1. Performance Tuning ................................................................. 4
2. Stream Processor Documentation ........................................ 7
   2.1 Introducing Stream Processor ........................................... 8
       2.1.1 Overview ................................................................. 8
       2.1.2 Architecture ............................................................ 9
       2.1.3 About This Release .................................................. 11
   2.2 Quick Start Guide .......................................................... 11
   2.3 Installation Guide .......................................................... 19
       2.3.1 Installing the Product .............................................. 19
           2.3.1.1 Installation Prerequisites ................................. 19
           2.3.1.2 Installing on Linux ................................... 20
           2.3.1.3 Installing on Windows .................................. 21
           2.3.1.4 Installing as a Linux Service ...................... 24
           2.3.1.5 Installing as a Windows Service ............... 26
       2.3.2 Running the Product ............................................. 32
       2.3.3 Upgrading from a Previous Release ......................... 33
   2.4 Key Concepts ............................................................. 45
   2.5 Tutorials ........................................................................ 45
       2.5.1 Creating a Simple Siddhi Application .................. 46
       2.5.2 Consuming Events ..................................................... 51
       2.5.3 Pre-processing Streaming Data .............................. 61
       2.5.4 Integrating Datastores .......................................... 65
       2.5.5 Summarizing Stream Data - Short Term .................. 71
       2.5.6 Streaming Data Summarization (Incremental Aggregation) 78
       2.5.7 Analyzing KPIs ......................................................... 82
       2.5.8 Publishing Processed Events .................................. 86
       2.5.9 Correlating Simple Events .................................... 89
       2.5.10 Correlating Events for Complex Event Processing .... 92
       2.5.11 Analyzing Trends .................................................... 96
       2.5.12 Making Real-time Predictions ............................... 100
       2.5.13 Streaming Machine Learning ................................ 101
       2.5.14 Detecting Anomalies ............................................. 104
   2.6 User Guide .................................................................. 140
       2.6.1 Understanding the Development Environment .......... 141
           2.6.1.1 Stream Processor Studio Overview ................. 141
           2.6.1.2 Working with the Design View ...................... 147
           2.6.1.3 Creating a Siddhi Application ....................... 172
       2.6.1.4 Testing a Siddhi Application .............................. 182
          2.6.1.4.1 Simulating Events ..................................... 184
       2.6.1.5 Debugging a Siddhi Application ....................... 205
       2.6.1.6 Exporting a Siddhi File ..................................... 210
       2.6.2 Developing Streaming Applications ....................... 211
           2.6.2.1 Siddhi Application Overview .......................... 212
           2.6.2.2 Collecting Events .......................................... 213
           2.6.2.3 Processing Streaming Events ......................... 215
       2.6.2.4 Storage Integration .......................................... 221
          2.6.2.4.1 Defining Data Tables ................................. 221
          2.6.2.4.2 Managing Stored Data via Streams ............. 227
          2.6.2.4.3 Managing Stored Data via REST APIs ........... 231
          2.6.2.4.4 Accessing and Manipulating Data in Multiple Tables 236
       2.6.2.5 Complex Event Processing ................................ 237
       2.6.2.6 Machine Learning ............................................. 238
       2.6.2.7 Incremental Analysis ....................................... 238
       2.6.2.8 Publishing Events ............................................ 243
       2.6.2.9 Converting to a Distributed Streaming Application 244
       2.6.3 Deploying Streaming Applications ............................ 246
       2.6.4 Visualizing Data ..................................................... 247
           2.6.4.1 Designing Dashboards ..................................... 248
              2.6.4.1.1 Creating New Dashboards ...................... 248
              2.6.4.1.2 Editing Dashboards .............................. 256
              2.6.4.1.3 Deleting Dashboards ............................ 258
       2.6.4.2 Viewing Dashboards ............................................ 259
       2.6.4.3 Generating Reports .......................................... 262
       2.6.4.4 Generating Widgets ......................................... 266
       2.6.4.5 Creating Custom Widgets ................................. 277
          2.6.4.5.1 Consume user information within a widget .... 284
          2.6.4.5.2 Inter Widget Communication via the PubSub Model 284
          2.6.4.5.3 Resolving CSS style conflicts .................... 286
Performance Tuning

This section describes some recommended performance tuning configurations to optimize the performance of WSO2 Product SP. It assumes that you have set up WSO2 Product SP on a server running Unix/Linux, which is recommended for a production deployment.

- **OS-Level Settings**
- **JVM settings**
- **JDBC Pool Configuration**
- **SP-Level settings**
  - Receiving events
  - Publishing events

**Important**

- Performance tuning requires you to modify important system files, which affect all programs running on the server. We recommend you to familiarize yourself with these files using Unix/Linux documentation before editing them.
- The parameter values we discuss below are just examples. They might not be the optimal values for the specific hardware configurations in your environment. We recommend that you carry out load tests on your environment to tune the product accordingly.

### OS-Level Settings

1. To optimize network and OS performance, configure the following settings in `/etc/sysctl.conf` file of Linux. These settings specify a larger port range, a more effective TCP connection timeout value, and a number of other important parameters at the OS-level.

   ```
   net.ipv4.tcp_fin_timeout = 30
   fs.file-max = 2097152
   net.ipv4.tcp_tw_recycle = 1
   net.ipv4.tcp_tw_reuse = 1
   net.core.rmem_default = 524288
   net.core.wmem_default = 524288
   net.core.rmem_max = 67108864
   net.core.wmem_max = 67108864
   net.ipv4.tcp_rmem = 4096 87380 16777216
   net.ipv4.tcp_wmem = 4096 65536 16777216
   net.ipv4.ip_local_port_range = 1024 65535
   ```

   When we have the localhost port range configuration lower bound to 1024, there is a possibility that some processes may pick the ports which are already used by WSO2 servers. Therefore, it's good to increase the lower bound as sufficient for production, e.g., 10,000.

2. To alter the number of allowed open files for system users, configure the following settings in `/etc/security/limits.conf` file of Linux.

   ```
   * soft nofile 4096
   * hard nofile 65535
   ```

   Optimal values for these parameters depend on the environment.

3. To alter the maximum number of processes your user is allowed to run at a given time, configure the following settings in `/etc/security/limits.conf` file of Linux (be sure to include the leading `*` character). Each carbon server instance you run would require up to 1024 threads (with default thread pool configuration). Therefore, you need to increase the nproc value by 1024 per each carbon server (both hard and soft).

   ```
   * soft nproc 20000
   * hard nproc 20000
   ```
### JVM settings

When an XML element has a large number of sub-elements and the system tries to process all the sub-elements, the system can become unstable due to a memory overhead. This is a security risk.

To avoid this issue, you can define a maximum level of entity substitutions that the XML parser allows in the system. You do this using the `entityExpansionLimit` attribute that is in the `<SP_HOME>/bin/editor.bat` (for Windows) or the `<SP_HOME>/bin/editor.sh` (for Linux/Solaris). The default entity expansion limit is 64000.

```bash
-DentityExpansionLimit=100000
```

### JDBC Pool Configuration

Within the WSO2 platform, we use Tomcat JDBC pooling as the default pooling framework due to its production ready stability and high performance. The table below indicates some recommendations on how to configure the JDBC pool using the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated from the connection pool at the same time. The default value is 100.</td>
<td>This value should match the maximum number of requests that can be expected at a time in your production environment. This is to ensure that, whenever there is a sudden increase in the number of requests to the server, all of them can be connected successfully without causing any delays. Note that this value should not exceed the maximum number of requests allowed for your database.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether connection objects will be validated before they are borrowed from the pool. If the object validation fails, it will be dropped from the pool, and we will attempt to borrow another connection.</td>
<td>Setting this property to 'true' is recommended as it will avoid connection requests from failing. The <code>validationQuery</code> property should be used if <code>testOnBorrow</code> is set to true. To increase the efficiency of connection validation and to improve performance, <code>validationInterval</code> property should also be used.</td>
</tr>
<tr>
<td>validationInterval</td>
<td>To avoid excess validation, run validation at most at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again. The default value is 30000 (30 seconds).</td>
<td>This time out can be as high as the time it takes for your DBMS to declare a connection as stale. For example, MySQL will keep a connection open for as long as 8 hours, which requires the validation interval to be within that range. However, note that having a low value for validation interval will not incur a big performance penalty, specially when database requests have a high throughput. For example, a single extra validation query run every 30 seconds is usually negligible.</td>
</tr>
<tr>
<td>validationQuery</td>
<td>The SQL query used to validate connections from this pool before returning them to the caller. If specified, this query does not have to return any data, it just can't throw an SQLException. The default value is null. Example values are <code>SELECT 1(mysql), select 1 from dual(oracle), SELECT 1(MS Sql Server)</code>.</td>
<td>Specify an SQL query, which will validate the availability of a connection in the pool. This query is necessary when <code>testOnBorrow</code> property is true.</td>
</tr>
</tbody>
</table>

When it comes to web applications, users are free to experiment and package their own pooling framework such BoneCP.

### SP-Level settings

Performance tuning can be tried out in the following areas at the SP level. The performance is considered in terms of throughput per second (TPS) and latency.

### Receiving events

The following parameters which affect the performance relating to the databridge communication are configured in the `<SP_HOME>/conf/edit or/deployment.yaml` under data-bridge-config property. These configurations are common for both thrift and binary protocols.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default Value</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>workerThreads</td>
<td>The number of threads reserved to handle the load of events received.</td>
<td>10</td>
<td>This value should be increased if you want to increase the throughput by receiving a higher number of events at a given time. The number of available CPU cores should be considered when specifying this value. If the value specified exceeds the number of CPU cores, higher latency would occur as a result of context switching taking place more often.</td>
</tr>
<tr>
<td>maxEventBufferCapacity</td>
<td>The maximum size allowed for the event receiving buffer in mega bytes. The event receiving buffer temporarily stores the events received before they are forwarded to an event stream.</td>
<td>10</td>
<td>This value should be increased when there is an increase in the receiving throughput. When increasing the value heap memory size also needs to be increased accordingly.</td>
</tr>
<tr>
<td>eventBufferSize</td>
<td>The number of messages that is allowed in the receiving queue at a given time.</td>
<td>2000</td>
<td>This value should be increased when there is an increase in the receiving throughput.</td>
</tr>
<tr>
<td>clientTimeoutMin</td>
<td>Session timeout value in minutes.</td>
<td>30</td>
<td>Cache that contains all the agent sessions are expired after this value is reached. This value should be increased when there is an...</td>
</tr>
</tbody>
</table>

**Publishing events**

The following parameters which affect the performance relating to the Data Agents - to publish events through databridge are configured in the `<S P_HOME>/conf/editor/deployment.yaml` under `data.agent.config` property. These configurations are common for both thrift and binary protocols.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default Value</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>queueSize</td>
<td>The size of the queue event disruptor which handles events before they are published to an application/data store.</td>
<td>32768</td>
<td>The value specified should always be the result of an exponent with 2 as the base. (e.g., 32768 is 2^15). A higher value should be specified when a higher throughput needs to be handled. However, the increase in the load handled at a given time can reduce the speed at which the events are processed. Therefore, a lower value should be specified if you want to reduce the latency.</td>
</tr>
<tr>
<td>batchSize</td>
<td>The maximum number of events in a batch sent to the queue event disruptor at a given time.</td>
<td>200</td>
<td>This value should be assigned proportionally to the throughput of events handled. Greater the batch size, higher will be the number of events sent to the queue event disruptor at a given time.</td>
</tr>
<tr>
<td>corePoolSize</td>
<td>The number of threads that will be reserved to handle events at the time you start the CEP server. This value will increase as throughput of events handled increases, but it will not exceed the value specified for the <code>MaxPoolSize</code> parameter.</td>
<td>1</td>
<td>The number of available CPU cores should be taken into account when specifying this value. Increasing the core pool size may improve the throughput, but latency will also be increased due to context switching.</td>
</tr>
<tr>
<td>maxPoolSize</td>
<td>The maximum number of threads that should be reserved at any given time to handle events.</td>
<td>1</td>
<td>The number of available CPU cores should be taken into account when specifying this value. Increasing the maximum core pool size may improve the throughput since more threads can be spawned to handle an increased number of events. However, latency will also increase since a higher number of threads would cause context switching to take place more frequently.</td>
</tr>
</tbody>
</table>

For better throughput you can configure the parameters as follows.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>queueSize</td>
<td>32768</td>
</tr>
<tr>
<td>batchSize</td>
<td>200</td>
</tr>
<tr>
<td>corePoolSize</td>
<td>1</td>
</tr>
<tr>
<td>socketTimeoutMS</td>
<td>30000</td>
</tr>
<tr>
<td>maxPoolSize</td>
<td>1</td>
</tr>
<tr>
<td>keepAliveTimeInPool</td>
<td>20</td>
</tr>
<tr>
<td>reconnectionInterval</td>
<td>30</td>
</tr>
<tr>
<td>maxTransportPoolSize</td>
<td>250</td>
</tr>
<tr>
<td>maxIdleConnections</td>
<td>250</td>
</tr>
<tr>
<td>evictionTimePeriod</td>
<td>5500</td>
</tr>
<tr>
<td>minIdleTimeInPool</td>
<td>5000</td>
</tr>
<tr>
<td>secureMaxTransportPoolSize</td>
<td>250</td>
</tr>
<tr>
<td>secureMaxIdleConnections</td>
<td>250</td>
</tr>
<tr>
<td>secureEvictionTimePeriod</td>
<td>5500</td>
</tr>
<tr>
<td>secureMinIdleTimeInPool</td>
<td>5000</td>
</tr>
</tbody>
</table>

For reduced latency, you can configure the parameters as follows.

```xml
<QueueSize>256</QueueSize>
<BatchSize>200</BatchSize>
<CorePoolSize>1</CorePoolSize>
<MaxPoolSize>1</MaxPoolSize>
```

**Stream Processor Documentation**

WSO2 Stream Processor is a Streaming SQL based, high performant, lightweight, open source stream processing platform, facilitating the creation of real-time, intelligent, actionable business insights, and data products for digital businesses. It allows you to collect events, analyze them in real-time, identify patterns, map their impacts, and react within milliseconds.
Get started with WSO2 SP

If you are new to using WSO2 Stream Processor, follow the steps given below to get started:

- Get familiar with WSO2 SP
  Understand the basics of the SP and its architecture.

- Quick Start Guide
  Download, install and run the SP in just 10 minutes.

- Try out the Tutorials
  Learn SP functions step by step.

For additional learning resources such as webinars and white papers, go to Library - Analytics. This is a great place for you to expand your knowledge on WSO2 SP.

Deep dive into WSO2 SP

To know more about WSO2 SP use the descriptions below to find the section you need, and then browse the topics in the left navigation panel. You can also use the Search box on the left to find a term or phrase in this documentation, or use the box in the top right-hand corner to search in all WSO2 product documentation.

Introducing Stream Processor

WSO2 Stream Processor (WSO2 SP) is packed with features that will enable any enterprise to build streaming analytics capabilities and derive meaningful insights out of the organization’s data. It is powered by Siddhi, a leading Open Source stream processing project. With WSO2 SP’s streaming SQL capabilities and with its inbuilt editor having event simulation and debugging support it can help you to create real-time applications much faster than before. Its high performance and low footprint also lead to more agile deployment: it is the only competing product that can handle 100K events per second in a high-availability deployment with just two commodity servers.

The topics in this section introduce WSO2 Stream Processor, including the business cases it solves, its features, and architecture.

- Overview
- Architecture
- About This Release

Overview

The availability of business insights and information is a significant factor for competitive advantage in modern businesses. This allows business owners to make decisions in real-time. Real-time stream processing and streaming analytics is the technology that makes this a reality.

The first generation Stream Processors pose several challenges.
WSO2 Stream Processor (SP) is a 100% open source streaming analytics and stream processing solution that allows you to build and deploy applications that collect, analyze, and present data in real-time.

<table>
<thead>
<tr>
<th>Collect</th>
<th>Analyse</th>
<th>Act</th>
</tr>
</thead>
</table>
| - Collects events from multiple event sources using various data formats.  
- Preprocesses by deploying at the edge. | - Processes stream of events in real-time using Streaming SQL queries.  
- Summarizes and correlates events in memory and by integrating with data stores. | - Notifies interesting event occurrences via alerts and service calls.  
- Visualizes the summarizations via dashboard. |

It has the following capabilities:

- **Process millions of events per second in real-time**  
  WSO2 SP is the sole analytics product in the market that facilitates high-performance analytics with only 2 nodes (minimum HA). It can process approximately 100,000 events per second with the ability to scale beyond with Apache Kafka.

- **Be updated at all times with incremental analytics**  
  WSO2 SP replaces periodic batch operations with out-of-the-box long-running incremental processing to achieve updated analysis at each arrival of data.

- **Adapt to the market faster with shortened development time**  
  Provides analysis in a specialized, easy-to-use, Siddhi Streaming SQL language using the state-of-the-art IDE, providing agile development experience with smart editing, simulation and debugging capabilities.

- **Investigate the past, predict the future**  
  Gain insights using past performances, build pre-trained and online machine learning models, and perform real-time predictions to drive business planning.

- **Enable insights into all your systems**  
  Work out-of-the-box with popular data formats, transport protocols and connect to over 100 legacy and cloud services via connectors and agents.

- **Enable managers to manage their business rules and visualize output**  
  Empower business users to create and dynamically deploy business rules through easy to use graphical UI, and let them make better decisions utilizing real-time dashboards.

- **Build smarter devices with edge analytics**  
  Make devices smarter by deploying WSO2 Siddhi (<2MB) for localized data analytics, and let centralized deployments such as IoT analytics handle a massive amount of data by filtering and summarising at the edge.

- **Build an event-driven architecture using streaming data integration**  
  Build information-rich streams by connecting to diverse data streams, letting organizations to get a better overall understanding of their data in real-time, and to build control flows.

**Architecture**

Stream processing and analytics refer to collecting, analyzing and acting on events generated during business activities. This definition is paramount when designing a solution to address a real-time streaming use case. Collecting refers to the collection of data from various data sources. Analysis refers to the manipulation of data to identify interesting patterns and to extract information. Acting refers to notifying the results to other systems and personals and representing the analyzed data visually. Streaming data that needs to be processed or analyzed, sequentially passes through these sections.

The WSO2 SP architecture reflects this natural flow in its design as illustrated below.

WSO2 SP contains Siddhi as its core to collect, analyze and act on the incoming events. The following are the major components of SP.

- Siddhi
Stream Processor Studio/Editor
Portal
Business Rules
Status Dashboard
Worker
Job Manager

Siddhi

Siddhi is the major component of SP which has the capability of running the stream processing and complex event processing logic. Stream processing logic can be scripted using a Streaming SQL language as Siddhi Application and deployed into stream processor for processing. It handles collection, analysis and performs actions based on the events which it receives and Siddhi Apps deployed.

Siddhi contains the following core elements:

<table>
<thead>
<tr>
<th>Siddhi functions</th>
<th>Siddhi core element and description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Collect</strong></td>
<td><strong>Source</strong>: Sources receive events via multiple transport protocols such as HTTP, TCP, Kafka, JMS, etc., and in different data formats such as XML, JSON, Text, Binary, etc. The events received via sources are mapped into streams for processing.</td>
</tr>
</tbody>
</table>
| **Analyze**      | **Stream**: Streams represent a continuous stream of events which adheres to a defined schema.  
**Table**: Tables represent a static set of events which adheres to a defined schema. Table can be manipulated with "insert", "update", "delete", and "update or insert" operations. Events in tables can be retrieved by joining them with streams or using REST API.  
**Window**: Windows represent a set of events which adheres to a defined schema which gets emitted based on the given window condition. Events in windows can be retrieved by joining them with streams or using REST API.  
**Aggregation**: Aggregations consume events from a stream and perform predefined aggregations. The results of the aggregations can be retrieved by joining them with streams or using REST API.  
**Query**: Queries help you process events. Queries process streams, tables, and windows and produce new streams, or update tables or windows. A query can contain filters, windows, aggregations, joins, patterns and/or sequence operations.  
**Store**: Stores are mapped to tables and it allows you to store events in various databases and systems such as RDBMS, Apache Cassandra, MongoDB, Apache Solr, Apache HBase, Hazelcast and many more.  
**Trigger**: Triggers produce periodic events to a stream to achieve periodic execution of query logic. |
| **Act**          | **Sink**: Sinks publish events arriving at streams via multiple transport protocols such as HTTP, Email, TCP, Kafka, JMS, etc., by mapping the events to different data formats such as XML, JSON, Text, Binary, etc. |

For more information about Siddhi, see the Siddhi Query Guide.

Stream Processor Studio/Editor

The Stream Processor Studio provides an environment for developers to build Siddhi applications with the support of syntax highlighting, auto-completion, and with integrated documentation support. It also allows them to test the application using simulations and debug the application to verify the processing logic.

For more information, see Understanding the Development Environment.

Portal

This is used for data visualization in WSO2 SP. The data from real-time streams and stored tables can be visualized via the portal. The portal can contain several dashboards and widgets that can be generated and customized by users based on their requirements.

For more information, see Visualizing Data.

Business Rules

Business rules provide a mechanism for business users to manage the rules themselves. Here business users can create/edit/delete simple filters using a form-based interface and or predefined parameterized rules created by the developers.

For more information, see Working with Business Rules.
Status Dashboard

This lets you monitor the system in operation by getting to fine-grain details about its throughput, latency, and how much load it is handling to better understand and manage the environment.

For more information, see Monitoring Stream Processor.

Worker

The worker provides a lightweight stream processing server that lets you deploy and run Siddhi applications in production.

Job Manager

This is used only on fully distributed deployments, to automatically deploy and manage Siddhi applications on multiple Stream Processor worker nodes.

About This Release

WSO2 SP version 4.3.0 is the successor of version 4.2.0. It contains the following new features and enhancements.

Features

Some of the new prominent capabilities of WSO2 SP are as follows:

- A change in the minimum HA deployment functionality where event synchronization in a Minimum HA setup takes place only when a failover takes place. For more information, see Minimum High Availability Deployment.
- The introduction of an Open Tracing Client. For more information, see Distributed Message Tracer - Using the WSO2 SP Message Trace Client.
- The possibility to download visualizations of data displayed in the Dashboard Portal as PDF documents. For more information, see Generating Reports.
- The introduction of automatic data purging for incremental aggregations. For more information, see Incremental Analysis.
- The introduction of JSON support for Siddhi.
- The possibility to join multiple RDBMS tables. For more information, see Accessing and Manipulating Data in Multiple Tables.
- The possibility to send and receive events via the IBM MQ transport.
- The possibility to send attachments when publishing processed data as emails.
- The possibility to import and export dashboards. For more information, see Importing and Exporting Dashboards.

Fixed Issues

For a list of fixed issues in this release, see WSO2 SP 4.3.0 - Fixed Issues.

Known issues

For a list of known issues in this release, see WSO2 SP 4.3.0 - Known Issues.

Quick Start Guide

WSO2 Stream Processor (SP) is a lightweight and lean, streaming SQL based stream processing platform that allows you to collect events, analyze them in real-time, identify patterns, map their impacts, and communicate the results within milliseconds. It is powered by Siddhi to be extremely high performing.

First, let's understand the following concepts that are used in this guide:

Stream Processing and Complex Event Processing Overview

Let's understand what an event is through an example. If we consider the transactions carried out via an ATM as a data stream, one withdrawal from it can be considered an event. This event contains data about the amount, time, account number etc. Many such transactions form a stream.
Stream processing engines allow you to create a processing graph and inject events into it. Each operator processes and sends events to the next processor.

A complex event is an event that summarizes, represents, or denotes a set of other events. Complex Event Processing is a subset of Stream Processing which involves analyzing multiple streams of events in real-time, recognizing particular sequences or patterns across streams and inferring a business significant event from correlated events.

The stream processing capabilities of WSO2 SP allow you to capture high volume data flows and process them in real-time, and present results in a streaming manner while its complex event processing capabilities detect patterns and trends for decision making via Patterns and Sequences supported for Siddhi.

**Siddhi overview**

WSO2 SP uses the Siddhi query language to write the processing logic for its Siddhi applications. Siddhi can:

- Accept event inputs from many different types of sources
- Process them to generate insights
- Publish them to many types of sinks.

To use Siddhi, you need to write the processing logic as a Siddhi Application in the Siddhi Streaming SQL language. After writing and starting a Siddhi application, it:
Takes data one-by-one as events
Processes the data in each event
Generates new high level events based on the processing done so far
Sends newly generated events as the output to streams.

Before you begin:
1. Install Oracle Java SE Development Kit (JDK) version 1.8.
2. Set the JAVA_HOME environment variable.
3. Download the latest WSO2 Stream Processor.
4. Extract the downloaded zip and navigate to the `<SP_HOME>/bin` directory (`<SP_HOME>` is the extracted directory).
5. Issue one of the following commands to start the WSO2 Stream Processor Studio.
   - For Windows: `editor.bat`
   - For Linux: `./editor.sh`

Once WSO2 SP server is successfully started, a log similar to the following is printed in the CLI.

Let's get started! You can write a simple Siddhi application to calculate the total weight with each cargo box loaded to the ship by following the steps below.

- **Step 1:** Create a Siddhi application
- **Step 2:** Simulate events
- **Step 3:** Edit Siddhi application to perform temporal processing
- **Step 4:** Simulate events for the edited Siddhi application
- What's next?

**Scenario**
In this scenario, you are creating an application for Shipping Wave, a fictitious large scale shipping company. Smith, the cargo manager needs to keep track of the total weight of cargo loaded to a ship at any given time. Measuring the weight of a cargo box when it is loaded to the ship is considered an event.

Let's get started! You can write a simple Siddhi application to calculate the total weight with each cargo box loaded to the ship by following the steps below.

- **Step 1:** Create a Siddhi application
- **Step 2:** Simulate events
- **Step 3:** Edit Siddhi application to perform temporal processing
- **Step 4:** Simulate events for the edited Siddhi application
- What's next?

**Step 1: Create a Siddhi application**

Smith needs to calculate the total weight of the cargo loaded into a ship with every cargo box added. In order to generate this output for him, let's create a Siddhi application as follows:

1. Access the Stream Processor Studio via the `http://<HOST_NAME>:<EDITOR_PORT>/editor` URL.

   The default URL is `http://localhost:9390/editor`

The Stream Processor Studio opens as shown below.
2. Enter a name for your Siddhi application. In this scenario, let's name the application CargoWeightApp as shown below.

```
@App:name("CargoWeightApp")
```

3. Defining the input stream. The stream needs to have a name and a schema defining the data that each incoming event should contain. The event data attributes are expressed as name and type pairs. In this example:
   - The name of the input stream: `CargoStream`
   - A name to refer to the data in each event: `weight`
   - Type of the data received as weight: `int`

   ```
   define stream CargoStream (weight int);
   ```

4. Define an output stream. This has the same info as the previous definition with an additional `totalWeight` attribute that contains the total weight calculated so far. Here, we need to add a `sink` configuration to log the events from the `OutputStream` so that we can observe the output values.

   ```
   @sink(type='log', prefix='LOGGER')
   define stream OutputStream(weight int, totalWeight long);
   ```

5. Enter a Siddhi query that defines the following.
   - A name for the query (i.e., `cargoWeightQuery`)
   - The input stream from which the events to be processed are taken (i.e., `CargoStream`)
   - The data that needs to be sent to the output stream (i.e., `weight` and `totalWeight`)
   - How the output needs to be calculated (i.e., by calculating the sum of the weight of all the events)
   - The stream to which the output needs to be sent (i.e., `OutputStream`)

   This query is as follows:
5. The completed Siddhi file is as follows:

```siddhi
@info(name='CargoWeightQuery')
from CargoStream
select weight, sum(weight) as totalWeight
insert into OutputStream;
```

Step 2: Simulate events

The Stream Processor Studio has in-built support to simulate events. To test whether the CargoWeightApp you created works as expected, let's simulate some events by following the steps given below.

1. To start the CargoWeight Siddhi application, click the play button.

   If the application is successfully started, the following is logged in the Stream Processor Studio console.

   ```none
   CargoWeight.siddhi - Started Successfully!
   ```

2. Click the following icon in the Stream Processor Studio to open the event simulation panel.

3. In the Single Simulation tab of the panel, select values as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Siddhi App Name</td>
<td>CargoWeight</td>
</tr>
<tr>
<td>Stream Name</td>
<td>CargoStream</td>
</tr>
</tbody>
</table>

6. To save the Siddhi file, click File => Save. This opens the Save to Workspace dialog box. Click Save to save this file in the `<SP_HOME>/wso2/editor/deployment/workspace` directory (which is the default location where Siddhi applications are saved).
As a result, the Weight attribute of the CargoStream stream is displayed as follows:

```
<table>
<thead>
<tr>
<th>Event No</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000</td>
</tr>
<tr>
<td>2</td>
<td>1500</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
</tr>
<tr>
<td>4</td>
<td>3000</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
</tr>
</tbody>
</table>
```

5. Send five more events with the following values.

The events are logged as follows.
Step 3: Edit Siddhi application to perform temporal processing

This section demonstrates how to carry out temporal window processing with Siddhi.

In the previous scenario, you carried out processing by having only the running sum value in-memory. No events were stored during this process.

Window processing is a method that allows us to store some events in-memory for a given period or a specific number of events so that we can perform operations such as calculating the average, maximum, etc values within them.

Let's consider that Smith, the cargo manager has an additional requirement to calculate the average for the last three cargo boxes loaded each time a new cargo box is loaded in order to balance the weight across the ship. Here, we are considering a window that consists of three events as shown in the image below.

To achieve this, edit the Siddhi application by following the steps below:

1. Add a new attribute named `averageWeight` to the definition of the `OutputStream` stream so that each output event presents the average weight in addition to the weight of the new box loaded and the total weight.

   ```siddhi
   define stream OutputStream(weight int, totalWeight long, averageWeight double);
   ```

2. To specify how to calculate the average weight, apply the `avg` Siddhi function to the `weight` attribute in the `select` statement as shown below. This indicates that the average is calculated for the `weight` attribute of incoming events.

   ```siddhi
   select weight, sum(weight) as totalWeight, avg(weight) as averageWeight
   ```

3. To specify that the calculations performed by this query with each event must be applied only to the last three events received, apply a `length` window to the input stream as shown below.

   ```siddhi
   from CargoStream#window.length(3)
   ```

This window applies to all the calculations performed for the events taken from the `CargoStream` stream. Therefore, adding this window also results in the total weight being calculated based on the last three events.
The completed query is as follows.

```sql
@info(name='CargoWeightQuery')
from CargoStream#window.length(3)
select weight, sum(weight) as totalWeight, avg(weight) as averageWeight
insert into OutputStream;
```

The complete CargoWeight Siddhi application is as follows.

```java
@App:name("CargoWeight")

define stream CargoStream (weight int);

@sink(type='log', prefix='LOGGER')
define stream OutputStream(weight int, totalWeight long, averageWeight double);

@info(name='CargoWeightQuery')
from CargoStream#window.length(3)
select weight, sum(weight) as totalWeight, avg(weight) as averageWeight
insert into OutputStream;
```

Step 4: Simulate events for the edited Siddhi application

In this step, let's start the edited Siddhi application and simulate the same six events that you simulated in Step 8.

<table>
<thead>
<tr>
<th>Event No</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>1500</td>
</tr>
<tr>
<td>4</td>
<td>2000</td>
</tr>
<tr>
<td>5</td>
<td>3000</td>
</tr>
<tr>
<td>6</td>
<td>1000</td>
</tr>
</tbody>
</table>

The output generated is logged as shown below:

```
[2017-12-19 11:09:05.679] INFO {org.wso2.siddhi.core.stream.output.sink.LogSink} - LOGGER : Event{timestamp=1513661944668, data=[1000, 1000, 1000.0], isExpired=false}
[2017-12-19 11:09:05.857] INFO {org.wso2.siddhi.core.stream.output.sink.LogSink} - LOGGER : Event{timestamp=1513661948855, data=[2000, 3000, 1500.0], isExpired=false}
[2017-12-19 11:09:14.400] INFO {org.wso2.siddhi.core.stream.output.sink.LogSink} - LOGGER : Event{timestamp=1513661954400, data=[1500, 4500, 1500.0], isExpired=false}
[2017-12-19 11:09:27.538] INFO {org.wso2.siddhi.core.stream.output.sink.LogSink} - LOGGER : Event{timestamp=1513661971078, data=[3000, 6500, 2166.6666666666667], isExpired=false}
[2017-12-19 11:09:34.373] INFO {org.wso2.siddhi.core.stream.output.sink.LogSink} - LOGGER : Event{timestamp=1513661974373, data=[1000, 6000, 2000.0], isExpired=false}
```
What's next?

Learn more about Siddhi and how it can be helpful in your user scenarios: Tutorials
Learn how to develop, deploy and monitor Siddhi applications: User Guide

Installation Guide

This chapter contains the following information.
- Installing the Product
- Running the Product
- Upgrading from a Previous Release

Installing the Product

Installing a WSO2 product is very fast and easy. Before you begin, be sure you have met the installation prerequisites, and then follow the installation instructions for your platform.
- Installation Prerequisites
- Installing on Linux
- Installing on Windows
- Installing as a Linux Service
- Installing as a Windows Service

Installation Prerequisites

Prior to installing any WSO2 SP, it is necessary to have the appropriate prerequisite software installed on your system. Verify that the computer has the supported operating system and development platforms before starting the installation.

System requirements

<table>
<thead>
<tr>
<th>CPU</th>
<th>A minimum of 2 CPUs is recommended.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>~ 4 GB minimum is recommended.</td>
</tr>
<tr>
<td>Disk</td>
<td>~ 1 GB minimum (excluding space allocated for log files and databases.)</td>
</tr>
</tbody>
</table>

Environment compatibility

Operating Systems / Databases
- All WSO2 Carbon-based products are Java applications that can be run on any platform that is Oracle JDK 1.8 compliant.
- All WSO2 products are generally compatible with most common DBMSs. For more information, see Working with Databases.
- It is not recommended to use Apache DS in a production environment due to issues with scalability.
- For environments that WSO2 products are tested with, see Compatibility of WSO2 Products.
- If you have difficulty in setting up any WSO2 product in a specific platform or database, please contact us.

Required applications

The following applications are required for running the Data Analytics Server and its samples, or for building from the source code. Mandatory installs are marked with *.

<table>
<thead>
<tr>
<th>Application</th>
<th>Purpose</th>
<th>Version</th>
<th>Download Links</th>
</tr>
</thead>
</table>
### Oracle Java SE Development Kit (JDK)*
- To launch the product as each product is a Java application.
- To build the product from the source distribution (both JDK and Apache Maven are required).
- To run Apache Ant.

**Note**
To launch WSO2 SP, you need to have Oracle JDK 1.8.*. You cannot launch WSO2 SP with Oracle JDK 1.7.* or lower.

1.8.*
http://java.sun.com/javase/downloads/index.jsp

**Do not use JDK 1.8.0_151 due to a known issue. This issue is fixed in JDK 1.8.0_162-ea. However, use JDK 1.8.0_144 until the latter mentioned version is released.**

#### We do not recommend OpenJDK as we do not support it or test our products with it.

<table>
<thead>
<tr>
<th>JDBC-compliant Connector for Java</th>
<th>1.7.0 or later</th>
<th><a href="http://dev.mysql.com/downloads/connector/">http://dev.mysql.com/downloads/connector/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Ant</td>
<td>1.7.0 or later</td>
<td><a href="http://ant.apache.org">http://ant.apache.org</a></td>
</tr>
<tr>
<td>Git</td>
<td>1.9.0 or later</td>
<td><a href="http://git-scm.com/downloads/">http://git-scm.com/downloads/</a></td>
</tr>
<tr>
<td>Apache Maven</td>
<td>3.0.*</td>
<td><a href="http://maven.apache.org">http://maven.apache.org</a></td>
</tr>
</tbody>
</table>

You are now ready to install. Click one of the following links for instructions:
- Installing on Linux
- Installing on Windows

### Installing on Linux

Follow the instructions below to install WSO2 SP on Linux.

**Before you begin:**
- See the known incompatibilities section to find out if this version of the product has issues running on your OS due to the JDK version.
- See the compatibility matrix to find out if this version of the product is fully tested on your operating system.

### Installing the required applications

1. Establish an SSH connection to the Linux machine or log in on the text Linux console.
2. Be sure your system meets the Installation Prerequisites. Java Development Kit (JDK) is essential to run the product.

### Installing the SP

1. Download the latest version of the SP.
2. Extract the archive file to a dedicated directory for the SP, which will hereafter be referred to as `<SP_HOME>`.

### Setting up JAVA_HOME

You must set your JAVA_HOME environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer.

Environment variables are global system variables accessible by all the processes running under the operating system.
1. In your home directory, open the BASHRC file in your favorite Linux text editor, such as vi, emacs, pico, or mcedit.
2. Assuming you have JDK 1.6.0_25 in your system, add the following two lines at the bottom of the file, replacing `/usr/java/jdk1.6.0_25` with the actual directory where the JDK is installed.

   ```bash
   export JAVA_HOME=/usr/java/jdk1.6.0_25
   export PATH=${JAVA_HOME}/bin:${PATH}
   ```

   The file should now look like this:

   ```bash
   # .bashrc
   # Source global definitions
   if [ -f /etc/bashrc ]; then
     . /etc/bashrc
   fi

   # User specific aliases and functions
   export JAVA_HOME=/usr/java/jdk1.6.0_25
   export PATH=${JAVA_HOME}/bin:${PATH}
   export M2_HOME=/opt/apache-maven-3.8.3
   export PATH=${M2_HOME}/bin:${PATH}
   ```

3. Save the file.

   If you do not know how to work with text editors in a Linux SSH session, run the following command:

   ```bash
   cat >> .bashrc
   Paste the string from the clipboard and press "Ctrl+D."
   ```

4. To verify that the `JAVA_HOME` variable is set correctly, execute the following command:

   ```bash
   echo $JAVA_HOME
   ```

   ```bash
   [suncom@ws02 ~]$ echo $JAVA_HOME
   /usr/java/jdk1.6.0_25
   [suncom@ws02 ~]$```

5. The system returns the JDK installation path.

### Setting system properties

If you need to set additional system properties when the server starts, you can take the following approaches:

- **Set the properties from a script**: Setting your system properties in the startup script is ideal, because it ensures that you set the properties every time you start the server. To avoid having to modify the script each time you upgrade, the best approach is to create your own startup script that wraps the WSO2 startup script and adds the properties you want to set, rather than editing the WSO2 startup script directly.

- **Set the properties from an external registry**: If you want to access properties from an external registry, you could create Java code that reads the properties at runtime from that registry. Be sure to store sensitive data such as username and password to connect to the registry in a properties file instead of in the Java code and secure the properties file with the secure vault.

When using SUSE Linux, it ignores `/etc/resolv.conf` and only looks at the `/etc/hosts` file. This means that the server will throw an exception on startup if you have not specified anything besides localhost. To avoid this error, add the following line above `127.0.0.1 localhost in the `/etc/hosts` file:

```bash
<ip_address> <machine_name> localhost
```
Installing the required applications

Make sure your system meets the Installation Prerequisites.

Java Development Kit (JDK) is essential to run the product.

Installing the SP

1. Download the latest version of the SP.
2. Extract the archive file to a dedicated directory for the SP, which will hereafter be referred to as `<SP_HOME>`.

Setting up JAVA_HOME

You must set your JAVA_HOME environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer. Typically, the JDK is installed in a directory under `C:\Program Files\Java`, such as `C:\Program Files\Java\jdk1.7.0_45`. If you have multiple versions installed, choose the latest one, which you can find by sorting by date.

Environment variables are global system variables accessible by all the processes running under the operating system. You can define an environment variable as a system variable, which applies to all users, or as a user variable, which applies only to the user who is currently logged in.

You set up JAVA_HOME using the System Properties, as described below. Alternatively, if you just want to set JAVA_HOME temporarily for the current command prompt window, set it at the command prompt.

Setting up JAVA_HOME using the system properties

1. Right-click the My Computer icon on the desktop and choose Properties.

2. In the System Properties window, click the Advanced tab, and then click the Environment Variables button.
3. Click the New button under **System variables** (for all users) or under **User variables** (just for the user who is currently logged in).

4. Enter the following information:
   - In the **Variable name** field, enter: `JAVA_HOME`
   - In the **Variable value** field, enter the installation path of the Java Development Kit, such as: `c:\Program Files\Java jdk1.7.0_45`

   The JAVA_HOME variable is now set and will apply to any subsequent command prompt windows you open. If you have existing command prompt windows running, you must close and reopen them for the JAVA_HOME variable to take effect, or manually set the JAVA_HOME variable in those command prompt windows as described in the next section. To verify that the JAVA_HOME variable is set correctly, open a command window (from the **Start** menu, click **Run**, and then type **CMD** and click **Enter**) and execute the following command:

   set JAVA_HOME

   The system returns the JDK installation path. You are now ready to **run the product**.

**Setting JAVA_HOME temporarily using the Windows command prompt (CMD)**

You can temporarily set the JAVA_HOME environment variable within a Windows command prompt window (CMD). This is useful when you have an existing command prompt window running and you do not want to restart it.

1. In the command prompt window, enter the following command where `<JDK_INSTALLATION_PATH>` is the JDK installation directory and
press Enter.

    set JAVA_HOME=<JDK_INSTALLATION_PATH>

For example:

    set JAVA_HOME=c:\Program Files\java\jdk1.7.0_45

The JAVA_HOME variable is now set for the current CMD session only.

2. To verify that the JAVA_HOME variable is set correctly, execute the following command:

    set JAVA_HOME

The system returns the JDK installation path.

Setting system properties

If you need to set additional system properties when the server starts, you can take the following approaches:

- **Set the properties from a script**: Setting your system properties in the startup script is ideal, because it ensures that you set the properties every time you start the server. To avoid having to modify the script each time you upgrade, the best approach is to create your own startup script that wraps the WSO2 startup script and adds the properties you want to set, rather than editing the WSO2 startup script directly.

- **Set the properties from an external registry**: If you want to access properties from an external registry, you could create Java code that reads the properties at runtime from that registry. Be sure to store sensitive data such as username and password to connect to the registry in a properties file instead of in the Java code and secure the properties file with the secure vault.

Installing the Bash Command Line features

When you install the Windows sub system for Linux(beta) feature in Windows 10, it results in a partial installation of bash Shell Command Line. When all the bash features are not available, you may be unable to run WSO2 SP on Windows. To complete the installation, you can issue the bash command from the Command Prompt. For detailed instructions to do this, see Windows 10 Installation Guide.

You are now ready to run the product.

Installing as a Linux Service

WSO2 Carbon and any Carbon-based product can be run as a Linux service as described in the following sections:

- Prerequisites
- Setting up CARBON_HOME
- Running the product as a Linux service

Before you begin:

- See the known incompatibilities section to find out if this version of the product has issues running on your OS due to the JDK version.
- See the compatibility matrix to find out if this version of the product is fully tested on your operating system.

Prerequisites

Install JDK and set up the JAVA_HOME environment variable. For more information, see Installation Prerequisites.

Setting up CARBON_HOME

Extract the WSO2 product that you want to run as a Linux service and set the environment variable CARBON_HOME to the extracted product directory location.

Running the product as a Linux service

1. To run the product as a service, create a startup script and add it to the boot sequence. The basic structure of the startup script has three parts (i.e., start, stop and restart) as follows:
#!/bin/bash

case "$1 in
  start)
    echo "Starting Service"
    ;;
  stop)
    echo "Stopping Service"
    ;;
  restart)
    echo "Restarting Service"
    ;;
  *)
    echo "$0 {start|stop|restart}"
    exit 1
esac

For example, given below is a startup script written for WSO2 Application Server 5.2.0:

```bash
#!/bin/sh
export JAVA_HOME="/usr/lib/jvm/jdk1.7.0_07"

startcmd='/opt/WSO2/wso2as-5.2.0/bin/wso2server.sh start > /dev/null &'
restartcmd='/opt/WSO2/wso2as-5.2.0/bin/wso2server.sh restart > /dev/null &'
stopcmd='/opt/WSO2/wso2as-5.2.0/bin/wso2server.sh stop > /dev/null &'

case "$1" in
  start)
    echo "Starting WSO2 Application Server ..."
    su -c "${startcmd}" user1
    ;;
  restart)
    echo "Re-starting WSO2 Application Server ..."
    su -c "${restartcmd}" user1
    ;;
  stop)
    echo "Stopping WSO2 Application Server ..."
    su -c "${stopcmd}" user1
    ;;
  *)
    echo "Usage: $0 {start|stop|restart}"
    exit 1
esac
```

In the above script, the server is started as a user by the name user1 rather than the root user. For example, `su -c "${startcmd}" user1`

2. Add the script to `/etc/init.d/` directory.
3. Install the startup script to respective runlevels using the command `update-rc.d`. For example, give the following command for the sample script shown in step 1:

