batching that should take place, if any, to efficiently utilize the available transport yet balance latency requirements.

2.3. Subject Space

We use this term to mean any publishable data available through the CEP engine. It means the sets of Esper EPL statements, CEP engine metrics or other metrics, named window, variable or more generally anything that can be turned into a one-time, infrequent or frequent or continuous stream including historical and reference data available through relational databases, for example.

A single subject space is identified by a subject space name. Its definition is an ordered list of subject space expressions including and excluding logical resources from that subject space. A subject space that includes all logical resources simply has no include or exclude expressions.

2.4. Logical Resource

Within the subject space, a logical resource identifies the specific publishable data available through the CEP engine.

One example of a logical resource is a single EPL statement that is uniquely identified by an Esper engine URI and a statement name.

A second example of a logical resource is dynamic set of EPL statements (statement set), for example all started statements within an Esper engine as identified by an Esper engine URI.

A third example of a logical resource is engine metrics for an Esper engine as identified by an Esper engine URI.

2.5. Statement Set

A statement set is a set of statements, for example all statements, or for example all statements that are tagged by a @Tag annotation with certain value(s).

Push Services manages statement sets dynamically: When a new statement comes online (gets compiled and started) that matches a statement set subscription then that new statement automatically becomes part of the resources for publishing.

In order to define which statements to include into a statement set, a subscription may provide a statement set filter expression. If you provide no filter expression with a statement set subscription request then Push Services includes all statements in the subscription. If you provide a filter expression, Push Services only includes those statements that match the filter expression provided.

The syntax for statement set filter expressions is outlined below.

2.6. Channel

Channel is the term used to tie together the subject space (any publishable data) with quality of service requirements and a binding to JMS destination(s).

2.7. Session

Clients (or consumers) that subscribe to logical resources must first establish a session. They are also responsible for keeping the session alive by sending heartbeats, and for indicating the end of a session when the subscription is no longer needed.

Clients to services are unidentified. That means clients that use services to manage CEP engine instances (e.g. that create statements, view variable values etc.) and that do not subscribe to logical resources to receive data are not specifically tracked.

2.8. Endpoint

An endpoint is a running Push Services.

2.9. Provider

A provider is a Java Message Service (JMS) provider (aka. broker).

2.10. Subject Space Expressions

Subject space expressions serve to define a subject space by including or excluding logical resources declaratively.

The table below outlines the properties available in expressions.

Table 2.1. Properties in Subject Space Expressions

Property Name	Description	Example
engineURI	The Esper engine URI.	`engineURI in ('uriOne', 'uriTwo')`

Property Name	Description	Example
statement	Populated if the logical resource i a single statement.	`statement is not null`
statement.statementName	The statement name.	`statement.statementName = 'MyStmt'`
statement.engineURI	The Esper engine URI for statement.	`statement.engineURI = 'uriOne' and statement.statementName = 'MyStmt'`
statement.iterate	Whether to iterate the statement.	`statement.iterate is not null`
statement.iterate.iterateNow	Whether to iterate the statement a start of subscription.	`statement.iterate.iterateNow = true`
statement.iterate.iteratePattern	Pattern expression to use fo iteration.	`statement.iterate.iteratePattern = 'every timer:interval(5)'`
statementSet	Populated if the logical resource i a dynamic set of statements.	`statementSet is not null`
statementSet.engineURI	The Esper engine URI for dynamic set of statements.	`statementSet.engineURI = 'uriOne'`
statementSet.filterExpression	The filter expression identifying dynamic set of statements.	`statementSet.filterExpression like '%tag='Shop'%'`
engine	Populated if the logical resource i an engine metric.	`engine is not null`
engine.engineURI	The Esper engine URI for engin metrics.	`engine.engineURI = 'uriOne'`
engine.type	The type of engine metric.	`engine.engineURI = 'uriOne'`

To understand subject space expressions, read the expression as follows: "For all logical resources where *expression*".

For example, the expression `"statement.statementName = 'MyStmt'"` means a single statement that has statement name `MyStmt`.

As a second example, the expression `"statement.statementName in ('MyStmtOne', 'MyStmtTwo')"` means statements that have statement name of either `MyStmtOne` or `MyStmtTwo`.

As a third example, the expression `"statementSet is not null and filterExpression is null"` means all statements (a dynamic statement set) for a given engine instance.

2.11. Statement Set Filter Expressions

A statement set filter expression is an optional parameter to a subscription for a statement set. A statement set subscription without a filter expression includes all statements.