```bash
sudo update-rc.d appserver defaults
```

The `defaults` option in the above command makes the service to start in runlevels 2, 3, 4 and 5 and to stop in runlevels 0, 1 and 6.

A **runlevel** is a mode of operation in Linux (or any Unix-style operating system). There are several runlevels in a Linux server and each of these runlevels is represented by a single digit integer. Each runlevel designates a different system configuration and allows access to a different combination of processes.

4. You can now start, stop and restart the server using `service <service name> {start|stop|restart}` command. You will be prompted for the password of the user (or root) who was used to start the service.

### Installing as a Windows Service

WSO2 Carbon and any Carbon-based product can be run as a Windows service as described in the following sections:

- Prerequisites
- Setting up the YAJSW wrapper configuration file
- Setting up `CARBON_HOME`
- Running the product in console mode
- Working with the WSO2CARBON service

#### Prerequisites

- Install JDK and set up the `JAVA_HOME` environment variable. For more information, see Installation Prerequisites.
- Download and install a service wrapper library to use for running your WSO2 product as a Windows service. WSO2 recommends Yet Another Java Service Wrapper (YAJSW) version 11.03, and several WSO2 products provide a default `wrapper.conf` file in their `<PROD UCT_HOME>/bin/yajsw/` directory. The instructions below describe how to set up this file.

#### Setting up the YAJSW wrapper configuration file

The configuration file used for wrapping Java Applications by YAJSW is `wrapper.conf`, which is placed in the `<YAJSW_HOME>/conf/` directory. Following is the minimal `wrapper.conf` configuration for running a WSO2 product as a Windows service. Create a file named `wrapper.conf` file, set its properties as follows, and save it in `<YAJSW_HOME>/conf/` directory and in the `<SP_HOME>/bin/yajsw/` directory.

If you want to set additional properties from an external registry at runtime, store sensitive information like usernames and passwords for connecting to the registry in a properties file and secure it with secure vault.

#### Minimal wrapper.conf configuration

```ini
# working directory
#******************************************************************************
wrapper.working.dir=${carbon_home}/
#******************************************************************************
```

If you want to keep the scripts in a location other than `/etc/init.d/` folder, you can add a symbolic link to the script in `/init.d/` and keep the actual script in a separate location. Say your script name is `appserver` and it is in `/opt/WSO2/` folder, then the commands for adding a link to `/etc/init.d/` is as follows:

- Make executable: `sudo chmod a+x /opt/WSO2/appserver`
- Add a link to `/etc/init.d/`: `sudo ln -snf /opt/WSO2/appserver /etc/init.d/appserver`
# YAJSW: default is "org.rzo.yajsw.app.WrapperJVMMain"
# DO NOT SET THIS PROPERTY UNLESS YOU HAVE YOUR OWN IMPLEMENTATION
# wrapper.java.mainclass=
#********************************************************************
# tmp folder
# yajsw creates temporary files named in_.. out_.. err_.. jna..
# per default these are placed in jna.tmpdir.
# jna.tmpdir is set in setenv batch file to <yajsw>/tmp
#********************************************************************
wrapper.tmp.path = ${jna_tmpdir}
#********************************************************************
# Application main class or native executable
# One of the following properties MUST be defined
#********************************************************************
# Java Application main class
wrapper.java.app.mainclass=org.wso2.carbon.bootstrap.Bootstrap
# Log Level for console output.  (See docs for log levels)
wrapper.console.loglevel=INFO
# Log file to use for wrapper output logging.
wrapperlogfile=${wrapper_home}/log/wrapper.log
# Format of output for the log file.  (See docs for formats)
#wrapper.logfile.format=LPTM
# Log Level for log file output.  (See docs for log levels)
#wrapper.logfile.loglevel=INFO
# Maximum size that the log file will be allowed to grow to before
# the log is rolled. Size is specified in bytes. The default value
# of 0, disables log rolling by size. May abbreviate with the 'k' (kB) or
# 'm' (mB) suffix. For example: 10m = 10 megabytes.
# If wrapper.logfile does not contain the string ROLLLNUM it will be
# automatically added as suffix of the file name
wrapper.logfile.maxsize=10m
# Maximum number of rolled log files which will be allowed before old
# files are deleted. The default value of 0 implies no limit.
wrapper.logfile.maxfiles=10
# Title to use when running as a console
wrapper.console.title="WSO2 Carbon"
#********************************************************************
# Wrapper Windows Service and Posix Daemon Properties
#********************************************************************
# Name of the service
wrapper.ntservice.name="WSO2CARBON"
# Display name of the service
wrapper.ntservice.displayname="WSO2 Carbon"
# Description of the service
wrapper.ntservice.description="Carbon Kernel"
#********************************************************************
# Wrapper System Tray Properties
#********************************************************************
# enable system tray
wrapper.tray = true
# TCP/IP port. If none is defined multicast discovery is used to find the port
# Set the port in case multicast is not possible.
wrapper.tray.port = 15002
# Exit Code Properties
# Restart on non zero exit code
wrapper.on_exit.0=SHUTDOWN
wrapper.on_exit.default=RESTART
# Trigger actions on console output
# On Exception show message in system tray
wrapper.filter.trigger.0=Exception
wrapper.filter.script.0=scripts\trayMessage.gv
wrapper.filter.script.0.args=Exception
# genConfig: further Properties generated by genConfig
placeHolderSoGenPropsComeHere=
wrapper.java.command = ${java_home}\bin\java
wrapper.java.classpath.1 = ${java_home}\lib\tools.jar
wrapper.java.classpath.2 = ${carbon_home}\bin\*.jar
wrapper.app.parameter.1 = org.wso2.carbon.bootstrap.Bootstrap
wrapper.app.parameter.2 = RUN
wrapper.java.additional.1 = -Xbootclasspath/a:${carbon_home}\lib\xboot\*.jar
wrapper.java.additional.2 = -Xms256m
wrapper.java.additional.3 = -Xmx1024m
wrapper.java.additional.4 = -XX:MaxPermSize=256m
wrapper.java.additional.5 = -XX:+HeapDumpOnOutOfMemoryError
wrapper.java.additional.6 = -XX:HeapDumpPath=${carbon_home}\repository\logs\heap-dump.hprof
wrapper.java.additional.7 = -Dcom.sun.management.jmxremote
wrapper.java.additional.8 = -Djava.endorsed.dirs=${carbon_home}\lib\endorsed;${java_home}\jre\lib\endorsed
wrapper.java.additional.9 = -Dcarbon.registry.root=\/
wrapper.java.additional.10 = -Dcarbon.home=${carbon_home}
wrapper.java.additional.11 = -Dwso2.server.standalone=true
wrapper.java.additional.12 = -Djava.command=${java_home}\bin\java
wrapper.java.additional.13 = -Djava.io.tmpdir=${carbon_home}\tmp
wrapper.java.additional.14 = -Dcatalina.base=${carbon_home}\lib\tomcat
wrapper.java.additional.15 = -Djava.util.logging.config.file=${carbon_home}\repository\conf\log4j.properties
wrapper.java.additional.16 = -Dcarbon.config.dir.path=${carbon_home}\repository\conf
wrapper.java.additional.17 = -Dcarbon.logs.path=${carbon_home}\repository\logs
wrapper.java.additional.18 = -Dcomponents.repo=${carbon_home}\repository\components\plugins
wrapper.java.additional.19 = -Dconf.location=${carbon_home}\repository\conf
wrapper.java.additional.20 =
-Dcom.atomikos.icatch.file=${carbon_home}\lib\transactions.properties
wrapper.java.additional.21 = -Dcom.atomikos.icatch.hide_init_file_path=true

wrapper.java.additional.22 =
-Dorg.apache.jasper.runtime.BodyContentImpl.LIMIT_BUFFER=true
wrapper.java.additional.23 =
Setting up CARBON_HOME

Extract the Carbon-based product that you want to run as a Windows service, and then set the Windows environment variable CARBON_HOME to the extracted product directory location. For example, if you want to run ESB 4.5.0 as a Windows service, you would set CARBON_HOME to the extracted wso2esb-4.5.0 directory.

Running the product in console mode

To verify that YAJSW is configured correctly for running the Carbon-based product as a Windows service, follow the steps below.

1. Open a Windows command prompt and navigate to the <YAJSW_HOME>/bat/ directory. For example:

```plaintext
cd C:\Documents and Settings\yajsw_home\bat
```

2. Start the wrapper in console mode by issuing the following command:

```plaintext
runConsole.bat
```

For example:

```plaintext
C:\Documents and Settings\yajsw_home\bat>runConsole.bat
```

If the configurations are set properly for YAJSW, you can see a console output similar to the following, and you can now access the UIs of your product from your web browser via the relevant hosts and ports.
Working with the WSO2CARBON service

To install the Carbon-based product as a Windows service, execute the following command in the <YAJSW_HOME>/bat/ directory:

```
installService.bat
```

The console displays a message confirming that the WSO2CARBON service was installed.

```plaintext
C:\Documents and Settings\yajsw_home\bat\installService.bat
C:\Documents and Settings\yajsw_home\bat\cd C:\Documents and Settings\yajsw_home\bat\
C:\Documents and Settings\yajsw_home\bat\call setenv.bat
"java" -Xmx30m -Djna_tmpdir="C:\Documents and Settings\yajsw_home\bat\..\tmp" -jar "C:\Documents and Settings\yajsw_home\bat\..\wrapper.jar" -i "C:\Documents and Settings\yajsw_home\bat\..\conf\wrapper.conf"
YAJSW: yajsw-stable-1.1.03
OS : Windows XP/5.2/amd64
JVM : Oracle Corporation/1.7.0_06
INFO: Using "C:\\DOCUME1\ADMINI1\LOCALS1\Temp\vfs_cache" as temporary files store.
platform null
*************************************************************************** INSTALLING "WSO2CARBON"***************************************************************************
Service "WSO2CARBON" installed
Press any key to continue . . .
```

To start the service, execute the following command in the same console window:

```
startService.bat
```

The console displays a message confirming that the WSO2CARBON service was started.

```plaintext
C:\Documents and Settings\yajsw_home\bat\startService.bat
C:\Documents and Settings\yajsw_home\bat\cd C:\Documents and Settings\yajsw_home\bat\
C:\Documents and Settings\yajsw_home\bat\call setenv.bat
"java" -Xmx30m -Djna_tmpdir="C:\Documents and Settings\yajsw_home\bat\..\tmp" -jar "C:\Documents and Settings\yajsw_home\bat\..\wrapper.jar" -t "C:\Documents and Settings\yajsw_home\bat\..\conf\wrapper.conf"
YAJSW: yajsw-stable-1.1.03
OS : Windows XP/5.2/amd64
JVM : Oracle Corporation/1.7.0_06
Dec 30, 2012 1:09:00 PM org.apache.commons.vfs2.VfsLog info
INFO: Using "C:\\DOCUME1\ADMINI1\LOCALS1\Temp\vfs_cache" as temporary files store.
platform null
*************************************************************************** STARTING "WSO2CARBON"***************************************************************************
Service "WSO2CARBON" started
Press any key to continue . . .
```

To stop the service, execute the following command in the same console window:

```
stopService.bat
```

The console will display a message confirming that the WSO2CARBON service has stopped.
To uninstall the service, execute the following command in the same console window:

```
uninstallService.bat
```

The console will display a message confirming that the WSO2CARBON service was removed.

```
C:\Documents and Settings\yajsw_home\bat\uninstallService.bat
C:\Documents and Settings\yajsw_home\bat\cd C:\Documents and Settings\yajsw_home\bat\n
C:\Documents and Settings\yajsw_home\bat\call setenv.bat
"java" -Xmx30m -Djna.tmpdir="C:\Documents and Settings\yajsw_home\bat/..\tmp" -jar "C:\Documents and Settings\yajsw_home\bat/..\wrapper.jar" -r "C:\Documents and Settings\yajsw_home\bat/..\conf\wrapper.conf"
JAVA: yajsw-stable-11.03
OS : Windows XP/5.2/amd64
JVM : Oracle Corporation/1.7.0_06
INFO: Using "C:\\DOCUMENTS\\ADMINI\\LOCALS\\Temp\\vfs_cache" as temporary files store.
platform null
*************** STOPPING "WSO2CARBON" ***************

Service "WSO2CARBON" stopped
Press any key to continue . . .
```

Running the Product

The following sections cover how to run WSO2 Stream Processor.

It has the following four profiles:

- **Editor**: Provide the developer environment to develop the Stream Processing Application on your local machine.
- **Worker**: Lets you deploy the Stream Processing Application and run it in production.
- **Dashboard**: Lets you visualize results via the Dashboard Portal, manage stream processing logic via Business Rules, and monitor the whole system via the Status Dashboard.
- **Manager**: Lets you deploy and manage streaming applications on a fully distributed Stream Processor deployment.

The following section demonstrates how to run each of those profiles:

- **Starting the Stream Processor Studio**
- **Starting a worker node**
- **Starting a dashboard node**
- **Starting a manager node**
Starting the Stream Processor Studio

To start the editor for developing Siddhi Applications, follow the steps below:

1. Navigate to the <SP_HOME>/bin directory.
2. Issue the following command to start the server of the Stream Processor Studio.
   - For Windows: editor.bat
   - For Linux: ./editor.sh
3. Use the following URL to access the editor UI.
   - http://localhost:<EDITOR_PORT>/editor (e.g., https://localhost:9390/portal)

Important
The Stream Processor Studio runtime is recommended to be used only when developing Siddhi applications. Therefore, it is designed to be run only in a development environment. It is not meant to be deployed in a remote or docker setup.

Starting a worker node

To start a worker node to deploy and run Siddhi Applications in production, follow the steps below:

1. Navigate to the <SP_HOME>/bin directory.
2. Use the following command to start the server of the Stream Processor Studio.
   - For Windows: worker.bat
   - For Linux: ./worker.sh

   In order to start a worker with only one Siddhi application deployed in it, use the -Dfile property with the path to the relevant Siddhi file as shown below.
   - For Windows: worker.bat -Dfile=<SIDDHI_FILE_PATH>
   - For Linux: ./worker.sh -Dfile=<SIDDHI_FILE_PATH>

Starting a dashboard node

To start a dashboard node to use the Dashboard Portal, Business Rules and Status Dashboard, follow the steps below:

1. Navigate to the <SP_HOME>/bin directory.
2. Issue the following command to start the server of the Stream Processor Studio.
   - For Windows: dashboard.bat
   - For Linux: ./dashboard.sh
3. Use the following URL to access the UIs.
   - Portal: https://<HOSTNAME>:<DASHBOARD_PORT>/portal (e.g., https://localhost:9643/portal)
   - Status Dashboard: https://<HOSTNAME>:<DASHBOARD_PORT>/monitoring (e.g., https://localhost:9643/monitoring)

Starting a manager node

To start a manager node to manage a fully distributed Stream Processor deployment, follow the steps below:

1. Navigate to the <SP_HOME>/bin directory.
2. Use the following command to start the manager node of the Stream Processor Studio.
   - For Windows: manager.bat
   - For Linux: ./manager.sh

Upgrading from a Previous Release

This section is currently under construction!

This section explains how you can upgrade to SP 4.3.0 from SP 4.2.0 or from DAS/CEP. Click on the relevant tab based on the product that you are currently using.
This section explains how to upgrade to WSO2 SP 4.3.0 from WSO2 SP 4.2.0.

Preparing to upgrade

The following prerequisites should be completed before upgrading.

- Make a backup of the SP 4.2.0 database and copy the `<SP_HOME_4.2.0>` directory in order to backup the product configurations.
- Download WSO2 SP 4.3.0 from here.

Migrating databases

To migrate the databases from SP 4.2.0 to SP 4.3.0, follow the steps below:

1. To connect WSO2 SP 4.3.0 to the same databases as WSO2 SP 4.2.0 so that the persisted data can be accessed, configure the datasources in `<SP_HOME>/conf/<runtime>/deployment.yaml` files similar to how they were configured in WSO2 SP 4.2.0. For the complete list of datasources configured for WSO2 SP, see Configuring Datasources.

2. To update the aggregation tables as required due to the changes in the Incremental Aggregation functionality of SP 4.3.0, run the `<SCRIPT_NAME>` script. This is can be done by issuing the following commands.

Migrating configurations

It is required to migrate configurations in the `<SP_HOME>/conf/<runtime>/deployment.yaml` files from SP 4.2.0 to SP 4.3.0. The following are the changes in the configurations included in these files between the two releases:

- The clustering configurations included under the `#deployment.config:` section, and under `type: ha`. For more information about the parameters you need to configure in this section for SP 4.3.0, see Minimum High Availability Deployment - Configuring a minimum HA cluster.

If you are deploying SP as a minimum HA cluster, note that the mechanism for sending requests to this setup has changed between the two releases. In SP 4.3.0, requests should be sent only to the active node unlike in SP 4.2.0 where you are required to send the requests to both the nodes. For more details about this, see Minimum High Availability Deployment - Publishing events to the cluster.

- The Siddhi Store Query API in WSO2 SP uses the HTTP/HTTPS transport to allow users to send requests to its endpoint. In SP 4.3.0, a new section named `siddhi.stores.query.api:` was added in both the `<SP_HOME>/conf/worker/deployment.yaml` file and the `<SP_HOME>/conf/editor/deployment.yaml` file. For more information about these parameters, see Managing Stored Data via REST APIs.

In addition to the above, compare the `deployment.yaml` files for both versions, and check whether you need to do the same changes to the other configuration parameters you may have made in the SP 4.2.0 version in the SP 4.3.0 version as well.

The `deployment.yaml` files must not be copied directly between servers due to certain differences in the parameters included in the two WSO2 SP versions.

Migrating Siddhi applications

To migrate the Siddhi applications that you have deployed in WSO2 SP 4.2.0, follow the procedure below:
In SP 4.3.0, data stored in tables for incremental aggregation queries are automatically purged by default. If you do not want the data to be purged or if you want to retain the data for a time period longer than the default data purging interval, you need to update the relevant aggregation queries with the required data purging related configurations. For more information, see Incremental Analysis - @purge.

1. Copy all the Siddhi applications in `<SP_HOME_4.2.0>/wso2/worker/deployment/siddhi-files` directory.
2. Place the Siddhi applications you copied to the `<SP_HOME_4.3.0>/wso2/worker/deployment/siddhi-files` directory.

Migrating dashboards

To migrate the dashboards you were using in SP 4.2.0 with all their contents, follow the procedure below:

1. Check the datasources configurations that you have added in the `<SP_4.2.0_HOME>conf/dashboards/deployment.yaml` file for the WSO2_DASHBOARD_DB datasource, and do a database dump.
2. Import the database dump according to the dashboard datasource configuration that you have added in the `<SP_4.3.0_HOME>conf/dashboards/deployment.yaml` file.
3. If you have created any custom widgets in SP 4.2.0, copy the relevant subdirectory for that widget from the `<SP_HOME_4.2.0>/wso2/dashboard/deployment/web-ui-apps/portal/extensions/widgets` directory, and place it in the `<SP_HOME_4.3.0>/wso2/dashboard/deployment/web-ui-apps/portal/extensions/widgets` directory.

Testing the migration

Simulate a few events to the Siddhi applications deployed in SP 4.3.0 to test whether they are generating the expected results.

This section provides information on how you can upgrade from DAS/CEP to Product SP.

Overview

WSO2 DAS 3.1.0 is the predecessor of WSO2 SP 4.0.0. Similar to SP, DAS processed events via an event flow that consisted of event streams, receivers, publishers, and execution plans. These elements of the event flow are defined separate from each other via the DAS Management Console.

WSO2 SP defines the complete event flow within a single application created via a Siddhi file. The application is then deployed in a SP worker node and executed at runtime. Due to this architectural difference between the two products, configurations cannot be directly migrated from DAS to SP. Instead, they need to be recreated as explained in this section.

With WSO2 SP’s streaming SQL capabilities and its inbuilt editor that has event simulation and debugging support, it can help you to create real-time applications much faster than before. The WSO2 SP is a new product focusing on solving stream processing and complex event processing use cases.

WSO2 SP uses a single Siddhi Streaming SQL file for script data collection, processing, and notification logic. The batch analytics aspect is handled via Siddhi aggregations.

For more information about the key capabilities of WSO2 SP, see About This Release.

Deployable Artifacts

Siddhi applications are the deployable artifact type of the Stream Processor.

To use Siddhi, you need to write the processing logic as a Siddhi application in the Siddhi Streaming SQL language. Once a Siddhi application is created and started, it does the following:

1. Takes data one-by-one as events
2. Processes the data per each event
3. Generates new high level events based on the processing carried out up to the current time
4. Sends newly generated events as the output to streams.
An element of Siddhi SQL can be composed together as a script in a Siddhi application. Here, each construct must be separated by a semicolon ( ; ) as shown in the syntax below.

```
<siddhi app>  
  <app annotation> *  
    ( <stream definition> | <table definition> | ... ) +  
    ( <query> | <partition> ) +  
;  
```

The following is a sample Siddhi application named Temperature-Analytics that includes a stream named TempStream and a query named 5minAvgQuery to process the events handled by it.

```
@app:name('Temperature-Analytics')

define stream TempStream (deviceID long, roomNo int, temp double);  

@name('5minAvgQuery')
from TempStream#window.time(5 min)
select roomNo, avg(temp) as avgTemp  
group by roomNo  
insert into OutputStream;
```

The following are Siddhi SQL element types in your DAS setup that you can redefine in a Siddhi application so that you can reuse them with WSO2 SP.

- Event Streams -> Stream Definition
- Event Receiver -> Source
- Event Publisher -> Sink
- Event Store -> Table
- Execution Plan -> Queries
- Upgrading the database

Event Streams -> Stream Definition

A stream such as the TempStream stream unifies common types of events together. This enables them to be processed via queries using their defined attributes in a streaming manner, and allow sinks and sources to map events to/from various data formats.

When migrating event stream definitions in WSO2 DAS, you must rewrite them in the syntax followed in Siddhi applications as illustrated in the table below.
Configuration in WSO2 CEP/DAS

```
{
    "streamId": "TempStream:1.0.0",
    "name": "TempStream",
    "version": "1.0.0",
    "metaData": [
        {
            "name": "ip",
            "type": "STRING"
        }
    ],
    "correlationData": [
        {
            "name": "id",
            "type": "LONG"
        }
    ],
    "payloadData": [
        {
            "name": "deviceID",
            "type": "LONG"
        },
        {
            "name": "roomNo",
            "type": "int"
        },
        {
            "name": "temp",
            "type": "DOUBLE"
        }
    ]
}
```

Configuration in Siddhi file

```
define stream TempStream
    (deviceID long, roomNo int, temp double);
```

Event Receiver -> Source

In WSO2 CEP/DAS, events are received by event receivers that manage the event retrieval process. In Siddhi files deployed in WSO2SP, you need to configure sources instead of event receivers to receive events.

To configure a stream that consumes events via a source, add the source configuration to a stream definition by adding the `@source` annotation with the required parameter values.
The `@source` parameter defines the source type that receives events. The other parameters to be configured depends on the source type selected. Some of the the parameters are optional.

For detailed information about the parameters see the documentation for the relevant source.
The following is the list of source types that are currently supported:

- HTTP
- Kafka
- TCP
- In-memory
- WSO2Event
- Email
- JMS
- File
- RabbitMQ
- MQTT

**Event Publisher -> Sink**

In WSO2 CEP/DAS, events are published via event publishers that manage the event publishing process. In Siddhi files deployed in WSO2SP, you need to configure sinks instead of event publishers to publish events.

To configure a stream that provides events via a sink, add the sink configuration to a stream definition by adding the `@sink` annotation with the required parameter values.

**Configuration in WSO2 CEP/DAS**

```xml
<eventPublisher
    name="httpLogger"
    statistics="disable"
    trace="disable"
    xmlns="http://wso2.org/carbon/eventpublisher">
    <from
        streamName="org.wso2.event.sensor.stream" version="1.0.0"/>
    <mapping
        customMapping="disable"
        type="text"/>
    <to
        eventAdapterType="logger">
        <property
            name="uniqueId">org.wso2.event.statistics.logger</property>
    </to>
</eventPublisher>
```

**Configuration in Siddhi file**

```siddhi
@Sink(type = 'log', @map(type = 'text'))
define stream sensorStream (sensorId int, temperature double)
```

For detailed information about the parameters see the documentation for the relevant sink.
The following is a list of currently supported sink types.

- HTTP
- Kafka
- TCP
- In-memory
- WSO2Event
- Email
- JMS
- File
- RabbitMQ
- MQTT

**Sink and Source Mappers**

In WSO2 CEP/DAS server the supported default format for the Message Format property is configured under Mapping Configuration when creating event receivers. In the Siddhi files, this is replaced with the type parameter of the @map annotation that defines the map type to be used to map the data.

Each @source and @sink configuration has a mapping denoted by the @map annotation that converts the incoming messages format to Siddhi events.

<table>
<thead>
<tr>
<th>Configuration in WSO2 CEP/DAS</th>
<th>Configuration in Siddhi file</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;mapping customMapping=&quot;disable&quot; type=&quot;map_type&quot;/&gt;</code></td>
<td><code>@map(type='map_type')</code></td>
</tr>
</tbody>
</table>

The other parameters to be configured depends on the mapper selected. Some of these parameters are optional. For detailed information about the parameters see the documentation for the relevant mapper.

The following is a list of currently supported source mapping types:

- WSO2Event
- XML
- TEXT
- JSON
- Binary
- Key Value

**Map Attributes**

@attributes is an optional annotation used with the @map annotation to define custom mapping that replaces `<mapping customMapping>` in the CEP/DAS server. When the @attributes annotation is not provided, each mapper assumes that the incoming events adhere to its own default data format. By adding the @attributes annotation, you can configure mappers to extract data from the incoming message selectively, and assign them to attributes.

There are two ways you can configure map attributes.

1. Defining attributes as keys and mapping content as values in the following format.
   ```
   @attributes({ attributeN='mapping_N', attribute1='mapping_1' })
   ```
2. Defining the mapping content of all attributes in the same order as how the attributes are defined in stream definition.
   ```
   @attributes({ 'mapping_1', 'mapping_N' })
   ```

**Example**
This query receives events via the HTTP source in the JSON data format, and directs them to the InputStream stream for processing. Here the HTTP source is configured to receive events on all network interfaces at the 8080 port and on the foo context. The source is also secured via basic authentication.

Configuration in WSO2 CEP/DAS

```xml
<eventReceiver ... xmlns="http://wso2.org/carbon/eventreceiver">
  <from ... />
  <mapping customMapping="enable" type="json">
    <from streamName="sensor.stream" version="1.0.6"/>
    <property>
      <from dataType="meta" name="time"/>
      <to name="meta_timestamp" type="long"/>
    </property>
    <property>
      <from dataType="meta" name="isPowerServed"/>
      <to name="meta_isPowerSaverEnabled" type="bool"/>
    </property>
    <property>
      <from dataType="meta" name="id"/>
      <to name="meta_sensorId" type="int"/>
    </property>
  </mapping>
  <to ... />
</eventReceiver>
```
**Event Store -> Table**

In CEP/DAS, event streams are persisted by creating a corresponding table in the WSO2 Data Access Layer for batch analysis. In WSO2 Stream Processor, this functionality is replaced by Table which is a stored version of an stream or a table of events. Its schema is defined via the table definition that is similar to a stream definition.

These events are by default stored in-memory, but Siddhi also provides store extensions to work with data/events stored in various data stores through the table abstraction.

Tables allow Siddhi to work with stored events. By defining a schema for tables, Siddhi allows them to be processed by queries using their defined attributes with the streaming data. You can also interactively query the state of the stored events in the table.

```siddhi
@PrimaryKey('symbol')
define table StockTable (symbol string, price float, volume long);
```

**Indexes**

Indexes allow tables to be searched/modified much faster.

Indexes are configured by including the `@Index('key1', 'key2')` annotation to the table definition. Each event table configuration can have 0-1 `@Index` annotations. Support for the `@Index` annotation and the number of attributes supported differ based on the table implementations. When more then one attribute is used for index, each one of them is used to index the table for fast access of data. Indexes can be configured together with primary keys.

**Example**

This query creates an indexed event table named `RoomTypeTable` with the `roomNo` attribute as the index key.

```siddhi
@Index('roomNo')
define table RoomTypeTable (roomNo int, type string);
```

**Execution Plan -> Queries**

Queries used in DAS/CEP execution plans and Stream Processor are almost same. There are a few newly introduced features for Siddhi 4.0.0 that are used with WSO2 Stream Processor. These features are listed below.
Incremental Aggregation

Incremental aggregation allows user to retrieve the aggregate value for different time durations. That is, it allows user to obtain aggregates such as sum, count, avg, min, max, and count) of stream attributes for durations such as sec, min, hour, etc.

Following is an example query.

```
define stream TradeStream (symbol string, price double, volume long, timestamp long);

define aggregation TradeAggregation
from TradeStream
select symbol, avg(price) as avgPrice, sum(price) as total
  group by symbol
aggregate by timestamp every sec ... year;
```

Set keyword

Set keyword allows you to update selected attributes from the table.

Here, for each assignment, the attribute specified in the left must be the table attribute, and the one specified in the right can be a stream/table attribute a mathematical operation, or other.

When the set clause is not provided, all the attributes in the table are updated. This works with the update and update or insert operations.

The following is a sample query.

```
FROM fooStream
SELECT roomNo, time: timestampInMilliseconds () as ts
UPDATE barTable
  SET barTable.timestamp = ts
ON  barTable.room_no == roomNo AND roomNo > 2
```

Pattern to identify non-occurrence of events

Patterns and sequences are the key features of a complex event processor to define a new complex event based on the order of one or more raw events. A pattern can be an atomic pattern that detects the arrival of a specific event, or a complex pattern that detects the arrival of more events in a defined order. Although patterns generally define complex events based on the order of events that arrive, sometimes a complex event may depend on an event that should not arrive.

Usually, Siddhi pattern processors wait for the events until they arrive. Once an event arrives, the pattern processor starts looking for the next event. When detecting events that have not arrived, the pattern processor must not wait for an infinite time period to declare the non-arrival of the event. Therefore, a time interval to wait for the event must be defined with absent pattern operators with an exception to the logical and pattern combining an event that is expected to arrive, and an event that must not arrive beforehand.

Following is a sample query.

```
from TemperatureStream[temp > 60] -> not FireAlarmStream[active == true] for 5 sec
select 'Fire alarm not working' as message
insert into AlertStream;
```

Defined Window
A defined window is a window that can be shared across multiple queries. Events can be inserted to a defined window from one or more queries and it can produce output events based on the defined window type. The following is a sample query.

```java
define stream TempStream(tempId string, temp double);
define window OneMinTempWindow(tempId string, temp double) time(1 min);

from TempStream
select *
insert into OneMinTempWindow;
```

**Upgrading the database**

WSO2 SP stores product-specific data in H2 databases by default. Those databases are located in the `<PRODUCT_HOME>/wso2/<Profile>/database` directory.

This embedded H2 database is suitable for development, testing, and for some production environments. However, we recommend that you use an industry-standard RDBMS such as Oracle, PostgreSQL, MySQL, MS SQL, etc., because they are more suitable for most production environments. Most table schemas are self-generated by the feature itself. For the other table schemas you can use the scripts provided with WSO2 SP in the `<SP_HOME>/wso2/<Profile>/dbscripts` directory to install and configure databases. This directory includes scripts for Oracle, PostgreSQL, MySQL and MS SQL.

CEP database migration is direct as it only uses RDBMS tables without any proprietary encoding.

DAS supports data persistence in two types of databases.

1. RDBMS Event tables
2. Analytics Tables (which also include Analytics Event Tables)

RDBMS Event table migration is straightforward because WSO2 SP supports RDBMS.

Analytics tables migration is an indirect process where we need to convert the Analytics Tables into RDBMS tables by running Spark scripts using the CarbonJDBC provider packed with DAS.

The following is an sample query that can be run on DAS to migrates the `ORG_WSO2_DAS_SAMPLE_SMART_HOME_DATA` event table into an RDBMS table.

```sql
CREATE TEMPORARY TABLE SMART_HOME_DATA
USING CarbonAnalytics OPTIONS (tableName "ORG_WSO2_DAS_SAMPLE_SMART_HOME_DATA", schema "house_id INT, metro_area STRING, state STRING, device_id INT, power_reading FLOAT, is_peak BOOLEAN");

CREATE TEMPORARY TABLE SMART_HOME_DATA_RDBMS using CarbonJDBC OPTIONS (dataSource "WSO2_SP_MIGRATION_DATA_STORE_DB", tableName "ORG_WSO2_DAS_SAMPLE_SMART_HOME_DATA", schema "house_id INTEGER -i, metro_area STRING, state STRING, device_id INTEGER, power_reading FLOAT, is_peak BOOLEAN");

INSERT INTO TABLE SMART_HOME_DATA_RDBMS SELECT * FROM SMART_HOME_DATA;
```
Key Concepts

The following are some of the concepts and terminology associated with WSO2 Stream Processor.

Streaming Analytics

Using analytical operators to orchestrate data flow, calculate analytics, and detect patterns on event data from multiple, disparate live data sources, and allowing you to build applications that sense, think, and act in real time.

Complex Event Processing (CEP)

A kind of computing in which incoming data about events is distilled into more useful, higher level “complex” event data that provides insight into what is happening. CEP is used for highly demanding, continuous-intelligence applications that enhance situation awareness and support real-time decisions.

Siddhi

This is the core processing library used within WSO2 Stream Processor to handle collecting events, processing them and notifying/acting based on the user-given Siddhi Applications.

Siddhi Streaming SQL Language

Siddhi Streaming SQL is an SQL-like language that facilitates writing stream processing and complex event processing logic. This defines how the events received to Stream Processing should be processed.

Siddhi Application

This is scripted using the Siddhi Streaming SQL Language. This represents a single unit of execution that can be deployed independently to process events in real-time.

Portal/Dashboard

The portal enables users to create dashboards that represent data visually to better understand the data being analyzed. The portal can contain multiple dashboards and each of these dashboards can also contain several widgets to represent different pieces of information.

Widget

This is a single visualization element that can represent data and can be reused in many dashboards.

Tutorials

This set of tutorials gives you a complete introduction to the fundamentals and most common usage scenarios of the WSO2 Stream Processor. It is recommended to try out these tutorials in the given sequence in order to understand the functional flows better.
Introduction

The first tutorial introduces you to WSO2 SP and explains some of its basic concepts.

For this tutorial, let's consider a scenario where Sam, the foreman of the Sweet Factory wants a name-wise total for all the categories of sweets produced until now. During a given run, each Sweet Bot in the factory produces a batch of sweets and generates an event that contains information about the name of the sweet it produced and the quantity produced.

Individual events generated by four different worker Sweet Bots are as follows:

<table>
<thead>
<tr>
<th>Input Event</th>
<th>Sweet Bot No</th>
<th>Item</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Toffee</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Gateau</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Gingerbread</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Gateau</td>
<td>8</td>
</tr>
</tbody>
</table>

The foreman expects to receive the following output from WSO2 SP if the above events are sent in the given order.

<table>
<thead>
<tr>
<th>Output Event</th>
<th>Item</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Toffee</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Gateau</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Gingerbread</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Gateau</td>
<td>10</td>
</tr>
</tbody>
</table>

This tutorial shows how this data is processed to generate the expected output if it is sent to WSO2 SP to be analyzed.

This tutorial covers the following:

- Creating a simple Siddhi Application
- Simulating input to the Siddhi Application

Before you begin:

1. Install Oracle Java SE Development Kit (JDK) version 1.8* and set the JAVA_HOME environment variable.
2. Download the WSO2 SP.

Tutorial steps

The Siddhi application required for this scenario can be created in two methods:

- **Source View** of WSO2 Stream Processor Studio where you can write the application in Siddhi Query Language.
- **Design View** of WSO2 Stream Processor where you can design the Siddhi application in a graphical interface by dragging and dropping Siddhi components and linking them as required.

Click on the relevant tab to create the Siddhi application in the preferred interface.
Let's get started!

1. Navigate to the `<SP_HOME>/bin` directory and issue the following command to start the WSO2 Stream Processor Studio.
   - For Windows: `editor.bat`
   - For Linux: `sh editor.sh`


3. Click **New** to start defining a new Siddhi application. A new file opens as shown below.

4. First, you need to specify the stream from which the data to be analyzed must be taken. For this tutorial, let's assume that all data to be analyzed is sent to a stream named `SweetProductionStream`. Let's add a query as follows to specify that information to be processed must be taken from the `SweetProductionStream` stream.

   ```
   from SweetProductionStream
   ```

5. To specify what information needs to be extracted from the events generated by the Sweet Bots, let's add a query as follows.
select name, sum(amount) as totalProduction

Here, the value for the name attribute of the incoming event is extracted with the same name. The sum of the values for the amount attribute of incoming events is extracted as hourlyTotal.

6. The amounts need to be grouped according to the sweet category name. To do this, add a group by clause as follows.

   group by name

7. The data taken from the input stream must be sent to an output stream before it is published. In this tutorial, let’s name the output stream as SweetTotalStream. This can be specified as shown below.

   insert into SweetTotalStream;

The Siddhi application should now look as follows.

   from SweetProductionStream
   select name, sum(amount) as totalProduction
   group by name
   insert into SweetTotalStream;

8. Before saving these queries as a Siddhi application, you must name your application. In this scenario, let’s name it as SweetTotalApp as shown below.

   @App:name('SweetTotalApp')

9. Let’s also specify a name for the queries you created as shown below to allow someone checking the Siddhi application to understand their purpose.

   @info(name='SweetTotalQuery')

10. The streams that you referred to in your queries need to be defined as follows.

   a. Let’s define the input stream as follows.

      define stream SweetProductionStream (name string, amount long);

      There are two attributes included in the schema to capture the name of the sweet category as a string value, and the amount produced as a long value.

   b. Let’s define the output stream as follows.

      @sink(type='log', prefix='Sweet Totals:')
      define stream SweetTotalStream(name string, totalProduction long);
This output stream definition includes the `name` attribute to present the name of the sweet category, and the `totalProduction` attribute to present the total production of each sweet category. In addition, there is a `@sink` annotation. This represents a logger sink that outputs all the data in the `SweetTotalStream` output stream to which it is connected. The output is presented as logs that are printed in the foreman's console.

Event sinks are covered in detail in Publishing Processed Events.

Now the Siddhi application is ready to be saved. It should look as follows.

```siddhi
@App:name('SweetTotalApp')
define stream SweetProductionStream (name string, amount long);
@sink(type='log', prefix='Sweet Totals:')
define stream SweetTotalStream(name string, totalProduction long);
@info(name='SweetTotalQuery')
from SweetProductionStream
select name, sum(amount) as totalProduction
group by name
insert into SweetTotalStream;
```

To save this Siddhi file, click **File**, and then click **Save**. The following dialog box appears. Click **Save**.

![Save To Workspace dialog box](image)

As a result, this file is saved in the `<SP_HOME>/wso2/editor/deployment/workspace` directory.

Let's get started!

1. Navigate to the `<SP_HOME>/bin` directory and issue the following command to start the WSO2 Stream Processor Studio.
   - For Windows: `editor.bat`
   - For Linux: `sh editor.sh`

   ![The default URL is](image)
3. Click **New** to start defining a new Siddhi application. A new file opens as shown below.

4. Click **Design View** to open the design view of the Stream Processor Studio.
5. First, you need to specify the stream from which the data to be analyzed must be taken. For this tutorial, let's assume that all data to be analyzed is sent to a stream named `SweetProductionStream`. Let's follow the substeps below.

   a. Drag and drop the following icon from the left pane to the grid. As a result, the **Stream Configuration** form appears below the grid.

   ![Stream Configuration Form](image)

   b. Let's enter information as follows in the **Stream Configuration** form.

      i. In the name field, let's enter `SweetProductionStream`.

      ii. The sweet bots publish the name of the sweet and the number produced in a given batch. To capture this information, let's add two attributes to the stream as follows.

      | Attribute Name | Attribute Type |
      |----------------|---------------|
      | name           | string        |
      | amount         | long          |
6. Once the information published by the sweet bots is analyzed, the result needs to be presented as the name of the sweet and the total produced. To achieve this, you need an output stream with a schema that captures this information. Let's define this stream as follows.
   a. Drag and drop the stream icon to the grid again.
   b. In the Stream Configuration form, enter `SweetProductionStream` as the name of the stream. Then enter two attributes as follows.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
</tr>
<tr>
<td>totalProduction</td>
<td>long</td>
</tr>
</tbody>
</table>
c. Click **Submit** to save the stream configuration.

7. Now the streams to capture the input and the output are ready. Now the Siddhi application you are creating needs to contain the logic based on which the input is processed to generate the required output. In this scenario, the processing to be carried out involved calculating the total sweet production for each sweet with each new batch. This can be achieved via a simple query. To add this query, follow the substeps below.

   a. The query to be added needs to project the output. Therefore, drag and drop the icon for projection queries from the left panel to the grid.

   b. Before defining the processing logic, you need to specify that the query needs to consider the `SweetProductionStream` stream as the input stream, and the `SweetTotalsStream` stream as the output stream. To show this connection, draw an arrow from the `SweetProductionStream` stream to the projection query, and another arrow from the projection query to the `SweetTotalsStream` stream as shown below.
c. To open the **Query Configuration** form and define the processing logic, click the settings icon on the projection query object as shown below.

![Query Configuration form](image)

The name of the sweet needs to be projected in the output with the `name` label (same as the input stream attribute). The total on the other hand needs to be derived by calculating the sum for the amount of sweets produced in all the batches, and output with the `totalProduction` label. To do this, let’s enter information in the **Query Configuration** form as follows.

![Query Configuration form with data](image)

<table>
<thead>
<tr>
<th>Output Attribute</th>
<th>Required Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>sum(amount)</td>
<td>totalProduction</td>
</tr>
</tbody>
</table>

i. The output is projected with the attributes used in the output stream (i.e., the SweetTotalStream). Therefore, in the **Select** section, the attributes in the SweetTotalStream stream are already listed under **As**. To map the attributes in the input stream (i.e., SweetProductionStream) with the same attributes, you need to enter expressions as explained below.

![Output Attribute vs Required Expression](image)
The values for the name attribute in the SweetProductionStream stream can be inserted into the SweetTotalStream stream with the same attribute name. To indicate this, enter name in the Expression field.

To derive the value for the totalProduction attribute, the sum of the total number produced of each sweet must be calculated with each new batch produced by sweet bots. To do this, let's specify the expression as `sum(amount)`.

Each output event indicating the total sweet production at the given time needs to be directed to the SweetTotalsStream output stream in order to be published. Therefore, make sure that all events is selected in the For field under Output.

e. Click Save to save the query configuration.

8. Click the Edit menu option, and then click Auto Align. The Siddhi application you created now looks as follows in the design view.

9. Click Source View. The application is displayed as follows.

10. Before saving these queries as a Siddhi application, you must name your application. In this scenario, let's name it as `SweetTotalApp` as shown below.

```
@App:name('SweetTotalApp')
```

11. Let's also specify a name for the queries you created as shown below to allow someone checking the Siddhi application to understand their purpose.

```
@info(name='SweetTotalQuery')
```

12. Click the File menu option, and then click Save to save the application you created. The following dialog box appears. Click Save.
As a result, this file is saved in the `<SP_HOME>/wso2/editor/deployment/workspace` directory.

Generating the output

Let's see how the Siddhi application you created functions when it is deployed and run in WSO2 SP. For the purpose of learning, the following four events generated by the Sweet Bots are simulated in WSO2 SP via the Event Simulation tool.

To check whether your Siddhi application generates this output, follow the steps below:

1. In the Stream Processor Studio, click the following icon for event simulation. This opens the left panel for event simulation.

2. In the **Single Simulation** tab, do the following:

   a. In the **Siddhi App Name** field, select **SweetTotalApp**.
   b. In the **Stream Name** field, select **SweetProductionStream**.
   c. In the **name** field, enter **Toffee**.
   d. In the **amount** field, enter **11**.
   e. Click **Start and Send**. This will start the siddhi app in run mode and send the event. The following is logged in the console.
In the previous tutorial, you covered how to create a simple Siddhi application and how to simulate virtual events to it. In this tutorial, let's look at how events are received in a real world scenario, and how to use the event capturing framework of WSO2 SP to receive incoming events even when their format is different to the format specified within your Siddhi application.

Let's consider a scenario where the events generated by the Sweet Bots are in the JSON format, and they are sent via HTTP calls. The manager requires them to be received at a specific endpoint that has a specific URL.

This tutorial covers the following topics:
- Event capturing with Siddhi sources
- Using maps with sources
- Custom mappings for inbound events

Before you begin:
- This tutorial uses the same Siddhi application that is created and deployed in Tutorial 1: Creating and Debugging a Simple Siddhi Application. Therefore, it is recommended that you try Tutorial 1 first.
- Sources are the type of extensions that are used in the WSO2 Stream Processor to indicate to the runtime that events would be arriving from the defined endpoint or entity. WSO2 Stream Processor supports many different Source types out of the box, which include HTTP, TCP, Kafka etc. For more information about the sources supported for WSO2 SP, see Collecting Events.

**Consuming Events**

- **Introduction**
- **Tutorial steps**

**Introduction**

In the previous tutorial, you covered how to create a simple Siddhi application and how to simulate virtual events to it. In this tutorial, let's look at how events are received in a real world scenario, and how to use the event capturing framework of WSO2 SP to receive incoming events even when their format is different to the format specified within your Siddhi application.

Let's consider a scenario where the events generated by the Sweet Bots are in the JSON format, and they are sent via HTTP calls. The manager requires them to be received at a specific endpoint that has a specific URL.

This tutorial covers the following topics:
- Event capturing with Siddhi sources
- Using maps with sources
- Custom mappings for inbound events

**Before you begin:**
- This tutorial uses the same Siddhi application that is created and deployed in Tutorial 1: Creating and Debugging a Simple Siddhi Application. Therefore, it is recommended that you try Tutorial 1 first.
- Sources are the type of extensions that are used in the WSO2 Stream Processor to indicate to the runtime that events would be arriving from the defined endpoint or entity. WSO2 Stream Processor supports many different Source types out of the box, which include HTTP, TCP, Kafka etc. For more information about the sources supported for WSO2 SP, see Collecting Events.

**Tutorial steps**

Let's get started!

To understand the difference between receiving events with default mapping and receiving them with custom mapping, this section is divided into the the following parts.
- Configure Siddhi application to receive HTTP events
- Add custom mapping

**Configure Siddhi application to receive HTTP events**

In this section, let's see how to add the required configurations to receive events via HTTP in JSON format.
1. Let's open the `SweetTotalApp` Siddhi application that you created in the previous tutorial. It currently looks as follows.

```siddhi
@App:name('SweetTotalApp')

define stream SweetProductionStream (name string, amount long);

@sink(type='log', prefix='Sweet Totals:')
define stream SweetTotalStream(name string, totalProduction long);

@info(name='SweetTotalQuery')
from SweetProductionStream
select name, sum(amount) as totalProduction
group by name
insert into SweetTotalStream;
```

2. To receive the events generated by the Sweet Bots via HTTP calls, you need an event source of the `http` type. Let's add it as shown below.

```siddhi
@source(type='http')
```

3. The HTTP events need to be received at a specific endpoint. Let's add the URL of this endpoint to the HTTP source configuration as follows:

```siddhi
@source(type='http', receiver.url='http://localhost:5005/SweetProductionEP')
```

4. The incoming messages generated by Sweet Bots are in the JSON format. You require a mapping configuration to transform them into a format that can be processed by WSO2 SP. Therefore, let's add an annotation for JSON mapping to the HTTP source as shown below.

```
For the complete list of mapping types supported for WSO2 SP, see Collecting Events - Event Format.
```

```siddhi
@source(type='http',
receiver.url='http://localhost:5005/SweetProductionEP', @map(type = 'json'))
```

Now the `SweetTotalApp` Siddhi application looks as follows.

```siddhi
@App:name('SweetTotalApp')

@source(type='http',
receiver.url='http://localhost:5005/SweetProductionEP', @map(type = 'json'))
define stream SweetProductionStream (name string, amount long);

@sink(type='log', prefix='Sweet Totals:')
define stream SweetTotalStream(name string, totalProduction long);

@info(name='SweetTotalQuery')
from SweetProductionStream
select name, sum(amount) as totalProduction
group by name
insert into SweetTotalStream;
```
5. Save the SweetTotalApp Siddhi application. Then click the following icon to start it so that it can process the events you send.

6. To see whether the Siddhi application functions as expected, let’s send two events with the following information.

   The input events generated by the Sweet Bots are in the JSON format. Therefore, the format is similar to the sample shown below.

   ```
   {
       "event": {
           "name": "Jaffa Cake",
           "amount": 10
       }
   }
   ``

   In order to send the two input events to WSO2 SP in the format shown above, issue the following two cURL commands.

   ```
curl -X POST \
   http://localhost:5005/SweetProductionEP \
   -H 'content-type: application/json' \
   -d '{
       "event": {
           "name": "Jaffa Cake",
           "amount": 10
       }
   }'

curl -X POST \
   http://localhost:5005/SweetProductionEP \
   -H 'content-type: application/json' \
   -d '{
       "event": {
           "name": "Jaffa Cake",
           "amount": 15
       }
   }'
   ```

   The two events sent contain the following information.

<table>
<thead>
<tr>
<th>Input Event No</th>
<th>Name</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jaffa Cake</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Jaffa Cake</td>
<td>15</td>
</tr>
</tbody>
</table>

   As a result, two output events are expected to be generated with the following information.

<table>
<thead>
<tr>
<th>Output Event No</th>
<th>Name</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jaffa Cake</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Jaffa Cake</td>
<td>25</td>
</tr>
</tbody>
</table>

   This generates the following output log in the CLI.
Add custom mapping

The previous step assumes that the message body of the events generated by Sweet Bots exactly match the schema of the SweetProduction Stream input stream that is defined in your Siddhi application. Now let’s consider a scenario where the attributes of the incoming JSON events are different to that of the input stream. Their format is as given in the sample below:

```json
{
    "sweet": "Jaffa Cake",
    "batch": {
        "batch id": "batch1",
        "count": 10
    }
}
```

From this event, you only need to extract the values for **sweet** and **count** (which is nested under **batch**) attributes. Therefore, let’s update the **map** annotation as follows:

1. To allow the Siddhi application to identify the elements that need to be extracted from the JSON message, add an **attributes** annotation to the mapping configuration as follows.

   ```yaml
   @map(type = 'json', @attributes(...))
   ```

2. The JSON mapper supports JsonPath. Therefore, the values for the attributes annotation can be JsonPath expressions. The JsonPath expressions for the name of the sweet and its total production are as follows.

<table>
<thead>
<tr>
<th>Stream Attribute Name</th>
<th>JSON Event Attribute Name</th>
<th>JsonPath Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>sweet</td>
<td>$.sweet</td>
</tr>
<tr>
<td>amount</td>
<td>count</td>
<td>$.batch.count</td>
</tr>
</tbody>
</table>

Based on the above JsonPath expressions you identified, you can update the **attributes** annotation as follows.

```yaml
@attributes(name = '$.sweet', amount = '$.batch.count')
```

Once the above substeps are completed, the mapping configuration looks as follows.

```yaml
@map(type = 'json', @attributes(name = '$.sweet', amount = '$.batch.count'))
```

The complete Siddhi application looks as follows.
@App:name('SweetTotalApp')

@source(type='http',
receiver.url='http://localhost:5005/SweetProductionEP', @map(type = 'json', @attributes(name = '$.sweet', amount = '$.batch.count')))
define stream SweetProductionStream (name string, amount long);

@sink(type='log', prefix='Sweet Totals: ')
define stream SweetTotalStream (name string, totalProduction long);

@info(name='SweetTotalQuery')
from SweetProductionStream
select name, sum(amount) as totalProduction
  group by name
insert into SweetTotalStream;

3. Let's save the changes you made to SweetTotalApp Siddhi application. Then click the following icon to start it so that it can process the events you send.

4. Now you can issue the following cURL commands to send the two incoming events as sent by the Sweet Bots.

```
curl -X POST \
  http://localhost:5005/SweetProductionEP \
  -H 'content-type: application/json' \ 
  -d '{
    "sweet": "Jaffa Cake",
    "batch": {
      "batch id": "batch1",
      "count": 10
    }
  }'
```

```
curl -X POST \
  http://localhost:5005/SweetProductionEP \
  -H 'content-type: application/json' \ 
  -d '{
    "sweet": "Jaffa Cake",
    "batch": {
      "batch id": "batch1",
      "count": 15
    }
  }'
```

As a result, you should see the same output in the CLI logs and you saw when you carried out step 6.
Pre-processing Streaming Data

- Introduction
- Tutorial steps

Introduction

In the previous tutorials, we learned how to create a simple Siddhi application and capture the events coming from external sources for analysis. This tutorial shows how to pre-process data before analyzing them.

Let's consider a scenario where the Sweet Factory foreman needs to identify events generated by Sweet Bots that match the following criteria:

- The sweet category produced is either eclair or gingerbread.
- The amount produced is greater than 10.

The results generated must be presented with the name of the sweet, the total value of the batch and the currency unit. Each eclair and gingerbread costs £0.4 to produce. In addition, the production overhead per batch is £2. The currency unit is GBP.

This tutorial covers the following concepts:
- Siddhi filters
- Transformations and defaults

Before you begin:
This tutorial reuses the source configuration that was first created and explained in Consuming Events. Therefore, it is recommended to try Tutorial 2 before following this tutorial.

Tutorial steps

Let's get started!

1. Start the WSO2 SP in the editor mode and login to the Stream Processor Studio. Then open a new Siddhi file.

2. In this tutorial, you can use the same input stream that you have been using in all the previous tutorials where the events receive include the sweet category name and the quantity produced. Let's add it as follows.

   ```
   define stream SweetProductionStream (name string, amount long);
   ```

3. Let's define the output stream based on what information you require as the output. As mentioned in the introduction, this scenario requires the output to include the name of the sweet, the value of the batch and the currency unit. Therefore, let's add an output stream named FilteredSweetStream with an attribute for each detail mentioned.

   ```
   define stream FilteredSweetStream (name string, value double, currency string);
   ```

4. To process the input data and insert the results into the FilteredSweetStream stream, let's add a query as follows.

   ```
   from SweetProductionStream select name, ((amount * 0.4) + 2) as value, "GBP" as currency insert into FilteredSweetStream;
   ```

   In this query, the amount produced of sweet category is multiplied by the production cost per unit, and 2 (representing the production overhead per batch of £2) is added to the result to derive the total production cost per batch. This is a Siddhi transformation where the value of an input attribute is changed while processing.

   Also note that the GBP static value is specified as the currency. This is how static values are added to data streams. Each event that goes through this preprocessing step contains this static value for the intended attribute.

5. The query added in the previous step outputs all the events after calculating the batch value. However, the output must consist of only events where the sweet category is either eclair or gingerbread, and the amount produced is greater than 10. To achieve this, let's add two filters as follows.

   a. Add a filter with the [ ] notation to specify the conditions based on which the events are filtered.

   ```
   [name == "eclair"], [name == "gingerbread"], [amount > 10]
   ```

   b. In the above step, you have specified the conditions to be met, but not how those conditions work together. You can use and, or, and not operators to specify how the conditions are related.
In this scenario, the sweet category can be eclair or gingerbread. Therefore, the `or` operator is used to denote the connection between the conditions `[name == "eclair"]` and `[name == "gingerbread"]` where it indicates that either condition can apply. The `and amount > 10` must apply together with either of the first two conditions. This is indicated via the `and` operator.

The query with the completed filter looks as follows.

```sql
from SweetProductionStream [(name == "eclair" or name == "gingerbread") and amount > 10]
select name, ((amount * 0.4) + 2) as value, "GBP" as currency
insert into FilteredSweetStream;
```

The completed Siddhi application (which you can name as `SweetProductionFilteringApp`) looks as follows.

```sql
@App:name('SweetProductionFilteringApp')

@source(type='http',
receiver.url='http://localhost:5005/SweetProductionEP', @map(type = 'json', @attributes(name = '$.sweet', amount = '$.batch.count'))) define stream SweetProductionStream (name string, amount long);

@sink(type='log', prefix='Conditionally filtered Sweets:') define stream FilteredSweetStream (name string, value double, currency string);

from SweetProductionStream [(name == "eclair" or name == "gingerbread") and amount > 10]
select name, ((amount * 0.4) + 2) as value, "GBP" as currency
insert into FilteredSweetStream;
```

6. To see how the output is generated by this Siddhi application, let's start the application in the editor and simulate five events by issuing the following cURL commands. Only the last two events sent match the filter conditions. Therefore, only those two events are returned with the pre-processed values.

```
curl -X POST \
http://localhost:5005/SweetProductionEP \
-H 'content-type: application/json' \
-d '{
  "sweet": "Jaffa Cake",
  "batch": {
    "batch id": "batch1",
    "count": 10
  }
}'
```
curl -X POST \
  http://localhost:5005/SweetProductionEP \
  -H 'content-type: application/json' \
  -d '{
    "sweet": "Jaffa Cake",
    "batch": {
      "batch id": "batch1",
      "count": 10
    }
  }'

curl -X POST \
  http://localhost:5005/SweetProductionEP \
  -H 'content-type: application/json' \
  -d '{
    "sweet": "gingerbread",
    "batch": {
      "batch id": "batch1",
      "count": 10
    }
  }'

curl -X POST \
  http://localhost:5005/SweetProductionEP \
  -H 'content-type: application/json' \
  -d '{
    "sweet": "eclair",
    "batch": {
      "batch id": "batch1",
      "count": 11
    }
  }'

curl -X POST \
  http://localhost:5005/SweetProductionEP \
  -H 'content-type: application/json' \
  -d '{
    "sweet": "gingerbread",
    "batch": {
      "batch id": "batch1",
      "count": 12
    }
  }'

The output is logged as follows:
Integrating Datastores

- Introduction
- Tutorial steps

Introduction

In the previous tutorials, you worked with events arriving in real time. Those tutorials only analyzed live events received over a specific period of time.

In this tutorial, let's understand how live data can work in conjunction with historic data as often required in real world scenarios.

Let's consider the scenario for purchasing raw materials for the Sweet Factory. Each time a consignment of raw materials is delivered, the supplier sends an event in JSON format to the WSO2 SP instance of the Sweet Factory. The manager wants the latest event from the supplier recorded for report generation.

This tutorial covers the following topics:
- Storing real-time data for later processing
- Correlating real-time data with historic data
- Manipulating static data based on real-time data

Before you begin:
- The user scenarios covered in this tutorial are supported by a set of extensions implemented for WSO2 SP called stores.

A store can be defined as any structure that acts as a data store for both data definition and data manipulation. Supported store implementations in Stream Processor 4.0 include the following:

- RDBMS store (which in turn supports the following)
  - H2
  - MySQL
  - Oracle Database
  - MS SQL Server
  - PostgreSQL
  - IBM DB2
- Apache HBase
- Apache Solr
- MongoDB

For more information on integrating data stores, please see the page on Storage Integration.

- In this scenario, the events generated when the supplier delivered raw materials are stored in a MySQL table named SweetFactorYouDB. You need to download and install MySQL and create this database before you try the tutorial steps.
  - Click here for instructions to configure the database.

1. Download and install MySQL Server.
2. Download the MySQL JDBC driver.
3. Unzip the downloaded MySQL driver zipped archive, and copy the MySQL JDBC driver JAR (mysql-connector-java-x.x.x-bin.jar) into the <SP_HOME>/lib directory.
4. Enter the following command in a terminal/command window, where username is the username you want to use to access the databases.

   ```bash
   mysql -u username -p
   ```

5. When prompted, specify the password you are using to access the databases with the username you specified.
6. Add the following configuration under the Data Sources Configuration section of the <SP_HOME>/conf/editor/deployment.yaml file.

   ```yaml
   You need to change the values for the username and password parameters to the username and password that you are using to access the MySQL database.
   ```
Let's get started!

1. Start the editor, and login to the WSO2 Stream Processor Studio. Then open a new Siddhi file to write a new Siddhi application. You can name it as ShipmentHistoryApp.

2. The information captured from the event sent by the supplier must include the name of the raw material, the name of the supplier and the amount of material purchased. Let's add an input stream definition as follows to capture this information.

```siddhi
define stream RawMaterialStream(name string, supplier string, amount double);
```

3. The incoming event from the supplier is in JSON format. Therefore, to convert it to the Siddhi format so that it can be processed by WSO2 Stream Processor, let's add a source configuration that includes a JSON mapping as follows.

```siddhi
@source(type = 'http', @map(type = 'json'))
define stream RawMaterialStream(name string, supplier string, amount double);
```

4. Let's define an output stream as follows.

```siddhi
define stream ShipmentDetailsStream(name string, supplier string, amount double);
```

5. To forward events from the input stream to the output stream, let's add a simple query as follows.
Now the Siddhi application looks as follows.

```siddhi
@App:name('ShipmentHistoryApp')
@source(type = 'http', @map(type = 'json'))
define stream RawMaterialStream(name string, supplier string, amount double);
define stream ShipmentDetailsStream(name string, supplier string, amount double);
from RawMaterialStream
    select name, amount
    insert into ShipmentDetailsStream;
```

The Stream Processor Studio indicates a syntax error at this stage because the `ShipmentDetailsStream` has an additional attribute other than the attributes included in the select clause. Ignore this error because the Siddhi application is still in an incomplete state. This error is corrected in the next steps.

6. The `ShipmentHistoryApp` in its current state can only forward all the events in the `RawMaterialStream` to the `ShipmentDetailsStream`. However, you need to save the incoming events in a store instead of passing them to a stream. Therefore, let's convert the output stream definition to a store definition by changing the `define stream` syntax to `define table`.

```siddhi
define table ShipmentDetails(name string, supplier string, amount double);
```

When the events are directed to a data store, they are not persisted, but they are stored within an in-memory table for later retrieval. The information stored in-memory is no longer available once the server is restarted.

7. Compared to a stream, a table supports a set of additional annotations that enables it to leverage on additional functionalities offered by data stores, such as unique keys and indexes.

A unique key (also referred to as primary key) is useful for uniquely identifying records. Fields set as unique keys are allowed to have duplicate values within the data store. Let's set a unique key using the `@PrimaryKey` annotation, as shown below.

```siddhi
@primaryKey('name')
define table ShipmentDetails(name string, supplier string, amount double);
```

8. The purchase records stored in the table need to be indexed by the supplier. To do this, you need to specify that the `supplier` attribute is an index attribute as shown below.

The `@index` annotation is used for specifying secondary indexes. This is useful for data retrieval scenarios. If the underlying data storage mechanism (e.g., RDBMS) supports secondary indexes, the fields specified here are indexed in the data store-level itself.

```siddhi
@primaryKey('name')
@index('supplier')
define table ShipmentDetails(name string, supplier string, amount double);
```

9. Let's specify a store of the RDBMS type to the `ShipmentDetails` table to bind RDBMS data storage mechanisms to it.

To do this, you need to use the `@store` annotation. Similar to the `@sink` and the `@source` annotations that you have already used in these tutorials, the properties used within the annotation vary depending on the type of the store.

```siddhi
@primaryKey('name')
@index('supplier')
@store(type='rdbms')
define table ShipmentDetails(name string, supplier string, amount double);
```
10. The RDBMS store has a set of properties that are required to be set in order to create the connection with the underlying DB instance. These include the JDBC URL of the database, the username, password and such. Let's add values to these properties so that the store definition becomes syntactically complete.

```java
@primaryKey('name')
@index('supplier')
@store(type='rdbms', jdbc.url="jdbc:mysql://localhost:3306/SweetFactoryDB", username="root", password="root", jdbc.driver.name="com.mysql.jdbc.Driver")
define table ShipmentDetails(name string, supplier string, amount double);
```

Now the updated Siddhi application is as follows:

```java
@App:name('ShipmentHistoryApp')
@source(type = 'http', @map(type = 'json'))
define stream RawMaterialStream(name string, supplier string, amount double);

@primaryKey('name')
@index('supplier')
@store(type='rdbms', jdbc.url="jdbc:mysql://localhost:3306/SweetFactoryDB", username="root", password="root", jdbc.driver.name="com.mysql.jdbc.Driver")
define table ShipmentDetails(name string, supplier string, amount double);

from RawMaterialStream
select name, supplier, amount
insert into ShipmentDetails;
```

11. The factory manager needs to see the latest shipments for each raw material. If a record for a particular raw material does not already exist in the database, it must be added as a new entry. If it already exists, its previous must be overwritten by the new incoming record.

This can be considered as an update or insert scenario. In Siddhi stores, the update or insert into directive can be used for this purpose in place of the normal insert into command. In order to carry out this operation, let's update the query as follows:

```java
from RawMaterialStream select name, supplier, amount update or insert into ShipmentDetails;
```

12. In the above query, the criteria based on which Siddhi can identify whether a record already exists in the database is not specified. In this scenario, we can specify that a record needs to be updated when the `name` attribute (which is the primary key of the table) matches any pre-existing record. This can be done by adding a condition to the query as shown below.

```java
from RawMaterialStream
select name, supplier, amount
insert into ShipmentDetails
on ShipmentDetailTables.name == name;
```

The completed query is as follows.
@App:name('ShipmentHistoryApp')

@source(type = 'http', @map(type = 'json'))
define stream RawMaterialStream(name string, supplier string, amount double);

@primaryKey('name')
@index('supplier')
@store(type='rdbms',
jdbc.url="jdbc:mysql://localhost:3306/SweetFactoryDB",
username="root", password="root",
jdbc.driver.name="com.mysql.jdbc.Driver")
define table ShipmentDetails(name string, supplier string, amount double);

from RawMaterialStream
select name, supplier, amount
update or insert into ShipmentDetails
on ShipmentDetails.name == name;

13. To store some data in the ShipmentDetails table, issue the following curl commands:

```bash
curl -X POST \
http://0.0.0.0:8280/ShipmentHistoryApp/RawMaterialStream \
-H 'content-type: application/json' \
-d '{
  "event": {
    "name": "Flour",
    "supplier": "Acme",
    "amount": 460.0
  }
}
'

curl -X POST \
http://0.0.0.0:8280/ShipmentHistoryApp/RawMaterialStream \
-H 'content-type: application/json' \
-d '{
  "event": {
    "name": "Sugar",
    "supplier": "Indigo6",
    "amount": 272.0
  }
}
'```
curl -X POST \
  http://0.0.0.0:8280/ShipmentHistoryApp/RawMaterialStream \
  -H 'content-type: application/json' \
  -d '{
    "event": {
      "name": "Honey",
      "supplier": "The BeeGees",
      "amount": 9.0
    }
  }'

curl -X POST \
  http://0.0.0.0:8280/ShipmentHistoryApp/RawMaterialStream \
  -H 'content-type: application/json' \
  -d '{
    "event": {
      "name": "Food Coloring",
      "supplier": "Wadjet Food Products",
      "amount": 30.0
    }
  }'

14. Once the events are sent, the event table is updated to be similar to the example shown below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Supplier</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flour</td>
<td>Acme</td>
<td>460</td>
</tr>
<tr>
<td>Sugar</td>
<td>Indigo6</td>
<td>272</td>
</tr>
<tr>
<td>Honey</td>
<td>The BeeGees</td>
<td>9</td>
</tr>
<tr>
<td>Food Coloring</td>
<td>Wadjet Food Products</td>
<td>30</td>
</tr>
<tr>
<td>Chocolate Chip</td>
<td>Larkspur Landing</td>
<td>34</td>
</tr>
</tbody>
</table>

The details in the store can be retrieved through the Store API as well as by deploying the Siddhi application in a worker node. To get the above details, issue the following cURL command.

In order to issue the following command, the server must be started in the worker node.
curl -X POST \
http://localhost:9090/stores/query \
-u admin:admin \
-H 'content-type: application/json' \
-d '{
"appName" : "ShipmentHistoryApp",
"query" : "from ShipmentDetails select *"
}'

The body of this request contains two parameters, which are **appName** and **query**.

- **appName**: This must be same as the name of the Siddhi application deployed in this scenario, it is `ShipmentHistoryApp`.
- **query**: Here, a valid `select` Siddhi query that includes the table name must be specified. In this scenario, the following query which includes the ShipmentDetails table created in this scenario is specified.

```
from ShipmentDetails select *
```

This retrieves all the data in the ShipmentDetails store. Further, Siddhi filters such as `having` can be applied in order to filter data.

### Summarizing Stream Data - Short Term

- **Introduction**
- **Tutorial steps**

#### Introduction

In the previous tutorials, you created a simple Siddhi application and understood how data arriving from outside sources can be captured and pre-processed by WSO2 SP. Further, you understood how to persist data in data stores to be used later.

In this tutorial, let's consider a more complex scenario which involves summarizing data in real time.

The foreman of the sweet factory requires the following information to understand the production capacity of the factory for each sweet category.

- The total sweet production for each sweet category for the last minute (at any given time).
- The highest amount of sweets produced during a production run needs to be identified for each 10 production runs.

For both expected results mentioned above, WSO2 SP needs to consider events that fall within a certain frame instead of considering all the events sent to a specific stream. WSO2 Siddhi supports this via the **window** concept.

A window allows you to capture a subset of events based on a specific criterion from an input stream to generate a result. The specific criterion can be time or length. Time windows capture events that occur during a specific time frame (e.g., within a minute), and a length windows capture events based on the number of events (e.g., every 10 events). Further, a window can be a sliding window (continuous window updates) or a batch/tumbling window (where window updates take place only when the specified time period has elapsed or the number of events have occurred).

#### Tutorial steps

This section covers the two scenarios mentioned above. Let's get started!

**Scenario 1 - Calculating the total sweet production for each sweet category for the last minute**
In this scenario, a Siddhi application is created to produce a time-based summarization.

1. Let's reuse the following input stream definition that you used in previous tutorials to capture data about the sweet production.

```java
define stream SweetProductionStream (name string, amount long);
```

To output the overall production during a minute per sweet category for the past minute, let's define an output stream as follows.

```java
define stream PastMinuteProductionStream (name string, pastMinuteTotal long);
```

2. To specify how the data must be derived from the `SweetProductionStream` input stream and inserted into the output stream, let's add a query as follows.

```java
from SweetProductionStream
select name, sum(amount) as pastMinuteTotal
group by name
insert into PastMinuteProductionStream;
```

This inserts the value for `name` into the `PastMinuteProductionStream` output stream with the same attribute name. The sum for the `amount` is calculated for all the events that have arrived, and inserted into the output stream as `pastMinuteTotal`. The output is grouped by the name of the sweet category.

3. The query given in the above step calculates the total produced for a sweet category based on all the events sent to the `SweetProductionStream` input stream. However, at any given time, you need to see only the total amount produced during the last minute. To achieve this, let's update the query as follows:
   a. To consider only events that are within a specific time frame, let's add a window as follows.

```java
from SweetProductionStream#window
select name, sum(amount) as pastMinuteTotal
group by name
insert into PastMinuteProductionStream;
```

   b. In this scenario, the subset of events to be captured by the window is based on time and the period of time considered is one minute. To specify this, update the window as follows.

```java
from SweetProductionStream#window.time(1 minute)
select name, sum(amount) as pastMinuteTotal
group by name
insert into PastMinuteProductionStream;
```

#window.time(1 minute) indicates that the window is a sliding window. This means that the window is of a fixed duration (i.e., 1 minute in this scenario), and it slides over incoming events to maintain this constant duration.

Once these changes are applied, the `SweetTotalsApp` Siddhi application looks as follows.
@App:name('PastMinuteSweetProductionApp')
@source(type='http',
receiver.url='http://localhost:5005/SweetProductionEP', @map(type = 'json', @attributes(name = '$.sweet', amount = '$.batch.count'))) define stream SweetProductionStream (name string, amount long);

@sink(type='log', prefix='Sweet totals over the past minute:') define stream PastMinuteProductionStream (name string, pastMinuteTotal long);

from SweetProductionStream#window.time(1 minute)
select name, sum(amount) as pastMinuteTotal
group by name
insert into PastMinuteProductionStream;

4. Let's try out this Siddhi application in the Stream Processor Studio. To do this, start and access the Stream Processor Studio. Then add the PastMinuteSweetProductionApp Siddhi application you created as a new file, and save it. Now you can start it by clicking the following icon for it while it is open.

5. To try out the SweetTotalsApp Siddhi application with the latest changes, let's send the following four cURL commands.

```bash
curl -X POST \  http://localhost:5005/SweetProductionEP \  -H 'content-type: application/json' \  -d '{  "sweet": "Toffee",  "batch": {    "batch id": "batch1",    "count": 11  }}'
curl -X POST \  http://localhost:5005/SweetProductionEP \  -H 'content-type: application/json' \  -d '{  "sweet": "Gateau",  "batch": {    "batch id": "batch1",    "count": 2  }}'
```
Scenario 2 - Identifying the highest amount of sweets produced during a production run

In this scenario, let's create a new Siddhi application named MaximumSweetProductionApp to capture the highest production reported for each sweet category during a production run, for 10 production runs.

1. The data arriving from the Sweet Bots is the same as in the previous scenario of this tutorial. Therefore, we can use the same input stream definition.

   ```siddhi
   define stream SweetProductionStream (name string, amount long);
   ```

2. The output should include the name of the sweet and the highest production total observed during the last 10 production runs. Therefore, let's define an output stream definition as follows.

   ```siddhi
   define stream DetectedMaximumProductionStream (name string, maximumValue long);
   ```

3. To calculate the highest production total observed in a production run, the `max()` Siddhi function can be used as follows.

   ```siddhi
   from SweetProductionStream select name, max(amount) as maximumValue group by name insert into DetectedMaximumProductionStream;
   ```

4. In this scenario, the output is derived based on events that fall within a fixed batch of 10 events. For this purpose, let's add a window as follows:

   a. Unlike the previous scenario, the window must be a length window and not a time window. Therefore, let's add a window and specify that it needs to be a length window as shown below. You also need specify the exact length of the length window (10 in
b. The above configuration has added a sliding length window of 10 production runs. However, the requirement of the foreman is to calculate the maximum once per 10 production runs. Therefore, let's convert the window you added to a batch window by adding `Batch` to the window configuration as shown below.

```sql
from SweetProductionStream#window.lengthBatch(10)
select name, max(amount) as maximumValue
group by name
insert into DetectedMaximumProductionStream;
```

The completed Siddhi application with source and sink mappings added should look as follows:

```sql
@App:name('MaximumSweetProductionApp')

@source(type='http',
receiver.url='http://localhost:5005/SweetProductionEP', @map(type = 'json', @attributes(name = '$.sweet', amount = '$.batch.count')))
define stream SweetProductionStream (name string, amount long);

@sink(type='log', prefix='Maximum detected production over 10 runs:')
define stream DetectedMaximumProductionStream (name string, maximumValue long);

from SweetProductionStream#window.lengthBatch(10)
select name, max(amount) as maximumValue
group by name
insert into DetectedMaximumProductionStream;
```

5. To test the `MaximumSweetProductionApp` Siddhi application, let's start the Siddhi application in the Stream Processor Studio and send 10 events by issuing 11 cURL commands as follows.

```bash
curl -X POST \
   http://localhost:5005/SweetProductionEP \
   -H 'content-type: application/json' \
   -d '{
   "sweet": "Jaffa Cake",
   "batch": {
   "batch id": "batch1",
   "count": 10
   }
}'
```
curl -X POST \
  http://localhost:5005/SweetProductionEP \
  -H 'content-type: application/json' \
  -d '{
    "sweet": "Jaffa Cake",
    "batch": {
      "batch id": "batch1",
      "count": 15
    }
  }' 

curl -X POST \
  http://localhost:5005/SweetProductionEP \
  -H 'content-type: application/json' \
  -d '{
    "sweet": "Jaffa Cake",
    "batch": {
      "batch id": "batch1",
      "count": 11
    }
  }'

curl -X POST \
  http://localhost:5005/SweetProductionEP \
  -H 'content-type: application/json' \
  -d '{
    "sweet": "Jaffa Cake",
    "batch": {
      "batch id": "batch1",
      "count": 12
    }
  }'

curl -X POST \
  http://localhost:5005/SweetProductionEP \
  -H 'content-type: application/json' \
  -d '{
    "sweet": "Jaffa Cake",
    "batch": {
      "batch id": "batch1",
      "count": 11
    }
  }'
curl -X POST \
   http://localhost:5005/SweetProductionEP \
   -H 'content-type: application/json' \
   -d '{
   "sweet": "Jaffa Cake",
   "batch": {
   "batch id": "batch1",
   "count": 13
   }
}'}

curl -X POST \
   http://localhost:5005/SweetProductionEP \
   -H 'content-type: application/json' \
   -d '{
   "sweet": "Jaffa Cake",
   "batch": {
   "batch id": "batch1",
   "count": 16
   }
}'}

curl -X POST \
   http://localhost:5005/SweetProductionEP \
   -H 'content-type: application/json' \
   -d '{
   "sweet": "Jaffa Cake",
   "batch": {
   "batch id": "batch1",
   "count": 15
   }
}'}

curl -X POST \
   http://localhost:5005/SweetProductionEP \
   -H 'content-type: application/json' \
   -d '{
   "sweet": "Jaffa Cake",
   "batch": {
   "batch id": "batch1",
   "count": 11
   }
}'}
curl -X POST \
  http://localhost:5005/SweetProductionEP \
  -H 'content-type: application/json' \
  -d '{
    "sweet": "Jaffa Cake",
    "batch": {
      "batch id": "batch1",
      "count": 17
    }
  }'

This generates the following log in the console.

Note that in the last event representing the last production run, the total production was 17, but the maximum detected total production output is 16. This is because you have used a batch window, and the 11th event does not belong to the fixed batch of 10 events.

Streaming Data Summarization (Incremental Aggregation)

- Introduction
- Tutorial steps
- User Scenario 2: Retrieval of data on demand

Introduction

In the previous tutorial, you looked at the Siddhi real time data summarization capabilities by calculating the total production in the past minute. Now let’s consider a more advanced scenario where you need to calculate the total value for a specific time period.

In this scenario, the foreman of the Sweet Factory needs to know the total production of Sherbet Lemon during each hour in November 2017. It is costly to do this by recalculating the total for each and every event. What you need is a time based aggregation of the events in real time and retrieval on demand. Siddhi supports this functionality through the Incremental Aggregation concept.

Incremental Aggregation calculates the aggregated values continuously and stores them. These values can be retrieved efficiently from the store on demand. Furthermore, Incremental Aggregators support out of order event arrival with in-memory buffers for higher accuracy.

This tutorial covers the following concepts:
- Introduction to incremental aggregation
- Retrieval from incremental aggregation

Before you begin:

In this scenario, information sent by the Sweet Bots are stored in a MySQL table named SweetFactoryDB. You need to download and install MySQL, and create this table before you carry out the tutorial steps.

Click here for instructions to configure the database table.

1. Download and install MySQL Server.
2. Download the MySQL JDBC driver.
3. Unzip the downloaded MySQL driver zipped archive, and copy the MySQL JDBC driver JAR (mysql-connector-java-x.x.x-bin.jar) into the <SP_HOME>/lib directory.
4. Enter the following command in a terminal/command window, where username is the username you want to use to access the databases.
   ```bash
   mysql -u username -p
   ```
5. When prompted, specify the password you are using to access the databases with the username you specified.
6. Add the following configuration under the Data Sources Configuration section of the <SP_HOME>/conf/editor/deployment.yaml file.

You need to change the values for the username and password parameters to the username and password.
Tutorial steps

Let's get started!

User Scenario 1: Defining incremental aggregation

In this scenario, let's define an incremental aggregation to calculate the total production in an incremental manner, and store the results.

1. Let's define an input stream as follows based on the data received from Sweet Bots. This is the same stream definition used in the previous tutorials to capture the name of the sweet category and the amount produced.

```sql
define stream SweetProductionStream (name string, amount long);
```

2. Now, let's define an aggregation for the input data. Here, you can assume that the foreman would like to know the production per hour, month and year for each sweet.

```sql
define aggregation SweetProductionAggregation
from SweetProductionStream
select name, sum(amount) as totalAmount
group by name
aggregate every hour...year
```

To create a database table named `SweetFactoryDB`, issue the following commands from the terminal.

```
mysql> create database SweetFactoryDB;
mysql> use SweetFactoryDB;
mysql> source <SP_HOME>/wso2/editor/dbscripts/metrics/mysql.sql;
mysql> grant all on SweetFactoryDB.* TO username@localhost identified by "password";
```
This calculates the total amount per hour, day, month and year by the arrival time of each event. Incremental Aggregation can also be done for seconds, minutes, hours, days, months and years. However, in this sweet production scenario, aggregating by second holds no information value. Therefore, the sweet production is aggregated from hour to year.

3. Now, comes the question of when the production occurs. In the above aggregation, event arrival time is the time used in aggregation. The Sweet Bots send information directly from the factory floor to the server in the same network. Therefore, we can assume that the event arrival time is the production time.

If you want you can be more accurate by appending the data sent by the SweetBots to include time as shown below.

a. First define the input stream to include a timestamp:

```java
define stream SweetProductionStream (name string, amount long, timestamp long);
```

b. Then use the timestamp for aggregation as shown below.

```java
aggregate by SweetProductionStream.timestamp every hour...year
```

For this tutorial, let's continue to use the format mentioned first instead of the format in these substeps because the time differences are very slight in the hourly calculations.

```java
from SweetProductionStream
select name, sum(amount) as totalAmount

group by name
```

This part of the aggregation specifies the following:

- From where the information to be processed is taken (i.e., SweetProductionStream input stream)
- The value you are aggregating. In this scenario, sum(amount) as totalAmount aggregates only the summation of values. The aggregation can also be minimum, maximum or average.
- The group by clause is optional and can be ignored if all production must be aggregated.

The completed Siddhi application looks as follows.

```java
define stream SweetProductionStream(name string, amount long);

@store(......)
define aggregation SweetProductionAggregation
from SweetProductionStream
select name, sum(amount) as totalAmount

group by name
aggregate every min...year
```

User Scenario 2: Retrieval of data on demand

In the previous scenario, you defined the aggregation. Now let's see how to retrieve from it. Siddhi supports this functionality through correlation of data. In this tutorial, you are retrieving data via aggregation joins. For more information on correlating data through joins see Siddhi Query Guide - Joins.

1. First, let's define a stream to retrieve data. The foreman needs to see the hourly production of Sherbet Lemon for November 2016. Therefore, the criteria to retrieve values are as follows.
Therefore, the input stream needs to be defined as follows:

```java
define stream GetTotalSweetProductionStream (name string, start_duration string, end_duration string, interval string);
```

2. A possible output of this retrieval is the timestamp (beginning of each hour), the name of the sweet and the total amount. Therefore, let's define an output stream with these values as follows.

```java
define stream HourlyProductionStream(AGG_TIMESTAMP long, name string, totalAmount long);
```

In the above definition, AGG_TIMESTAMP is the internal reference of the aggregation defining the start of the time interval.

3. Now, let's use the aggregation, retrieval stream, and the output stream to define data correlation from an aggregation. Aggregation for the selected period contains aggregation for all sweets. Therefore, let's join the aggregation, and the retrieval stream based on the sweet name to filter aggregations for Sherbet Lemon.

```java
from GetTotalSweetProductionStream as b join SweetProductionAggregation as a
on a.name == b.name
```

4. You need to retrieve data relevant only for November 2017. Therefore, let's add it in the retrieval stream as the duration.

```java
from GetTotalSweetProductionStream as b join SweetProductionAggregation as a
on a.name == b.name
within b.duration
```

In the output event, the duration for which the data is retrieved must be represented in a specific format. For example, November 2017 can be represented as 2017-11-** **:***:***. The supported date formats are <yyyy>-<MM>-<dd> (if time is in GMT) and <yyyy>-<MM>-<dd> <HH>:<mm>:<ss> <Z> (if the time is not in GMT), here the ISO 8601 UTC offset must be provided for <Z> (e.g., +05:30, -11:00).

If the user needs a specific time duration, the query must be changed as follows. Both durations specified must adhere to the data formats required by Siddhi.

```java
from GetSweetProductionStream as b join SweetProductionAggregation as a
on a.name == b.name
within b.start_duration, b.end_duration
```

5. Let's add interval for the retrieval to specify for which intervals you want the data to be retrieved.

```java
from GetTotalSweetProductionStream as b join SweetProductionAggregation as a
on a.name == b.name
within b.duration
per b.interval
```

Interval can be in the format of SECONDS, MINUTES, HOURS, DAYS, MONTHS or YEARS (these values are not case sensitive).

The completed statement including the output stream looks as follows:

```java
from GetTotalSweetProductionStream as b join SweetProductionAggregation as a
on a.name == b.name
within b.duration
per b.interval
select a.AGG_TIMESTAMP, a.name, a.totalAmount
insert into HourlyProductionStream;
```

In the above definition, a.AGG_TIMESTAMP is the internal data of the aggregation defining the start of the time interval. For instance, in
1. Analyzing KPIs

- Introduction
- Tutorial steps

Introduction

A Key Performance Indicator (KPI) is a quantifiable metric that can be used to evaluate the success or the failure of the object being measured. WSO2 SP with its real-time stream processing engine allows you perform KPI Analysis. In this tutorial, let's look at how to create a Siddhi application that performs KPI analysis.

In this scenario, the foreman wants the production per hour of each sweet to not fall below 5000 units in order to meet the customer demand. At the same time, he also needs to know if the production per hour of a category is over 10000 in order to identify surpluses. Let's see how to write a query to derive this information by analyzing data sent in JSON format by the Sweet Bots on the factory floor.

This tutorial covers the following concepts:
- Grouping and filtering
- Siddhi functions
Tutorial steps

In this tutorial, there are two scenarios as follows:

- Detecting sweets of which the production per hour falls below 5000
- Detecting sweets of which the production per hour rises above 10000

**Scenario 1: Detecting sweets of which the production per hour falls below 5000**

Let's get started!

1. Start the WSO2 SP in the editor mode and login to the Stream Processor Studio. Then open a new Siddhi application.
2. Let's revisit the following configurations that you created in Tutorial 5.
   
   ```
   define stream SweetProductionStream (name string, amount long);
   define stream PastHourProductionStream (name string, pastHourTotal long);
   from SweetProductionStream#window.time(1 hour)
   select name, sum(amount) as pastHourTotal
   group by name
   insert into LowProductionStream;
   ```

   Let's add these configurations to your new Siddhi application in order to edit them in a way that addresses the requirements of this scenario.

   For this tutorial, the output stream name which was previously `PastHourProductionStream` is changed to `LowProductionStream`.

3. Only events with a per-hour production that is less than 5000 need to be sent to the `LowProductionStream` output stream. Therefore, add the having clause to the query as follows.

   ```
   from SweetProductionStream#window.time(1 hour) select name, sum(amount) as hourlyTotal
   group by name
   insert into LowProductionStream;
   having hourlyTotal < 5000
   ```

   The `having hourlyTotal < 5000` clause is added just after the `select` statement so that all the subsequent operations are applied to the filtered result.

4. At present, the foreman who is receiving the output of this query cannot identify the hour to which output event applies. Therefore, you need to add an additional attribute named `currentHour` to the output stream definition as shown below.

   ```
   define stream LowProductionStream (name string, hourlyTotal long, currentHour int);
   ```

5. The exact hour to which the reported production per hour applies is not sent in the incoming events generated by the Sweet Bots. This needs to be inferred by the Siddhi application. To infer the exact hour during which the specific sweet production total was calculated, let's add the `currentTimeMillis()` time function in Siddhi as follows:
6. To specify HOUR as the unit of time to be extracted for the purpose of reporting the production totals calculated, add the `time:extract()` expression as shown below.

```
select name, sum(amount) as hourlyTotal, time:extract(currentTimeMillis(), 'HOUR') as currentHour
```

7. The working hours of the sweet factory are between 9.00AM and 5.00PM. However, the Siddhi application runs throughout the day, and (as at now) calculates the sweet production totals even for the hours during which the Sweet Bots are inactive. To avoid reporting during non-working hours where the production was below the target, you need to add a filter that excludes non-working hours as follows:

```
from SweetProductionStream#window.time(1 hour) select name, sum(amount) as hourlyTotal, time:extract(currentTimeMillis(), 'HOUR') as currentHour group by name having hourlyTotal < 5000 and currentHour > 9 and currentHour < 17 insert into LowProductionStream;
```

Apply the same source and sink configurations that you created in Tutorial 2 to complete the Siddhi application. The completed Siddhi application looks as follows.

```
@App:name('SweetProductionKPIApp')
@source(type='http', receiver.url='http://localhost:5005/SweetProductionEP', @map(type = 'json', @attributes(name = '$.sweet', amount = '$.batch.count')))
define stream SweetProductionStream (name string, amount long);

@sink(type='log', prefix='Low production alert:')
define stream LowProductionStream (name string, hourlyTotal long, currentHour int);

from SweetProductionStream#window.time(1 hour)
select name, sum(amount) as hourlyTotal, time:extract(currentTimeMillis(), 'HOUR') as currentHour group by name
having hourlyTotal < 5000 and currentHour > 9 and currentHour < 17
insert into LowProductionStream;
```

8. To check whether this Siddhi application works as expected, send four cURL commands as follows.

```
curl -X POST \\http://localhost:5005/SweetProductionEP \\-H 'content-type: application/json' \\
-d '{
"sweet": "Toffee",
"batch": {
  "batch id": "batch1",
  "count": 6000
}
}'
```
curl -X POST \
http://localhost:5005/SweetProductionEP \
-H 'content-type: application/json' \
-d '{
    "sweet": "Gateau",
    "batch": {
        "batch id": "batch1",
        "count": 6200
    }
}''

curl -X POST \
http://localhost:5005/SweetProductionEP \
-H 'content-type: application/json' \
-d '{
    "sweet": "Gingerbread",
    "batch": {
        "batch id": "batch1",
        "count": 4500
    }
}''

curl -X POST \
http://localhost:5005/SweetProductionEP \
-H 'content-type: application/json' \
-d '{
    "sweet": "Gateau",
    "batch": {
        "batch id": "batch1",
        "count": 5800
    }
}''

The production per hour in the third event is less than 5000. This is logged in the output logs as follows.

Low production alert: : Event(timestamp=1527238932317, data=[Gingerbread, 4500, 13], isExpired=false)

Scenario 2 Detecting sweets of which the production per hour rises above 10000

In this scenario, events of which the per-hour production of a sweet exceeds 10000 must be marked with Possible Surplus so that the foreman identifies any surpluses that occur.

Let's get started!

1. In this scenario, the foreman needs to see the name of the sweet, the total produced during an hour and the hour to which the information applies in order to identify surpluses. Therefore, let's add an output stream definition with attributes to capture this information as follows.

```sql
define stream ProductionStatusStream (name string, hourlyTotal long, currentHour int, status string);
```

2. In this scenario, you are receiving information from the same SweetProductionStream input stream used in the previous scenario and in previous tutorials. Let's also add the query to pick values from the SweetProductionStream input stream and add it to the Product
2. From SweetProductionStream#window.time(1 hour) select name, sum(amount) as hourlyTotal, time:extract(currentTimeMillis(), 'HOUR') as currentHour, "Possible surplus" group by name having hourlyTotal > 10000 and currentHour > 9 and currentHour < 17 insert into ProductionStatusStream;

Here, you are using the having clause as you did in the previous scenario. In this query, the events filtered are ones that are generated later that the 9.00 AM and earlier than 5.00PM with a production during an hour that exceeds 10000.

3. The query added in the previous step only sends the filtered events to the ProductionStatusStream output stream. However, the requirement in this scenario is to send all the events to the ProductionStatusStream output stream while events that match the filtering criteria are marked with Possible Surplus. To achieve this, you can use the ifThenElse() function that is configured as follows.

```sql
ifThenElse(<condition>, <valueIfTrue>, <valueIfFalse>)
```

Let's add a condition to the query using this function as shown below.