The table below outlines the properties available in expressions:

Table 2.2. Properties in Statement Set Filter Expressions

Property Name	Description	Example
name	The statement name.	`name in ('MyStmtOne', 'MyStmtTwo')`
epl	EPL expression of statement.	`epl like 'select * from MyEvent'`
state	A statement state: STARTED STOPPED or DESTROYED.	`state = 'STOPPED'`
userObject	The user object (as a string value attached to a statement.	`userObject = 'myUserObject'`
description	Statement description.	`description like '%to be removed%'`
hint	Hint values as a comma separated list.	`hint is null`
priority	Statement priority.	`priority = 1`

Property Name	Description	Example
drop	True if @Drop specified.	`drop = true`
tag	A map property for all tags (se @Tag, string key-value pairs attached to a statement.	`tag('myTag')='myValue'`

To understand statement set expressions, read the expression as follows: "For all statements where *expression*".

For example, the expression `"name = 'MyStmt'"` means a statement that has statement name `MyStmt`.

As a second example, the expression `"name in ('MyStmtOne', 'MyStmtTwo')"` means statements that are named either `MyStmtOne` or `MyStmtTwo`.

As a third example, the expression `"tag('myTag') is not null or tag('mySecondTag') = 'id'"` means all statements that are tagged with `myTag` with any value or that are tagged with `mySecondTag` and value `id`.

Chapter 3. Getting Started

Esper Enterprise Edition comes with Push Service and EsperHQ configured. Please simply follow the directions in file `RUNNING.txt` or the index web page to start the preconfigured server.

After starting the preconfigured server the EsperHQ client application is available at *http://localhost:8400/esperhqapp*.

Esper Push Services consists of the following components:

1. The Push Services configuration file `ceppushsvc-default.xml` (the file name is referenced by the Esper configuration file which is `esper-default.xml`).
2. The Push Services jar file `esper-ceppushsvc-`*version*`.jar`.

Esper Push Services minimally requires the following for operation:

1. Java 7 or higher.
2. An Push Services configuration file.
3. A provider for Java Message Service (JMS), see below.
4. A JAX-RS Web Services container.

Push Services requires a Java Message Service (JMS) provider. By default, Push Services ships with configuration and jar files for Apache ActiveMQ (see http://activemq.apache.org). We do not specifically recommend any particular Java Message Service (JMS) provider.

There are two options to start Esper Push Services:

1. As an Push Services endpoint via the Push Services endpoint API:

   For use when your application requires complete control of Push Services lifecycle (start and stop).

2. As an Esper plug-in as part of an Esper configuration XML file or object:

   This requires your application to add the Push Services plug-in class to the Esper configuration (XML or API). Push Services is then started automatically as part of Esper engine initialization.

The above options are mutually exclusive.

3.1. Push Services Endpoint API

The class that provides an Push Service endpoint is `com.espertech.esper.ceppushsvc.endpoint.EsperPushSvcEndpoint`.

The class for configuring a Push Services endpoint is `com.espertech.esper.ceppushsvc.client.config.ConfigurationPushSvc`.

Your application can start an endpoint with minimal configuration, and assuming the default JMS provider as shipped with the distribution, as follows:

```
ConfigurationPushSvc config = new ConfigurationPushSvc();

// Name a default subject space that encompasses all logical resources.
config.getSubjectSpaces().put("AllDataSubjectSpace", new SubjectSpace());

// Default channel maps to the subject space,
// uses the default QoS and "esper.databus" JMS destination.
Channel defaultChannel = new Channel();
defaultChannel.setName("DefaultChannel");
defaultChannel.setSubjectSpace("AllDataSubjectSpace");
defaultChannel.setJms(new ChannelJMSConfig("esper.databus"));
config.addChannel(defaultChannel);

// Set JMS provider name to use
config.getSettings().setPushProvider(new PushProvider("defaultProvider"));

// Add JMS provider
Provider provider = new Provider();
provider.setObjectName("ConnectionFactory");
Map<String, String> context = new HashMap<String, String>();
context.put(Context.PROVIDER_URL,                        "tcp://localhost:61616?
wireFormat.maxInactivityDuration=0");
context.put(Context.INITIAL_CONTEXT_FACTORY,
 ActiveMQInitialContextFactory.class.getName());
provider.setContext(context);
config.getProviders().put("defaultProvider", provider);

EsperPushSvcEndpoint endpoint = new EsperPushSvcEndpoint(config);
endpoint.start();
```

You may alternatively have the `ConfigurationPushSvc` class read a configuration file via one of the `configure(...)` methods. The distribution provides additional sample configuration files.

Please ensure to use the JMS provider startup scripts or API to start the JMS provider, if required, before starting Push Services.

3.2. Push Services Plug-in Configuration

In alternative to the endpoint API as described above, you may use Push Services as a plug-in adapter to an Esper configuration. The Esper engine initialization then also initializes and starts the Push Services endpoint.