```sql
from SweetProductionStream#window.time(1 hour) select name, sum(amount) as hourlyTotal, time:extract(currentTimeMillis(), 'HOUR') as currentHour, ifThenElse(sum(amount) > 10000, "Possible surplus", "OK") as status group by name having currentHour > 9 and currentHour < 17 insert into ProductionStatusStream;
```

Here, if the production within the current hour is more than 10000, Possible Surplus is included in the output event as the value for the status attribute.

The completed Siddhi application looks as follows.

```java
@App:name('SweetProductionKPIApp')
@source(type='http', receiver.url='http://localhost:5005/SweetProductionEP', @map(type = 'json', @attributes(name = '$.sweet', amount = '$.batch.count'))) define stream SweetProductionStream (name string, amount long);

@sink(type='log', prefix='Low production alert:') define stream LowProductionStream (name string, hourlyTotal long, currentHour int);

@sink(type='log', prefix='Sweet production status:') define stream ProductionStatusStream (name string, hourlyTotal long, currentHour int, status string);

from SweetProductionStream#window.time(1 hour) select name, sum(amount) as hourlyTotal, time:extract(currentTimeMillis(), 'HOUR') as currentHour, ifThenElse(sum(amount) > 10000, "Possible surplus", "OK") as status group by name having currentHour > 9 and currentHour < 17 insert into ProductionStatusStream;
```

4. To check whether this Siddhi application works as expected, you can send the same four cURL commands you issued in the previous scenario after you save the Siddhi application. The following output is logged.

Publishing Processed Events
Introduction

In the previous tutorials, you covered how to capture events with WSO2 Stream Processor and how these events are preprocessed. In all the previous tutorials, you published the output as logs in the console.

In this tutorial, let's look at how events can be published via WSO2 Stream Processor to external endpoints. Similar to receiving events, WSO2 SP also supports many functionalities for publishing events, allowing you to publish events using different transports and in different formats.

In the previous tutorial, you wrote a query to identify the hours during which the per hour production was less than 5000. Let's consider a scenario where the factory foreman needs to send an email to the factory manager when there is such a shortage in the production of a sweet.

This tutorial covers the following concepts:
- Publishing events with Siddhi sinks
- Custom mappings for sinks

Before you begin:
- Try Tutorial 5: Summarizing Stream Data - Short Term.
- To understand sink configurations and the different types of sinks supported in Siddhi, see Publishing Events.

Tutorial steps

Let's get started!

1. Start the editor, and log in to the WSO2 Stream Processor Studio. Then open the PastHourSweetProductionApp Siddhi application that you created in Tutorial 5. It already has the following queries to identify the hours during which the total production of a sweet has been less than 5000.

```siddhi
define stream SweetProductionStream (name string, amount long);

define stream LowProductionStream (name string, hourlyTotal long, currentHour long);

from SweetProductionStream#window.time(1 hour)
select name, sum(amount) as hourTotal, time:extract(currentTimeMillis(), 'HOUR') as currentHour group by name having hourlyTotal < 5000 and currentHour > 9 and currentHour < 17 insert into LowProductionStream;
```

2. To send an email when the production of a sweet falls below 5000 during a hour, you need to configure an email sink. Let's add it to the Siddhi application as follows.

```siddhi
@sink(type='email');
define stream LowProductionStream (name string, hourlyTotal long, currentHour long)
```

This sink configuration is added above the LowProductionStream output stream definition to indicate that it is connected to this stream. The events sent via this sink are taken from the LowProductionStream output stream.

3. For the email to reach the factory manager, the sweet factory foreman needs to send it from the factory's email address which is factor012@sweets-r-us.com, to the factory manager's email address which is bossman@sweets-r-us.com. Taking this into account, let's update the email sink as follows:

```siddhi
@sink(type='email',
  address='factory012@sweets-r-us.com', username='factory012',
  password='secret_password', subject='Low production alert',
  to='bossman@sweets-r-us.com')
define stream LowProductionStream (name string, hourlyTotal long, currentHour long);
```

This configuration includes the following information.
- **username**: The email username of the sender. (Note: this should be without the @ and the latter part of the address.
  eg: @gmail.com)
- **address**: The email address of the sender.
- **password**: The email password of the sender.
- **subject**: The email password.
- **to**: The recipient's email address.

4. At present, the sink configuration does not have a mapping configuration (similar to what you have already defined for the HTTP event source). Without the mapping, this configuration only allows you to publish events in the same format in which they are sent to WSO2 SP to be processed. This format is not readable and the email sinks do not support that format. Therefore, let's add a mapping configuration by following these sub steps.
   - The published event must be in a format that can be read by humans. Therefore, you can add a basic mapping of the text type.
b. To transform the output event into a readable text message, the `@payload` annotation needs to be added in the following format.

```java
@map(type = 'text', @payload("Some text body"))
```

Within the text payload, the stream variables can be specified via the `{{ }}` placeholder so that the runtime values are dynamically assigned. For example, if the stream has an attribute named `attr1`, a payload body that refers to this attribute looks as follows.

```java
@payload("The value is {{attr1}}")
```

When the total production per hour falls below 5000, the following information in the output event varies

- The name of the sweet category
- The exact amount produced during the hour
- The specific hour during which the total production was less than 5000

Let's compose the body of the email to be sent to the factory manager via the `@payload` annotation with a placeholder for each detail mentioned above.

```java
@payload("Hello,

The production of {{name}} has fallen to below 5000 units, to {{hourlyTotal}} in the past hour (hour {{currentHour}} of the day).

This message was generated automatically.
")
```

Note that `{{name}}, {{hourlyTotal}} and {{currentHour}}` are placeholders for the attributes in the `LowProductionStream` stream, to which the sink is bound.

Now, let's look at the completed output stream definition with the sink, and with the newlines represented by the `\n` newline character.

```java
@sink(type='email', address='factory012@sweets-r-us.com', username='factory012', password='secret_password', subject='Low production alert', to='bossman@sweets-r-us.com', @map(type = 'text', @payload("Hello,\n\nThe production of {{name}} has fallen to below 5000 units, to {{hourlyTotal}} in the past hour (hour {{currentHour}} of the day).\n\nThis message was generated automatically."))
```

define stream LowProductionStream (name string, hourlyTotal long, currentHour long)

The completed Siddhi application looks as follows:
Correlating Simple Events

Introduction

In the previous tutorials, you covered basic Siddhi functionality including ingesting data, preprocessing, data store integration and publishing events. In all of these tutorials, you have considered events arriving from a single source, (either data sent by the SweetBots on the factory floor or data from the suppliers of raw material).

In a real world user scenario, data from various sources need to be analyzed in tandem. This can be anything from multiple disparate entities as well as pre-received data stored elsewhere. Hence, correlating events from multiple sources needs to be studied. Siddhi offers this functionality through stream and store joins. A join can be used to match events arriving from different sources based on given criteria and then process them together.

To understand this concept, let's consider a scenario in the sweet factory where events are sent to denote the following:

- The raw materials arriving at the factory (name of the material and the amount)
- The raw materials consumed per production run by all SweetBots (name and amount)

The factory needs to ensure that the production of flour and sugar does not consume more than 95% of the supply in the past hour for sustainability reasons. Any material that exceeds this limit should be recorded.

Before you begin:
Try Integrating Datastores.

Tutorial steps

Let's get started!

1. To create your Siddhi application, let's start the editor, and log in to the WSO2 Stream Processor Studio. Then open a new Siddhi file to write a new Siddhi application.
2. Now let's define two input streams as follows.

```siddhi
define stream MaterialConsumptionStream(name string, user string, amount double);
```

```siddhi
define stream MaterialSupplyStream(name string, supplier string, amount double);
```

3. The results output must include the name of the raw material, amounts of the supplied and produced material, and the name of the raw material user and supplier if the threshold is reached. Let's define an output stream based on these details:

```siddhi
define stream MaterialThresholdAlertStream(name string, supplyAmount double, consumptionAmount double, user string, supplier string);
```

4. To select events from the `MaterialConsumptionStream` for analysis, write a query as follows.

```siddhi
from MaterialConsumptionStream
select name, amount, user
```

5. Let's define an alias for the `MaterialConsumptionStream` so that it can be identified more easily once you have used the join statement.

```siddhi
from MaterialConsumptionStream as c select c.name, c.amount, c.user
```

The same needs to be done for the `MaterialSupplyStream`.

```siddhi
from MaterialSupplyStream as s select s.name, s.amount, s.supplier
```

6. Now that we have both input streams ready, let's join both inputs together using the `join` keyword.

```siddhi
from MaterialConsumptionStream as c
join MaterialSupplyStream as s
```

7. To correlate the two streams, you need a common attribute in both of them. In this scenario, both streams have the name of the material produced. Therefore, `name` can be used as the attribute as shown below.

```siddhi
from MaterialConsumptionStream as c
join MaterialSupplyStream as s
on c.name == s.name
```

8. The time period to be considered is one hour, a time window of one hour should be added to both streams as shown below.

```siddhi
from MaterialConsumptionStream#window.time(1 hour) as c
join MaterialSupplyStream#window.time(1 hour) as s
on c.name == s.name
```

9. To ensure that the consumption does not exceed 95% of the supply, you can add a query as follows.

```siddhi
s.amount * 0.95 < c.amount
```

The complete statement in a usable format looks as follows.

```siddhi
from MaterialConsumptionStream#window.time(1 hour) as c
join MaterialSupplyStream#window.time(1 hour) as s
on c.name == s.name
select s.name, s.amount as supplyAmount, c.amount as consumptionAmount, user, supplier
group by s.name
having s.amount * 0.95 < c.amount
insert into MaterialThresholdAlertStream
```

The completed Siddhi application with source and sink configurations added looks as follows.
@App:name('MaterialThresholdAlertApp')

@source(type = 'http', @map(type = 'json'))
define stream MaterialConsumptionStream(name string, user string, amount double);

@source(type = 'http', @map(type = 'json'))
define stream MaterialSupplyStream(name string, supplier string, amount double);

@sink(type='log', prefix='Materials that go beyond sustainability threshold:')
define stream MaterialThresholdAlertStream(name string, supplyAmount double, consumptionAmount double, user string, supplier string);

from MaterialConsumptionStream#window.time(1 hour) as c
join MaterialSupplyStream#window.time(1 hour) as s
on c.name == s.name
select s.name, s.amount as supplyAmount, c.amount as consumptionAmount, user, supplier
group by s.name
having s.amount * 0.95 < c.amount
insert into MaterialThresholdAlertStream;

Now, if you want to correlate the events not just from streams but also with previously stored data, the store that was used for storing the events can be used in place of either stream (but not both).

define table ShipmentDetailsTable(name string, supplier string, amount double);
define stream MaterialConsumptionStream(name string, user string, amount double);

from MaterialConsumptionStream#window.time(1 hour) as c
join ShipmentDetailsTable as s
on c.name == s.name
select ...

The following is the completed Siddhi application. It contains the same query as above that uses historical data from the ShipmentDetailsTable.
Correlating Events for Complex Event Processing

- Introduction
- Tutorial steps

Introduction

In the previous tutorial, you looked at correlating events from multiple sources in a simple manner using joins. You used join statements between streams and stores and observed how aliases can be used on streams for easy reference.

In this tutorial, let's see how event correlation can be applied to Complex Event Processing (CEP) scenarios.

Let's consider the following two scenarios in the sweet factory.

- **Scenario 1**: The factory foreman needs to monitor the supply of materials and send an alert if there is a decrease of 10 units within 10 minutes.
- **Scenario 2**: The factory foreman needs to monitor both supply and production, and send an alert if production does not start within 15 minutes after the raw materials are received.

This tutorial covers the following concepts:

- Simple patterns
- Logical patterns for applying logical operations to events arriving in a temporal order

WSO2 Siddhi also supports another type of patterns known as Counting Patterns. For more information, see the Siddhi Query Guide - Counting Pattern.
Simple Siddhi patterns can be identified by the following syntax.

```
from (every) x -> y
within t
select ...
```

This means that for events in the x stream are followed by events from the y stream within a time gap of t, the subsequent operations are carried out.

Let's consider the following real world example to understand this.

```
from every (el=MyStream) -> e2=MyStream[el.val1 <= e2.val1]
within 1 hour
select el.val1, e2.val2
insert into OutStream
```

In the above example, each event that arrives at the MyStream stream is checked against subsequent events arriving at the same stream (because the every keyword is used) and checked against new events matching the given filter condition. This is done continuously for a time period of one hour. If any matching events are found, then they are sent to the OutStream output stream based on the select clause.

If the every keyword is not used, then the calculation is invoked only once, when the first event arrives at the MyStream stream.

### Tutorial steps

This covers the steps for the following two user scenarios.

- User Scenario 1: Detecting a decrease in supply within a set time period
- User Scenario 2: Detecting production delays after supplies are received

### User Scenario 1: Detecting a decrease in supply within a set time period

Let's get started!

1. Let's define an input stream for the raw material supply. This can be the same input stream you defined in Correlating Simple Events.

   ```
   define stream MaterialSupplyStream(name string, supplier string, amount double);
   ```

2. Now let's define an output stream as follows.

   ```
   define stream MaterialSupplyAlertStream(name string, supplier string, amount double);
   ```

3. Let's add a query to perform a simple insertion from the input stream to the output stream.

   ```
   from MaterialSupplyStream
   select name, supplier, amount
   insert into MaterialSupplyAlertStream;
   ```

4. To define a pattern to this stream so that events within a 10-minute window are considered, let's update the query as follows.
from every (e1=MaterialSupplyStream) -> e2=MaterialSupplyStream
within 10 min
select e1.name, e1.amount as originalAmount, e2.amount as laterAmount,
e1.supplier
insert into MaterialSupplyAlertStream;

5. The foreman requires each event be checked against the subsequent events to see if there's a decrease in the amount by 10 units. The Siddhi filter for this is as follows.

[e1.name == e2.name and e1.amount - e2.amount > 10]

Let's add it to the query as shown below.

from every (e1=MaterialSupplyStream) -> e2=MaterialSupplyStream[e1.name == e2.name and
 e1.amount - e2.amount > 10] within 10 min select e1.name, e1.supplier, e1.amount insert
into MaterialSupplyAlertStream;

6. To generate a more complete output, you need to add the following information to it.
   • The name of the raw material
   • The earlier amount
   • The later amount (which is less than the earlier detected amount by 10 units or more)
   • The supplier whose supply has fallen below the threshold

Let's update the output stream definition as follows to include this information.

define stream MaterialSupplyAlertStream(name string, originalAmount
double, laterAmount double, supplier string);

The completed Siddhi application is as follows.

@App:name('MaterialDecreaseDetectionApp')
@source(type = 'http', @map(type = 'json'))
define stream MaterialSupplyStream(name string, supplier string, amount
double);
@sink(type='log', prefix='Decrease in supply detected:')
define stream MaterialSupplyAlertStream(name string, originalAmount double,
laterAmount double, supplier string);

from every (e1=MaterialSupplyStream) -> e2=MaterialSupplyStream[e1.name ==
e2.name and e1.amount - e2.amount > 10]
within 10 min
select e1.name, e1.amount as originalAmount, e2.amount as laterAmount,
e1.supplier
insert into MaterialSupplyAlertStream;

User Scenario 2: Detecting production delays after supplies are received

For this user scenario, let's consider a situation where events from 2 different streams contain data. Here, you need to identify delays in production, which means you have to check for events not arriving at a particular stream.

1. Let's begin by defining input streams for both the raw material supply and the sweet production.
1. ```
define stream MaterialConsumptionStream(name string, user string, amount double);
```

```
define stream MaterialSupplyStream(name string, supplier string, amount double);
```

2. The information to be output is as follows.
   - Name of the raw material
   - Production amount

   To generate this output, let's define an output stream as follows.

   ```
define stream ProductionDelayAlertStream(name string, amount double);
```

3. To correlate events from the consumption and supply streams, let's define a simple pattern as follows.

   ```
from every (e1=MaterialSupplyStream) -> e2=MaterialConsumptionStream
within 10 min
select e1.name, e1.amount
insert into MaterialSupplyAlertStream;
```

4. The above query you added does not currently contain a condition based on which the correlation is done. The condition to be added needs to consider events that have not arrived at the MaterialSupplyStream input stream instead of the new events that arrive there. To do this, you can use the logical NOT operator as a part of a logical pattern specification as shown below.

   ```
from every (e1=MaterialSupplyStream) -> not MaterialConsumptionStream[name == e1.name and amount == e1.amount]
for 15 min
select e1.name, e1.amount
insert into MaterialSupplyAlertStream;
```

The pattern you defined here is different to the pattern defined in User Scenario 1 in the following ways.

In this scenario, you are not defining a stream reference for the MaterialConsumptionStream stream (i.e. similar to `e2=MaterialConsumptionStream`). This is because the criterion to be met is the non-arrival of events. Therefore, you cannot check for event `e2` in the non-arrival stream.

Instead of the `within` keyword, you are using the `for` keyword. This is because the `not` pattern has to be terminated either by a single and clause (which denotes an event arriving at a different stream can terminate the clause), or a `for <time>` clause (which denotes that the wait time for events not arriving is `<time>`). More details, see the `not` pattern in Siddhi Query Guide - Logical Patterns.

The completed Siddhi application looks as follows.
@App:name('ProductionDelayDetectionApp')

@source(type = 'http', @map(type = 'json'))
define stream MaterialSupplyStream(name string, supplier string, amount double);

@source(type = 'http', @map(type = 'json'))
define stream MaterialConsumptionStream(name string, user string, amount double);

@sink(type='log', prefix='Decrease in supply detected: ')
define stream ProductionDelayAlertStream(name string, amount double);

from every (e1=MaterialSupplyStream) -> not MaterialConsumptionStream[name == e1.name and amount == e1.amount]
  for 15 sec
select e1.name, e1.amount
insert into ProductionDelayAlertStream;

Analyzing Trends

- Introduction
- Tutorial steps

Introduction

In the previous two tutorials, you understood how event correlation is handled through joins as well as patterns. Now, let's look at how a more advanced form of event processing (namely trend analysis) can be carried out with Siddhi.

The factory foreman needs to detect all the input from the Sweet Bots to detect overall decreases in the production of a sweet. For example, if the production of any given sweet is showing a downward trend for 10 minutes, the factory manager needs to be alerted.

Sequences are used in Siddhi for considering consecutive events for analysis. A sequence can be considered a special, more advanced form of Siddhi pattern where only the immediate subsequent event is analyzed. In contrast, a partition can be used to analyze multiple following events. Therefore, sequences use the \( \text{->} \) notation instead of \( \text{->} \) during definition.

The following is an sample sequence definition that considers \( n \) sequential events from the \( \text{stream1} \) stream.

from every el=stream1, e2=stream1, e3=stream1, \ldots, en=stream1
select ...
insert into ...

Tutorial steps

Let's get started!

1. First, let's add the \text{SweetProductionStream} input stream that you first created in Creating a Simple Siddhi Application to capture information from the Sweet Bots.
define stream SweetProductionStream(name string, amount long);

2. The performance analysis needs to be done based on time. Let's also assume that each Sweet Bot is sending events with time information, in the form of an additional field that denotes the UNIX. This additional field needs to be added as an attribute to the SweetProductionStream stream as shown below.

define stream SweetProductionStream(name string, amount long, timestamp long);

An example event (in JSON form) from a SweetBot is as follows:

```json
{
   "event": {
      "name": "Bonbon",
      "amount": 10,
      "timestamp": 1415463675
   }
}
```

3. Let's also define an output stream from which alerts are generated. The following information needs to be included in the output event.
   - The name of the sweet of which the production is showing a downward trend
   - Highest detected value within the 10-minute window (i.e. the initial amount based on which the checks are done)
   - The final amount detected
   - The timestamp at the end of processing

The output stream definition with this information looks as follows.

define stream DecreasingTrendAlertStream(name string, initialAmount long, finalAmount long, timestamp long);

4. To enable the sequential processing of events, let's first define a simple sequence with no processing.

```sql
from every e1=SweetProductionStream, e2=SweetProductionStream
select e1.name, e1.amount as initialAmount, e2.amount as finalAmount, e2.timestamp
insert into DecreasingTrendAlertStream;
```

Here, you are using the comma (,) to delimit events that you need to process in sequence.

5. Next, you need to define the logic for the user scenario. It requires detecting a decreasing trend within 10 minutes in the production figures. This can be deconstructed into the following points:
   i. Every event has a value for the amount parameter.
   ii. If the amount value for event e1 is v1, then we can say that a decreasing trend exists if the next event e2 has a value v2 as the amount, and v1 > v2.
   iii. The above decreasing trend has to continue for 10 minutes to be flagged for alerting.
   iv. If no event with a decreasing trend (i.e., v1 < v2) is detected within 10 minutes, no alert is generated.

Let's construct a query taking the above points into consideration.

a. Taking the two sequential events e1 and e2 into consideration, point ii can be converted into a filter form as follows.
   ```sql
   [e1.amount > e2.amount]
   ```

b. As mentioned in point iii, the processing should consider a period of 10 minutes (10 * 60000 milliseconds). The filter form to do this is as follows.
   ```sql
   [e2.timestamp - e1.timestamp < 10 * 60000]
   ```

c. When the above two filter forms are combined, the following created.
   ```sql
   [e1.amount > e2.amount and (e2.timestamp - e1.timestamp) < 10 * 60000]
   ```

Let's apply this filter to the sequence, so that the above conditions are applied to the second event e2.

```sql
from every e1=SweetProductionStream,
   e2=SweetProductionStream[el.amount > amount and (timestamp - el.timestamp) < 10 * 60000]
select el.name, el.amount as initialAmount, e2.amount as finalAmount, e2.timestamp
insert into DecreasingTrendAlertStream;
```

Here, you are not referring to e2's attributes as e2.amount etc. You can refer to it as amount because it is self evident within the filter.
6. For a complete trend analysis, the pattern matching must run continuously. When an event e2 that meets the conditions you specified arrives, you need to make sure that matching does not stop there and that the next event is also evaluated against the same criterion. In order to achieve this, sequences can be specified with regular expressions. This may include symbols such as * (0 or more instances), + (one or more instances) and ? (zero or one instance).

In this scenario, you need the processing for the decreasing trend to be done continuously (for 0 or more matching events). Therefore, let's add the * regular expression to the filter as shown below.

```sql
from every e1=SweetProductionStream, e2=SweetProductionStream
select e1.name, e1.amount as initialAmount, e2.amount as finalAmount, e2.timestamp
insert into DecreasingTrendAlertStream;
```

7. Finally, you need the processing to conclude at the end of 10 minutes. This means, there should be a clause to check timestamps and match events that timestamps greater than 10 * 60000 of the original event. In other words, adding a filter to match timestamps greater than 10 minutes at the end ensures that the processing can break out of the loop described above and proceed to the output stream.

Therefore, let's define a new event e3 that follows the initial event e1, and zero or more e2 events:

```sql
e3=SweetProductionStream[timestamp - e1.timestamp > 10 * 60000 and e1.amount > amount]
```

Let's add this to the query as follows.

```sql
from every e1=SweetProductionStream,
e2=SweetProductionStream[e1.amount > amount and (timestamp - e1.timestamp) < 10 * 60000]*,
e3=SweetProductionStream[timestamp - e1.timestamp > 10 * 60000 and e1.amount > amount]
select e1.name, e1.amount as initialAmount, e2.amount as finalAmount, e2.timestamp
insert into DecreasingTrendAlertStream;
```

The completed query is as follows.

```sql
define stream SweetProductionStream(name string, amount long, timestamp long);

define stream DecreasingTrendAlertStream(name string, initialAmount long, finalAmount long);

from every e1=SweetProductionStream,
e2=SweetProductionStream[e1.amount > amount and (timestamp - e1.timestamp) < 10 * 60000]*,
e3=SweetProductionStream[timestamp - e1.timestamp > 10 * 60000 and e1.amount > amount]
select e1.name, e1.amount as initialAmount, e2.amount as finalAmount, e2.timestamp
insert into DecreasingTrendAlertStream;
```

8. The Siddhi application still does not generate a product-wise decreasing trend alert. For example, the following two events arriving in succession must be considered a valid decreasing trend.

<table>
<thead>
<tr>
<th>Product</th>
<th>Amount</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bonbon</td>
<td>10</td>
<td>1415463675</td>
</tr>
<tr>
<td>Pretzel</td>
<td>7</td>
<td>1415474284</td>
</tr>
</tbody>
</table>

To address this, you need to consider the processing of each sweet as a self-contained unit instead of considering them all together. This means, when processing events that contain the name attribute set to a particular sweet are only matched with other events with the same value for the attribute.
Siddhi offers this functionality through partitions. With partitions, streams are virtually divided and incoming events are processed in isolated groups (known as partitions) that are completely independent from one another. Each partition (which can contain multiple operations within) is tagged with a key and only processes events that match their key. For more information about partitions, see Siddhi Query Guide - Partition.

A typical partition definition looks as follows.

```plaintext
partition with (myAttribute2 of MyStream )
begin
    from MyStream
    select myAttribute1, myAttribute2, max(myAttribute3) as maxAttribute3
    insert into MyOutputStream;
end;
```

Here, the partition key is `myAttribute2`, and if there are three possible values for `myAttribute2`, each of these values are considered a separate partition.

For this scenario, let's define a partition with the `name` attribute as follows so that each sweet group is considered separately based on its name.

```plaintext
partition with (name of SweetProductionStream)
begin
...
End;
```

Let's copy the query with partitions you created above and place it within the partition as shown below.

```plaintext
partition with (name of SweetProductionStream)
begin
    from every e1=SweetProductionStream,
        e2=SweetProductionStream[e1.amount > amount and (timestamp - e1.timestamp) < 10 * 60000],
        e3=SweetProductionStream[timestamp - e1.timestamp > 10 * 60000 and e1.amount > amount]
    select e1.name, e1.amount as initialAmount, e2.amount as finalAmount, e2.timestamp
    insert into DecreasingTrendAlertStream;
end;
```

The completed Siddhi application looks as follows.
Making Real-time Predictions

- Introduction
- Tutorial steps

Introduction

In the previous tutorials, we looked at the core Siddhi functionalities including event ingestion, publishing and many forms of processing such as preprocessing, correlation, KPI analysis, trend analysis etc.

In this tutorial, let’s move on to the Machine Learning capabilities offered by WSO2 Stream Processor including real-time prediction, online machine learning, anomaly detection etc. Let’s look at how static real-time predictions can be made via WSO2 SP using PMML serializations.

The factory foreman of the Sweet Factory needs to predict the nature of next shipment of sugar syrup based on the shipments he has received so far. A predictive solution has been trained with inputs, temperature and density of the latest shipment. Using these, a prediction is required on whether the shipment received meets his requirements before it is dispatched to the factory. We can also assume that this pre-trained model is exported in PMML serialization and that it is available in the system. To build and train a model, you can use this PMML sample.

**Before you begin:**

PMML (Predictive Model Markup Language) is a standardized serialization that is used for exporting predictive solutions (machine learning models). PMML works by defining the model in one system and transferring the model to a different system via an XML file. This allows predictions to be made using events from the new system. This XML file can contain various data transformation and preprocessing steps in addition to one or more predictive models.

In addition to PMML, TensorFlow serialization is also supported by Stream Processor for static real-time predictions. For more details, see Siddhi Extensions Documentation - Tensorflow.
**Tutorial steps**

Let's get started!

1. Let's add an input event stream definition to capture the events generated by the supplier before shipping the sugar syrup to the sweet factory.

   ```sql
   define stream SugarSyrupDataStream (temperature double, density double);
   ```

2. Now let's define the output stream. To include a prediction on whether the shipment will be acceptable or not in the output, this definition must include an attribute for the prediction as shown below.

   ```sql
   define stream PredictedSugarSyrupDataStream (nextTemperature double, nextDensity double, decision boolean);
   ```

3. As you have learnt from previous tutorials, a reading from an input stream looks similar to the following.

   ```sql
   from SugarSyrupDataStream
   ```

   For this scenario, you need to update it as follows.
   
   a. To enable the PMML extension, you need to add the `#pmml:predict()` annotation as shown below.

   ```sql
   from SugarSyrupDataStream#pmml:predict()
   ```

   b. To access the pre-trained PMML model via which the predictions are made, specify the path as follows.

   ```sql
   from SugarSyrupDataStream#pmml:predict("/home/user/decision-tree.pmml")
   ```

   c. Let's also add the attributes that are needed by the model for prediction.

   ```sql
   from SugarSyrupDataStream#pmml:predict("/home/user/decision-tree.pmml", temperature, density)
   ```

   Based on the model definition, the output attributes can differ. Here, you have defined the model so that it can return a prediction on whether the shipment can be accepted, based on the given temperature and density.

4. Let's route this output to the output stream as shown below.

   ```sql
   from SugarSyrupDataStream#pmml:predict("/home/user/decision-tree.pmml", temperature, density)
   select *
   insert into PredictedSugarSyrupDataStream;
   ```

The completed Siddhi application is as follows.

```sql
@App:name('SugarSyrupPredictionApp')
@source(type='http', receiver.url='http://localhost:5006/SugarSyrupEP', @map(type = 'json'))
define stream SugarSyrupDataStream (temperature double, density double);

@sink(type='log', prefix='Predicted next sugar syrup shipment:')
define stream PredictedSugarSyrupDataStream (nextTemperature double, nextDensity double, decision boolean);

from SugarSyrupDataStream#pmml:predict("/home/user/decision-tree.pmml", temperature, density)
select *
insert into PredictedSugarSyrupDataStream;
```

---

**Streaming Machine Learning**
Introduction

In the previous tutorial, we looked at how static machine learning and prediction can be done via PMML. In this tutorial, let's see how real-time online machine learning support is offered in Stream Processor through various mechanisms.

After the production of each sweet in the Sweet Factory, its density and volume are measured and sent to WSO2 SP. Based on this data, WSO2 SP needs to predict what the produced sweet is out of 5 possibilities.

- **This tutorial covers the following concepts:**
  - A Hoeffding Classifier model is trained using real-time data to get predictions.
  - The other types of algorithms supported are MRulesRegressor, ClusTree, K-Means, and PerceptronClassifier.

**Before you begin:**
The siddhi-gpl-execution-streamingml extension needs to be added to WSO2 SP.

Tutorial steps

Let's get started!

1. The input data captured for the training phase must include the name of the sweet, its density and volume. Let's define an input stream as shown below to capture this information.

```sql
define stream ProductionTrainingStream (density double, volume double, sweetType string);
```

2. You need another input stream through which the volume and the density of the unknown sweet must be sent during the predicting phase. Let's define it as follows.

```sql
define stream SweetProductionStream (density double, volume double);
```

3. Finally, let's define an output stream to present the output of the predictions.

```sql
define stream PredictionStream (density double, volume double, prediction string, confidenceLevel double);
```

4. Now, let's write a simple query to select values from the trainer stream and write them to a temporary stream.

```sql
from ProductionTrainingStream
select *
insert into TemporaryStream;
```

Here, the TemporaryStream stream is an in-memory stream with no definition and no sources/sinks. Therefore, it can be used for routing events that are not needed elsewhere.

5. Now, let's update the query to enable it to train the Hoeffding tree. This can be achieved by specifying the #streamingml annotation to the input stream, and setting its type to HoeffdingTree. Here, we can either specify whether to train the tree (updateHoeffdingTree) or to produce an output based on the learnt model (hoeffdingTreeClassifier).

```sql
from ProductionTrainingStream#streamingml:updateHoeffdingTree()
select *
insert into TemporaryStream;
```

Here, the TemporaryStream stream is an in-memory stream with no definition and no sources/sinks. Therefore, it can be used for routing events that are not needed elsewhere.

6. You need to add two parameters to the updateHoeffdingTree directive for the following.
   - The name of the model that is built or updated.
   - The number of classes in the predicted output. e.g., if the training data can contain five types of sweets, the value for this parameter must be 5 in order to indicate that there are five possible values for the attribute that is being predicted.
In addition, you can also specify the attributes in the input stream that are needed for the training operation. e.g., If you are using the attributes X, Y, and Z from the stream to train the model so that it can predict the value for the K attribute, you must also include the X, Y, Z, and K attributes in the trainer stream definition. The last attribute is considered as the attribute for which the prediction is needed.

Taking the above into account, let’s update the trainer stream definition as follows.

```java
from ProductionTrainingStream#streamingml:updateHoeffdingTree('SweetTypeModel', 5, density, volume, sweetType )
select *
insert into TemporaryStream;
```

Now that the annotation is complete, the events going through to the TemporaryStream stream contain the accuracy evaluation of the model. This is not needed in the end result, and therefore, you can omit it.

7. Now let’s consider the other end of the prediction process where the model makes predictions based on the density and volume. Let’s write a simple query to select values from the prediction stream and direct them to the output stream.

```sql
from SweetProductionStream
select density, volume, prediction, confidenceLevel
insert into PredictionStream;
```

8. Let’s also define the second #streamingml annotation to the SweetProductionStream stream to train the model based on events arriving (as you previously did in step 5).

```sql
from SweetProductionStream#streamingml select density, volume, prediction, confidenceLevel
insert into PredictionStream;
```

9. This time, we need to use the classifier to perform a prediction instead of training the tree. Therefore, let’s add the hoeffdingTreeClassifier annotation instead of the classifier annotation you previously added.

```sql
from SweetProductionStream#streamingml:hoeffdingTreeClassifier()
select density, volume, prediction, confidenceLevel
insert into PredictionStream;
```

10. Let’s specify the following parameters for the model.

   - The name of the model to which you want to refer. You can refer to the SweetTypeModel model that you previously trained.
   - The stream attributes to be used for querying the model for predictions. In this scenario, you need to query by the density and the volume of the unknown sweet.

```sql
from SweetProductionStream#streamingml:hoeffdingTreeClassifier('SweetTypeModel', density, volume)
select density, volume, prediction, confidenceLevel
insert into PredictionStream;
```

Here, the confidenceLevel attribute is a standard returnable from the classifier. It indicates the extent to which the generated prediction can be accurate.

The final query (with source and sink configurations added) looks as follows.
Detecting Anomalies

- Introduction
- Tutorial steps

Introduction

In the previous two tutorials, you looked at how machine learning can be used in conjunction with the real-time processing capabilities offered by the Stream Processor. Now, let's consider a common use-case that is often encountered, namely anomaly detection. In this tutorial, you will look at how the outlier function offered by the time series execution extension can be used to detect anomalies in a stream of events.

The Stream Processor offers various time series-based anomaly detection mechanisms, such as `lengthTime` and `outlier`. In addition, clustering algorithms such as K-Means and ClusTree can also be used for detecting outliers.

In this scenario, the Stream Processor receives a stream of events from the sugar syrup supplier. This stream contains information on the temperature, density and the viscosity of the syrup supplied. The factory foreman needs to monitor the supply of sugar syrup and ensure that the density and the viscosity of each shipment is within an acceptable range. What is required is to have a mechanism that evaluates the viscosity readings based on the supplied syrup shipment and produce an alert if the viscosity is not in the expected range for the given temperature and density. It is also required that the outlier calculation takes place for every last five shipments received so that at any given time, the detection is carried out based on the latest events.

Tutorial steps

Let's get started!

1. Let's start by defining a stream for the input received from the supplier. This needs to contain the temperature, density and viscosity of the new shipment.
define stream SugarSyrupDataStream (viscosity double, temperature double, density double);

2. Next, let's define an output stream to be used for alerting if an outlier is found.

define stream OutlierStream (viscosity double, temperature double, density double, outlier boolean);

3. Now let's add a simple query to select all input and add it to the OutlierStream stream as follows.

from SugarSyrupDataStream
select *
insert into OutlierStream;

This query does not function at this stage because the OutlierStream stream contains an additional attribute named outlier that is not included in the definition of the SugarSyrupDataStream stream.

4. To enable the outlier execution extension, let's add the timeseries annotation. This annotation indicates that a time series extension is being invoked.

from SugarSyrupDataStream#timeseries
select *
insert into OutlierStream;

5. The function to be used for anomaly detection is outlier(). Therefore, let's add it as follows.

from SugarSyrupDataStream#timeseries:outlier()
select *
insert into OutlierStream;

6. Next, let's add parameters for the outlier function as follows.

from SugarSyrupDataStream#timeseries:outlier(5, viscosity, temperature, density)
select *
insert into OutlierStream;

The outlier extension allows a set of parameters that include the following that are used in the query above.

- **The window size**: This specifies the length of the sliding window to be considered when the outlier is calculated. In this scenario, you need to consider five shipments. Therefore, the number of events must be 5.

- **The dependant variable**: This is the variable of which the change needs to be evaluated based on the given independent variables and a linear function. The factory foreman is interested in detecting anomalies in viscosity based on the changes in temperature and density. Therefore, the viscosity is the dependent variable in this scenario.

- **A set of independent variables**: These dictate the nature of the graph, and form the baseline upon which the changes in the dependent variables are measured. In this scenario, the temperature and the density can be considered the independent variables.

The above parameters are relevant for this user scenario. For the complete list of parameters that can be used with the outlier function, see Siddhi Query Guide 4.0 - Time Series - outlier.

This completes the query. The complete Siddhi application looks as follows.
Presenting Data

- Introduction
- Tutorial steps

Introduction

In previous tutorials, you created Siddhi applications to analyze data using the Siddhi logic. In this tutorial, let’s look at how you can present this data in a visual manner.

At the end of this tutorial, you will be able to:

- Create your own widget to represent data stored in an RDBMS table
- Create a dashboard to render a widget
- Secure your dashboard with required permission

Before you begin:

- This tutorial is a continuation of Integrating Datastores. You need to complete it before you try this tutorial.
- Download the com.mysql.jdbc.Driver and save it in the <SP_HOME>/lib directory.

Tutorial steps

Let’s get started!

1. After completing Tutorial 4, you have an RDBMS table where information is saved as follows.

<table>
<thead>
<tr>
<th>Name</th>
<th>Supplier</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flour</td>
<td>Acme</td>
<td>460</td>
</tr>
<tr>
<td>Sugar</td>
<td>Indigo6</td>
<td>272</td>
</tr>
<tr>
<td>Honey</td>
<td>The BeeGees</td>
<td>9</td>
</tr>
<tr>
<td>Food Coloring</td>
<td>Wadjet Food Products</td>
<td>30</td>
</tr>
<tr>
<td>Chocolate Chip</td>
<td>Larkspur Landing</td>
<td>34</td>
</tr>
</tbody>
</table>

WSO2 SP needs to identify this table. To enable this, let’s add a datasource configuration as follows in the <SP_HOME>/conf/dashboard/deployment.yaml file under the wso2:datasources: => datasources subsection.
Now, you are ready to visualize data in the Dashboard portal of WSO2 SP.

2. Now let’s access the dashboard portal as follows.
   a. To start the dashboard server, issue one of the following commands from `<SP_HOME>/bin directory.
      - On Windows: dashboard.bat --run
      - On Linux/Mac OS: sh dashboard.sh
   b. You can access the dashboard via the following URL.
      https://<Your_IP>:9643/portal
   c. Login to the dashboard by entering admin as both the username and the password.
3. Once you access the dashboard portal, the home page is displayed as follows with the two sample dashboard that are shipped with WSO2 SP by default.
Before creating a dashboard, let's create a widget as follows

a. Click **CREATE WIDGET**. This opens the widget generation wizard as follows.

b. In the first page where you are prompted to enter a name for your widget, let's add **Shipment Details** as the name of the widget.
c. Click Next to open the page for configuring the data provider. In this scenario, you are visualizing data stored in the database table you have already created. Therefore, select RDBMS Batch Data Provider.

Once you select this, you are prompted to enter more data relating to the datasource you configured in the `<SP_HOME>/conf/dashboard/deployment.yaml` file.

i. In your datasource configuration in the deployment.yaml file, you have entered SweetFactoryDB as the datasource name. Therefore, you need to enter the same name in the Datasource Name field here.

ii. In the Query field, you need to enter the database query you need to run in order to retrieve the data to be visualized. Let's enter the following query to retrieve that data from the ShipmentDetailsTable.

   ```sql
   select * from ShipmentDetailsTable
   ```

iii. In the Table Name field, enter the name of your table, which is ShipmentDetailsTable. Leave the default values unchanged in the rest of the fields. The information you entered for the data provider is as follows.
Create Widget

Select a data provider type and configure its properties.

**RDBMS Batch Data Provider**

RDBMS provider supports retrieving data for widgets using any SQL database table. The Batch provider allows the user to get all the data in the table batch-wise, and the Streaming data provider will allow the user to retrieve data in a streaming manner using an incremental data column.

**Datasource Name**

`SweetFactoryDB`

Name of the datasource defined in the deployment yaml.

**Dynamic Query Generation Configuration**

**Add widget inputs**

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Enter your JS function to create the query:

```javascript
function generateQuery(username) {
    return "select * from ShipmentDetails";
}
```
d. Let's click Next to configure the chart following the steps below.
   i. In this scenario, let's create a simple line chart to view your data. Therefore, select **Line/Area/Bar Chart Parent** as the chart type. You can select **name** for the **x axis field** field. Leave the default values for the rest of the fields unchanged.

For more information about the other fields to be configured for the data provider, see [Generating Widgets](#).
ii. Before you create the chart, click Preview to see a preview of the chart. The preview is displayed as follows.
If you are satisfied with the preview, click **CREATE** to create the widget.

4. To render the widget you created, you need a dashboard. Let's create the dashboard as follows.
   a. The previous step brings you back to the home page of the Dashboard portal. To create the dashboard, click **CREATE DASHBOARD**.
   b. In the wizard for creating dashboards that opens, enter information as follows.

   ![Create Dashboard Interface]

   **Create Dashboard**

   Name of your Dashboard
   
   **Sweet Factory**

   Description
   
   **Sweet Factory Shipment Details**

   ADD CANCEL
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name of your Dashboard</td>
<td>Enter a unique name for your dashboard.</td>
<td>Sweet Factory</td>
</tr>
<tr>
<td>URL</td>
<td>Enter a URL for the dashboard.</td>
<td>sweetfactory</td>
</tr>
<tr>
<td>Description</td>
<td>Enter a description for the dashboard.</td>
<td>Sweet Factory Shipment Details</td>
</tr>
</tbody>
</table>

c. Click **Add** to proceed. You are directed to the home page that is created by default.

Click the widgets icon (marked in the image below) to open the widgets panel.
d. You are directed to the dashboard designer with a list of widgets displayed in the left panel. Search for the Shipment Details widget that you previously created. You can either type the name of the widget in the search bar or scroll down to locate it.
e. To render the Shipment Details widget, drag and drop it to the dashboard container.

Now you have successfully created your dashboard.

5. To be redirected to the home page with the list of available dashboards, click Portal. The Sweet Factory dashboard you created should be displayed there. Now you need to ensure that only the intended users are able to view your dashboard. As an owner of the dashboard, you can restrict access to the Sweet Factory dashboard for other users. To do this, click the menu icon for your dashboard, and then click Settings.
This opens the **Dashboard Settings** page. Enter user roles under each category of users as follows. Depending on your configured identity provider, the available list of roles is displayed in the drop-down list for each user type.
a. **Owners** are the most privileged users of the dashboard. They are the only users with access to this Dashboard Settings page. They can also edit and view the dashboard.

b. **Editors** of a dashboard can either edit the dashboard or view the dashboard. However, they do not have access to the Dashboard Settings page.

c. **Viewers** are the least privileged users of a dashboard. They can view the dashboard, but cannot edit it or change its settings.

**Managing Business Rules via Templates**

- Introduction
- Tutorial steps

**Introduction**

In the previous tutorials, you covered various Siddhi concepts and created many Siddhi applications to try them out. In many scenarios, the basic outline of the Siddhi application has been the same. For example, the sources defined for SweetBots are same across multiple Siddhi applications. Furthermore, Siddhi application functionality can be templated so that conditions can be changed over time. Business Rules
Management dashboard is a user friendly interface that can be used to build business rules (Siddhi applications customized as per business requirements).

Previously (in tutorial 7), you created a Siddhi application to observe KPIs. The Factory Foreman of the Sweet Factory needs to set different KPIs for the different categories of sweets. For this purpose, you need to template the Siddhi application for KPI analysis that you created in tutorial 7. Using this template, you can create different business rules based on sweet names, surplusMinThresholdInput, and the start and end of work hours.

This tutorial covers Business Rules Management.

**Before you begin:**
Before creating business rules, the required templates must be defined. The following is a sample template you can use in this tutorial.

Click here to view the sample template

```json
{
    "templateGroup" : {
        "name" : "Sweet Production Factory",
        "uuid": "sweet-production-factory",
        "description" : "Analyzes productions from sweet factory",
        "ruleTemplates" : [
            {
                "name" : "KPI Analysis",
                "uuid" : "sweet-production-kpi-analysis",
                "type" : "template",
                "instanceCount" : "many",
                "description" : "Key Performance Indicator analysis for Sweet Production",
                "script" :
                    "// Validates given surplus minimum threshold
                    function validateAndGetSurplusMinThreshold(amount) {
                        if (isNaN(amount) || amount < 1) {
                            throw 'Minimum value for surplus should be a positive number';
                        }
                        return amount;
                    }

                    // Gets working hour in 24 hours format
                    function getWorkingHour(hour, amOrPm) {
                        // Validate given hour
                        if (isNaN(hour)) {
                            throw 'A valid number is expected for hour instead of ' + hour;
                        }
                        if (hour % 1 != 0) {
                            throw 'A full working hour expected instead of ' + hour;
                        }
                        if ((hour < 1) || (hour > 12)) {
                            throw 'Working hour should be in between 1 and 12';
                        }
                    }
        }
    }
}
```
var workHoursStart = getWorkingHour('${workHoursStartInput}', '${workHoursStartAM_PM}');
var workHoursEnd = getWorkingHour('${workHoursEndInput}', '${workHoursEndAM_PM}');
var surplusMinThreshold = 
validateAndGetSurplusMinThreshold('${surplusMinThresholdInput}');

"templates" : [
    {
        "type" : "siddhiApp",
        "content" : 
            "@App:name('SweetProductionKPIApp')
            @source(type='http', receiver.url='http://localhost:5005/SweetProductionEP', @map(type = 'json', @attributes(name = '${.sweet}', amount = '${.batch.count}'))
            define stream SweetProductionStream (name string, amount long);
            @sink(type='log', prefix='${lowProductionAlertPrefix}')
            define stream LowProductionStream (name string, hourlyTotal long, currentHour int);
            @sink(type='log', prefix='${sweetProductionStatusPrefix}')
            define stream ProductionStatusStream (name string, hourlyTotal long, currentHour int, status string);
            from SweetProductionStream#window.time(${timeInterval} hour)
            select name, sum(amount) as hourlyTotal, time:extract(currentTimeMillis(), 'HOUR') as currentHour, ifThenElse(sum(amount) > ${surplusMinThreshold}, "Possible surplus", "OK") as status
            group by name
            having currentHour > ${workHoursStart} and currentHour < ${workHoursEnd}
            insert into ProductionStatusStream;"
    }
],
"properties" : {
    "lowProductionAlertPrefix": {"fieldName": "Low production alert prefix", "description": "Displayed when a low production alert is logged", "defaultValue": "Low production alert:"},
    "sweetProductionStatusPrefix": {"fieldName": "Sweet production status prefix", "description": "Displayed when a sweet production status is logged", "defaultValue": "Sweet production status:"}
Tutorial steps

Let's get started!

In this tutorial, the dashboard environment is started in the same computer as the worker environment for simplicity.

1. To start the dashboard profile of WSO2 SP, issue one of the following commands from the `<SP_HOME>/bin` directory.
   - For Windows: `dashboard.bat`
   - For Linux: `sh dashboard.sh`

2. Now start the worker profile of WSO2 SP, issue one of the following commands from the `<SP_HOME>/bin` directory. This is needed

---

This template must be placed in the `<SP_HOME>/wso2/dashboard/resources/businessRules/templates` directory as `Sweet_Production_Factory.json`.

The group UUID (sweet-production-factory) which is in the ruleTemplates must be added in the `<SP_HOME>/conf/dashboard/deployment.yaml` file under the `wso2.business.rules.manager` namespace as shown below.

```yaml
wso2.business.rules.manager:
  datasource: BUSINESS_RULES_DB
  # rule template wise configuration for deploying business rules
  deployment_configs:
    - # ip:port of the node
      localhost:9090:
        # uuids of rule templates which are needed to be deployed on the node given above
        - sweet-production-factory
        # credentials for worker nodes
        username: admin
        password: admin
```

For more information, see Business Rules Templates.
when saving and deploying a business rule.

- For Windows: `worker.bat`
- For Linux: `sh worker.sh`

3. You can access the Business Rules dashboard via the following URL:

   https://localhost:9643/business-rules

   Then login by specifying `admin` as both the username and the password.

   This step uses the default URL and credentials. If the dashboard user store is changed, use a valid credential and ensure that the user has valid permissions for adding and deleting workers. For more information, see User Management.

4. To create a new rule that can be used for KPI analysis, let's click **CREATE**.

   ![No business rules found](image)

   If previously defined business rules exist, they are displayed as shown in the example below. You can click `+` to add a new business rule.

   ![Business Rules](image)

5. The following view opens when you click **CREATE**. You can observe that a business rule can be created from a business template or from scratch. You have already created and saved a business rules template. Therefore, let's click **From Template** to create a rule from it.
6. When you click From Template the Sweet Production Factory template that you previously created and saved is displayed as follows.

To create a rule from this template, let's click on this template. It opens as follows.
7. In the **Rule Template** field, you can select the **KPI Analysis** rule template.

8. You can enter values in the rule template that opens as shown below.
Click save and deploy to proceed. As a result, the following log is displayed in the worker console.

INFO [org.wso2.carbon.stream.processor.core.internal.StreamProcessorService] - Siddhi App sweetproductionkpianalysis_0 deployed successfully

Monitoring the Stream Processor via the Status Dashboard
Introduction

In previous tutorials, you learnt different Siddhi concepts and created many Siddhi applications to try them out. However, those Siddhi applications were created in the Stream Processor Studio and they were not deployed in a production environment.

In this tutorial, let's consider a scenario where your Siddhi applications are deployed in a production environment, and in a worker profile. The node is configured to execute Siddhi applications, and it does not have a UI (User Interface).

The staff at the Sweet Factory need to check whether the SP nodes in which their Siddhi applications are deployed are working as expected, or whether any Siddhi application requires performance improvements. They need a user-friendly UI in order to monitor this. For this purpose, you can use the Monitoring Dashboard of WSO2 SP.

Tutorial steps

Let's get started!

1. Let's start the dashboard profile of WSO2 Stream Processor by issuing one of the following commands from the `<SP_HOME>/bin` directory.
   - **For Windows:** `dashboard.bat`
   - **For Linux:** `sh dashboard.sh`

2. To access the Monitoring Dashboard, you can use the following URL.
   https://localhost:9643/monitoring

   You can login to the dashboard by specifying `admin` as both the username and the password.

   Here, we are using the credentials available by default to access the dashboard. If you are using a different user, make sure that it is granted the relevant permissions to add/delete workers from the dashboard. For more details, see User Management via the IdP Client Interface.

   The following is displayed once you login.

   ![Node Overview](image)

3. Let's add a worker to view statistics for it. To do this, follow the substeps below.
   a. Click **Add New**. This opens the following page.
b. Let’s add **localhost** as the host and **9443** as the HTTPS port.

These are the default host and port of a single node. If the worker node is not run in the same computer as the dashboard node or if it is run on a different port, change these values accordingly.

Click **Add Node** to proceed. As a result, the following is displayed.

**Unreachable Node**

Node details you entered is currently unreachable. Please choose the run time environment.

- **Worker**
- **Manager**
- **Cancel**

To monitor a worker node (which we will run later on **localhost:9443**), click **Worker**.
3. The dashboard indicates that there is no worker running at the host and port you specified because you have not started a worker node yet.

4. To view an active worker, let's start a worker node by issuing one of the following commands from the `<SP_HOME>/bin` directory.

   - For Windows: `worker.bat`
   - For Linux: `sh worker.sh`

   In this tutorial, you can run the worker and the dashboard in the same computer for simplicity.

   Now, if you refresh the **OVERVIEW** page of the Monitoring Dashboard, an active worker is displayed as follows.
5. To view more information about the worker node, let's click on it. This opens a page named `localhost:9443`, displaying information specific to the worker node you started as shown below.

This information is displayed in this page after a time period of about 30 seconds after opening it.

Note that no information is currently displayed under **Overall Throughput (events/second)** and **Siddhi Applications**. This is because no Siddhi applications are deployed in this worker.

6. Let's deploy and run a Siddhi application in the worker. For this purpose, you can use the `SweetTotalApp` that you created in **Tutorial 1** and updated in **Tutorial 2**.
You can deploy the Siddhi application by copying the `SweetTotalApp.siddhi` file from the `<SP_HOME>/wso2/editor/deployment/workspace` directory and pasting it in the `<SP_HOME>/wso2/worker/deployment/siddhi-files` directory.

7. To view the Siddhi application you deployed, let's refresh the `localhost:9443` page. The `SweetTotalApp` Siddhi application is displayed in the Siddhi Applications section as follows.

8. To view information specific to the Siddhi application, click on it. This opens a page named `localhost:9443 > SweetTotalApp` as shown below.

9. Metrics are not enabled for Siddhi applications by default. Let's enable metrics for the `SweetTotalApp` Siddhi application by clicking the `Metrics` switch. This displays available information relating to the latency, overall throughput and memory used, as well as available statistics for the Siddhi application as shown below.
Performing Real-time and Periodic ETL

Introduction

In many real life scenarios that involve integrated enterprise, data needs to be loaded and moved from one location to another to be processed for future reference. This process is commonly known as Extract, Transform, and Load (ETL). In this tutorial, you can learn how ETL is carried out in real-time and periodically.

Let's consider that the Sweet Factory has shops distributed all over the world. In addition, there are travelling sales people who operate from mobile sweet trucks. Each sales person transactions via a system that generates JMS messages. This data needs to be saved in a database so that it can be used later for sales analysis and financial analysis. Later, they need to be moved to other databases depending on the nature of the analysis carried out.

Tutorial steps

This section covers two scenarios that are simple examples for performing real-time and periodic ETL in real world business scenarios.

Scenario 1: Extract data from JMS, perform a stateless transformation, and load to a database.

Once the head office of the Sweet Factory receive events with information about the sales transactions, that information needs to be cleaned and enriched via a connection with a data store. Then the enriched version of the data needs to be saved in another RDBMS store.
Let's get started!

1. The sales transactions reported as JMS messages include the user ID (i.e., the ID of the salesman), the transaction amount and the location. Let's begin by adding the stream definition to capture this information.

   ```
   define stream TransactionStream (userId long, transactionAmount double, location string);
   ```

2. Before enriching the data for further analysis, it needs to be cleaned. This involves checking for null values, and replacing them with default values. To do this, let's follow the steps below:

   a. First, let's select the data you need to extract and clean. This includes the user ID, transaction amount and the location. For travelling sales people, specific locations are not registered in the system. As a result, no value is specified in the location attribute of the events generated for some sales actions. To clean the data before enriching it, you need to replace the null value for the location attribute of such events with unknown. This can be achieved via the `ifThenElse()` function of Siddhi by including it in the select clause as shown below.

   ```
   select userId, 
   transactionAmount, 
   ifThenElse(location is null, "UNKNOWN", location) as location
   ```

   b. The information specified in the select clause given above is taken from the `TransactionsStream` input stream that you previously defined. To specify this, let's add a `from` statement as follows.

   ```
   from TransactionStream 
   select userId, 
   transactionAmount, 
   ifThenElse(location is null, "UNKNOWN", location) as location
   ```

   c. Now you can add an output stream to be inferred so that the information extracted and cleaned can be directed to it.

   ```
   from TransactionStream 
   select userId, 
   transactionAmount, 
   ifThenElse(location is null, "UNKNOWN", location) as location 
   insert into CleanedTransactionsStream;
   ```

   d. Now you have completed the query for extracting and cleaning the information you need. You will be adding more queries to this Siddhi application in order to enrich and load data as you proceed with this tutorial. Therefore, to make it easy for other users to understand this Siddhi application, let's name this query as `CleaningData`. The completed query looks as follows.

   ```
   @info(name = 'CleaningData')
   from TransactionStream 
   select userId, 
   transactionAmount, 
   ifThenElse(location is null, "UNKNOWN", location) as location 
   insert into CleanedTransactionsStream;
   ```

3. The information you have extracted and cleaned needs to be enriched. To do this, let's create another query named `EnrichData` as follows.

   a. You have already directed the extracted and cleaned information to the `CleanedTransactionsStream` stream. Therefore, let's specify that stream as the input stream for this query.

   ```
   from CleanedTransactionsStream
   ```

   b. The user IDs of the registered sales people are already saved in a table known as the `userTable`. To enrich the data, user IDs of the events in the `CleanedTransactionsStream` stream need to be joined with the user IDs stored in the table. For this, let's add a join query as follows.

   ```
   from CleanedTransactionsStream as c join UserTable as u 
   on c.userId == u.userId
   ```

   c. Now, let's include a concatenation to derive a user name ID from the first name and the last name of the sales people as follows.

   ```
   select c.userId, 
   str:concat( u.firstName, " ", u.lastName) as userName, 
   transactionAmount, 
   location
   ```

   d. The enriched data can be directed to another inferred output stream named `EnrichedTransactionsStream`. 
from CleanedTrasactionStream as c join UserTable as u
on c.userId == u.userId
select c.userId,
str:concat( u.firstName, " ", u.lastName) as userName,
transactionAmount,
location
insert into EnrichedTrasactionStream;

4. In the EnrichData Siddhi query you created, you included a join between the CleanedTrasactionStream stream and a table named UserTable. For this query to be executed, the definitions for both the stream and the table need to be included in the same Siddhi application. You have already added the stream definition. Now let’s add the table definition above the Siddhi queries as follows.

More table definitions are added as you proceed with the tutorial. Therefore, it is useful to specify the purpose of each table definition via a comment as shown below.

define stream TrasactionStream (userId long, 
transactionAmount double, location string);

-- Table used to enrich data
@store(type = 'rdbms', datasource = 'TRANSACTION_DATA_SOURCE')
define table UserTable (userId long, firstName string,
lastName string);

@info(name = 'CleaningData')
from TrasactionStream
select userId,
transactionAmount,
ifThenElse(location is null, "UNKNOWN", location) as location
insert into CleanedTrasactionStream;

@info(name = 'EnrichData')
from CleanedTrasactionStream as c join UserTable as u
on c.userId == u.userId
select c.userId,
str:concat( u.firstName, " ", u.lastName) as userName,
transactionAmount,
location
insert into EnrichedTrasactionStream;

5. Let’s create another table to store the enriched data as shown below.
5. Final table to load the data

```plaintext
@store(type = 'rdbms', datasource = 'TRANSACTION_DATA_SOURCE')
define stream TransactionTable (userId long, transactionAmount double, location string);
```

6. To insert the enriched data to the table you created, let's add another Siddhi Query as follows:

```plaintext
@info(name = 'LoadData')
from EnrichedTransactionStream
insert into TransactionTable;
```

7. The transactions need to be generated via JMS in JSON format. Therefore, let's add an event sink with JMS as the transport and JSON as the format. You can add this above the stream definition as shown below.

```plaintext
@sink(type = 'jms', destination = 'transactions',
      factory.initial = '...', provider.url = '...',
      @map(type = 'json',
           @attributes('$.userId', '$.transactionAmount', '$.location')))

define stream TransactionStream (userId long, transactionAmount double, location string);
```

Let's name this Siddhi application as JMS-to-DB-ETL. You can optionally add a description too. The complete Siddhi Application looks as follows:
Scenario 2: Extract data from a database, perform a stateful transformation, and load data to Kafka

Let's consider a scenario where the head office of the Sweet Factory needs to analyze sales by the location based on the latest transactions
every five minutes. To do this, you need to poll the database with the sales transactions every five minutes. Before the aggregates are calculated, you also need to ensure that the database contains the latest transactions. You can perform real-time ETL via the WSO2 Stream Processor to achieve this as shown below.

Let's get started!

1. The information you need to transform and load to Kafka needs to be extracted from a datastore. Therefore, let's start by adding the definition of this data table to the Siddhi application as follows.

   -- In-memory Table to keep last processed transaction ID
   @primaryKey('key')
   define table TriggerStateTable (key string, lastProcessedId long);

2. Once the information is extracted and transformed, the transformed data needs to be directed to an output stream. This output stream can be defined as follows.

   define stream TransformedDataStream(transactionLocation string,
   totalTransactionAmount double, avgTransactionAmount double,
   transactionId long);

3. The database table needs to be polled every five minutes for the last processed ID. To this, you can compare the last processed ID with a stream in which a new event is triggered every five minutes. Therefore, let's add a trigger as follows:

   define trigger TriggerStream at every 5 min;
   define stream TransformedDataStream(transactionLocation string,
   totalTransactionAmount double, avgTransactionAmount double,
   transactionId long);

   -- In-memory Table to keep last processed transaction ID
   @primaryKey('key')
   define table TriggerStateTable (key string, lastProcessedId long);

4. To collect the last processed transaction ID, you need to add a Siddhi query as follows.

   a. To extract the last processed ID, the events triggered in the TriggerStream stream as well as the events stored in the TriggerStateTable table need to be considered. Therefore, let's add a Siddhi join to join this stream and the table as follows.

      from TriggerStream as s right outer join TriggerStateTable as t
      select ifThenElse(t.lastProcessedId is null, 0l, t.lastProcessedId )
      as lastProcessedId
      insert into DataRetrievalStream;

   b. The data collected needs to be inserted into another stream so that it can be used for further processing. Therefore, let's add the insert clause with an inferred stream as follows.

      from TriggerStream as s right outer join TriggerStateTable as t
      insert into DataRetrievalStream;

   c. Now you need to specify the condition based on which the last processed ID is selected from the joined TriggerStream stream and the TransactionTable table, and inserted into the DataRetrievalStream. Let's add it as follows.

      from TriggerStream as s right outer join TriggerStateTable as t
      select ifThenElse(t.lastProcessedId is null, 0l, t.lastProcessedId )
      as lastProcessedId
      insert into DataRetrievalStream;

5. Let's define a data store to save the transaction data as follows.

   -- Store table
   @store(type = 'rdbms', datasource = 'TRANSACTION_DATA_SOURCE')
   define table TransactionTable(transactionId long, userId long,
   transactionAmount double, transactionLocation string);

6. Now you have created a query to collect the last processed IDs from a triggered stream and a table. The data is taken from two locations via a join. Therefore, you need to process the data collected from both to extract the last processed ID. To do this, let's create a Siddhi query as follows.

   a. The last processed transaction ID should be taken from the TransactionTable only if the last processed ID in that table is later
than the last processed ID of an event in the DataRetrievalStream stream. Therefore, let's join this stream and table so that the last processed ID is selected by comparing events in both.

```
from DataRetrievalStream as s join TransactionTable as t
on s.lastProcessedId < t.transactionId
```

b. The data extracted needs to be directed to an output stream. Therefore, let's add an output stream named ExtractedDataStream that can be inferred.

```
from DataRetrievalStream as s join TransactionTable as t
on s.lastProcessedId < t.transactionId
insert into ExtractedDataStream;
```

c. Now let's select the information to be inserted into the ExtractedDataStream stream from the event that is extracted as the last processed event.

```
from DataRetrievalStream as s join TransactionTable as t
on s.lastProcessedId < t.transactionId
select t.transactionId, t.transactionAmount, t.transactionLocation
insert into ExtractedDataStream;
```

d. The completed query looks as follows.

```
@info(name = 'ExtractData')
from DataRetrievalStream as s join TransactionTable as t
on s.lastProcessedId < t.transactionId
select t.transactionId, t.transactionAmount, t.transactionLocation
insert into ExtractedDataStream;
```

7. You need to update the TriggerStateTable table with the last processed ID identified after the join between the table and the stream. To do this, let's create a Siddhi query as follows.

a. The information with which the TriggerStateTable table is updated is taken from the ExtractedDataStream stream to which the last processed transaction is being sent every five minutes. Therefore, let's add this stream as the input stream.

```
from ExtractedDataStream
```

b. The event with the last processed ID may already exist in the TriggerStateTable table. In such a scenario, the existing record should be updated with the latest details taken from the stream. If that event does not already exist in the table, it should be inserted as a new record. To indicate this, let's add an update or insert clause as follows.

```
update or insert into TriggerStateTable
on TriggerStateTable.key == "lastProcessedId";
```

c. When selecting records to be updated/inserted to the table, the unique identity by which the records are identified should be the last processed ID. Therefore, let's add a select clause with the lastProcessedID attribute specified as the key.

```
from ExtractedDataStream
select "lastProcessedId" as key, transactionId as lastProcessedId
update or insert into TriggerStateTable
on TriggerStateTable.key == "lastProcessedId";
```

d. The completed query looks as follows.

```
@info(name='UpdateLastProcessedId')
from ExtractedDataStream
select "lastProcessedId" as key, transactionId as lastProcessedId
update or insert into TriggerStateTable
on TriggerStateTable.key == "lastProcessedId";
```

8. Now let's perform a stateful aggregation. You need to analyze sales the sales for each location by calculating the total sales and the average sales per location every five minutes. Let's create a Siddhi query to carry out this analysis.
a. The information to be analyzed is taken from the `ExtractedDataStream` stream, and the results are inserted into an output stream named `TransformedDataStream`. Therefore, let's add the `from` and `insert into` clauses accordingly.

```sql
from ExtractedDataStream
insert into TransformedDataStream
```

b. As mentioned, the calculations need to be done every five minutes in a tumbling manner. Therefore, let's add a time batch window as follows.

```sql
from ExtractedDataStream#window.timeBatch(5 min)
insert into TransformedDataStream
```

c. The attributes in the `ExtractedDataStream` stream are `transactionID`, `transactionLocation` and `transactionAmount`. The values for the `transactionLocation` attribute need to be taken without any further processing. However, you need to derive the total transaction amount and the average transaction amount from the values for the `transactionAmount` attribute. This can be achieved via the `sum()` and `avg()` Siddhi functions. Therefore, let's add the select clause as follows.

```sql
from ExtractedDataStream#window.timeBatch(5 min)
select transactionLocation,
  sum(transactionAmount) as totalTransactionAmount,
  avg(transactionAmount) as avgTransactionAmount,
  transactionId
insert into TransformedDataStream;
```

d. The analysis is carried out to evaluate the sales by location. Therefore, let's add a group by clause to group the calculations by the location when presenting the results.

```sql
from ExtractedDataStream#window.timeBatch(5 min)
select transactionLocation,
  sum(transactionAmount) as totalTransactionAmount,
  avg(transactionAmount) as avgTransactionAmount,
  transactionId
group by transactionLocation
insert into TransformedDataStream;
```

e. Now let's name the query and complete it as follows.

```sql
@info(name = 'StatefulAggregation')
from ExtractedDataStream#window.timeBatch(5 min)
select transactionLocation,
  sum(transactionAmount) as totalTransactionAmount,
  avg(transactionAmount) as avgTransactionAmount,
  transactionId
group by transactionLocation
insert into TransformedDataStream;
```

9. The processed information needs to be published via Kafka. Therefore, let's add a Kafka sink as shown below.

```sql
-- Sink to load the data
@sink(type='kafka', bootstrap.servers='...',
topic='...',is.binary.message='...',
@map(type='xml'))
```

This sink publishes data from the `TransformedDataStream` stream that is added as the output stream to direct the data processed by the `StatefulAggregation` Siddhi query. Therefore, let's connect the sink to that stream by adding the stream definition below it as shown below.

```sql
-- Sink to load the data
@sink(type='kafka', bootstrap.servers='...',
topic='...',is.binary.message='...',
@map(type='xml'))
define stream TransformedDataStream(transactionLocation string,
totalTransactionAmount double, avgTransactionAmount double,
transactionId long);
```
Now let's name the Siddhi application and complete it. The completed version looks as follows.

```siddhi
@App:name('DB-to-Kafka-ETL')
@App:description('Extract data from database, perform stateful
transformation, and load data to kafka')

declare trigger TriggerStream at every 5 min;
-- Sink to load the data
@sink(type='kafka', bootstrap.servers='...',
   topic='...',is.binary.message='...',
   @map(type='xml'))
define stream TransformedDataStream(transactionLocation string,
totalTransactionAmount double, avgTransactionAmount double,
transactionId long);

-- Store table
@store(type = 'rdbms' , datasource = 'TRANSACTION_DATA_SOURCE')
define table TransactionTable(transactionId long, userId long,
transactionAmount double, transactionLocation string);

-- In-memory Table to keep last processed transaction ID
@primaryKey('key')
define table TriggerStateTable (key string, lastProcessedId long);

@info(name = 'CollectLastProcessedId')
from TriggerStream as s right outer join TriggerStateTable as t
select ifThenElse(t.lastProcessedId is null, 0l, t.lastProcessedId)
as lastProcessedId
insert into DataRetrievalStream;

@info(name = 'ExtractData')
from DataRetrievalStream as s join TransactionTable as t
   on s.lastProcessedId < t.transactionId
select t.transactionId, t.transactionAmount, t.transactionLocation
insert into ExtractedDataStream;

@info(name='UpdateLastProcessedId')
from ExtractedDataStream
select "lastProcessedId" as key, transactionId as lastProcessedId
update or insert into TriggerStateTable
   on TriggerStateTable.key == "lastProcessedId";

@info(name = 'StatefulAggregation')
from ExtractedDataStream#window.timeBatch(5 min)
select transactionLocation,
   sum(transactionAmount) as totalTransactionAmount,
   avg(transactionAmount) as avgTransactionAmount,
   transactionId
group by transactionLocation
insert into TranformedDataStream;
```
User Guide

The section demonstrates how to use some of the important functionalities of WSO2 SP:

- **Understanding the Development Environment**: Provides the developer relevant knowledge on how to use the Stream Processor Editor and its functionalities.
- **Developing Streaming Applications**: Provides information on how to write Stream Processing Applications using Siddhi Streaming SQL.
- **Deploying Streaming Applications**: Provides information on how a user can deploy a Stream Processing application in a production environment.
- **Visualizing Data**: Explains how to create visualizations using dashboards and widgets.
- **Managing Business Rules**: Demonstrates how developers can define templates, and how business users can create rules from those templates and manage them.
- **Monitoring Stream Processor**: Explains how to configure the Status Dashboard to monitor Stream Processor nodes.

**Understanding the Development Environment**

This section explains the WSO2 Stream Processor Studio which is shipped with WSO2 SP to serve as the development environment to develop Siddhi applications.

Watch the following screencast for an introduction to the Stream Processor Studio.

**Using Stream Processor Studio/Editor**

The following topics describe how to use Stream Processor Studio/Editor to develop Siddhi stream processing applications.

- **Stream Processor Studio Overview**
- **Creating a Siddhi Application**
- **Testing a Siddhi Application**
- **Debugging a Siddhi Application**
- **Exporting a Siddhi File**

**Using IDE Plugins**

You can also develop, run and debug Siddhi stream processing applications via the IntelliJ IDEA Plugin.

**Stream Processor Studio Overview**

The Stream Processor Studio is a developer tool that is shipped with WSO2 SP to develop Siddhi applications. It allows provides two interfaces to develop Siddhi applications

- **From source view**: This allows you to write Siddhi applications in the Siddhi Query Language. This supports auto-completion, tracking syntax errors and debugging.
- **From design view**: This interface visualizes the event flow of a Siddhi application, and allows you to compose the applications by dragging and dropping Siddhi components to a graph.

Once a Siddhi application is created, you can simulate events via the Stream Processor Studio to test whether it works as expected. You can also run the Siddhi application in the debug mode to detect errors in the Stream Processor logic.

**Starting Stream Processor Studio**

To start and access the Stream Processor Studio, follow the steps below:

1. Start the Stream Processor Studio by issuing one of the following commands from the `<SP_HOME>/bin` directory.
   - For Windows: `editor.bat`
   - For Linux: `./editor.sh`

The default URL is `http://localhost:9390/editor`

**Welcome Page**
- **New**
  Click this to open a new untitled Siddhi file.

- **Open**
  Click this to open a Siddhi file that is already saved in the `workspace` folder of the Stream Processor Studio. If the file is already opened in a new tab, clicking **Open** does not open it again. The default path to the `workspace` directory is `<SP_Home>/wso2/editor/deploy`.

- **Try out samples**
  The pre-created samples provided out of the box are listed in this section. When you click on a sample, it opens in a new tab without a title.

- **More Samples**
  Click this to view the complete list of samples in the samples directory. This allows you to access samples other than the ones that are displayed by default in the **Try out samples** section. When you click on a sample, it opens in a new tab without a title.

- **Quick links**
  This section provides links to more resources.

### Menu Items

This section explains the options that are available in the **File**, **Edit** and **Run** menus.

#### File menu items

The **File** menu includes the following options.
• New
  Click this to open a new untitled Siddhi file.

• Open File
  Click this to open a Siddhi file that is already saved in the workspace directory of the Stream Processor Studio. If the file is already opened in a new tab, clicking this menu item does not open it again in another tab. The default path to the workspace directory is ${SP Home}/wso2/editor/deployment.

When a Siddhi file is opened, its source view is displayed by default.

To view a design view where the elements of the Siddhi application are graphically represented, click Design View. As a result, a graphical view of the Siddhi application is displayed as shown in the following example.
- **Import Sample**
  Click this to import a sample from the samples directory to a new tab. The sample opens in an untitled Siddhi file. Once you save it, it can be accessed from the workspace directory.

- **Save**
  Click this to save an edited or new file to the workspace directory.

- **Save As**
  Click this if you want to save an existing saved file with a different name. If you click this for an untitled Siddhi file, the normal save operation is executed (i.e., same operation carried out when you click **Save**).

- **Import File**
  Click this to open a file from a system location. This file is opened in a new tab in the saved state with the same file name with which it is imported.

- **Export File**
  Click this to export a saved file to a system location. This is only applicable to Siddhi application tabs that are in a saved state.

- **Close File**
  Click this to close a currently active Siddhi application that is already open in a tab.

- **Close All Files**
  Click this to close all the Siddhi files that are currently open.

- **Delete File**
  Click this to delete the currently active Siddhi file from the workspace directory. Only Siddhi files that are already saved can be deleted.

- **Settings**
  Click this to change the theme and the font size used in the Stream Processor Studio. The default theme is **Twilight**.

**Edit menu Items**

The **Edit** menu includes the following options.
- **Undo**
  Click this to undo the last edit made to the Siddhi application that you are currently editing. Only unsaved edits can be undone.

- **Redo**
  Click this to redo the edit that was last undone in the Siddhi application that you are currently editing. The redo operation can be carried out only if you have not saved the Siddhi application after you undid the change.

- **Find**
  Click this to search for a specific string in the currently active Siddhi application tab.

- **Find and Replace**
  Click this to search for a specific string in the currently active Siddhi application tab, and replace it with another string.

- **Reformat Code**
  Click this to reformat the Siddhi queries in the Siddhi application you are currently creating/editing in the source view.

  This menu option is only visible when you are working in the source view.

- **Auto-Align**
  Click this to horizontally align all the Siddhi components in a Siddhi application that you are creating/editing in the design view.

  This menu option is only visible when you are working in the design view.

**Run menu Items**

The **Run** menu includes the following options.

- **Run**
  Click this to start the Siddhi application in the Run mode. Only saved Siddhi applications can be run.

  This menu option is enabled only when a Siddhi application is being created/edited in the source view.

- **Debug**
  Click this to start the Siddhi application in the Debug mode. Only saved Siddhi applications can be run in this mode.

  This menu option is enabled only when a Siddhi application is being created/edited in the source view.

- **Stop**
  Click this to stop a Siddhi application that is already started in either the Run or Debug mode.

**Side Panel**

**File Explorer**
This provides a view of all the files saved.

_event_simulator_

Simulation can be carried out in two ways:

- Single Simulation
- Feed Simulation

For detailed information about event simulation, see Simulating Events.

_output_console_
This provides feedback on various user activities carried out on the Stream Processor Studio.

**Toolbar**

- **Run icon**
  Click this to start a currently open Siddhi application in the Run mode. This icon is enabled only for saved Siddhi applications.

- **Debug icon**
  Click this to start a currently open Siddhi application in the Debug mode. This icon is enabled only for saved Siddhi applications.

- **Stop icon**
  Click this to stop a Siddhi application that is currently running in either the Run or Debug mode.

- **Revert icon**
  Click this to revert the unsaved changes in the Siddhi application that is currently being created/edited.

**Working with the Design View**

This section provides an overview of the design view of the Stream Processor Studio.

- **Accessing the Design View**
- **Adding Siddhi components**
- **Connecting Siddhi components**
- **Saving, running and debugging Siddhi applications**
Accessing the Design View

To open the design view of the Stream Processor Studio:

1. Start the Stream Processor Studio and log in with your credentials. For detailed instructions, see Stream Processor Studio Overview - Starting Stream Processor Studio.
2. Click New and open a new Siddhi file, or click Open and open an existing Siddhi file.
3. Click Design View to open the Design View.

The design view opens as shown in the example below. It consists of a grid to which you can drag and drop the Siddhi components represented by icons displayed in the left panel to design a Siddhi application.

Adding Siddhi components

To add a Siddhi component to the Siddhi application that you are creating/editing in the design view, click on the relevant icon in the left pane, and then drag and drop it to the grid as demonstrated in the example below.
The following is the complete list of Siddhi components that you can add to the grid of the design view when you create a Siddhi application.

Stream

Table:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
</table>
| ![Stream Icon](image) | A stream represents a logical series of events ordered in time. For a detailed description of streams, see Siddhi Query Guide - Stream. | Once you add a stream component to a Siddhi application in the design view, the Define Stream form appears below the grid. To configure the stream, following information can be specified in this form:

**Stream Name**: A unique name for the stream. This should be specified in title caps, and without spaces (e.g., `ProductionDataStream`).

**Attributes**: Attributes of streams are specified as name and type pairs in the Attributes table.

The above creates a stream with the following configuration:

```java
define stream SweetProductionStream (amount double, name string);
```
| Source | • Sources  
|        | • Projection queries  
|        | • Filter queries  
|        | • Window queries  
|        | • Join queries  |

| Target | • Sinks  
|        | • Projection queries  
|        | • Filter queries  
|        | • Window queries  
|        | • Join queries  |

**Source**

**Icon**

| Description | A source receives events in the specified transport and in the specified format. For more information, see Siddhi Query Guide - Source. |
| Form | Once you add a source component to a Siddhi application in the design view, the Define Source form appears below the grid. To configure the event source, following information can be specified in this form:  
|        | • **Type**: This specifies the transport type via which the events are received. The value should be entered in lower case (e.g., tcp).  
|        | • **Options**: To add this section, you need to click Properties next to Source and select the Options check box. This section allows you to add properties you want to configure for the selected transport. You can click + Option to add a new option. The value must be entered in the `<PROPERTY_NAME>=<PROPERTY_VALUE>` format (e.g., context='SweetProduct onData').  
|        | • **Properties**: To add this section, you need to click Properties next to Source and select the Map check box. This section specifies the format in which the events are received via the source, and allows you to define the mapping so that incoming events can be converted to Siddhi events. The event format is specified in the Name field of the Type section in lower case (e.g., binary). To configure the mapping based on the mapper type you selected, you can click Properties next to Map and select the Type check box and/or the Attribute Mapping check box. |
This source configuration is represented as follows in the source view.

@Source(type = 'tcp', context='SweetProductionData',
@map(type='binary'))

<table>
<thead>
<tr>
<th>Source</th>
<th>No connection can start from another Siddhi component and link to a source because a source is the point from which events selected into the event flow of the Siddhi application start.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>Streams</td>
</tr>
</tbody>
</table>

**Sink**

<table>
<thead>
<tr>
<th>Icon</th>
<th><img src="image" alt="Sink Icon" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>A sink publishes events via the specified transport and in the specified format. For more information, see Siddhi Query Guide - Sink.</td>
</tr>
</tbody>
</table>
Form

Once you add a source component to a Siddhi application in the design view, the Sink Configurations form appears below the grid. To configure the event sink, following information can be specified in this form:

- **Type**: This specifies the transport via which the sink publishes processed events. The value should be entered in lower case (e.g., log).
- **Properties**: To add more properties to configure the source, you can click Properties next to Sink. Then select the **Options** checkbox and/or the **Map** checkbox to add sink options and/or to define the mapping details as required.
  - **Options**: This section allows you to add properties you want to configure for the selected transport. You can click + Option to add a new property. The value must be entered in the `<PROPERTY_NAME>=<PROPERTY_VALUE>` format (e.g., prefix='Sweet Totals').
  - **Map**: In this section, you can add the mapping type (i.e., the format in which the event is published) in the Name sub section. If you want to add more configurations to the mapping, click Properties next to Map, and then select the Options checkbox and/or Attribute Mappings check box. For more information, see Siddhi Query Guide - Sink Mapper.
    - **Options**: This can be used to add additional properties to the @map annotation.
    - **Attribute Mapping**: You can add attributes as key and value pairs or as single attributes.

Example

![Sink Configuration Form](image)

<table>
<thead>
<tr>
<th>Source</th>
<th>Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>N/A</td>
</tr>
</tbody>
</table>

A sink cannot be followed by another Siddhi component because it represents the last stage of the event flow where the results of the processing carried out by the Siddhi application are communicated via the required interface.

Table

<table>
<thead>
<tr>
<th>Icon</th>
<th><img src="image" alt="Table Icon" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>A table is a stored version of an stream or a table of events. For more information, see Siddhi Query Guide - Table.</td>
</tr>
</tbody>
</table>

Form

Once you add a table component to a Siddhi application in the design view, the Table Configuration form appears below the grid. To configure the table, following information can be specified in this form:

- **Name**: This field specifies unique name for the table. This should be specified in title caps, and without spaces (e.g., Product ionDataTable).

- **Attributes**: Attributes of tables are specified as name and type pairs in the Attributes table. To add a new attribute, click +Attribute.
The details entered in the above Table Configuration form creates a table definition as follows:

```sql
define table ShipmentDetails(name string, supplier string, amount double);
```

### Source
- Projection queries
- Window queries
- Filter queries
- Join queries

### Target
- Projection queries
- Window queries
- Filter queries
- Join queries

### Window

**Icon**

![Window Icon](image)

**Description**
This icon represents a window definition that can be shared across multiple queries. For more information, see Siddhi Query Guide - (Defined) Window.

**Form**

Once you add a window component to a Siddhi application in the design view, the Window Configuration form appears below the grid. To configure the window, following information can be specified in this form:

- **Name**: This field specifies a unique name for the window. **PascalCase** is used for window names as a convention.
- **Attributes**: Attributes of windows are specified as name and type pairs in the Attributes table.
- **Function Name**: This specifies the function of the window (i.e., the window type such as `time`, `length`, `frequent` etc.). The window types supported include `time`, `timeBatch`, `timeLength`, `length`, `lengthBatch`, `sort`, `frequent`, `lossyFrequent`, `cron`, `externalTime`, `externalTimeBatch`.
- **Parameters**: This section allows you to define one or more parameters for the window definition based on the window type you entered in the Function Name field.
The details entered in the above Window Configuration form creates a window definition as follows:

```sql
define window FiveMinTempWindow (roomNo int, temp double) time(5 min) output all events;
```

**Source**
- Projection queries
- Window queries
- Filter queries
- Join queries

**Target**
- Projection queries
- Window queries
- Filter queries
- Join queries

**Trigger**

**Icon**

**Description**
a trigger allows you to generate events periodically. For more information, see Siddhi Query Guide - Trigger.

**Form**
Once you add a trigger component to a Siddhi application in the design view, the Trigger Configuration form appears below the grid. To configure the trigger, following information can be entered in this form:

- **Name**: A unique name for the trigger.
- **At**: This specifies the time interval at which the events must be triggered, or the cron expression based on which they are triggered. For more information about cron expressions, see the quartz-scheduler.
The details entered in the above Trigger Configuration form creates a trigger definition as follows:

```sql
define trigger FiveMinTriggerStream at every 5 min;
```

<table>
<thead>
<tr>
<th>Source</th>
<th>N/A</th>
</tr>
</thead>
</table>
| Target       | • Projection queries  
               • Window queries  
               • Filter queries  
               • Join queries |

### Aggregation

<table>
<thead>
<tr>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Icon" /></td>
</tr>
</tbody>
</table>

**Description**

Incremental aggregation allows you to obtain aggregates in an incremental manner for a specified set of time periods. For more information, see Siddhi Query Guide - Incremental Aggregation.

**Before you add an aggregation:**

Make sure that you have already added the stream with the events to which the aggregation is applied is already defined.
Once a connection is made from a stream to an aggregation, move the cursor over the aggregation and click the settings icon on the aggregation (shown below) to open the **Aggregation Configuration** form.

The following information can be entered in this form:

- **Input**: In this section, define a unique name for the aggregation in the **Name** field.
- **Select**: This section specifies the attributes to be included in the aggregation query. In the **Select** field, you can select **All** attributes to perform the aggregation for all the attributes of the stream specified under **Input**, or select **User Defined Attributes** to select specific attributes. If you select **User Defined Attributes**, the **User Defined Attributes** table appears. In this table, you can enter the specific attributes to be selected, and specify the name with which each attribute is included in the aggregation. If an attribute value needs to be processed before it is included in the aggregation, you can enter the required expression to process it.
- **Aggregate By**: Here, you can specify the time values based on which the aggregates are calculated.

The details entered in the above **Aggregation Configuration** form creates an aggregation definition as follows:

```siddhi
define aggregation TradeAggregation
    from TradeStream
    select symbol, avg(price) as avgPrice, sum(price) as total

aggregate every seconds...years;
```

Source: N/A

Target: Join queries

**Function**

**Icon**

The function icon represents [Script in Siddhi Query Language](#). It allows you to write functions in other programming languages and execute them within Siddhi queries. A function component in a Siddhi application is not connected to their Siddhi components in the design UI. However, the configuration of one or more Query components can include a reference to it.
Once you add a function component to a Siddhi application in the design view, the **Function Configuration** appears below the grid. To configure the function, following information can be specified in this form:

- **Name**: A unique name for the function.
- **Script Type**: The language in which the function is written.
- **Return Value**: The data format of the value that is generated as the output via the function.
- **Script Body**: This is a free text field to write the function in the specified script type.

### Example

![Function Configuration Form](image)

The details entered in the above **Function Configuration** form creates a function definition as follows:

```javascript
define function concatFN[JAVASCRIPT] return string {
    var str1 = data[0];
    var str2 = data[1];
    var str3 = data[2];
    var response = str1 + str2 + str3;
    return response;
}
```

### Projection Query

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
</table>
| ![Projection Icon](image) | **Before you add a projection query:**

You need to add and configure the following:

- The input stream with the events to be processed by the query.
- The output stream to which the events processed by the query are directed.

This icon represents a query to project the events in an input stream to an output stream. This involves selecting the attributes to be included in the output, renaming attributes, introducing constant values, and using mathematical and/or logical expressions. For more information, see [Siddhi Query Guide - Query Projection](#).
Once you connect the query to an input stream (source) and an output stream (target), move the cursor over the query and click the settings icon on the projection query (shown below) to open the **Query Configuration** form.

The following information needs to be entered in this form:

- **Input**: This section specifies the stream to be considered as the input stream with the events to which the query needs to be applied. The input stream connected to the query as the source is automatically displayed.
- **Select**: This section specifies the attributes to be included in the output. In the **Select** field, you can select **All Attributes** to select all the attributes of the events, or select **User Defined Attributes** to select specific attributes, and the expressions to convert them as required to generate output events.
- **Output**: This section specifies the action to be performed on the output event. The fields to be configured in this section are as follows:
  - **Operation**: This field specifies the operation to be performed on the generated output event (e.g., **Insert** to insert events to a selected stream/table/window).
  - **Into**: This field specifies the stream/table/window in which the operation specified need to be performed.
  - **For**: This field specifies whether the operation needs to be performed for all output events, only current events or for only expired events.
The details entered in the above **Query Configuration** form creates a query definition as follows:

```sql
from TradeStream
select symbol, avg(price) as averagePrice, sum(amount) as total
insert all events into OutputStream;
```
Once you connect the query to an input stream (source) and an output stream (target), move the cursor over the query and click the settings icon on the filter query (shown below) to open the Query Configuration form.

The following information needs to be entered in this form:

- **Input**: This section specifies the stream to be considered as the input stream with the events to which the query needs to be applied. The input stream connected to the query as the source is automatically displayed.
- **Select**: This section specifies the attributes to be included in the output. In the Select field, you can select All Attributes to select all the attributes of the events, or select User Defined Attributes to select specific attributes, and the expressions to convert them as required to generate output events.
- **Output**: This section specifies the action to be performed on the output event. The fields to be configured in this section are as follows:
  - **Operation**: This field specifies the operation to be performed on the generated output event (e.g., Insert to insert events to a selected stream/table/window).
  - **Into**: This field specifies the stream/table/window in which the operation specified need to be performed.
  - **For**: This field specifies whether the operation needs to be performed for all output events, only current events or for only expired events.
- **Stream Handlers**: This section can be used to specify the filter criterion. To do this, select Filter in the Stream Handler field, and then enter the filter condition in the Filter Condition field.

A Siddhi application can have multiple stream handlers. To add another stream handler, click the + Stream Handler. Multiple functions, filters and windows can be defined within the same form as stream handlers.

The details entered in the above Query Configuration form creates a query definition with a filter as follows:

```siddhi
from TradeStream[sum(amount) > 10000]
select symbol, avg(price) as averagePrice, sum(amount) as total
insert all events into OutputStream;
```
Window queries include a window to select a subset of events to be processed based on a specific criterion. For more information, see Siddhi Query Guide - (Defined) Window.

Before you add a window query:
You need to add and configure the following:

- The input stream with the events to be processed by the query.
- The output stream to which the events processed by the query are directed.

A Siddhi application can have multiple stream handlers. To add another stream handler, click the + Stream Handler. Multiple functions, filters and windows can be defined within the same form as stream handlers.
The details entered in the above **Query Configuration** form creates a query definition with a window as follows:

```sql
from TradeStream#window.time(1 month)
select symbol, avg(price) as averagePrice, sum(amount) as total
insert all events into OutputStream;
```

<table>
<thead>
<tr>
<th>Source</th>
<th>A window query can have only one source at a given time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streams</td>
<td></td>
</tr>
<tr>
<td>Tables</td>
<td></td>
</tr>
<tr>
<td>Triggers</td>
<td></td>
</tr>
<tr>
<td>Windows</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target</th>
<th>Streams</th>
<th>Tables</th>
<th>Triggers</th>
<th>Windows</th>
</tr>
</thead>
</table>

| Join Query | |
|------------|--
| **Icon** | ![Join Icon](image)

<table>
<thead>
<tr>
<th><strong>Description</strong></th>
<th>A join query derives a combined result from two streams in real-time based on a specified condition. For more information, see Siddhi Query Guide - Join.</th>
</tr>
</thead>
</table>
Once you connect two Siddhi components to the join query as sources and another Siddhi component as the target, move the cursor over the join query and click the settings icon on the projection query (shown below) to open the **Join Query Configuration** form.