Under this option Push Services is initialized when your application first obtains an `EPServiceProvider` instance for a given URI or when your application calls the `initialize`

method on an `EPServiceProvider`. Push Services is destroyed when your application calls the `destroy` method on an `EPServiceProvider` instance or when it calls the `initialize` method on `EPServiceProvider` instance that had the adapter in its configuration.

3.2.1. Via Esper Configuration XML

The XML as below configures an engine instance with Push Services. It specifies the same configuration options as outlined above.

```
<esper-configuration>
                  <plugin-loader         name="CEPPushServices"         class-
name="com.espertech.esper.ceppushsvc.client.EndpointMgmtPlugin">
    <init-arg name="esperceppushsvc.configuration.file" value="conf/ceppushsvc-
default.xml" />
  </plugin-loader>
</esper-configuration>
```

3.3. Start a JMS Provider

Push Services provides a plug-in to start a JMS provider, for use when your technical environment does not have a JMS provider.

Under this option the JMS provider is initialized when your application first obtains an `EPServiceProvider` instance for a given URI or when your application calls the `initialize` method on an `EPServiceProvider`. The JMS provider is destroyed when your application calls the `destroy` method on an `EPServiceProvider` instance or when it calls the `initialize` method on `EPServiceProvider` instance that had the adapter in its configuration.

The below extract starts the default JMS provider on port 61616 and uses the data directory for JMS provider data files.

```
<esper-configuration>
               <plugin-loader       name="JMS_Provider_Bootstrap"       class-
name="com.espertech.esper.ceppushsvc.jmsbroker.JMSProviderMgmtPlugin">
    <init-arg name="type" value="ApacheActiveMQ" />
              <init-arg    name="connectorURL"    value="tcp://localhost:61616?
wireFormat.maxInactivityDuration=0" />
        <init-arg  name="dataDirectory"  value="${ESPEREE_BASE}/data/jmsbroker-
endpoint-default" />
  </plugin-loader>
</esper-configuration>
```

Chapter 4. Configuration

If using XML to configure Push Services, then your XML configuration file should adhere to the XML XSD schema file provided in the `conf` folder of the distribution by name `ceppushsvc-configuration-5-0.xsd`.

4.1. Subject Spaces

A subject space has a name and a list of includes- and excludes-expressions. The expression syntax for subject spaces has been explained in *Chapter 2, Key Concepts*.

The following XML configuration configures a subject space by name `defaultSpace` that includes all logical resources:

```
<ceppushsvc-configuration>
  <subjectspaces>
    <subjectspace name="defaultSpace"/>
  </subjectspaces>
</ceppushsvc-configuration>
```

The next code snippet shows the configuration API to configure the same subject space:

```
ConfigurationPushSvc config = new ConfigurationPushSvc();
config.getSubjectSpaces().put("defaultSpace", new SubjectSpace());
```

By adding includes- and excludes-expressions to a subject space you may narrow down its scope from all logical resources to only a subset of logical resources.

The next example adds an additional subject space named `sampleSpaceOne` that includes only a specific statement named `MyStmt`:

```
<subjectspaces>
  <subjectspace name="sampleSpaceOne">
    <includes expr="statement.statementName='MyStmt'"/>
  </subjectspace>
  <subjectspace name="defaultSpace"/>
</subjectspaces>
```

The next code snippet shows the configuration API to configure the additional subject space:

```
ConfigurationPushSvc config = new ConfigurationPushSvc();
```

```
SubjectSpace space = new SubjectSpace();
space.addExpression(new                    SubjectSpace.SubjectSpaceEntry(true,
 "statement.statementName='MyStmt'"));
config.getSubjectSpaces().put("sampleSpaceOne", space);
config.getSubjectSpaces().put("defaultSpace", new SubjectSpace());
```

4.2. Channels

A channel has a name, an assigned subject space, a quality-of-service setting and a JMS provider setting.

In the next XML configuration we configure a single channel by name `defaultChannel`. The channel is mapped to subject space `defaultSpace`. The quality of service parameters call for a batched publishing of data every 1000 milliseconds or every 100 events, whichever comes first. The JMS destination for the data is topic `esper.data.default`.

```
<ceppushsvc-configuration>
  ... more settings...
  <channels>
    <channel name="defaultChannel" subjectspace="defaultSpace">
      <qos>
        <batched-service interval-msec="1000" count="100"/>
      </qos>
      <jms>
        <destination topic="esper.data.default"/>
      </jms>
    </channel>
  </channels>
  ... more settings...
</ceppushsvc-configuration>
```

The next code snippet shows the configuration API to configure the same channel as above:

```
ConfigurationPushSvc config = new ConfigurationPushSvc();
Channel channel = new Channel();
channel.setName("defaultChannel");
channel.setSubjectSpace("defaultSpace");
BatchedService batched10Sec = new BatchedService();
batched10Sec.setIntervalMSec(1000);
batched10Sec.setCount(100);
channel.setService(batched10Sec);
channel.setJms(new ChannelJMSConfig("esper.data.default"));
config.addChannel(channel);
```

4.3. JMS Providers

This section describes how to configure Java Message Service (JMS) provider(s) (aka. brokers). You may configure multiple providers, however only a single JMS provider may be active for Push Services at any given time.