The following information needs to be entered in this form:

**Configurations common for both sources**

- **Select**: This section specifies the attributes to be included in the output. In the **Select** field, you can select **All Attributes** to select all the attributes of the events, or select **User Defined Attributes** to select specific attributes, and the expressions to convert them as required to generate output events.
- **Output**: This section specifies the action to be performed on the output event. The fields to be configured in this section are as follows:
  - **Operation**: This field specifies the operation to be performed on the generated output event (e.g., **Insert** to insert events to a selected stream/table/window).
  - **Into**: This field specifies the stream/table/window in which the operation specified need to be performed.
  - **For**: This field specifies whether the operation needs to be performed for all output events, only current events or for only expired events.

The stream handlers need to be separately configured for the left source and the right source of the join.

**Configurations for each source**

The following configurations are entered under **Left Source** and **Right Source**.

- **Input**: This section specifies the input. This can be a stream, aggregation, window or a table for each source.
- **Stream Handlers**: This section defines the functions, filters, and windows that can be included in the query to extract events from the left/right source.

**Example**

A join query is configured as follows:

- **Common configurations**

![Join Query Configuration form](image)
Left source configuration

Input
From
TempStream
Attributes { deviceId: LONG, roomNo: INT, temp: DOUBLE }

Stream Handlers
Stream Handler
Stream Handler
Filter
Filter Condition
temp > 30.0

Stream Handler
Window

Window Name
time

Parameters
Parameter Name
1 min

+ Attribute

+ Stream Handler

As
Name
T

Is Unidirectional

Join
Type
join
The above configurations result in creating the following join query.
from TempStream[temp > 30.0]#window.time(1 min) as T
join RegulatorStream[isOn == false]#window.length(1) as R
on T.roomNo == R.roomNo
select T.roomNo, R.deviceID, 'start' as action
insert into RegulatorActionStream;

A join query must always be connected to two sources, and at least one of them must be a defined stream/trigger/window.

- Streams
- Tables
- Aggregations
- Windows

A join query must always be connected to a single target.

- Streams
- Tables
- Windows

**Pattern Query**

**Icon**

![Pattern Query Icon]

**Description**

**Before you add a pattern query:**

You need to add and configure the following:

- The input stream with the events to be processed by the query.
- The output stream to which the events processed by the query are directed.

A pattern query detects patterns in events that arrive overtime. For more information, see Siddhi Query Guide - Patterns.
Once you connect the query to an input stream (source) and an output stream (target), move the cursor over the query and click the settings icon on the pattern query component (shown below) to open the **Pattern Query Configuration** form.

The following information is configured in the **Pattern Query Configuration** form:

- **Name**: This field specifies a unique name for the pattern query.
- **Annotations**: One or more annotations to be included in the pattern query can be entered in this section. To expand this section to add annotations, click **Properties** and select the **Add Annotations** check box. As a result, a single field appears to add an annotation as shown below. To add more annotation fields, click + **Annotation**.

- **Input**: This section displays the input streams that are currently connected to the pattern query as sources.
- **Conditions**: This section defines the conditions based on which patterns are identified. This involves specifying a unique ID and the input stream considered for each condition. Multiple conditions can be added. To add a new condition, click + **Condition**.
- **Logic**: This section defines the detailed logic based on which the pattern query output is generated.
The above configuration results in creating the following query.

```sql
from every (el=MaterialSupplyStream) -> not MaterialConsumptionStream[name == el.name and amount == el.amount]
  for 15 sec
select el.name, el.amount
insert into ProductionDelayAlertStream;
```
Sequence Query

**Description**

**Before you add a sequence query:**
You need to add and configure the following:

- The input stream with the events to be processed by the query.
- The output stream to which the events processed by the query are directed.

A sequence query detects sequences in event occurrences over time. For more information, see Siddhi Query Guide - Sequence.

**Form**

Once you connect the query to an input stream (source) and an output stream (target), move the cursor over the query and click the settings icon on the sequence query component (shown below) to open the **Sequence Query Configuration** form.

The following information is configured in the **Sequence Query Configuration** form:

- **Name**: This field specifies a unique name for the sequence query.
- **Annotations**: One or more annotations to be included in the sequence query can be entered in this section. To expand this section to add annotations, click **Properties** and select the **Add Annotations** check box. As a result, a single field appears to add an annotation as shown below. To add more annotation fields, click + **Annotation**.

- **Input**: This section displays the input streams that are currently connected to the sequence query as sources.
- **Conditions**: This section defines the conditions based on which sequences are identified. This involves specifying a unique ID and the input stream considered for each condition. Multiple conditions can be added. To add a new condition, click + **Condition**.
- **Logic**: This section defines the detailed logic based on which the sequence query output is generated.
The above configuration results in creating the following query:

```sql
from every e1 = SweetProductionStream, e2 = SweetProductionStream[e1.amount > amount and (timestamp - e1.timestamp) < 10 * 60000], e3 = SweetProductionStream[timestamp - e1.timestamp > 10 * 60000 and e1.amount > amount]
select e1.name as name, e1.amount as initialAmount, e2.amount as finalAmount, e2.timestamp as timestamp
insert into DecreasingTrendAlertStream;
```
Partitions divide streams and queries into isolated groups in order to process them in parallel and in isolation. For more information, see Siddhi Query Guide - Partition.

**Form**

Once the stream to be partitioned is connected as a source to the partition, move the cursor over the partition and click the settings icon to open the **Partition** form.

In this form, you can enter expressions to convert the attributes of the stream selected to be partitioned, to create output events.

**Example**

The above configuration creates the following partition query.

```sql
partition with ( roomNo >= 1030 as 'serverRoom' or
    roomNo < 1030 and roomNo >= 330 as 'officeRoom'
  or
  roomNo < 330 as 'lobby' of TempStream)

begin
  from TempStream#window.time(10 min)
  select roomNo, deviceID, avg(temp) as avgTemp
  insert into AreaTempStream
end;
```

**Source** Streams

**Target** N/A
Connecting Siddhi components

In order to define how the Siddhi components in a Siddhi application interact with each other to process events, you need to define connections between Siddhi components. A connection is defined by drawing an arrow from one component to another by dragging the cursor as demonstrated below.

![Diagram of Siddhi components and connections]

Saving, running and debugging Siddhi applications

To save a Siddhi application that you created in the design view, you need to switch to the source view. You also need to switch to the source view to run or debug a Siddhi application. For more information, see the following sections:

- Stream Processor Studio Overview
- Debugging a Siddhi Application

Creating a Siddhi Application

Siddhi applications are files that define the Siddhi logic to process the events sent to WSO2 SP. They are written in the Siddhi Query Language using the Stream Processor Studio tool shipped with WSO2 SP.

A Siddhi file contains the following configurations:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream</td>
<td>A logical series of events ordered in time with a uniquely identifiable name, and set of defined attributes with specific data types defining its schema.</td>
</tr>
<tr>
<td>Source</td>
<td>This consumes data from external sources (such as TCP, Kafka, HTTP, etc) in the form of events, then converts each event (that can be in XML, JSON, binary, etc. format) to a Siddhi event, and passes that to a stream for processing.</td>
</tr>
<tr>
<td>Sink</td>
<td>This takes events arriving at a stream, maps them to a predefined data format (such as XML, JSON, binary, etc), and publishes them to external endpoints (such as E-mail, TCP, Kafka, HTTP, etc).</td>
</tr>
</tbody>
</table>
| Executional Element | An executional element can be one of the following:
  - Stateless query: Queries that only consider currently incoming events when generating an output. e.g., filters
  - Stateful query: Queries that consider both currently incoming events as well as past events when generating an output. e.g., windows, sequences, patterns, etc.
  - Partitions: Collections of stream definitions and Siddhi queries separated from each other within a Siddhi application for the purpose of processing events in parallel and in isolation. |

A Siddhi application can be created from the source view or the design view of the WSO2 SP Stream Processor Studio.

Creating a Siddhi application in the source view

To create a Siddhi application via the source view of the WSO2 SP Stream Processor Studio, follow the steps below:

1. Start WSO2 SP in the editor mode and access the Stream Processor Studio. For detailed instructions, see Starting Stream Processor Studio. The Stream Processor Studio opens as shown below.
1. Click **New** to start defining a new Siddhi application. A new file opens as shown below.

2. Add the following sample Siddhi application to the file.

```siddhi
@App:name("SweetProductionAnalysis")
@Source(type = 'tcp', context='SweetProductionData',
    @map(type='binary'))
define stream SweetProductionStream (name string, amount double);

@sink(type='log', @map(type='json'))
define stream ProductionAlertStream (name string, amount double);

from SweetProductionStream
select *
insert into ProductionAlertStream;
```

3. Note the following in this Siddhi application:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream</td>
<td>This stream contains two stream configurations:</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>- SweetProductionStream</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>define stream SweetProductionStream (name string, amount double);</td>
</tr>
<tr>
<td></td>
<td>This is the input stream that defines the schema based on which events are selected to be processed by the SweetProductionAnalysis Siddhi application. Events received via the source in this application are directed to this stream.</td>
</tr>
<tr>
<td></td>
<td>- ProductionAlertStream</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>define stream ProductionAlertStream (name string, amount double);</td>
</tr>
<tr>
<td></td>
<td>This is the output stream from which the sink configured in this application takes events to be published as the output.</td>
</tr>
<tr>
<td>Source</td>
<td>This source configuration has the following sections:</td>
</tr>
<tr>
<td></td>
<td>- @Source(type = 'tcp', context='SweetProductionData', @map(type='binary'))</td>
</tr>
<tr>
<td></td>
<td>This configuration defines tcp as the transport via which events are received to be processed by the SweetProductionAnalysis Siddhi application.</td>
</tr>
<tr>
<td></td>
<td>- @map(type='binary'))</td>
</tr>
<tr>
<td></td>
<td>This configuration defines the input mapping. In this scenario, Binary Mapper is used which converts input events into binary events and feeds them into siddhi.</td>
</tr>
<tr>
<td>Sink</td>
<td>This source configuration has the following sections:</td>
</tr>
<tr>
<td></td>
<td>- @sink(type='log', @map(type='json'))</td>
</tr>
<tr>
<td></td>
<td>This configuration defines log as the transport via which the processed events are published from the ProductionAlertStream output stream. Log sink simply publishes events into the console.</td>
</tr>
<tr>
<td></td>
<td>- @map(type='json'))</td>
</tr>
<tr>
<td></td>
<td>This configuration defines the output mapping. Events are published with the json mapping type. Json mapper converts the events in the ProductionAlertStream to the Json format.</td>
</tr>
<tr>
<td>Executional Elements</td>
<td>from SweetProductionStream select * insert into ProductionAlertStream;</td>
</tr>
<tr>
<td></td>
<td>This is where the logic of the siddhi app is defined. In this scenario, all the events received in the SweetProductionStream input stream are inserted into the ProductionAlertStream output stream.</td>
</tr>
</tbody>
</table>
4. To save this Siddhi application, click **File**, and then click **Save**. By default, Siddhi applications are saved in the `<SP_HOME>/wso2/editor/deployment/workspace` directory.

5. To export the Siddhi application to your preferred location, click **File**, and then click **Export File**.

6. To see a graphical view of the event flow you defined in your Siddhi application, click **Design View**.

Creating a Siddhi application in the design view

To create a Siddhi application via the design view of the WSO2 SP Stream Processor Studio, follow the steps below:

1. Start WSO2 SP in the editor mode and access the Stream Processor Studio. For detailed instructions, see **Starting Stream Processor Studio**. The Stream Processor Studio opens as shown below.
2. Click New to start defining a new Siddhi application. A new file opens as shown below.

3. To open the design view, click Design View.

4. To define the input stream into which the events to be processed via the Siddhi application should be received, drag and drop the stream icon (shown below) into the grid.

As a result, the Stream Configuration form appears below the grid as shown below.
Fill this form as follows to define a stream named `SweetProductionStream` with two attributes named `name` and `amount`:

a. In the **Name** field, enter `SweetProductionStream`.

b. In the **Attributes** table, enter two attributes as follows. You can click **+Attribute** to add a new row in the table to define a new attribute.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
</tr>
<tr>
<td>amount</td>
<td>double</td>
</tr>
</tbody>
</table>

c. Click **Submit** to save the new stream definition. As a result, the stream is displayed on the grid with the `SweetProductionStream` label as shown below.
5. To define the output stream to which the processed events need to be directed, drag and drop the stream icon again. Place it after the SweetProductionStream stream. This stream should be named ProductionAlertStream and have the following attributes.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
</tr>
<tr>
<td>totalProduction</td>
<td>long</td>
</tr>
</tbody>
</table>

6. To add the source from which events are received, drag and drop the source icon (shown below) into the grid.

As a result, the Source Configuration form appears below the grid. Enter values in this form as follows to configure a source via which events can be received.

a. In the Name field, enter a name for the source. Then click **Save** to add the source.

b. To configure the source by entering values for the rest of the parameters, the source must be connected to a stream. This tcp source needs to be connected to the SweetProductionStream input stream so that the events received via TCP can be directed there. To connect the source to the stream, draw an arrow from the source to the stream by dragging the cursor as demonstrated below.

Then open the Source Configuration form again by clicking the **Settings** icon of the tcp source.

c. Click **Properties**. Based on the properties you want to configure for the source, select the relevant check box to select the required annotation type. In this example, let's select both **Options** and **Map**.
As a result, the **Options** and **Map** sections appear as shown below.

Now the **Source Configuration** form looks as follows:

i. Enter `context='SweetProductionData'` in the **Options** field to indicate the context. You can add multiple options by clicking + **Option** to add more fields under **Options**.

ii. For this example, assume that events are received in the **binary** format. To allow this, enter **binary** in the **Name** field under **Type** in the **Map** section.

Now the **Source Configuration** form looks as follows:
d. Click **Save** to save the source configuration.

7. To add a query that defines the execution logic, drag and drop the projection query icon (shown below) to the grid.

8. The query uses the events in the `SweetProductionStream` input stream as inputs and directs the processed events (which are its output) to the `ProductionAlertStream` output stream. Therefore, create two connections as demonstrated below.

9. To define the execution logic, move the cursor over the query in the grid, and click on the settings icon that appears.
This opens the **Query Configuration** form. Enter information in it as follows:

<table>
<thead>
<tr>
<th>Expression</th>
<th>As</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
<td>When creating the output events, the name attribute is output with the same attribute name.</td>
</tr>
<tr>
<td>sum(amount)</td>
<td>amount</td>
<td>With each event that arrives in the SweetProductionStream stream, the query calculates the sum for the amount attribute, and outputs the result with amount as the attribute value.</td>
</tr>
</tbody>
</table>

10. To add a sink to publish the output events that are directed to the ProductionAlertStream output stream, drag and drop the sink icon (shown below) into the grid.

As a result, the **Sink Configuration** form appears below the grid. Enter values in this form as follows to configure a sink via which events can be published.

a. In the **Name** field, enter a name for the sink.

b. Click **Properties**. Based on the properties you want to configure for the sink, select the relevant check box to select the required annotation type. In this example, assume that the events need to be published in the **json** format. Therefore, let's select the **Map** check box, and enter `type=json` to specify `json` as the mapping type.

11. Connect the sink you added to the ProductionAlertStream output stream by dragging and dropping the cursor from the ProductionAlertStream output stream to the sink.

12. To align the Siddhi components that you have added to the grid, click **Edit** and then click **Auto-Align**. As a result, all the components are
horizontally aligned as shown below.

13. Click **Source View**. The siddhi application is displayed as follows.

![Siddhi Application Source View](image)

14. Click **File** and then click **Save as**. The **Save to Workspace** dialog box appears. In the **File Name** field, enter **SweetProductionAnalysis** and click **Save**.

![Save to Workspace Dialog](image)

### Testing a Siddhi Application

WSO2 Stream Processor Studio allows the following tasks to be carried out to ensure that the Siddhi applications you create and deploy are validated before they are run in an actual production environment.

- Validate Siddhi applications that are written in the Stream Processor Studio.
- Run Siddhi applications that were written in the Stream Processor Studio in either Run or Debug mode.
- Simulate events to test the Siddhi applications and analyze events that are received and sent. This allows you to analyze the status of each query within a Siddhi application at different execution points.

#### Validating a Siddhi application

To validate a Siddhi application, follow the procedure below:

1. Start and access the Stream Processor Studio. For detailed instructions, see **Starting Stream Processor Studio**.
2. In this example, let's use an existing sample as an example. Click on the **ReceiveAndCount** sample to open it.
3. Sample opens in a new tab. This sample does not have errors, and therefore, no errors are displayed in the editor. To create an error for demonstration purposes, change the `count()` function in the `query1` to `totalCountNew` as shown below.

```siddhi
@info(name='query1') from SweetProductionStream select count() as amount
insert into ProductionAlertStream;
```

Now, the editor indicates that there is a syntax error. If you move the cursor over the error icon, it indicates that **totalCountNew** is an invalid function name as shown below.
Running or debugging a Siddhi application

You can run or debug a siddhi application to verify whether the logic you have written is correct. To start a Siddhi application in the run/debug mode, follow the procedure below:

1. Start and access the Stream Processor Studio. For detailed instructions, see Starting Stream Processor Studio.
2. For this example, click the existing sample ReceiveAndCount. It opens in a new untitled tab.
3. Save the Siddhi file so that you can run it in the Run or Debug mode. To save it, click File => Save. Once the file is saved, you can see the Run and Debug menu options enabled as shown below.
4. Go to File and click Save. (Default saved in default workspace folder). Once saved you can see the above options are now enabled.

   a. To start the application in Run mode, click Run => Run. This logs the following output in the console.

   b. Start the application in the Debug mode, click Run => Debug. As a result, the following message is logged in the console. You can also note that another console tab is opened with debug options.
5. To create an error for demonstration purposes, change the `count()` function in the `query1` query to `countNew()`, and save. Then click Run => Run. As a result, the following output is logged in the console.

```
ReceiveAndCount.siddhi - Siddhi App ReceiveAndCount is in faulty state.
```

**Simulating events**

This section demonstrates how to test a Siddhi application via event simulation. There are multiple methods to simulate events. In this example, you can use single simulation. For more information about all available methods, see [Simulating Events](#).

To simulate a single event in order to check the status of your Siddhi queries, follow the procedure below:

1. Run the `ReceiveAndCount` Siddhi application.
2. Click the following icon for event simulation.

It opens the left panel for event simulation as follows.

<table>
<thead>
<tr>
<th>Single Simulation</th>
<th>Feed Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S1</strong> +</td>
<td></td>
</tr>
</tbody>
</table>

**Siddhi App Name**

- Please Select a Siddhi App -

**Stream Name**

<table>
<thead>
<tr>
<th>Timestamp(ms) - optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current System Time</td>
</tr>
</tbody>
</table>

3. In the **Siddhi App Name** field, select the name of the application for which you want to simulate events. For this example, select `ReceiveAndCount`.

4. In the **Stream Name** field, select the stream defined in the Siddhi application for which you want to simulate events. For this example, select `SweetProductionStream`.

5. Enter values for the attributes of the selected stream that appear in the pane, and click **Send**.

6. In this Siddhi application, you have a sink of the `log` type connected to the `TotalCountStream` output stream. Therefore, once the events are received at the input stream, a log is printed in the output console as shown below.

```
[2017-12-21 20:36:58.339] INFO {org.wso2.siddhi.core.stream.output.sink.LogSink} - ReceiveAndCount : TotalCountStream : Event(timestamp=1513868818338, dowDay=2, isExpired=0 false)
```
Simulating Events

Simulating events involves simulating predefined event streams. These event stream definitions have stream attributes. You can use event simulator to create events by assigning values to the defined stream attributes and send them as events. This is useful for debugging and monitoring the event receivers and publishers, execution plans and event formatters.

<table>
<thead>
<tr>
<th>Function</th>
<th>REST API</th>
</tr>
</thead>
</table>
| Saving a simulation configuration | • **Single Event Simulation**: POST http://<SP_HOST>:<API_PORT>/simulation/single  
                               • **Multiple Event Simulation**: POST http://<SP_HOST>:<API_PORT>/simulation/feed |
| Editing a simulation configuration | • **Single Event Simulation**: PUT http://<SP_HOST>:<API_PORT>/simulation/single  
                               • **Multiple Event Simulation**: PUT http://<SP_HOST>:<API_PORT>/simulation/feed |
| Deleting a simulation configuration | • **Single Event Simulation**: DELETE http://<SP_HOST>:<API_PORT>/simulation/single  
                               • **Multiple Event Simulation**: DELETE http://<SP_HOST>:<API_PORT>/simulation/feed |
| Retrieving a simulation configuration | • **Single Event Simulation**: GET http://<SP_HOST>:<API_PORT>/simulation/single  
                               • **Multiple Event Simulation**: GET http://<SP_HOST>:<API_PORT>/simulation/feed |
| Uploading a CSV file             | POST http://<SP_HOST>:<API_PORT>/simulation/feed                           |
| Editing and uploaded CSV file    | PUT -F 'file=~/[path to csv file]'  http://<SP_HOST>:<API_PORT>/simulation/files/{fileName}?fileName={fileName} |
| Deleting an uploaded CSV file    | DELETE http://<SP_HOST>:<API_PORT>/simulation/files/{fileName}             |
| Pausing an event simulation      | POST http://<SP_HOST>:<API_PORT>/simulation/feed/{simulationName}?action=pause |
| Resuming an event simulation     | POST http://<SP_HOST>:<API_PORT>/simulation/feed/{simulationName}?action=resume |
| Stopping an event simulation     | DELETE http://<SP_HOST>:<API_PORT>/simulation/feed/{simulationName}        |

The following sections cover how events can be simulated.

- Saving a simulation configuration
- Editing a simulation configuration
- Deleting a simulation configuration
- Retrieving a simulation configuration
- Uploading a CSV file
- Editing an uploaded CSV file
- Deleting an uploaded CSV file
- Pausing an event simulation
- Resuming an event simulation
- Stopping an event simulation

Saving a simulation configuration

To simulate events for WSO2 SP, you should first save the event simulator configuration in the <SP_HOME>/deployment/simulator/simulationConfigs directory by sending a POST request to a REST API as described below.

**REST API**

The REST API to be called depends on the type of event simulation you are carrying out as shown in the table below.

<table>
<thead>
<tr>
<th>Event Simulation Type</th>
<th>REST API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulating a single event</td>
<td>POST http://&lt;SP_HOST&gt;:&lt;API_PORT&gt;/simulation/single</td>
</tr>
<tr>
<td>Simulating multiple events</td>
<td>POST http://&lt;SP_HOST&gt;:&lt;API_PORT&gt;/simulation/feed/</td>
</tr>
</tbody>
</table>

*Sample cURL command*
curl -X POST \
http://localhost:9390/simulation/feed/ \
-H 'content-type: text/plain' \
-d '{
  "properties": {
    "simulationName": "simulationPrimitive",
    "startTimestamp": "",
    "endTimestamp": "",
    "noOfEvents": "",
    "description": "",
    "timeInterval": "1000"
  },
  "sources": [
    {
      "siddhiAppName": "TestExecutionPlan",
      "streamName": "FooStream",
      "timestampInterval": "1000",
      "simulationType": "RANDOM_DATA_SIMULATION",
      "attributeConfiguration": [
        {
          "type": "PRIMITIVE_BASED",
          "primitiveType": "STRING",
          "length": "5"
        },
        {
          "type": "PRIMITIVE_BASED",
          "primitiveType": "INT",
          "min": "0",
          "max": "999"
        }
      ]
    }
  ]
}'

Sample output
{
  "status": "CREATED",
  "message": "Successfully uploaded simulation configuration 'simulationPrimitive'"
}

REST API response
- 200 if the simulation configuration is successfully saved.
- 409 if a simulation configuration with the specified name already exists.
- 400 if the configuration provided is not in a valid JSON format.

For descriptions of the HTTP status codes, see HTTP Status Codes.

Editing a simulation configuration

To edit a simulation configuration that is already saved, a PUT request should be sent to a REST API as explained below.
**REST API**

The REST API to be called depends on the type of event simulation you are carrying out as shown in the table below.

<table>
<thead>
<tr>
<th>Event Simulation Type</th>
<th>REST API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulating a single event</td>
<td>PUT http://&lt;SP_HOST&gt;:&lt;API_PORT&gt;/simulation/single</td>
</tr>
<tr>
<td>Simulating a multiple events</td>
<td>PUT http://&lt;SP_HOST&gt;:&lt;API_PORT&gt;/simulation/feed/{feed name}</td>
</tr>
</tbody>
</table>

**Sample cURL command**

```bash
curl -X PUT \
    http://localhost:9390/simulation/feed/simulationPrimitive \
    -H 'content-type: text/plain' \
    -d '
    "properties": {
        "simulationName": "updatedSimulationPrimitive",
        "startTimestamp": "",
        "endTimestamp": "",
        "noOfEvents": "10",
        "description": "Updating the simulation configuration",
        "timeInterval": "1000"
    },
    "sources": [
        {
            "siddhiAppName": "TestExecutionPlan",
            "streamName": "FooStream",
            "timestampInterval": "1000",
            "simulationType": "RANDOM_DATA_SIMULATION",
            "attributeConfiguration": [
                {
                    "type": "PRIMITIVE_BASED",
                    "primitiveType": "STRING",
                    "length": "5"
                },
                {
                    "type": "PRIMITIVE_BASED",
                    "primitiveType": "INT",
                    "min": "0",
                    "max": "999"
                }
            ]
        }
    ]
    
}''
```

**Sample output**

```json
{
    "status": "OK",
    "message": "Successfully updated simulation configuration 'simulationPrimitive'."
}
```

**REST API response**
• 200 if the simulation configuration is successfully updated.
• 404 if the file specified does not exist in the `<SP_HOME>/wso2/editor/deployment/simulation-configs` directory.
• 400 if the file specified is not a CSV file, or if the file does not exist in the path specified.
• 403 if the size of the file specified exceeds the maximum size allowed.

For descriptions of the HTTP status codes, see HTTP Status Codes.

Deleting a simulation configuration

To delete an event simulation file that is already saved in the `<SP_HOME>/wso2/editor/deployment/simulation-configs` directory, a DELETE request should be sent to a REST API as explained below.

**REST API**

The REST API to be called depends on the type of event simulation you are carrying out as shown in the table below.

<table>
<thead>
<tr>
<th>Event Simulation Type</th>
<th>REST API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulating a single event</td>
<td>DELETE http://&lt;SP_HOST&gt;:&lt;API_PORT&gt;/simulation/single</td>
</tr>
<tr>
<td>Simulating a multiple events</td>
<td>DELETE http://&lt;SP_HOST&gt;:&lt;API_PORT&gt;/simulation/feed/</td>
</tr>
</tbody>
</table>

**Sample cURL command**

```
curl -X DELETE 'http://localhost:9390/simulation/feed/simulationPrimitive'
```

**Sample output**

```
{
    "status": "OK",
    "message": "Successfully deleted simulation configuration 'simulationPrimitive'"
}
```

**REST API response**

• 200 if the simulation configuration is successfully deleted.
• 404 if the file specified does not exist in the `<SP_HOME>/wso2/editor/deployment/simulation-configs` directory.

For descriptions of the HTTP status codes, see HTTP Status Codes.

Retrieving a simulation configuration

To view a simulation configuration saved in the `<SP_HOME>/wso2/editor/deployment/simulation-configs` directory via the CLI, a GET request should be sent to a REST API as explained below.

**REST API**

The REST API to be called depends on the type of event simulation you are carrying out as shown in the table below.

<table>
<thead>
<tr>
<th>Event Simulation Type</th>
<th>REST API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulating a single event</td>
<td>GET http://&lt;SP_HOST&gt;:&lt;API_PORT&gt;/simulation/single</td>
</tr>
<tr>
<td>Simulating a multiple events</td>
<td>GET http://&lt;SP_HOST&gt;:&lt;API_PORT&gt;/simulation/feed/</td>
</tr>
</tbody>
</table>

**Sample cURL command**

```
curl -X GET 'http://localhost:9390/simulation/feed/simulationPrimitive'
```

**Sample output**
REST API Response

- 200 if the simulation configuration is successfully retrieved.
- 404 if the file specified does not exist in the `<SP_HOME>/wso2/editor/deployment/simulation-configs` directory.

For descriptions of the HTTP status codes, see HTTP Status Codes.

Uploading a CSV file

To simulate events from a CSV file, the required CSV file needs to exist in the `<SP_HOME>/wso2/editor/deployment/csv-files` directory.

REST API

A POST request should be sent to the following API.

POST http://<SP_HOST>:<API_PORT>/simulation/feed

Sample cURL command

curl -X POST \
http://localhost:9390/simulation/feed/ \
-H 'content-type: text/plain' \
-d '{
  "properties": {
    "simulationName": "FeedSimulationTest",
    "startTimestamp": "",
    "endTimestamp": "",
    "noOfEvents": "",
    "description": "",
    "timeInterval": "1000"
  },
  "sources": [
    {
      "siddhiAppName": "TestExecutionPlan",
      "streamName": "FooStream",
      "timestampInterval": "1000",
      "simulationType": "CSV_SIMULATION",
      "fileName": "myEvents.csv",
      "delimiter": ",",
      "isOrdered": true,
      "indices": "0,1"
    }
  ]
}'

Sample output
REST API response

- 200 if the CSV file is successfully uploaded.
- 409 if a CSV file with the file name specified already exists in the `<SP_HOME>/wso2/editor/deployment/csv-files` directory.
- 400 if the specified file is not a CSV file or if the specified file path is not valid.
- 403 if the size of the file specified exceeds the maximum file size allowed.

For descriptions of the HTTP status codes, see [HTTP Status Codes](#).

Editing an uploaded CSV file

This section explains how to edit a CSV file that is already uploaded to the `<SP_HOME>/wso2/editor/deployment/csv-files` directory.

**REST API**

A PUT request should be sent to the following API.

```
PUT -F 'file=@/{path to csv file}'
http://<SP_HOST>:<API_PORT>/simulation/files/{fileName}?fileName={fileName}
```

**Sample cURL command**

```
    curl -X PUT -F 'file=/home/nadeeka/Desktop/editedMyEvents.csv'
    http://localhost:9390/simulation/files/myEvents.csv?fileName=myEvents.csv
```

**Sample output**

```
    {
        "status": "OK",
        "message": "Successfully updated CSV file 'myEvents.csv' with file ' editedMyEvents.csv'."
    }
```

**REST API response**

- 200 if the CSV file is successfully updated.
- 404 if the specified CSV file does not exist in the `<SP_HOME>/deployment/simulator/csvFiles` directory.

For descriptions of the HTTP status codes, see [HTTP Status Codes](#).

Deleting an uploaded CSV file

This section explains how to delete a CSV file that is already uploaded to the `<SP_HOME>/wso2/editor/deployment/csv-files` directory.

**REST API**

A DELETE request should be sent to the following API.

```
DELETE http://<SP_HOST>:<API_PORT>/simulation/files/{fileName}
```

**Sample cURL command**

```
    curl -X DELETE http://localhost:9390/simulation/files/myEvents.csv
```

**Sample output**

```
    {
        "status": "OK",
        "message": "Successfully deleted file 'myEvents.csv'"
    }
```
Pausing an event simulation

This section explains how to pause an event simulation that has already started.

REST API

A POST request should be sent to the following API.

POST http://<SP_HOST>:<API_PORT>/simulation/feed/{simulationName}/?action=pause

Sample cURL command

```bash
curl -X POST
http://localhost:9390/simulation/feed/simulationPrimitive/?action=pause
```

Sample output

```json
{
  "status": "OK",
  "message": "Successfully paused event simulation 'simulationPrimitive'."
}
```

REST API response

- 200 if the event simulation is successfully paused.
- 409 if the event simulation is already paused.

Resuming an event simulation

This section explains how to resume an event simulation that has already paused.

REST API

A POST request should be sent to the following API

POST http://<SP_HOST>:<API_PORT>/simulation/feed/{simulationName}/?action=resume

Sample cURL command

```bash
curl -X POST
http://localhost:9390/simulation/feed/simulationPrimitive/?action=resume
```

Sample output

```json
{
  "status": "OK",
  "message": "Successfully resumed event simulation 'simulationPrimitive'."
}
```

REST API response

- 200 if the event simulation is successfully resumed.

Stopping an event simulation

This section explains how to stop an event simulation.

REST API
A POST request should be sent to the following API
POST http://<SP_HOST>:<API_PORT>/simulation/feed/{simulationName}/?action=stop

Sample cURL command

curl -X POST
http://localhost:9390/simulation/feed/simulationPrimitive/?action=stop

Sample output

{  "status": "OK",  "message": "Successfully stopped event simulation 'simulationPrimitive'."}

REST API response
  • 200 if the event simulation is successfully stopped.

Simulating a Single Event
This section demonstrates how to simulate a single event to be processed via WSO2 SP.

Prerequisites
Before simulating events, a Siddhi application should be deployed.

Simulating an event
To simulate a single event, follow the steps given below.

1. Access the Stream Processor Studio via the http://localhost:<EDITOR_PORT>/editor URL. The Stream Processor Studio opens as shown below.

   The default URL is http://localhost:9090/editor.
2. Click the **Event Simulator** icon in the left pane of the editor to open the **Single Simulation** panel.

It opens the left panel for event simulation as follows.

![Image of Single Simulation panel]

3. Enter Information in the **Single Simulation** panel as described below.
   a. In the **Siddhi App Name** field, select a currently deployed Siddhi application.
   b. In the **Stream Name** field, select the event stream for which you want to simulate events. The list displays all the event streams defined in the selected Siddhi application.
   c. If you want to simulate the event for a specific time different to the current time, enter a valid timestamp in the **Timestamp** field.
      To select a timestamp, click the time and calendar icon next to the **Timestamp** field.

      Then select the required date, hour, minute, second and millisecond. Click **Done** to select the time stamp entered. If you want to select the current time, you can click **Now**.
   d. Enter values for the attributes of the selected stream.

4. Click **Send** to start to send the event. The simulated event is logged similar to the sample log given below.

```
[2017-12-20 03:58:33,393] INFO [org.wso2.siddhi.core.stream.output.sink.LogSink] - ReceivedMsgCount : TotalConnStream : Event{timestamp=1513868518338, descr=[1], isExpired=false}
```

**Simulating Multiple Events via CSV Files**

This section explains how to generate multiple events via CSV files to be analyzed via WSO2 SP.

**Prerequisites**

Before simulating events, a Siddhi application should be deployed.

**Simulating events**

To simulate multiple events from a CSV file, follow the steps given below.

1. Access the Stream Processor Studio via the http://localhost:<EDITOR_PORT>/editor URL. The Stream Processor Studio opens as shown below.

   The default URL is http://localhost:9090/editor
2. Click the Event Simulator icon in the left pane of the editor.

3. In the event simulation left panel that opens, click on the Feed Simulation tab.

4. To create a new simulation, click Create. This opens the following panel.

5. Enter values for the displayed fields as follows.
   a. In the Simulation Name field, enter a name for the event simulation.
   b. In the Description field, enter a description for the event simulation.
   c. If you want to receive events only during a specific time interval, enter that time interval in the Time Interval field.
   d. Click Advanced Configurations if you want to enter detailed specifications to filter events from the CSV file. Then enter
information as follows.

i. If you want to include only events that belong to a specific time interval in the simulation feed, enter the start time and the end time in the Starting Event's Timestamp and Ending Event's Timestamp fields respectively. To select a timestamp, click the time and calendar icon next to the field.

Then select the required date, hour, minute, second and millisecond. Click Done to select the time stamp entered. If you want to select the current time, you can click Now.

ii. If you want to restrict the event simulation feed to a specific number of events, enter the required number in the No of Events field.

e. In the Simulation Source field, select CSV File.

f. Click Add Simulation Source to open the following section.

In the Siddhi App Name field, select the required Siddhi application. Then more fields as shown below.

Enter details as follows:

i. In the Stream Name field, select the stream for which you want to simulate events. All the streams defined in the Siddhi App you selected are available in the list.

ii. In the CSV File field, select an available CSV file. If no CSV files are currently uploaded, select Upload File from the
list. This opens the **Upload File** dialog box.

Click **Choose File** and browse for the CSV file you want to upload. Then click **Upload**.

iii. In the **Delimiter** field, enter the character you want to use in order to separate the attribute values in each row of the CSV file.

iv. If you want to enter more detailed specifications, click **Advanced Configuration**. Then enter details as follows.

1. To use the index value as the event timestamp, select the **Timestamp Index** option. Then enter the relevant index.
2. If you want to increase the value of the timestamp for each new event, select the **Increment event time by(ms)** option. Then enter the number of milliseconds by which you want to increase the timestamp of each event.
3. If you want the events to arrive in order based on the timestamp, select **Yes** under the **Timestamp Interval** option.

v. Click **Save** to save the information relating to the CSV file. The name of the CSV file appears in the **Feed Simulation** tab in the left panel.

6. To simulate a CSV file that is uploaded and visible in the **Feed Simulation** tab in the left panel, click on the arrow to its right. The simulated events are logged in the output console.

### Simulating Multiple Events via Databases

This section explains how to generate multiple events via databases to be analyzed via WSO2 SP.

#### Prerequisites

Before simulating events via databases, the following prerequisites must be completed:

- A Siddhi application must be created.
- The database from which you want to simulate events must be already configured for WSO2 SP.

#### Simulating events

To simulate multiple events from a database, follow the procedure below:

1. Access the Stream Processor Studio via the `http://localhost:<EDITOR_PORT>/editor` URL. The Stream Processor Studio opens as shown below.

   ```
   The default URL is http://localhost:9090/editor
   ```
2. Click the Event Simulator icon in the left pane of the editor.

3. Click the Feed tab to open the Feed Simulation panel.

4. To create a new simulation, click Create. This opens the following panel.
Enter values for the displayed fields as follows.

a. In the **Simulation Name** field, enter a name for the event simulation.
b. In the **Description** field, enter a description for the event simulation.
c. If you want to simulate events at time intervals of a specific length, enter that length in milliseconds in the **Time Interval(ms)** field.
d. If you want to enter more advanced conditions to simulate the events, click **Advanced Configurations**. As a result, the following section is displayed.

Then enter details as follows.

i. If you want to include only events that belong to a specific time interval in the simulation feed, enter the start time and the end time in the **Starting Event's Timestamp** and **Ending Event's Timestamp** fields respectively. To select a timestamp, click the time and calendar icon next to the field.

Then select the required date, hour, minute, second and millisecond. Click **Done** to select the time stamp entered. If you want to select the current time, you can click **Now**.

ii. If you want to restrict the event simulation feed to a specific number of events, enter the required number in the **No of Events** field.

iii. In the **Simulation Source** field, select **Database**. To connect to a new database, click **Add Simulation Source** to open the following section.
Enter information as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Siddhi App Name</td>
<td>Select the Siddhi Application in which the event stream for which you want to simulate events is defined.</td>
</tr>
<tr>
<td>Stream Name</td>
<td>Select the event stream for which you want to simulate events. All the streams defined in the Siddhi Application you selected are available to be selected.</td>
</tr>
<tr>
<td>Data Source</td>
<td>The JDBC URL to be used to access the required database.</td>
</tr>
<tr>
<td>Driver Class</td>
<td>The driver class name of the selected database.</td>
</tr>
<tr>
<td>Username</td>
<td>The username that must be used to access the database.</td>
</tr>
<tr>
<td>Password</td>
<td>The password that must be used to access the database.</td>
</tr>
</tbody>
</table>

1. Once you have entered the above information, click **Connect to Database**. If the datasource is correctly configured, the following is displayed to indicate that WSO2 SP can successfully connect to the database.
g. To use the index value as the event timestamp, select the **Timestamp Index** option. Then enter the relevant index. If you want the vents in the CSV file to be sorted based on the timestamp, select the **Yes** option under **CSV File is Ordered by Timestamp**.

h. To increase the timestamp of the published events, select the **Timestamp Interval** option. Then enter the number of milliseconds by which you want to increase the timestamp of each event.

6. Click **Save**. This adds the fed simulation you created as an active simulation in the **Feed Simulation** tab of the left panel as shown below.

![Feed Simulation Tab](image)

7. Click on the play button of this simulation to open the **Run or Debug** dialog box.

![Run or Debug Dialog Box](image)

8. If you want to run the Siddhi application you previously selected and simulate events for it, select **Run**. If you want to simulate events in the **Debug** mode, select **Debug**. Once you have selected the required mode, click **Start Simulation**. A message appears to inform you that the feed simulation started successfully. Similarly, when the simulation is completed, a message appears to inform you that the event simulation has finished.

### Generating Random Data

This section explains how to generate random data to be analyzed via WSO2 SP.

**Prerequisites**

Before simulating events, a Siddhi application should be deployed.

**Simulating events**

To simulate random events, follow the steps given below:

The default URL is http://localhost:9090/editor

2. Click the Event Simulator icon in the left pane of the editor.

3. Click the Feed tab to open the Feed Simulation panel.

4. To create a new simulation, click Create. This opens the following panel.
5. Enter values for the displayed fields as follows.
   a. In the **Simulation Name** field, enter a name for the event simulation.
   b. In the **Description** field, enter a description for the event simulation.
   c. If you want to include only events that belong to a specific time interval in the simulation feed, enter the start time and the end time in the **Starting Event's Timestamp** and **Ending Event's Timestamp** fields respectively. To select a timestamp, click the time and calendar icon next to the field.
   
   Then select the required date, hour, minute, second and millisecond. Click **Done** to select the time stamp entered. If you want to select the current time, you can click **Now**.
   d. If you want to restrict the event simulation feed to a specific number of events, enter the required number in the **No of Events** field.
   e. If you want to receive events only during a specific time interval, enter that time interval in the **Time Interval** field.
   f. In the **Simulation Source** field, select **Random**.
   g. If the random simulation source from which you want to simulate events does not already exist in the Feed Simulation pane, click **Add New** to open the following section.
h. Enter information relating to the random source as follows:
   i. In the Siddhi App Name field, select the name of the Siddhi App with the event stream for which the events are simulated.
   ii. In the Stream Name field, select the event stream for which you want to simulate events. All the streams defined in the Siddhi Application you selected are available to be selected.
   iii. In the Timestamp Interval field, enter the number of milliseconds by which you want to increase the timestamp of each event.
   iv. To enter values for the stream attributes, follow the instructions below.
      - To enter a custom value for a stream attribute, select Custom data based from the list. When you select this value, data field in which the required value can be entered appears as shown in the example below.

      ![Custom data based example]

      - To enter a primitive based value, select Primitive based from the list. The information to be entered varies depending on the data type of the attribute. The following table explains the information you need to enter when you select Primitive based for each data type.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Values to enter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To enter values for the stream attributes, follow the instructions below.

To enter a custom value for a stream attribute, select **Custom data based** from the list. When you select this value, a data field in which the required value can be entered appears as shown in the example below.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Values to enter</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>Specify a length in the <strong>Length</strong> field that appears. This results in a value of the specified length being auto-generated.</td>
</tr>
</tbody>
</table>
| FLOAT or DOUBLE | The value generated for the attribute is based on the following values specified.  
  - **Min**: The minimum value.  
  - **Max**: The maximum value.  
  - **Precision**: The precise value. The number of decimals included in the auto-generated values are the same as that of the value specified here.  
| INT or LONG | The value generated for the attribute is based on the following values specified.  
  - **Min**: The minimum value.  
  - **Max**: The maximum value. |
| BOOL | No further information is required because **true** and **false** values are randomly generated. |

To randomly assign values based on a pre-defined set of meaningful values, select **Property based** from the list. When you select this value, a field in which the set of available values are listed appears as shown in the example below.

To assign a regex value, select **Regex based** from the list. To enter values for the stream attributes, follow the instructions below.

To enter a primitive based value, select **Primitive based** from the list. The information to be entered varies depending on the data type of the attribute. The following table explains the information you need to enter when you select **Primitive based** for each data type.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Values to enter</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>Specify a length in the <strong>Length</strong> field that appears. This results in a value of the specified length being auto-generated.</td>
</tr>
</tbody>
</table>
| FLOAT or DOUBLE | The value generated for the attribute is based on the following values specified.  
  - **Min**: The minimum value.  
  - **Max**: The maximum value.  
  - **Precision**: The precise value. The number of decimals included in the auto-generated values are the same as that of the value specified here.  
| INT or LONG | The value generated for the attribute is based on the following values specified.  
  - **Min**: The minimum value.  
  - **Max**: The maximum value. |
| BOOL | No further information is required because **true** and **false** values are randomly generated. |
To assign a regex value, select Regex based from the list.

6. Click Save to save the simulation information. The saved random simulation appears in the Feed tab of the left panel.

7. To simulate events, click the arrow to the right of the saved simulation (shown in the example below).

The simulated events are logged in the CLI as shown in the extract below.

```
[2017-08-11 11:46:33,720] INFO [org.wso2.siddhi.core.query.processor.stream.LogStreamProcessor] - randomApp: randomStream, streamEvent[timestamp=1501811230000, beforeWindowData=null, onAfterWindowData=null, outputData=[t, 0.98125, 8, 6, 11.0175, false, type=CURRENT, next=null]]
```

8. Click Save to save the simulation information. The saved random simulation appears in the Feed tab of the left panel.

9. To simulate events, click the arrow to the right of the saved simulation (shown in the example below).

The simulated events are logged in the CLI as shown in the extract below.

```
[2017-08-11 11:46:45,437] INFO [org.wso2.siddhi.core.query.processor.stream.LogStreamProcessor] - randomApp: randomStream, streamEvent[timestamp=1501811230000, beforeWindowData=null, onAfterWindowData=null, outputData=[t, 0.98125, 8, 6, 11.0175, false, type=CURRENT, next=null]]
```

**Debugging a Siddhi Application**

WSO2 Stream Processor Studio allows debugging tasks to be carried out to ensure that the Siddhi applications you create and deploy are validated before they are run on an actual production environment. To debug a Siddhi application, you can run it in the debug mode, apply debug point and then run event simulation so that the specific debug points are analyzed.

To run a Siddhi application in the debug mode, follow the procedure below:

A Siddhi application can be run in the debug mode only in the source view.

1. Start the Stream Processor Studio following the instructions in Stream Processor Studio Overview.
2. You are directed to the welcome-page. In this scenario, let’s use the existing sample Siddhi application named ReceiveAndCount to demonstrate the debugging functionality. To open this Siddhi application, click on the sample.
The ReceiveAndCount Siddhi application opens in a new tab.

3. In order to debug the Siddhi file, you need to first save it in the workspace directory. To do this, click File => Save. In the Save to Workspace dialog box that appears, click Save.

4. To run the Siddhi application in the debug mode, click Run => Debug.

As a result, the following log is printed in the console.

![Console output]

Also, another console tab is opened with debug options as shown below.

![Debug options]

5. Apply debug points for the required queries. To mark a debug point, you need to click on the left of the required line number so that it is marked with a dot as shown in the image below.
6. Simulate one or more events for the `SweetProductionStream` stream in the Siddhi application. The first line that is marked as a debug point is highlighted as shown below when they are hit.

For detailed instructions to simulate events, see the following sections:

- Simulating a Single Event
- Simulating Multiple Events via CSV Files
- Simulating Multiple Events via Databases
- Generating Random Data
Two viewing options are provided under both Event State and the Query State sections of the Debug tab for each debug point hit as shown above. 

To expand the tree and understand the details of the processed attributes and their values etc., click the following icon for the relevant query.

When you observe the details, note that the value for outputData in the Event State section is null. This is because the debug point is still at the beginning of the query. Also note that the value calculated via the count() function is still displayed as 0 in the Query State section.

The following icons are displayed in the Debug tab of the console:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Click this to proceed from the current debug point to the next available debug point. If there is no debug point marked after the current debug point, the existing debug point continues to be displayed in the tab.

Click this to proceed from the current debug point even if no debug point exists after it.

Once you navigate to next debug point and see the details by clicking the plus signs as mentioned above you can further analyze the processed attributes and its values as shown below. Note that after the count() aggregate function, a value of 1 has been calculated.
Exporting a Siddhi File

The Siddhi files you create in the Stream Processor Studio are saved in the `<SP_HOME>/wso2/editor/deployment/workspace` directory by default. If you want to save a Siddhi file in a different location, you can export it to the required location as explained below.

1. Start WSO2 SP in the editor mode and access the Stream Processor Studio. For detailed instructions, see Starting Stream Processor Studio. The Stream Processor Studio opens as shown below.

2. Open the Siddhi application you want to export. You can click **Open** and select a Siddhi application that is already saved in the `<SP_HOME>/wso2/editor/deployment/workspace` directory.

3. Click **File => Export File**.
This opens the native file browser opens as shown below.

![Native File Browser](image)

4. Browse for the required directory path and click **Save**.

This functionality differs based on the web browser you are using and its settings. e.g., if you have set a default download location and disabled the **Ask where to save each file before downloading** feature as shown below, the file is downloaded to the default location without prompting you for any further input.

![Browser Settings](image)
Developing Streaming Applications

The following topics cover information on writing Siddhi Streaming Applications for WSO2 Stream Processor:

- Siddhi Application Overview
- Collecting Events
- Processing Streaming Events
- Storage Integration
- Complex Event Processing
- Machine Learning
- Incremental Analysis
- Publishing Events
- Converting to a Distributed Streaming Application

Siddhi Application Overview

A Siddhi application (.siddhi) file is the deployment artifact containing the Stream Processing logic for WSO2 Stream Processor.

The format of a Siddhi application is as follows:

```
@App:name("ReceiveAndCount")
@App:description('Receive events via HTTP transport and view the output on the console')

/*
Sample Siddhi App block comment
*/

-- Sample Siddhi App line comment

@Source(type = 'http',
receiver.url='http://localhost:8006/productionStream',
basic.auth.enabled='false',
@map(type='json'))
define stream SweetProductionStream (name string, amount double);

@sink(type='log')
define stream TotalCountStream (totalCount long);

-- Count the incoming events
@info(name='query1')
from SweetProductionStream
select count() as totalCount
insert into TotalCountStream;
```

Basic information about Siddhi applications

Following are some important things to note about Siddhi applications:

- The file name of each Siddhi application must be equal to the name specified via the @App:name() annotation. e.g., In the sample Siddhi application given above, the application name is ReceiveAndCount. Therefore, the Siddhi file name must be ReceiveAndCount.Siddhi.
- It is optional to provide a description via the @App:description() annotation.
- The definitions of the required streams, windows, tables, triggers and aggregations need to be included before the Siddhi queries. e.g., In the above sample Siddhi file, the streams (lines 14 and 17) are defined before the queries (lines 21-23).
- Siddhi can infer the definition of the streams. It is not required to define all the streams. However, if annotations need to be added to a stream, that stream must be defined.
- In the above sample, lines 4-6 nd 8 demonstrate how to include comments within Siddhi applications.
For more information about Siddhi applications, see Siddhi Application at Siddhi Streaming SQL Guide.

Common elements of a Siddhi application

This section explains the common types of definitions and queries that are included in Siddhi application:

**Queries**

Queries define the logical processing and selections that must be executed for streaming events. They consume from the pre-defined streams/windows/tables/aggregations, process them in a streaming manner, and insert the output to another stream, window or table. For more information about Siddhi queries, see Queries at Siddhi Streaming SQL Guide.

**Streams**

Streams are one of the core elements of a stream processing application. A stream is a logical series of events ordered in time with a uniquely identifiable name and set of defined attributes with specific data types defining its schema. In Siddhi, streams are defined by giving it a name and the set of attributes it contains. Lines 14 and 17 of the above sample are examples of defined streams. For more information on Siddhi streams, see Streams at Siddhi Streaming SQL Guide.

**Tables**

A table is a collection of events that can be used to store streaming data. The capability to store events in a table allows you to query for stored events later or process them again with a different stream. The generic table concept holds here as well, however, Siddhi tables also support numerous table specific data manipulations such as defining primary keys, indexing, etc. For more information on Siddhi tables, see Storage and Integration Tables at Siddhi Streaming SQL Guide.

**Windows**

Windows allow you to retain a collection of streaming events based on a time duration (time window), or a given number of events (length window). It allows you to process events that fall into the defined window or expire from it. For more information on Siddhi windows, see Windows at Siddhi Streaming SQL Guide.

**Aggregations**

Aggregation allows you to aggregate streaming events for different time granularities. The time granularities supported are seconds, minutes, hours, days, months and years. Aggregations such as sum, min, avg can be calculated for the desired duration(s) via Siddhi aggregation. For more information on Siddhi aggregations, see Aggregations at Siddhi Streaming SQL Guide.

The elements mentioned above work together in a Siddhi application to form an event flow. To understand how the elements os a Siddhi application are interconnected, you can view the design view of a Siddhi application. For more information, see Stream Processor Studio Overview.

**Collecting Events**

The first step in stream processing is to collect the data that needs to be analyzed. Collection of data is done through Siddhi source which can be defined via @source() annotation on a stream.

To collect data to be processed a stream should have been defined in the Siddhi Application along with the @source() annotation and deployed in WSO2 SP. Here a single SiddhiApp can contain multiple streams and each of those steams can also contain multiple @source() annotations.

**Example Input Source definition as bellow.**

```plaintext
@Source(type = 'http',
   receiver.url='http://localhost:8006/productionStream',
   basic.auth.enabled='false',
   @map(type='json', @attributes( name='$.name',
   amount='$.quantity'))
define stream SweetProductionStream (name string, amount double);
```
The source defines the following:

- **Input source type:** via `type = 'http'`
- **Source type configurations** (This is optional. In this example http source type configurations are defined via `receiver.url='http://localhost:8006/productionStream', basic.auth.enabled='false'`)
- **Input message format:** with optional `@map()` sub-annotation.
- **Custom attribute mapping of the input message format (optional):** with optional `@attributes( name='$.name', amount='$.quantity')` sub-annotation of `@map`.

A source could also be defined externally, and referred to from several siddhi applications as described below. Multiple sources can be defined in the `<SP HOME>/conf/<PROFILE>/deployment.yaml` file. A `<PROFILE>` could refer to dashboard, editor, manager or worker. The following is a sample configuration of a source.

```yaml
siddhi:
  refs:
    - ref:
        name: 'source1'
        type: '<store.type>'
        properties:
          <property1>: <value1>
          <property2>: <value2>
```

You can refer to a source configured in the `<SP HOME>/conf/<PROFILE>/deployment.yaml` file from a Siddhi application as shown in the example below.

```siddhi
@Source(ref='source1', basic.auth.enabled='false',
        @map(type='json', @attributes( name='$.name',
                                    amount='$.quantity')))
define stream SweetProductionStream (name string, amount double);
```

For detailed instructions to configure a source, see [Siddhi Guide - Source](#).

**Source types**

WSO2 SP supports following source types out of the box, to receive events via corresponding transports. Click on the required source type for instructions to configure a source to receive events via them.

- HTTP
- Kafka
- TCP
- In-memory
- WSO2 Event
- Email
- JMS
- File
- RabbitMQ
- MQTT

**Event format**
WSO2 Siddhi allows events to be received in multiple formats. The following formats are currently supported. Once an event is received in a specific format, it is internally converted to a Siddhi event so that it can be processed by applying the WSO2 Siddhi logic. Click on the required format for detailed information on how a source can be configured to receive events in that format.

- WSO2Event
- XML
- Text
- JSON
- Binary
- Key Value

**Processing Streaming Events**

*Forrester* defines Streaming Analytics as follows:

> "Software that provides analytical operators to orchestrate data flow, calculate analytics, and detect patterns on event data from multiple, disparate live data sources to allow developers to build applications that sense, think, and act in real time."

The stream processing capabilities of WSO2 SP allow you to capture high volume data flows and process them in real time, and present results in a streaming manner.

Following are a few stream processing capabilities of WSO2 SP.

**Functions**

The following functions shipped with Siddhi, consume zero, one or more parameters from streaming events and produce a desired value as output. These functions are executed per event. For more information on Siddhi functions, please refer to [Siddhi Query Guide - Functions](#). More functions are made available as [Siddhi Extensions](#).

- **eventTimestamp** - Returns the timestamp of the processed event

  ```siddhi
  from fooStream
  select symbol as name, eventTimestamp() as eventTimestamp
  insert into barStream
  ```

  **eventTimestamp example**

- **UUID** - Generates a UUID (Universally Unique Identifier)

  ```siddhi
  from fooStream
  select UUID() as messageID, messageDetails
  insert into barStream;
  ```

  **UUID example**

- **default** - Checks if the 'attribute' parameter is null and if so returns the value of the 'default' parameter. The function is given as `default(attribute, default value)`

  ```siddhi
  from fooStream
  select default(temp, 0.0) as temp, roomNum
  insert into barStream;
  ```

  **default example**

- **cast** - Converts the first parameter according to the cast-to parameter. Incompatible arguments cause Class Cast

  ```siddhi
  from fooStream
  select cast(temp, 'float') as temp, roomNum
  insert into barStream;
  ```

  **cast example**
exceptions if further processed.

<table>
<thead>
<tr>
<th>cast example</th>
</tr>
</thead>
<tbody>
<tr>
<td>from fooStream</td>
</tr>
<tr>
<td>select symbol as name, cast(temp, 'double') as temp</td>
</tr>
<tr>
<td>insert into barStream;</td>
</tr>
</tbody>
</table>

- `convert` - Converts the first input parameter according to the convert-to parameter

<table>
<thead>
<tr>
<th>convert example</th>
</tr>
</thead>
<tbody>
<tr>
<td>from fooStream</td>
</tr>
<tr>
<td>select convert(temp, 'double') as temp</td>
</tr>
<tr>
<td>insert into barStream;</td>
</tr>
</tbody>
</table>

- `ifThenElse` - Evaluates the 'condition' parameter and returns value of the 'if.expression' parameter if the condition is true, or returns value of the 'else.expression' parameter if the condition is false. The function is given as `ifThenElse(<condition>, <if.expression>, <else.expression>)`

<table>
<thead>
<tr>
<th>ifThenElse example</th>
</tr>
</thead>
<tbody>
<tr>
<td>from fooStream</td>
</tr>
<tr>
<td>ifThenElse(sensorValue&gt;35,'High','Low')</td>
</tr>
<tr>
<td>insert into barStream;</td>
</tr>
</tbody>
</table>

- `minimum` - Returns the minimum value of the input parameters

<table>
<thead>
<tr>
<th>minimum example</th>
</tr>
</thead>
<tbody>
<tr>
<td>from fooStream</td>
</tr>
<tr>
<td>select minimum(price1, price2, price3) as minPrice</td>
</tr>
<tr>
<td>insert into barStream;</td>
</tr>
</tbody>
</table>

- `maximum` - Returns the maximum value of the input parameters. This function could be used similar to how 'minimum' function is used in a query.
- `coalesce` - Returns the value of the first input parameter that is not null. All input parameters have to be of the same type.

<table>
<thead>
<tr>
<th>coalesce example</th>
</tr>
</thead>
<tbody>
<tr>
<td>from fooStream</td>
</tr>
<tr>
<td>select coalesce('123', null, '789') as value</td>
</tr>
<tr>
<td>insert into barStream;</td>
</tr>
</tbody>
</table>

- `instanceOfBoolean` - Returns 'true' if the input is a instance of Boolean. Otherwise returns 'false'.

instanceOfBoolean example

```
from fooStream
select instanceOfBoolean(switchState) as state
insert into barStream;
```

- `instanceOfDouble` - Returns 'true' if the input is a instance of Double. Otherwise returns 'false'. This function could be used similar to how 'instanceOfBoolean' function is used in a query.
- `instanceOfFloat` - Returns 'true' if the input is a instance of Float. Otherwise returns 'false'. This function could be used similar to how 'instanceOfBoolean' function is used in a query.
- `instanceOfInteger` - Returns 'true' if the input is a instance of Integer. Otherwise returns 'false'. This function could be used similar to how 'instanceOfBoolean' function is used in a query.
- `instanceOfLong` - Returns 'true' if the input is a instance of Long. Otherwise returns 'false'. This function could be used similar to how 'instanceOfBoolean' function is used in a query.
- `instanceOfString` - Returns 'true' if the input is a instance of String. Otherwise returns 'false'. This function could be used similar to how 'instanceOfBoolean' function is used in a query.

Filters

Filters are applied to input data received in streams to filter information based on given conditions. For more information, see Siddhi Query Guide - Filters.

e.g., Filtering cash withdrawals from an ATM machine where the withdrawal amount is greater tha $100, and the withdrawal data is between 01/12/2017-15/12/2017.

Windows

Windows allow you to capture a subset of events based on a duration or number of events criterion, from an input stream for calculation. Each input stream can only have a maximum of one window.

**Criterion - Time windows vs length windows**

The subset of events can be captured based on one of the following.

- **Time**: This involves capturing all the events that arrive during a specific time interval (e.g., writing a query that is applicable to events that occur during a period of 10 minutes).
- **Length**: This involves capturing a subset of events based on the number of events (e.g., writing a query applicable to each group that consists of 10 events).

**Method of processing - Sliding windows vs batch windows**

Consider 10 events that have arrived in a stream.

When a sliding length window is included in a Siddhi query, the following event groups are identified:

- Events 1-5
- Events 2-6
- Events 3-7
- Events 4-8
- Events 5-9
- Events 6-10

When a batch window is included in a Siddhi query, the following event groups are identified:

- Events 1-5
- Events 6-10

This window feature differs from the Defined Window concept elaborated here due to this being specific to a single query only. If a window is to be shared among queries, the Defined Window must be used.

For more information about windows, see Siddhi Query Guide - Window.

Aggregate Functions
Aggregation functions allow executing aggregations such as sum, avg, min, etc. on a set of events grouped by a window. If a window is not defined, the aggregation(s) would be calculated by considering all the events arriving at a stream.

Consider the following events arriving at a stream, where the prices vary from one another.

- Event 1: price = 10.00
- Event 2: price = 20.00
- Event 3: price = 30.00
- Event 4: price = 40.00
- Event 5: price = 50.00

Consider the following two queries, where sum of price is calculated based on a length window of 2, and without a window respectively.

**Query 1: Aggregate based on length window**

```
from fooStream#window.length(2)
select sum(price) as totalPrice
insert into barStream;
```

**Query 2: Aggregate without a window**

```
from fooStream
select sum(price) as totalPrice
insert into barStream;
```

The following output would be generated for Query 1.

- totalPrice = 10.00
- totalPrice = 30.00
- totalPrice = 50.00
- totalPrice = 70.00
- totalPrice = 90.00

The following output would be generated for Query 2.

- totalPrice = 10.00
- totalPrice = 30.00
- totalPrice = 60.00
- totalPrice = 100.00
- totalPrice = 150.00

For more information on aggregate function, please refer to Siddhi Query Guide - Aggregate Functions.

**Group By**

With the group by functionality, events could be grouped based on a certain attribute, when performing aggregations.

Consider the following events, which have a symbol attribute and a price attribute.

- Event 1: symbol = wso2, price = 10.00
- Event 2: symbol = wso2, price = 20.00
- Event 3: symbol = abc, price = 30.00
- Event 4: symbol = abc, price = 40.00
- Event 5: symbol = abc, price = 50.00

When the sum aggregation is calculated for a window of length 3, after grouping by symbol, the given output is generated.
Query1: Aggregate based on length window

```sql
from fooStream#window.length(3)
select symbol, sum(price) as totalPrice
group by symbol
insert into barStream;
```

Output:
- symbol = wso2, totalPrice = 10.00
- symbol = wso2, totalPrice = 30.00
- symbol = abc, totalPrice = 30.00
- symbol = abc, totalPrice = 70.00
- symbol = abc, totalPrice = 120.00

For more information on group by, please refer to Siddhi Query Guide - Group By.

**Having**

Having allows to filter events after processing the select statement.

This is useful if the filtering is based on some value derived by applying a function/ aggregation. For example, if you want to find all the events where maximum production total across 3 days is less than 1000 units, such filtering could be achieved with a query as follows:

```
Having example
from fooStream
select item, maximum(productionOnDay1, productionOnDay2, productionOnDay3) as maxProduction
having maxProduction < 1000
insert into barStream;
```

For more information on having clause, please refer to Siddhi Query Guide - Having.

**Join**

Join is an important feature of Siddhi, which allows combining pair of streams, pair of windows, stream with window, stream/ window with a table and stream/window with an aggregation.

The join logic can be defined with ‘on’ condition as well, which restricts the events combined in a join.

For example, assume that we need to combine a transaction stream with a table containing blacklisted credit card numbers, to identify fraudulent transactions. Following query helps achieve such a requirement:

```
Join query example
from transactionStream as t join blacklistedCardsTable as b
on t.cardNumber = b.cardNumber
select t.cardNumber, t.transactionDetails, b.fraudDescription
insert into suspiciousTransactionStream;
```

For more information on join queries, please refer to Siddhi Query Guide - Join.

Output Rate Limiting
Output rate limiting allows queries to output events periodically based on a specified condition. This helps to limit continuously sending events as output.

For more information on output rate limiting, please refer to Siddhi Query Guide - Output Rate Limiting

Partitioning

Partitioning in Siddhi allows to logically separate events arriving at a stream, and to process them separately, in parallel.

For example, assume that the total number of transactions per company needs to be monitored at a stock exchange. However, if all the transactions are arriving at a single stream, we would need to logically separate them based on the company symbol. The following example depicts how this can be achieved with Siddhi partitioning.

```
Partition example

partition with (symbol of stockStream)
begin
    from stockStream
    select symbol, count() as transactionCount
    insert into transactionsPerSymbol;
end;
```

Partitioning can be done based on an attribute value as above, or based on a condition. For more information on partitioning, please refer to Siddhi Query Guide - Partitioning

Trigger

Triggers could be used to get events generated by the system itself, based on some time duration.

An example for a trigger definition is as follows.

```
Trigger example

define trigger FiveMinTriggerStream at every 5 sec;
```

This would generate an event every 5 seconds. The generated event would contain an attribute of type 'Long' named 'triggered_time', reflecting the time at which event was triggered in milliseconds (epoch time).

Trigger could be defined as a time interval, a cron job or to generate an event when Siddhi is started. For more information on triggers, please refer to Siddhi Query Guide - Trigger

Script

Scripts allow to define function operations in a different programming language. An example is as follows.
Define function `concatFn` [javascript] return string {
    var str1 = data[0];
    var str2 = data[1];
    var str3 = data[2];
    var response = str1 + str2 + str3;
    return response;
};

define stream TempStream(deviceID long, roomNo int, temp double);

from TempStream
select concatFn(roomNo,'-',deviceID) as id, temp
insert into DeviceTempStream;

For more information on scripts, please refer to Siddhi Query Guide - Script

Storage Integration

The following sections cover how storage of events is handled in WSO2 Stream Processor.

- Defining Data Tables
- Managing Stored Data via Streams
- Managing Stored Data via REST APIs
- Accessing and Manipulating Data in Multiple Tables

Defining Data Tables

This section explains how to configure data tables to store the events you need to persist to carry out time series aggregation.

The data handled by WSO2 Stream Processor are stored in the following two types of tables:

- **In-memory tables**: If no store-backed tables are defined, data is stored in in-memory tables by default.
- **Store-backed tables**: These are tables that are defined by you in an external database. For a list of database types supported and instructions to define table for different database types, see Defining Tables for External Data Stores.

Adding primary and index keys

Both in-memory tables and tables backed by external databases support primary and index keys. These are defined to allow stored information to be searched and retrieved in an effective and efficient manner.

Adding primary keys

Attribute(s) within the event stream for which the event table is created can be specified as the primary key for the table. The purpose of primary key is to ensure that the value for a selected attribute is unique for each entry in the table. This prevents the duplication of entries saved in the table.

Primary keys are configured via the `@PrimaryKey` annotation. Only one `@PrimaryKey` annotation is allowed per event table.

When several attributes are given within Primary key annotation (e.g. `@PrimaryKey('key1', 'key2')`), those attributes would act as a composite primary key.

Syntax

```
@PrimaryKey('attribute_1')
define table <event_table> (attribute_1 attribute_type, attribute_2 attribute_type, attribute_3 attribute_type);
```
Example

```java
@PrimaryKey('symbol')
define table StockTable (symbol string, price float, volume long);
```

The above configuration ensures that each entry saved in the StockTable event table should have a unique value for the symbol attribute because this attribute is defined as the primary key.

**Adding indexes**

An attribute within the event stream for which the event table is created can be specified as the primary key for the table. This allows the entries stored within the table to be indexed by that attribute.

Indexes are configured via the @Index annotation.

An event table can have multiple attributes defined as index attributes. However, only one @Index annotation can be added per event table.

**Syntax**

To index by a single attribute:

```java
@Index('<attribute_1>')
define table <event_table> (<attribute_1> <attribute_type>, <attribute_2> <attribute_type>, <attribute_3> <attribute_type>);
```

To index by multiple attributes:

```java
@Index('<attribute_1>' '<attribute_2>')
define table <event_table> (<attribute_1> <attribute_type>, <attribute_2> <attribute_type>, <attribute_3> <attribute_type>);
```

Example

```java
@Index('symbol')
define table StockTable (symbol string, price float, volume long);
```

The above configuration ensures that the entries stored in the StockTable event table are indexed by the symbol attribute.

**Defining Tables for Physical Stores**

This section explains how to define data tables to store data handled by WSO2 Stream Processor in physical databases. The @store annotation syntax for defining these tables differ based on the database type as well as where the properties are defined.

The store properties such as URL, username and password can be defined in the following ways:

- **Inline definition:** The data store can be defined within the Siddhi application as shown in the example below:

  ```java
  @Store(type='hbase', hbase.zookeeper.quorum='localhost')
  @primaryKey('name')
  define table SweetProductionTable (name string, amount double);
  ```

  This method is not recommended in a production environment because it is less secure compared to the other methods.

- **As references in the deployment file:** In order to do this, the store configuration needs to be defined for the relevant deployment
The database connection is started when a Siddhi application is deployed, and disconnected when the Siddhi application is undeployed. Therefore, this method is not recommended if the same database is used across multiple Siddhi applications.

```yaml
siddhi:
  refs:
    - ref:
        name: 'store1'
        type: 'rdbms'
        properties:
          jdbc.url: 'jdbc:h2:./repository/database/ANALYTICS_EVENT_STORE'
          username: 'root'
          password: '${sec:store1.password}
          field.length='currentTime:100'
          jdbc.driver.name: 'org.h2.Driver'
```

Then you need to refer to that store via the `@store` annotation as in the Siddhi application as shown in the example below.

```java
@Store(ref='store1')
@PrimaryKey('id')
@Index('houseId')
define table SmartHomeTable (id string, value float, property bool, plugId int, householdId int, houseId int, currentTime string);
```

- **Using WSO2 data sources configuration:** Once a data source defined in the `wso2.datasources` section of the file, `<SP_HOME>/conf/<PROFILE>/deployment.yaml`, the same connection can be used across different Siddhi applications. This is done by specifying the data source to which you need to connect via the `@store` annotation in the following format.

  ```java
  @Store(type='<DATABASE_TYPE>', datasource='<carbon.datasource.name>')
  ```

  The database connection pool is initialized at server startup, and destroyed at server shut down.

  This is further illustrated by the following example.

  ```java
  @Store(type='rdbms', datasource='SweetFactoryDB')@PrimaryKey("symbol")
define table FooTable (symbol string, price float, volume long);
  ```

  For more information about defining datasources, see Configuring Datasources.

The following database types are currently supported for WSO2 SP.
In order to create and use an event table to store data, the following should be completed:

- The required database (MySql, MongoDB, Oracle Database, etc) should be downloaded and installed.
- A database instance should be started.
- The user IDs used to perform the required table operations should be granted the relevant privileges.
- The relevant JDBC Driver must be downloaded and the jar must be put in the `<SP_HOME>/lib` directory.

### Parameters

The following parameters are configured in the definition of an RDBMS event table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>jdbc.url</td>
<td>The JDBC URL via which the RDBMS data store is accessed.</td>
<td>Required</td>
</tr>
<tr>
<td>username</td>
<td>The username to be used to access the RDBMS data store.</td>
<td>Required</td>
</tr>
<tr>
<td>password</td>
<td>The password to be used to access the RDBMS data store.</td>
<td>Required</td>
</tr>
<tr>
<td>pool.properties</td>
<td>Any pool parameters for the database connection must be specified as key value pairs.</td>
<td>Required</td>
</tr>
<tr>
<td>jndi.resource</td>
<td>The name of the JNDI resource through which the connection is attempted. If this is found, the pool properties described above are not taken into account.</td>
<td>Optional</td>
</tr>
<tr>
<td>table.name</td>
<td>The name of the RDBMS table created.</td>
<td>Optional</td>
</tr>
<tr>
<td>field.length</td>
<td>The number of characters that the values for fields of the <code>STRING</code> type in the table definition must contain. If this is not specified, the default number of characters specific to the database type is considered.</td>
<td>Optional</td>
</tr>
</tbody>
</table>

In addition to the above parameters, you can add the `@primary` and `@index` annotations in the RDBMS table configuration.

- **@primary** : This specifies a list of comma-separated values to be treated as unique fields in the table. Each record in the table must have a unique combination of values for the fields specified here.
- **@index** : This specifies the fields that must be indexed at the database level. You can specify multiple values as a comma-separated list.

**Example**

The following is an example of an RDBMS table definition:
The query syntax to define an HBase table is as follows.

```java
@Store(type="hbase", any.hbase.property="<STRING>", table.name="<STRING>", column.family.name="<STRING>")
@PrimaryKey("PRIMARY_KEY")
@Index("INDEX")
```

**Parameters**

The following parameters are configured in the definition of an HBase event table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>table.name</td>
<td>The name with which the table should be persisted in the store. If no table name is specified, the table in the store is assigned the same name as the corresponding Siddhi table.</td>
<td>Optional</td>
</tr>
<tr>
<td>column.family.name</td>
<td>The name of the HBase column family from which data must be stored/referred to.</td>
<td>Required</td>
</tr>
<tr>
<td>any.hbase.property</td>
<td>Any property that can be specified for HBase connectivity in hbase-site.xml is also accepted by the HBase Store implementation. The most frequently used properties are...</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>hbase.zookeeper.quorum - The hostname of the server in which the zookeeper node is run.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hbase.zookeeper.property.clientPort - The port of the zookeeper node.</td>
<td></td>
</tr>
</tbody>
</table>

**Example**

```java
@Store(type="hbase", table.name="SomeTestTable", column.family.name="SomeCF", hbase.zookeeper.quorum="localhost", hbase.zookeeper.property.clientPort="2181")
@PrimaryKey(symbol)
define table FooTable (symbol string, price float, volume long);
```

Apache Solr

**Query syntax**

The query syntax to define an SOLR table is as follows.
@PrimaryKey("id")
@store(type="solr", url=<solr-cloud-zookeeper-url>,
collection=<solr-collection-name>, base.config=<config-name>,
shards=<no-of-shards>, replicas=<no-of-replicas>,
schema=<schema-definition>, commit.async=true|false)
define table Footable (time long, date string);

Parameters
The following parameters are configured in an SOLR table definition.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection</td>
<td>The name of the solr collection/table.</td>
<td>Required</td>
</tr>
<tr>
<td>url</td>
<td>The URL of the zookeeper master of SOLR cloud.</td>
<td>Required</td>
</tr>
<tr>
<td>base.config</td>
<td>The default configuration that should be used for the SOLR schema.</td>
<td>Optional</td>
</tr>
<tr>
<td>shards</td>
<td>The number of shards.</td>
<td>Optional</td>
</tr>
<tr>
<td>replica</td>
<td>The number of replica.</td>
<td>Optional</td>
</tr>
<tr>
<td>schema</td>
<td>The SOLR schema definition.</td>
<td>Optional</td>
</tr>
</tbody>
</table>
| commit.async| If this is set to true, the results all the operations carried out for the table (described below) are applied at a specified time interval. If this is set to false, the results of the operations are applied soon after they are performed with the vent arrival.
  e.g., If this is set to false, an event selected to be inserted into the table is inserted as soon as it arrives to the event stream. | N/A               |

Example
This query defines an SOLR table named FooTable in which a schema that consists of the two attributes time (of long type) and date (of the string type) is maintained. The values for both attributes are stored.

@store(type='solr', url='localhost:9983', collection='TEST1',
base.config='gettingstarted', " +
"shards='2', replicas='2', schema='time long stored, date string stored',
commit.async='true') " +
"define table Footable(time long, date string);

MongoDB
Query syntax
The following is the query syntax to define a MongoDB event table.

@store(type="mongodb", mongodb.uri="<MONGODB CONNECTION URI>")
@PrimaryKey("ATTRIBUTE_NAME")
@IndexBy("<ATTRIBUTE_NAME> <SORTING ORDER> <INDEX OPTIONS>")
define table <TABLE_NME> (<ATTRIBUTE1_NAME> <ATTRIBUTE1_TYPE>,
<ATTRIBUTE2_NAME> <ATTRIBUTE2_TYPE>, <ATTRIBUTE3_NAME> <ATTRIBUTE3_TYPE>,
...};

The mongodb.uri parameter specifies the URI via which MongoDB user store is accessed.
In addition, the following annotations are used in the MongoDB definition.

- **@primary**: This specifies a list of comma-separated values to be treated as unique fields in the table. Each record in the table must have a unique combination of values for the fields specified here.
- **@index**: This specifies the fields that must be indexed at the database level. You can specify multiple values as a comma separated list.

**Example**

The following query defines a MongoDB table named `FooTable` with the `symbol`, `price`, and `volume` attributes. The `symbol` attribute is considered the primary key and it is also indexed.

```
@Store(type="mongodb",
    mongodb.uri="mongodb://admin:admin@localhost:27017/Foo?ssl=true")
@PrimaryKey("symbol")
@indexBy("symbol 1 {background:true}")
define table FooTable (symbol string, price float, volume long);
```

**Managing Stored Data via Streams**

This section covers how to manage stored data in event tables via streams.

- Inserting records
- Retrieving records
- Updating a table
- Deleting Records
- Searching records
- Inserting/updating records

**Inserting records**

**Prerequisites**

In order to insert events to a table:

- General prerequisites should be completed.
- The event stream from which the events to be inserted are taken should be defined.
- The table to which events are to be inserted should be defined. For more information, see Defining a table.

**Query syntax**

The following is the syntax to insert events into a table.

```
from <STREAM_NAME>
select <ATTRIBUTE1_NAME>, <ATTRIBUTE2_NAME>, <ATTRIBUTE3_NAME> ...
insert into <TABLE_NAME>;
```

**Example**

The following query inserts events from the `FooStream` stream to the `FooTable` table with the `symbol`, `price`, and `volume` attributes.

```
from FooStream
select symbol, price, volume
insert into FooTable;
```

**Retrieving records**

**Prerequisites**
In order to retrieve events from a table:

- **General prerequisites** should be completed.
- The table to be read should be already defined. For more information, see Defining a table.
- One or more events should be inserted into the table. For more information, see Inserting events.

**Query syntax**

The following is the query syntax to retrieve events from an existing table. For more information, please refer to Siddhi Query Guide - Join Table.

```sql
from <STREAM_NAME> join <TABLE_NAME>
    on <CONDITION>
select (<STREAM_NAME>|<TABLE_NAME>).<ATTRIBUTE1_NAME>,
    (<STREAM_NAME>|<TABLE_NAME>).<ATTRIBUTE2_NAME>, ...
insert into <OUTPUT_STREAM>
```

**Example**

The following query joins the FooStream events with the events stored in the StockTable table. An output event is created for each matching pair of events, and it is inserted into another stream named OutputStream.

```sql
from FooStream#window.length(1) join StockTable
select FooStream.symbol as checkSymbol, StockTable.symbol as symbol,
    StockTable.volume as volume
insert into OutputStream
```

The information inserted with the output event is as follows.

<table>
<thead>
<tr>
<th>Source</th>
<th>Value of</th>
<th>Attribute name in the output event</th>
</tr>
</thead>
<tbody>
<tr>
<td>FooStream</td>
<td>symbol</td>
<td>checkSymbol</td>
</tr>
<tr>
<td>StockTable</td>
<td>symbol</td>
<td>symbol</td>
</tr>
<tr>
<td>StockTable</td>
<td>volume</td>
<td>volume</td>
</tr>
</tbody>
</table>

**Updating a table**

This section explains how to update the selected records of an existing table.

**Prerequisites**

In order to update events in a table:

- **General prerequisites** should be completed.
- The table to be updated should be already defined. For more information, see Defining a table.
- One or more events should be inserted into the table. For more information, see Inserting events.
- The event stream with the events based on which the updates are made must be already defined.

**Query syntax**

```sql
from <STREAM_NAME>
select <ATTRIBUTE1_NAME>, <ATTRIBUTE2_NAME>, ...
update <TABLE_NAME> (for <OUTPUT_EVENT_TYPE>)?
    set <TABLE_NAME>.<ATTRIBUTE_NAME> = (<ATTRIBUTE_NAME>|<EXPRESSION>)?,
    <TABLE_NAME>.<ATTRIBUTE_NAME> = (<ATTRIBUTE_NAME>|<EXPRESSION>)?, ...
    on <CONDITION>
```
Example

The following query updates the events in the FooTable table with values from the latest events of the FooStream event stream. The events in the table are updated only if the existing record in the table has the same value as the new event for the symbol attribute, and a value greater than 50 for the price attribute.

```
from FooStream
select symbol, price, volume
update FooTable
set FooTable.symbol = symbol, FooTable.price = price, FooTable.volume = volume
on (FooTable.symbol == symbol and price > 50)
```

Methods of Updating the columns in a table

This section gives further information on methods of updating the columns in an existing table.

The value used for updating a table column can be any of the following:

- A constant

```
FROM sensorStream
SELECT sensorId, temperature, humidity
UPDATE sensorTable
    SET sensorTable.column_temp = 1
ON sensorId < sensorTable.column_ID
```

- One of the attributes specified in the `SELECT` clause

```
FROM fooStream
SELECT roomNo, time: timestampInMilliseconds() as ts
UPDATE barTable
    SET barTable.timestamp = ts
ON barTable.room_no == roomNo AND roomNo > 2
```

- A basic arithmetic operation applied on an output attribute

```
FROM sensorStream
SELECT sensorId, temperature, humidity
UPDATE sensorTable
    SET sensorTable.column_temp = temperature + 10
ON sensorId < sensorTable.column_ID
```

- A basic arithmetic operation applied to a column value in the event table

```
FROM sensorStream
SELECT sensorId, temperature, humidity
UPDATE sensorTable
    SET sensorTable.column_temp = sensorTable.column_temp + 10
ON sensorId < sensorTable.column_ID
```
**Deleting Records**

This section explains how to delete existing records in a table based on a specific condition.

**Prerequisites**

In order to delete selected events in a table:

- **General prerequisites** should be completed.
- The table with the records to be deleted should be already defined. For more information, see Defining a table.
- The event stream with the events with which the records in the table are compared (i.e., to apply the condition based on which the events are deleted) must be already defined.
- One or more events should be inserted into the table. For more information, see Inserting events.

**Query syntax**

```
from <STREAM_NAME>
select <ATTRIBUTE1_NAME>, <ATTRIBUTE2_NAME>, ...
delete <TABLE_NAME> (for <OUTPUT_EVENT_TYPE>)?
on <CONDITION>
```

**Example**

This query deletes the events in the RoomTypeTable table if its value for the roomNo attribute is equal to the roomNumber attribute value of DeleteStream.

```
from DeleteStream
delete RoomTypeTable
  on RoomTypeTable.roomNo == roomNumber;
```

**Searching records**

This section explains how to check whether a specific record exists in an event table.

**Prerequisites**

In order to search for a record in a table that matches a specific condition:

- **General prerequisites** should be completed.
- The table to be searched should be already defined. For more information, see Defining a table.
- The event stream with the events with which the records in the table are compared (i.e., to apply the condition based on which the events are searched) must be already defined.
- One or more events should be inserted into the table. For more information, see Inserting events.

**Query syntax**

```
from <STREAM_NAME>[<CONDITION> in <TABLE_NAME>]
select <ATTRIBUTE1_NAME>, <ATTRIBUTE2_NAME>, ...
insert into <OUTPUT_STREAM_NAME>
```

**Example**

The following query matches events arriving from the FooStream event stream with the existing recored stored in the StockTable table. If the symbol attribute of an event saved in the table has the same value as the event from the FooStream stream, that event is inserted into the OutputStream stream.
Inserting/updating records

This section explains how to update a selection of records in a table based on the new events from a specific event stream. The selection is made based on a specific condition that matches events from the stream with events in the table. When the events from the stream have no matching events in the table, they are inserted into the table as new events.

Prerequisites

- General prerequisites should be completed.
- The table for which this operations is to be performed must be already defined. For more information, see Defining a table.
- The event stream from which the events with which the records in the table are compared (i.e., to apply the condition based on which the events are inserted/updated) must be already defined.

Query syntax

The query syntax to perform the insert/update operation for a table is as follows.

```
from <STREAM_NAME>
select <ATTRIBUTE1_NAME>, <ATTRIBUTE2_NAME>, ...
update or insert into <TABLE_NAME> (for <OUTPUT_EVENT_TYPE>)
set <TABLE_NAME>.<ATTRIBUTE_NAME> = <EXPRESSION>,
  <TABLE_NAME>.<ATTRIBUTE_NAME> = <EXPRESSION>, ...
on <CONDITION>
```

Example

This query matches events from the FooStream stream with the events stored in the StockTable table. When an event in the table has the same value for the symbol attribute as the matching new event from the event stream, it is updated based on the new event. If a new event from the event stream does not have a matching event in the table (i.e., an event with the same value for the symbol attribute), that event is inserted as a new event.

```
from FooStream
select *
update or insert into StockTable
on StockTable.symbol == symbol
```

Managing Stored Data via REST APIs

The actions such as inserting, searching, updating, retrieving, and deleting records can be carried out by invoking the POST/stores/query REST API. These actions can be performed for tables as well as windows and aggregations. The following sections provide sample commands for each action.

- Retrieving records
- Updating records
- Deleting records
- Inserting/updating records

Before you begin:

The Siddhi store query endpoint needs to be configured as follows:

1. In the siddhi.stores.query.api: section of the <SP_HOME>/conf/worker/deployment.yaml file, configure the following properties. The following is a sample configuration with default values.
siddhi.stores.query.api:
  transportProperties:
   - name: "server.bootstrap.socket.timeout"
     value: 60
   - name: "client.bootstrap.socket.timeout"
     value: 60
   - name: "latency.metrics.enabled"
     value: true
  listenerConfigurations:
   - id: "default"
     host: "0.0.0.0"
     port: 7070
   - id: "msf4j-https"
     host: "0.0.0.0"
     port: 7443
     scheme: https
     keyStoreFile: "${carbon.home}/resources/security/wso2carbon.jks"
     keyStorePassword: wso2carbon
     certPass: wso2carbon

* transportProperties
  * server.bootstrap.socket.timeout: The number of seconds after which the connection socket of the bootstrap server times out.
  * client.bootstrap.socket.timeout: The number of seconds after which the connection socket of the bootstrap server times out.
  * latency.metrics.enabled: If this is set to true, the latency metrics are enabled and logged for the HTTP transport.

* listenerConfigurations: Multiple listeners can be configured as shown in the above sample.
  * id: A unique ID for the listener.
  * host: The host of the listener.
  * port: The port of the listener.
  * scheme: This specifies whether the transport scheme is HTTP or HTTPS.
  * keyStoreFile: If the transport scheme is HTTPS, this parameter specifies the path to the key store file.
  * keyStorePassword: If the transport scheme is HTTPS, this parameter specifies the key store password.

2. In the siddhi.stores.query.api: section of the <SP_HOME>/conf/editor/deployment.yaml file, configure the following properties:
Inserting records

This allows you to insert a new record to the table with the attribute values you define in the select section.

**Syntax**

```plaintext
select <attribute name>, <attribute name>, ...
insert into <table>;
```

**Sample cURL command**

The following cURL command submits a query that inserts a new record with the specified attribute values to the table RoomOccupancyTable.

```bash
-d '{"appName" : "RoomService", "query" : "select 10 as roomNo, 2 as people
insert into RoomOccupancyTable;" }' -k
```

Retrieving records

This store query retrieves one or more records that match a given condition from a specified table/window/aggregator.

**Retrieving records from tables and windows**

This is the store query to retrieve records from a table or a window.

**Syntax**

```plaintext
```

The above sample shows default values. The same parameter descriptions provided for the `<SP_HOME>/conf/worker/deployment.yaml` file apply to this configuration.
Sample cURL command

The following cURL command submits a query that retrieves room numbers and types of the rooms starting from room no 10, from a table named `roomTypeTable`.

The `roomTypeTable` table must be defined in the RoomService Siddhi application.

```
curl -X POST https://localhost:9443/stores/query -H "content-type: application/json" -u "admin:admin" -d '{"appName" : "RoomService", "query" : "from roomTypeTable select roomNo, type on roomNo >= 10; " }' -k
```

Sample response

The following is a sample response to the sample cURL command given above.

```
{"records":[
 [10,"single"],
 [11, "triple"],
 [12, "double"]
 ]}
```

Retrieving records from aggregations

This is the store query to retrieve records from an aggregation.

Syntax

```
from <aggregation>
<on condition>?
within <time range>
per <time granularity>
select <attribute name>, <attribute name>, ...
<groupby>?
<having>?
<order by>?
<limit>?
```

Sample cURL command

The following cURL command submits a query that retrieves average price of a stock.
curl -X POST https://localhost:9443/stores/query -H "content-type: application/json" -u "admin:admin" -d "{"appName" : "StockAggregationAnalysis", "query" : "from TradeAggregation on symbol=='FB' within '2018-**-** +05:00' per 'hours' select AGG_TIMESTAMP, symbol, total, avgPrice" }" -k

Sample response

The following is a sample response to the sample cURL command given above.

```json
{"records": [ [1531180800, 'FB', 10000.0, 250.0], [1531184400, 'FB', 11000.0, 260.0], [1531188000, 'FB', 9000.0, 240.0] ]}
```

Updating records

This store query updates selected attributes stored in a specific table based on a given condition.

Syntax

```bash
select <attribute name>, <attribute name>, ...
update <table>
    set <table>.<attribute name> = (<attribute name>|<expression>)?,
    <table>.<attribute name> = (<attribute name>|<expression>)?, ...
on <condition>
```

Sample cURL command

The following cURL command updates the room occupancy for selected records in the table, RoomOccupancyTable. The records that are updated are ones of which the room number is greater than 10. The room occupancy is updated by adding 1 to the existing value of the people attribute.

```bash
curl -X POST https://localhost:9443/stores/query -H "content-type: application/json" -u "admin:admin" -d "{"appName" : "RoomService", "query" : "select 10 as roomNumber, 1 as arrival update RoomTypeTable set RoomTypeTable.people = RoomTypeTable.people + arrival on RoomTypeTable.roomNo == roomNumber;" }" -k
```

Deleting records

This store query deletes selected records from a specified table.