The below XML configuration configures a provider by name `defaultProvider` and configures the default settings for the default JMS provider.

```
<ceppushsvc-configuration>
  ... more settings...
  <providers>
    <provider name="defaultProvider">
      <context object-name="ConnectionFactory">
        <env name="java.naming.factory.initial"
           value="org.apache.activemq.jndi.ActiveMQInitialContextFactory"/>
        <env name="java.naming.provider.url"
           value="tcp://localhost:61616?wireFormat.maxInactivityDuration=0"/>
      </context>
    </provider>
  </providers>
  ... more settings...
</ceppushsvc-configuration>
```

The next code snippet shows the configuration API to configure the same provider as above:

```
ConfigurationPushSvc config = new ConfigurationPushSvc();
Provider provider = new Provider();
provider.setObjectName("ConnectionFactory");
Map<String, String> context = new HashMap<String, String>();
context.put(Context.PROVIDER_URL,
    "tcp://localhost:61616?wireFormat.maxInactivityDuration=0");
context.put(Context.INITIAL_CONTEXT_FACTORY,
    org.apache.activemq.jndi.ActiveMQInitialContextFactory.class.getName());
provider.setContext(context);
configJMS.getProviders().put("defaultProvider", provider);
```

4.4. Push Services Settings

4.4.1. Push Provider

The JMS provider name by default is `defaultProvider`. Push Services will attempt to connect to the provider configured by this name, unless you explicitly configure another provider name.

The following XML configures the default settings:

```
<ceppushsvc-configuration>
  ... more settings...
  <settings>
    <pushprovider provider-name="defaultProvider"/>
   </settings>
</ceppushsvc-configuration>
```

The next code snippet shows the configuration API to configure the same control bus settings:

```
ConfigurationPushSvc config = new ConfigurationPushSvc();
config.getSettings().setPushProvider(new PushProvider("defaultProvider"));
```

4.4.2. Session Management

The default session management is disabled. Session management should be enabled when running a distributed architecture: When CEP engine(s) and web application layer are separated between JVMs or hosts the session management checks that web application layer sessions are alive.

You may enable session management and perform session keep-alive checks every 10 seconds and expire sessions that have been inactive for 60 seconds. Unless you plan to override the defaults, you do not need to configure this (the settings are optional).

The following XML configures the default settings:

```
<ceppushsvc-configuration>
  ... more settings...
  <settings>
    <session-mgmt enabled="false" interval-msec="10000" expiry-msec="60000"/>
  </settings>
</ceppushsvc-configuration>
```

The next code snippet shows the configuration API to configure the same settings:

```
ConfigurationPushSvc config = new ConfigurationPushSvc();
config.getSettings().getSessionMgmt().setEnabled(false);
config.getSettings().getSessionMgmt().setIntervalMSec(10000);
config.getSettings().getSessionMgmt().setExpiryMSec(60000);
```

4.4.3. Work Queue

The number of threads in the work queue is configurable. By default, 1 thread is allocated. By setting the number of threads to zero all work items are performed by the originating thread.

The following XML configures the default settings:

```
<ceppushsvc-configuration>
  ... more settings...
  <settings>
    <work-queue num-threads="1">
  </settings>
</ceppushsvc-configuration>
```

The next code snippet shows the configuration API to configure the same settings:

```
ConfigurationPushSvc config = new ConfigurationPushSvc();
config.getSettings().getWorkQueue().setNumThreads(1);
```

Chapter 5. Examples

Enterprise Edition comes with a Java example for CEP push services. The example is completely standalone, thereby please shut down any existing server process to avoid overlapping ports.

The example can be found in the root folder of the installation under `examples/examples-ceppushsvc`.

The example consists of two parts:

1. A tiny server that starts Esper, an embedded JMS broker, Push Services as well as a HTTP/REST container.
2. A tiny client that uses an HTTP client to subscribe to a statement by invoking REST services, and that receives messages pushed over a JMS destination.

To run the examples, change directory to the `traffic/etc` folder and build first by invoking the `compile` script in the `etc` folder. The server startup script is `run_traffic_server` and the client startup script is `run_traffic_client`.