Syntax
Sample cURL command

The following cURL command submits a query that deletes a record in the table named `RoomTypeTable` if it has value for the `roomNo` attribute that matches the value for the `roomNumber` attribute of the selection that has 10 as the actual value.

```
curl -X POST https://localhost:9443/stores/query -H "content-type: application/json" -u "admin:admin" -d '{"appName": "RoomService", "query": "select 10 as roomNumber delete RoomTypeTable on RoomTypeTable.roomNo == roomNumber;" }' -k
```

Inserting/updating records

Syntax

```
select <attribute name>, <attribute name>, ...
update or insert into <table>
    set <table>.<attribute name> = <expression>, <table>.<attribute name> = <expression>, ...
    on <condition>
```

Sample cURL command

The following cURL command submits a query that attempts to update selected records in the `RoomAssigneeTable` table. The records that are selected to be updated are ones with room numbers that match the numbers specified in the select clause. If matching records are not found, it inserts a new record with the values provided in the select clause.

```
curl -X POST https://localhost:9443/stores/query -H "content-type: application/json" -u "admin:admin" -d '{"appName": "RoomService", "query": "select 10 as roomNo, "single" as type, "abc" as assignee update or insert into RoomAssigneeTable set RoomAssigneeTable.assignee = assignee on RoomAssigneeTable.roomNo == roomNo;" }' -k
```

For more information and examples for store queries, see Siddhi Query Guide - Store Query.

Accessing and Manipulating Data in Multiple Tables

WSO2 SP allows you to perform CUD operations (i.e., inserting, updating, and deleting data) and retrieval queries for multiple normalized tables within a single data store. This is supported via the `siddhi-store-rdbms` extension.

Performing CUD operations for multiple tables

In order to perform CUD operations, the system parameter named `perform.CUD.operations` needs to be set to `true` in `deployment.yaml` file.

The syntax for a Siddhi query to perform a CUD operation in multiple tables is as follows:
from TriggerStream
select rdbms:cud(<STRING> datasource.name, <STRING> query)
insert into OutputStream;

e.g., If you need to change the details of a customer in customer details table connected to a datasource named SAMPLE_DB a Siddhi query can be written as follows:

from Trigger Stream
select rdbms:cud('SAMPLE_DB', 'UPDATE Customers ON CUSTOMERS SET ContactName='Alfred Schmidt', City='Frankfurt' WHERE CustomerID=1;')
insert into OutputStream

Retrieving data from multiple tables

In order to retrieve information from multiple tables in a data store, the syntax is as follows:

from TriggerStream
select rdbms:query(<STRING> data source.name, <STRING> query, <STRING> stream definition)
insert into OutputStream

e.g., If you need to find matching records in both customer and orders table based on orderId and customerId in the SAMPLE_DB database, a Siddhi query can be written as follows:

from TriggerStream
select rdbms:query('SAMPLE_DB', 'SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate FROM Orders INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID', 'orderId string, customerName string, orderDate string')
insert into OutputStream;

Complex Event Processing

Gartner’s IT Glossary defines CEP as follows:

“CEP is a kind of computing in which incoming data about events is distilled into more useful, higher level “complex” event data that provides insight into what is happening.”

“CEP is event-driven because the computation is triggered by the receipt of event data. CEP is used for highly demanding, continuous-intelligence applications that enhance situation awareness and support real-time decisions.”

WSO2 SP allows you to detect patterns and trends for decision making via Patterns and Sequences supported for Siddhi.

Patterns

Siddhi patterns allow you to detect patterns in the events that arrive over time. This can correlate events within a single stream or between multiple streams. A pattern can be a counting pattern that allows you to match multiple events received for the same matching condition, or a logical pattern that match events that arrive in temporal order and correlate them with logical relationships.
Sequences

This allows you to carry out analysis based on the sequential order in which events matching certain conditions arrive. The difference between sequences and patterns is, in Siddhi sequence, events arriving consecutively are compared based on given conditions. For patterns, events do not have to arrive in consecutive manner.

For more information, see Siddhi Query Guide - Sequence.

Machine Learning

Predictive analytics is a branch of advanced analytics that makes predictions about unknown future events. Historical data is collected to build a predictive model in order to predict the future. However, models can be outdated with time if it is not updated with new data that arrives. WSO2 SP addresses this issue by supporting online machine learning algorithms that enable you to evolve your models while they are alive. A key characteristic of an online predictive algorithm is that the memory and time requirements of the predictive analytics process do not grow over time.

Online predictive analytics for regression, classification, and clustering problems are supported via Siddhi queries. You can update the models by providing required data (and labels for the supervised case). You can also predict using the models by providing unseen data.

WSO2 SP keeps the state of the models in memory and persist the state time to time to recover from failures. The usage of system memory does not grow when new data points are detected. There are no great variations in the time taken to perform predictions and updates over time.

For detailed information about Machine Learning functionalities offered by WSO2 SP, see the documentation for the following extensions:

- Streaming Machine Learning
- PMML Model
- Machine Learning with R
- Tensorflow Model

Incremental Analysis

Incremental aggregation allows you to obtain aggregates in an incremental manner for a specified set of time periods.

This not only allows you to calculate aggregations with varied time granularity, but also allows you to access them in an interactive manner for reports, dashboards, and for further processing. Its schema is defined via the aggregation definition.

This section explains how to calculate aggregates in an incremental manner for different sets of time periods, store the calculated values in a data store and then retrieve that information.

Prerequisites

You should disable data purging if the aggregation query in included in the Siddhi application for read-only purposes. In order to define a query to calculate aggregate values, a pre-defined stream must exist.

Step 1: Calculate and persist time-based aggregate values

In this example, the Siddhi application defines an aggregation named TradeAggregation calculate the average and sum for the price and quantity attributes respectively, of events arriving at the TradeStream stream. These aggregates are calculated per second, minute, hour, day, month and year durations as specified by sec...year (If minute...day was given, aggregation would be done for minute, hour, day. Giving comma separated values as every day, month is also supported. Furthermore, the aggregate calculations would be done per each symbol value due to the presence of 'group by' clause.
In the given example, aggregation would be executed based on an external timestamp, which is reflected by the value of the 'timestamp' attribute (i.e. the event's time is determined by the 'timestamp' attribute's value). This attribute's value could either be a long value (reflecting Unix timestamp in milliseconds) or a string value adhering to one of the following formats.

- If a timezone needs to be given explicitly: `<yyyy>-<MM>-<dd> <HH>:<mm>:<ss> <Z>` (Here the ISO 8601 UTC offset must be provided for `<Z>`. For example +05:30 reflects the India Time Zone. If time is not in GMT this value must be provided)
- If timezone is GMT: `<yyyy>-<MM>-<dd> <HH>:<mm>:<ss>`

Aggregation can also be done based on the event arrival time. To achieve this, the last line of the given example must be changed as 'aggregate every sec ... year;'

The group by clause is optional for incremental aggregation. When the group by clause is not given, all the events would be aggregated together, for each duration.

Following annotations are supported for incremental aggregation. All of these are optional. Hence, an aggregation can be defined without any of these annotations.

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
</table>

The group by clause is optional for incremental aggregation. When the group by clause is not given, all the events would be aggregated together, for each duration.

Following annotations are supported for incremental aggregation. All of these are optional. Hence, an aggregation can be defined without any of these annotations.
The aggregated results would be stored in the database defined by @store definition. For more information on how to define a store, please refer to 'Configuring Event Tables to Store Data' section (Please note that @PrimaryKey annotations are not allowed with aggregation definition since a primary key is defined internally).

When a store is defined, tables corresponding to each aggregate duration would get created in the given database. If a store is not defined, all of these tables would be in-memory tables by default. Giving a store definition is useful in a production environment since otherwise, all aggregations would be lost in the case of a system failure. Names of the tables created would be of the format `<aggregation name>_<duration>`.

The following tables get created in the given example:

- TradeAggregation_SECONDS
- TradeAggregation_MINUTES
- TradeAggregation_HOURS
- TradeAggregation_DAYS
- TradeAggregation_MONTHS
- TradeAggregation_YEARS

Internally, the primary key of these tables would be set to AGG_TIMESTAMP attribute (Additional AGG_EVENT_TIMESTAMP, in case the aggregation is done based on external timestamp sent in the event). If aggregation has a group by clause, a composite primary key of AGG_TIMESTAMP(AGG_EVENT_TIMESTAMP optional) and group by key(s) is defined. AGG_TIMESTAMP & AGG_EVENT_TIMESTAMP are internally calculated value, reflecting the time bucket of an aggregation for a particular duration. For example, for the 'day' aggregations, AGG_TIMESTAMP = 1515110400000 reflects that it's the aggregation for the 5th of January 2018 (1515110400000 is the Unix time for 2018-01-05 00:00:00). All aggregations are based on GMT timezone.

The other values stored in the table would be aggregations and other function calculations done in the aggregate definition. (If it's a function calculation that is not an aggregation, the output value would correspond to the function calculation for the latest event for that time bucket. e.g. If a multiplication is done, the multiplication value would correspond to the latest event's multiplication as per the duration). Please note that certain aggregations are internally stored as a collection of other aggregations. For example, the average aggregation is internally calculated as a function of sum and count. Hence, the table would only reflect a sum and a count. The actual average would be returned when the user retrieves the aggregate values as described in 'Step 2'

When providing aggregation names, please be mindful of the name length, since, for certain databases such as Oracle, table name length limitations exist (e.g. For Oracle it's 30 characters). Hence, the table name (`<Aggregate name>_<Duration>`) length must not exceed the maximum character length specified for the relevant database.

@purge

This specifies the whether automatic data purging is enabled or not. If automatic data purging is enabled, you need to specify the time interval at which the data purging should be carried out. In addition, you need to specify the time period for which the data should be retained based on the granularity by including the @retentionPeriod annotation (described below). If this annotation is not included in an incremental aggregation query, the data purging is carried out every 15 minutes based on the default retention periods mentioned in the description of the @retentionPeriod annotation.

If you want to disable automatic data purging, you can use this annotation as follows:

@purge(enable=false)

You should disable data purging if the aggregation query is included in the Siddhi application for read-only purposes.
@retentionPeriod

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Retention Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>second</td>
<td>120 seconds</td>
</tr>
<tr>
<td>minute</td>
<td>120 minutes</td>
</tr>
<tr>
<td>hour</td>
<td>25 hours</td>
</tr>
<tr>
<td>day</td>
<td>32 days</td>
</tr>
<tr>
<td>month</td>
<td>13 months</td>
</tr>
<tr>
<td>year</td>
<td>none</td>
</tr>
</tbody>
</table>

If the retention period is not specified, the default retention period for each granularity is applied as given below:

- second: 120 seconds
- minute: 24 hours
- hour: 30 days
- day: 1 year
- month: All
- year: All

Step 2: Retrieve calculated and persisted aggregate values

In this query, we are retrieving the aggregate values that you calculated and persisted in Step 1.

This query matches events in the TradeSummaryRetrievalStream stream and the TradeAggregation aggregation that was defined in step 1 based on the value for the symbol attribute, and performs a join for each matching pair. Based on that join, it produces an output event(s) with the symbol, total and avgPrice attributes for the day granularity within the time range 2014-02-15 00:00:00 to 2014-03-16 00:00:00. The time zone is represented by +05:30. The output would contain total and avgPrice per Symbol, for all the days falling within the given time range.

```plaintext
define stream TradeSummaryRetrievalStream (symbol string);

from TradeSummaryRetrievalStream as b join TradeAggregation as a
on a.symbol == b.symbol
within "2014-02-15 00:00:00 +05:30", "2014-03-16 00:00:00 +05:30"
per "days"
select a.symbol, a.total, a.avgPrice
insert into TradeSummaryStream;
```

The `per` clause defines for which time granularity, aggregations need to be retrieved. The `per` value could be seconds, minutes, hours, days, months or years. Furthermore, this `per` value must be a duration for which aggregation is already done as per the aggregate definition. For example, one is not allowed to have an aggregate by timestamp *every sec ... day* in the aggregate definition, and then attempt to retrieve *per 'months'* since month aggregation has not been done.

Apart from providing these within and per values as constants in the query itself, the user has the capability of retrieving those values from
attributes in the incoming stream. An example is as follows.

```sql
define stream TradeSummaryRetrievalStream (symbol string, startTime long, endTime long, perDuration string);

from TradeSummaryRetrievalStream as b join TradeAggregation as a
    on a.symbol == b.symbol
    within b.startTime, b.endTime
    per b.perDuration
select a.symbol, a.total, a.avgPrice
insert into TradeSummaryStream;
```

Ultimately, the user would be produced with the aggregate values for given 'per' time granularity, which happens to be inside the duration specified by the 'within' clause.

- Please note that we can include the AGG_TIMESTAMP attribute in the select clause. If AGG_TIMESTAMP is in the select statement, the `order by AGG_TIMESTAMP` clause could be used if the user needs to get the final aggregations in ascending order, based on time. Such a query would look as follows

```sql
from TradeSummaryRetrievalStream as b join TradeAggregation as a
    on a.symbol == b.symbol
    within "2014-02-15 00:00:00 +05:30", "2014-03-16 00:00:00 +05:30"
    per "days"
select AGG_TIMESTAMP, a.symbol, a.total, a.avgPrice
order by AGG_TIMESTAMP
insert into TradeSummaryStream;
```

- For detailed information about incremental aggregation, see Siddhi Query Guide - Incremental Aggregation.

**Incremental Aggregation Behavior in Single-node Deployment and HA Deployment**

Prior to describing how Incremental Aggregation behaves in a single node and HA setup, it would be worthy to know how the aggregation is executed internally.

Consider an instance, where aggregation is done for second, minute and hour durations. Assume that events are arriving in the following sequence, for the given timestamp

- event 0 2018-01-01 05:59:58
- event 1 2018-01-01 05:59:58
- event 2 2018-01-01 05:59:59
- event 3 2018-01-01 06:00:00
- event 4 2018-01-01 06:00:01
- event 5 2018-01-01 06:00:02

Consider that no buffer has been defined (Hence buffer size is 0). When event 0 arrives, it would be initially processed at the second level. This aggregation would be retained in memory until the 58th second elapses. When event 1 arrives, since that too belongs to the 58th second (same as event 0), event 0 and event 1 data would be aggregated together. When event 2 arrives, we would know that the 58th second has now elapsed (since event 2 has the timestamp for the 59th second). Hence the aggregation for the 58th second (that is the total aggregation for event 0 and event 1) would be written to the <Aggregate name>_SECONDS table, and then forwarded to be processed at the minute level. At this phase, 59th second's data (event 2) would be processing in-memory at the second executor level. Aggregation which was forwarded would be processing in-memory for the 59th minute at minute executor level. When event 3 arrives, the 59th seconds' aggregation (event 2) would get expired and get written to <Aggregate name>_SECONDS table. It would also be forwarded to the minute executor. At the minute executor level, this forwarded aggregation (corresponding to event 2) would be aggregated in-memory, for the 59th minute. Effectively, the running in-memory aggregation at the minute level would have an aggregation for events 0, 1 and 2 at this phase. Once event 4 arrives, aggregation for timestamp 2018-01-01 06:00:00 (event 3) would be written to Aggregate name>_SECONDS table and forwarded to the minute executor. With the arrival of event 3 at the minute executor, it would identify that the 59th minute has now elapsed. Hence the aggregation for the 59th minute would be written to <Aggregate name>_MINUTES table and then forwarded to the hour executor. The hour executor would process this forwarded aggregation for the 5th hour. At this stage, event 4 would be processing in-memory at the second level; aggregation for event 3 would be processing in-memory at minute level; aggregation for the 5th hour would be processing in-memory at hour level. When event 5 arrives, aggregation for event 4 would be forwarded to the minute window. Hence event 3 and 4 would be aggregated together since they belong to the same minute (0th minute of the 6th hour). Event 5 would be processed in-memory at the second level.

Hence the aggregation status would be as follows after all 5 events have arrived.
1. **Second level aggregation**
   a. `<Aggregate name>_SECONDS table`
      i. Aggregation for 2018-01-01 05:59:58 (Aggregation of event 0 and 1)
      ii. Aggregation for 2018-01-01 05:59:59 (event 2)
      iii. Aggregation for 2018-01-01 06:00:00 (event 3)
      iv. Aggregation for 2018-01-01 06:00:01 (event 4)
   b. Processing in-memory: Aggregation for 2018-01-01 06:00:02 (event 5)

2. **Minute level aggregation**
   a. `<Aggregate name>_MINUTES table`
      i. Aggregation for 2018-01-01 05:59:00 (Aggregation of event 0, 1 and 2)
   b. Processing in-memory: Aggregation for 2018-01-01 06:00:00 (That is the 0th minute of the 6th hour. Aggregation of event 3 and 4)

3. **Hour level aggregation**
   a. `<Aggregate name>_HOURS table`: No aggregation data is written to this table
   b. Processing in-memory: Aggregation for 2018-01-01 05:00:00 (That is the 5th hour. Aggregation of event 0, 1 and 2)

**Single-node Deployment**

In a single node deployment, if an @store configuration is not given, the older aggregations would be stored in in-memory tables (That is the `<Aggregate name>_MINUTES table`, etc would be in-memory tables). In such a situation, if the server fails for some reason, all the aggregations done so far would be lost. 'Older aggregations' refers to aggregations which are not running in-memory (aggregations which have already been completed for a particular second, minute, etc.)

When some DB configuration has been provided with @store annotation, the older aggregations would be stored in the defined external database. Hence, if the node fails, once we restart the server, the in-memory running aggregations would be recreated from these stored data. In the given example, notice how the in-memory aggregations for the Minute level can be recreated with data iii and iv in the Seconds table. Likewise, Hour in-memory aggregations can be recreated with data in the Minute table. However, this recreation can only be done for all the durations except the most granular duration for which aggregation is done (For the given example, we cannot recreate the in-memory aggregations for the second level, since there's no table for a prior duration).

Hence, for the given example, (when @store details are provided) if the server fails after all 5 events have arrived after we restart the server, the in-memory aggregations for minute and hour level would be recreated from table information. Effectively, only event 5 would be lost, since that was aggregating in-memory at the second level.

**HA Deployment**

Let's now explore the behavior of Incremental Analytics in an HA setup. Please refer to Minimum High Availability (HA) Deployment documentation for more information regarding an HA setup and to see how we can configure one.

In a minimum HA setup, one WSO2 SP node would be active whereas the other would be passive. If the HA setup is configured properly, even if the @store configuration is not provided, none of the processed aggregations would be lost due to snapshotting the current state of the SP.

When a snapshot has already been created, the recreation from tables does not happen (since there's no necessity to do so). Hence, the events coming in after a server restart in an HA setup would be processed as in the case where no server failure occurs. That is, the details provided under the 'info' section of 'Single-node Deployment' is not applicable here.

**Publishing Events**

Once data is analyzed by WSO2 SP, the resulting data is output via a selected transport to the required interface in the required data format. In order to output results, a Siddhi application that processes events must have one or more sinks configured in it.

A sink configuration specifies the following:

- **Source types**
- **Event format**

**Source types**

Each sink configuration must specify the transport type via which the events to be published by the Siddhi application should be published. The parameters to be configured for the transport differs based on the transport types. The following is the list of transport types supported by WSO2 SP. For detailed instructions to configure a sink of a specific transport type, click on the relevant link.

- HTTP
- Kafka
- TCP
- In-memory
Event format

The format in which the output events need to be published is specified as the mapping type in the sink configuration. The parameters related to mapping that need to be configured differ based on the mapping type. The following is a list of supported mapping types. For detailed instructions to configure each type, click on the relevant link.

- WSO2Event
- XML
- TEXT
- JSON
- Binary
- Key Value

Converting to a Distributed Streaming Application

A Siddhi Application is a combination of multiple Siddhi executional elements. A Siddhi executional element can be a Siddhi Query or a Siddhi Partition. When defining a Siddhi application, you can specify a number of parallel instances to be created for each executional element, and how each executional element must be isolated for an SP instance. Based on this, the initial Siddhi application is divided into multiple Siddhi applications and deployed in different SP instances.

This deployment pattern is supported so that a high volume of data can be distributed among multiple SP instances instead of having them accumulated at a single point. Therefore, it is suitable to be used in scenarios where the volume of data handled is too high to be managed in a single SP instance.

Creating a distributed Siddhi application

A sink could also be defined externally, and referred to from several siddhi applications as described below. A <PROFILE> could refer to dashboard, editor, manager or worker.

Multiple sinks can be defined in the <SP HOME>/conf/<PROFILE>/deployment.yaml file. The following is a sample configuration of a sink.

```yaml
siddhi:
  refs:
    - ref:
        name: 'sink1'
        type: '<sink.type>'
        properties:
          <property1>: <value1>
          <property2>: <value2>
```

You can refer to a source configured in the <SP HOME>/conf/<PROFILE>/deployment.yaml file from a Siddhi application as shown in the example below.

```yaml
@Sink(ref='sink1', @map(type='json', @attributes(name='$.name', amount='$.quantity')))
define stream SweetProductionStream (name string, amount double);
```
This section explains how to write distributed Siddhi applications by assigning executional elements to different execution groups.

**Executional elements**

A distributed Siddhi application can contain one or more of the following elements:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateless queries</td>
<td>Queries that only consider currently incoming events when generating an output. e.g., Filters</td>
</tr>
<tr>
<td>Stateful queries</td>
<td>Queries that consider both currently incoming events as well as past events when generating an output. e.g., windows, sequences, patterns, etc.</td>
</tr>
<tr>
<td>Partitions</td>
<td>Collections of stream definitions and Siddhi queries separated from each other within a Siddhi application for the purpose of processing events in parallel and in isolation.</td>
</tr>
</tbody>
</table>

**Annotations**

The following annotations are used when writing a distributed Siddhi application.

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@dist(execGroup='name of the group')</td>
<td>All the executional elements with the same execution group are executed in the same Siddhi application. When different execution groups are mentioned within the same distributed Siddhi application, WSO2 SP initiates a separate Siddhi Application per execution group. In each separated Siddhi application, only the executional elements assigned to the relevant execution group are executed. Executional elements that have no execution group assigned to them are executed in a separate SP instance.</td>
</tr>
<tr>
<td>@dist(parallel='number of parallel instances')</td>
<td>The number of instances in which the executional element must be executed in parallel. All the executional elements assigned to a specific execution group (i.e., via the @dist(execGroup) annotation) must have the same number of parallel instances specified. If there is a mismatch in the parallel instances specified for an execution group, an exception occurs. When the number of parallel instances to be run is not given for the executional elements assigned to an execution group, only one Siddhi application is initiated for that execution group.</td>
</tr>
</tbody>
</table>

**Example**

The following is a sample distributed Siddhi application.
When this Siddhi application is deployed, it is executed as shown in the table below.

<table>
<thead>
<tr>
<th>Execution Group</th>
<th>Number of Siddhi Application Instances</th>
<th>Queries executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>group1</td>
<td>1</td>
<td>query1, query2, query3</td>
</tr>
<tr>
<td>group2</td>
<td>2</td>
<td>query4</td>
</tr>
<tr>
<td>group3</td>
<td>2</td>
<td>query5</td>
</tr>
</tbody>
</table>

Deploying Streaming Applications

A Siddhi application (.siddhi file) can be deployed to run that on production.

There are two modes deployment

- Standard deployment
  Here we deploy Siddhi Apps in worker node (In Single node, Minimum HA Deployments)
• Fully distributed deployment
  Here we deploy Siddhi Apps in manager node (Only in Fully Distributed Deployment)

Refer below sections to get to know how to deploy on the above deployments:
  • Standard Deployment
  • Fully Distributed Deployment

Standard Deployment

To start a worker node in single node mode, issue one of the following commands:
  • For Windows: worker.bat
  • For Linux: ./worker.sh

Siddhi applications can be deployed to the worker node by using one of the following methods:

1. Dropping the .siddhi file in to the <SP_HOME>/wso2/worker/deployment/siddhi-files/ directory before or after starting the worker node.
2. Sending a "POST" request to http://<host>:<port>/siddhi-apps with the Siddhi App included in the body of the request. Refer Stream Processor REST API Guide for more information on using WSO2 Stream Processor APIs.

When a Siddhi application is successfully deployed, a message similar to the following example appears in the startup logs.

To configure a Minimum HA deployment refer Minimum High Availability Deployment documentation.

Fully Distributed Deployment

To successfully set up, configure and run Siddhi applications in a fully distributed environment refer Fully Distributed Deployment documentation.

Visualizing Data

The Dashboard Portal allows you to visualize the results of the analysis carried out by WSO2 SP in graphical representations. The following topics explain how to start, run and work with the Dashboard Portal of WSO2 SP.

• Starting the Dashboard Portal
• Accessing the Dashboard Portal
• Using the Dashboard Portal

Starting the Dashboard Portal

To start the Dashboard Portal, run one of the following commands from the <SP_HOME>/bin directory:

• On Windows: dashboard.bat --run
• On Linux/Mac OS: sh dashboard.sh

Accessing the Dashboard Portal

Once the server has started, use one of the following URLs to access the Dashboard Portal:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>URL Format</th>
<th>Example</th>
</tr>
</thead>
</table>

Using the Dashboard Portal

When you access the Dashboard Portal using one of the links given above, you will be prompted for login to the dashboard. Let's provide default admin login credentials for the moment. (username : admin password : admin)
Upon successful login, following page appears with the list of dashboards that are currently available.

You can carry out the following tasks in the Dashboard Portal:

- Designing Dashboards
- Viewing Dashboards
- Generating Reports
- Generating Widgets
- Creating Custom Widgets
- Dashboard Authorization Configuration
- Managing Widgets
- Importing and Exporting Dashboards

**Designing Dashboards**

Dashboard Portal component of the Stream Processor provides the capability to create dashboards and display widgets in them. It provides a designer view that allows you to easily design your dashboard with any layout, and with multiple pages.

The following sections describe how dashboard designers can work with dashboards using the Dashboard Portal.

- Creating New Dashboards
- Editing Dashboards
- Deleting Dashboards

**Creating New Dashboards**

To create a new dashboard, follow the procedure below:

1. Start and access the WSO2 SP Dashboard Portal. For more information, see Visualizing Data.
2. Click + to open the **Create a Dashboard** page.
The following buttons appear as a result.

Click **Create Dashboard** to open the **Dashboard Designer** wizard.

3. In the **Create a Dashboard** page, enter information as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name of your Dashboard</td>
<td>Enter a unique name for your dashboard.</td>
<td>Sales Statistics</td>
</tr>
<tr>
<td>URL</td>
<td>Enter a URL for the dashboard.</td>
<td>sales-stats</td>
</tr>
<tr>
<td>Description</td>
<td>Enter a description about the dashboard.</td>
<td>Monthly Sales Statistics</td>
</tr>
</tbody>
</table>

4. Click **Add** to save the dashboard. A basic dashboard is created and you are redirected to the home page of the dashboard that is created by default.
5. To add a widget to this page, click the **Widgets** icon.

The widget list is displayed in the left panel as follows:

Widgets that you see in the left panel are a set of sample widgets that are shipped with the portal. You can generate or manually create your own widgets and get them listed here. For more information, see [Generating Widgets](#) and [Creating Custom Widgets](#).
6. Drag and drop a required widget into the dashboard container as shown below.

You can click the Delete icon on the widget container to remove a widget that is already added to the dashboard.

7. To view the dashboard you created, click Portal.

The list of available dashboards opens as shown below, and the Sales Statistics dashboard you created is included in the list.

Where to go next?
You can continue to add pages and widgets to the dashboard and also control access to your dashboard. For detailed instructions see the following topics:

- Managing Dashboard Pages
- Securing Dashboards
- If you click Back without further editing the dashboard, you can continue to edit this dashboard later by following the instructions in Editing Dashboards.
Managing Dashboard Pages

This section covers the following topics.

- Adding multiple pages to a dashboard
- Marking a page as landing page
- Deleting pages from a dashboard

**Before you begin:**
Create a dashboard following the instructions in Creating New Dashboards.

Adding multiple pages to a dashboard

To add pages to a dashboard, follow the steps below:

1. Click **Create Page** in the left panel of the dashboard.

   ![Create Page](image)

   A new page opens.

2. Drag and drop widgets to the new page as described in Creating New Dashboards.

3. Open the **Dashboard Designer** wizard for the dashboard again as described in Editing Dashboards. The newly added page appears in the left panel as shown in the example below.
4. Click on the new page (**New Page-3** in the above example) to expand it. As a result, the section appears displaying the current page settings as marked in the image below.

Edit the page URL as required in the **URL** field.

5. Repeat the above steps to add as many pages as required.

**Marking a page as landing page**

The landing page (also known as Home Page) of a dashboard is the page that is displayed by default when you access the dashboard. You can change the current home page by marking a new page as the home page. To do this, follow the steps below:

1. Open the **Dashboard Designer** wizard for the relevant dashboard.
2. In the left panel, click on the page you want to mark as the home page, and select the **Mark as Home Page** check box.
Deleting pages from a dashboard

To delete a page, follow the steps below:

1. Open the Dashboard Designer wizard for the relevant dashboard.
2. In the left panel, click on the page you want to delete, and then click the **Delete** icon.

It is not allowed to delete the page that is currently marked as the home page. If you want to delete the current home page, you need to first mark a different page as the home page.

Securing Dashboards

By default any created dashboard will be accessible to any user who has access to Dashboard Portal. If you want to secure your dashboard, you have to explicitly configure permissions for the dashboard.

Dashboard permissions are configured by editing dashboard settings as follows:
1. Log in to the WSO2 Dashboard Portal and access it. For instructions, see Visualizing Data. The available dashboards are displayed as shown below.

2. Click the menu icon of the dashboard that you want to secure, and then click Settings. The Dashboard Settings page opens. Here, you can assign roles for different permission levels.
There are three different permission levels for a given dashboard.

- **Owners** - Owners of a dashboard are the most privileged users of the dashboard. Only they have access to settings page once access control is set. They also can edit or view the dashboard.

- **Editors** - Editors of a dashboard can either edit the dashboard or view the dashboard. But they don't have access to the dashboard settings page.

- **Viewers** - Viewers of a dashboard are the least privileged users of a dashboard. The can only view the dashboard and cannot edit or change the settings of a dashboard.

Depending on the configured user store, available list of roles will be populated for each permission level. Dashboard owners can set the required roles for each of those permission levels. Each permission level can have multiple roles.

**Editing Dashboards**

A dashboard can be edited in the **Dashboard Designer** wizard. To access this wizard for an existing dashboard, follow the steps below:

1. Open the menu of the dashboard by clicking its menu icon.
2. Click **Design**.

The **Design** menu item is available for a given dashboard only if the user who currently logged in is an owner or an editor of the dashboard, or if the dashboard is not secured. For more information, see [Securing Dashboards](#).

A user who has access to edit a dashboard can do the following:

- Perform page level changes such as add/remove pages, change page name/URL, change home page, etc. For more information, see [Managing Dashboard Pages](#).
- Add/remove widgets from dashboard. For more information, see [Creating New Dashboards](#).
- Resize widget containers.

**Resize widget containers**

Once you have dragged and dropped a widget into a particular dashboard, the size of the widget container can be altered by dragging its borders as demonstrated below.
Deleting Dashboards

To delete an existing dashboard, follow the procedure below:

1. Log in to the WSO2 Dashboard Portal and access it. For instructions, see Visualizing Data. The available dashboards are displayed as shown below.

2. Click the menu icon of the dashboard you want to delete.
3. Click **Delete**.

The **Delete** menu item is available for a given dashboard only if the user who currently logged is an owner of the dashboard, or if the dashboard is not secured. For more information, see Securing Dashboards.

A message appears to confirm whether you want to proceed to delete the selected dashboard. Click **YES** to continue.

**Viewing Dashboards**

To view an existing dashboard, follow the procedure below:

1. Log in to the WSO2 Dashboard Portal and access it. For instructions, see Visualizing Data. The available dashboards are displayed as shown below.

2. Click on the dashboard you want to view. In this example, the **Sales Dashboard** is clicked. The home page of the dashboard opens as shown in the example below.
3. To view the other pages in the dashboard, open the left pane that displays the list of pages by clicking the following icon.

You can click the same icon to hide the left pane.

Then click on the relevant page in the left pane.
4. If you want to toggle the color theme of the dashboard, click the Light-Dark switch in the top bar.
4. You also can maximize and minimize the view of the widgets in the dashboard by clicking on the maximize and minimize icons in the widget container as shown below.

5. Generating Reports

The Dashboard Portal of WSO2 Stream Processor allows you to generate reports with graphical illustrations of processed data. This is done by creating PDF documents with selected dashboards and widgets.
Generating a report for a single widget

To create a PDF document with a single widget, follow the steps below:

1. Log in to the WSO2 Dashboard Portal and access it. For instructions, see Visualizing Data. The available dashboards are displayed as shown below.

2. Click on the dashboard you want to view. In this example, the Sales Dashboard is clicked. The home page of the dashboard opens as shown in the example below.
3. To download a widget in the dashboard as a PDF document, click the Generate Report icon of that widget.

Generating a report with multiple dashboard pages

To create a PDF document that includes one or more dashboard pages, follow the procedure below:

1. Open the existing Sales Dashboard as instructed in the first two steps of Generating a report with a single widget section.
2. To generate a PDF document with all the widgets displayed in the current page, open the left pane by clicking the icon marked in red below at the top left corner of the page.
a. In the left navigator, select a page (e.g., the Home page) you want to include in the report, and then click Capture the Current Page.
Once you click the **Capture the Current Page** for a page, the **Generate Report** button is enabled.

b. If there is another page you want to include in the report (e.g., **Regional Statistics**) page, select that page and click **Capture the Current Page** again. Repeat this step for all the pages you need to include.

3. Click **Generate Report** to create the PDF document with the selected pages.

### Generating Widgets

Widgets are the way to visualize your data in the dashboards. Widgets display your data or information in a selected format that you can configure through the Widget Generation Wizard.

### Prerequisites

The following prerequisites are required only in the following scenarios:

- The data provider selected is **RDBMS Batch Data Provider** or RDBMS Streaming Data Provider.
- If the data provider used is **Siddhi Store Data Provider** and the Siddhi application makes a reference to the data source defined in the **deployment.yaml** file.

Before generating widgets, the following prerequisites must be completed:

- A data source must be configured for the widget. Data sources are configured in the `<SP_HOME>/conf/dashboard/deployment.yaml` file. The following is a sample H2 data source.
- name: SAMPLE_DB
  description: Sample datasource used for gadgets generation wizard
  jndiConfig:
    name: jdbc/SAMPLE_DB
    useJndiReference: true
    definition:
      type: RDBMS
      configuration:
        jdbcUrl: 'jdbc:h2:${sys:carbon.home}/wso2/${sys:wso2.runtime}/database/SAMPLE_DB;IFEXISTS=TRUE;DB_CLOSE_ON_EXIT=FALSE;LOCK_TIMEOUT=60000;MVCC=TRUE'
        username: wso2carbon
        password: wso2carbon
        driverClassName: org.h2.Driver
        maxPoolSize: 50
        idleTimeout: 60000
        connectionTestQuery: SELECT 1
        validationTimeout: 30000
        isAutoCommit: false

- Download the required JDBC driver (i.e., based on the database type you are using) and copy it to the `<SP_HOME>/lib` directory.

  This driver must be OSGi compatible in order to be detected.

Generating a widget

To generate create a widget and display it in a dashboard, follow the steps below:

1. Start and access the WSO2 SP Dashboard Portal. For detailed instructions, see Visualizing Data.
2. Click +.

Then click Create Widget to open the Create Widget wizard.

3. In the first page, enter a name for the new widget. In this example, let's enter Transactions as the name.
4. Click Next. Then select a data provider for the widget. The data provider is the source from which the widget extracts information to be displayed. In this example, let's select **RDBMS Batch Data Provider** as the data provider.

As a result, the page is populated with the default configurations of the Sample_DB datasource. Click Next to proceed.
The parameters you need to configure vary depending on the data provider you select. The following are the parameters you need to configure for the other available data providers.

The parameters available to be configured for these data providers depend on how the data providers are configured.
Siddhi Store Data Provider

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Siddhi App</td>
<td>The Siddhi application that stores the data to be visualized in the Siddhi store.</td>
<td>define stream MasterpassTransactionDataStream (merchant_client_id string, merchant_name string, checkout_transaction_id string, card_brand_id string, card_brand_name string, card_bin long, card_holder_name string, card_expiry_mm int, card_expiry_yy int, card_aav int, card_sli int, card_billing_address1 string, card_billing_address2 string, card_billing_city string, card_billing_subdivision string, card_billing_postal string, card_billing_country string, contact_email string, contact_firstname string, contact_lastname string, contact_nationalid long, contact_gender string, contact_dob_mm int, contact_dob_yyyy int, contact_address1 string, contact_address2 string, contact_city string, contact_subdivision string, contact_country string, contact_postal string, shipping_address1 string, shipping_address2 string, shipping_city string, shipping_subdivision string, shipping_country string, shipping_postal string, shipping_recipient_name string, wallet_id int, region string, total_usd float);</td>
</tr>
</tbody>
</table>
|                | Only the siddhi event table/ aggregation definition needs to be added here, with applicable annotations (@store, @PrimaryKey) (Note: For aggregation, aggregation input stream should also be defined) | } via WebSocket messages. For more information, see Working with Data Providers.
Dynamic Query Generation Configuration

The javascript function which returns the query that the data provider needs to execute in order to get the required information from the database. This can be provided in two ways:

- **Static query**: This involves providing a simple query without using any external values. Here, you do not need to enter widget inputs.
  
  *e.g.* from `MasterpassTransactionDataStreamByCardType` within '2018-**-** **:**:**' per 'hour s' select `card_brand_name,region as re, sum(totalAmount) as total` group by `region`

- **Dynamic query(with widget inputs)**: This involves adding external inputs received from publisher widgets under Add widget inputs. You can click + to add more inputs. You need to specify the default values to be used in the query if no input is received from publisher widgets.
  
  *e.g.* return "from `MasterpassTransactionDataStreamByCardType` within " + from + "L, " + to + "L per " + per + "s' select `card_brand_name,region as re, sum(totalAmount) as total` group by `region";"

Publishing Interval

The frequency with which you want the data provider to poll data from the Siddhi store. The time interval is specified in seconds.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Type</td>
<td>The map type to convert the data received via WebSocket in order to update them in the widget. Supported mapping types are, xml json and text.</td>
<td>xml</td>
</tr>
<tr>
<td>Topic</td>
<td>The WebSocket topic to which WSO2 SP needs to listen in order to display information in the widget</td>
<td>topic1</td>
</tr>
</tbody>
</table>

**Websocket Provider**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Type</td>
<td>The map type to convert the data received via WebSocket in order to update them in the widget. Supported mapping types are, xml json and text.</td>
<td>xml</td>
</tr>
<tr>
<td>Topic</td>
<td>The WebSocket topic to which WSO2 SP needs to listen in order to display information in the widget</td>
<td>topic1</td>
</tr>
</tbody>
</table>
Metadata related to the stream

The metadata related to the stream. The attributes of the stream into which the events published in the widget are received can be added here. The fields in the table are as follows:

- **Metadata Name**: The name of the stream attribute.
- **Metadata Type**: The metadata type of the attribute. This can be one of the following.
  - **Linear**: Data is received in numerical format.
  - **Ordinal**: Data is received in string format.
  - **Time**: data is received as timestamps.

The definition of the stream from which events published in the widget are fetched is as follows:

```java
define stream SweetTotalStream (name string, totalProduction long);
```

This can be entered as metadata as shown below:

<table>
<thead>
<tr>
<th>Metadata Name</th>
<th>Metadata Type</th>
<th>Linear</th>
<th>Ordinal</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>ORDINAL</td>
<td>-</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>totalProduction</td>
<td>LINEAR</td>
<td>-</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

### RDBMS Streaming Data Provider

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Datasource Name</strong></td>
<td>The name of the data source from which the widget extracts information to be displayed.</td>
<td>SAMPLE_DB</td>
</tr>
<tr>
<td><strong>Dynamic Query Generation Configuration</strong></td>
<td>The javascript function which returns the query that the data provider needs to execute in order to get the required information from the database. This can be provided in two ways:</td>
<td>Entering a query: return &quot;select * from TRANSACTIONS_TABLE&quot;;</td>
</tr>
<tr>
<td><strong>Static query</strong></td>
<td>This involves providing a simple query without using any external values. Here, you do not need to enter widget inputs.</td>
<td></td>
</tr>
<tr>
<td><strong>Dynamic query(with widget inputs)</strong></td>
<td>This involves adding external inputs received from publisher widgets under <strong>Add widget inputs</strong>. You can click + to add more inputs. You need to specify the default values to be used in the query if no input is received from publisher widgets.</td>
<td></td>
</tr>
<tr>
<td><strong>Table Name</strong></td>
<td>The name of the table from which the data provider must poll data.</td>
<td>TRANSACTIONS_TABLE</td>
</tr>
<tr>
<td><strong>Incremental Column</strong></td>
<td>A data column that is used by the data provider to identify new records.</td>
<td>CREDITCARDNO</td>
</tr>
<tr>
<td><strong>Time Columns</strong></td>
<td>Columns in the Table that contain time data, separated by commas.</td>
<td>timeFieldOne, timeFieldTwo</td>
</tr>
<tr>
<td><strong>Publishing Interval</strong></td>
<td>The frequency with which you want the data provider to poll data from the database. The time interval is specified in seconds.</td>
<td>1</td>
</tr>
</tbody>
</table>
5. Select a chart type of your choice. For this example, let’s select **Line/Area/Bar Chart Parent**.

6. Configure your chart as follows:
Select the data fields that should represent the X and Y axes of your graph. In this example, let's select **Country** for the X axis and **Amount** for the Y axis.

- If you want to do a color categorization, select the field based on which you want it to be done.
- If you want to design another chart within the same graph, click **Add Chart** and configure that chart as required.
- If you want to do any advance configurations, expand the **Advanced Settings** section and do the required configurations. For more information about the advanced settings you can configure, see [Customizing charts](#).
- If the chart is a publisher widget, expand the **Data Publishing Configuration** section. Then specify how the widget outputs need to be displayed by entering the required text as shown in the example below. To add a new widget output, you can click +.

This configuration is done only if the widget is a publisher (i.e., its output is used by one or more other widgets as input). For more information, see [Inter-widget Communication via the Publisher/Subscriber Model](#).
7. Once you have configured your chart(s) as required, click **Preview** to see how it will look when it is added. In this example, the widget configured looks as follows.

8. To save the widget, click **Create**. A message appears to inform you that the widget is successfully created. Now this widget can be used in any dashboard. Once you click **Create** you are also redirected to the landing page.

9. Open the **Dashboard Designer** wizard for an existing dashboard (for detailed instructions, see Editing Dashboards). Then click on the required page.

10. Click the **Widgets** icon.

This opens the list of available widgets in the left pane. The **Transactions** widget you created is included in this list.
11. Drag and drop the widget to the dashboard. It is displayed in the dashboard as shown below.
Customizing charts

The following are advanced settings that you can use when you configure widgets.

- **Time formatting regex:** The charts in WSO2 SP Dashboard Portal handle time series data automatically and format them accordingly. However, if you want to customize how they are formatted, you can specify the required time format in a regex format as explained in [d3 documentation](#).
- **Tick axis angles:** If you want to rotate the ticks of an axis, you can specify the amount (in degrees) as a numerical value.
- **Maximum length of the dataset:** This property can be used if you want to limit the number of data points that are visible in a given chart at a time.
- **Add Chart (when generating Line/Area/Bar charts):** This can be used if you want to draw another line, area or bar chart on the same XY plane using the same data set. Properties to configure another chart are provided in the form.
- **Color set to use in the charts:** Charts are shipped with a default color set that is used to visualize the data. If you want to override this color set, you can define your own color set by using this option. You can define a list of colors in hex format that can be used in the charts.

Creating Custom Widgets

In addition to creating a widget using widget generation wizard, you can implement your own custom widgets and use them within the Dashboard portal.

If you want to refer the source of a sample widget to develop your custom widget, you can find the available sample widgets here.

This section explains:

- How to write a custom widget for the Dashboard portal
- The features that are provided for widget developers in the Dashboard portal

A widget in Dashboard portal is a simple ReactJS component that can be used to visualize information. This widget functions as a chart, table, set of components to interact with, etc.

In order to identify a particular ReactJS component as a widget by the dashboard portal, it needs to register itself as a widget in the portal.

Let's have a look at a simple widget, Hello World.

Creating HelloWorld widget

The directory structure of your HelloWorld widget source is as follows

```bash
HelloWorld (referred as <WIDGET_ROOT>)
package.json
src
    HelloWorld.jsx
resources
    widgetConf.json
webpack.config.js
```

1. Copy following content to package.json and webpack.config.js respectively. Webpack is required for building a widget.
package.json
{
"name": "hello-world-widget",
"version": "1.0.0",
"private": true,
"dependencies": {
"react": "^16.1.1",
"react-dom": "^16.1.1"
},
"scripts": {
"build": "webpack -p"
},
"devDependencies": {
"webpack": "^3.5.6",
"ajv": "^5.2.2",
"babel-core": "^6.25.0",
"babel-loader": "^7.1.1",
"babel-preset-es2015": "^6.24.1",
"babel-preset-react": "^6.24.1",
"babel-register": "^6.26.0",
"copy-webpack-plugin": "^4.2.0",
"node-sass": "^4.5.3",
"sass-loader": "^6.0.6"
}
}

webpack.config.js
const path = require('path');
const webpack = require('webpack');
const CopyWebpackPlugin = require('copy-webpack-plugin');
module.exports = {
context: path.resolve(__dirname, './src'),
entry: {
index: './HelloWorld.jsx'
},
output: {
path: path.resolve(__dirname, './dist/HelloWorld/'),
filename: 'HelloWorld.js'
},
module: {
loaders: [
{
test: /\.html$/,
use: [{loader: 'html-loader'}]
},
{
test: /\.js$/,
exclude: /node_modules/,


use: [
    {
        loader: 'babel-loader',
        query: {
            presets: ['es2015', 'react']
        }
    }
],

test: /
    \.(png|jpg|svg|cur|gif|eot|svg|ttf|woff|woff2)$/,
    use: ['url-loader']
},

test: /\.(jsx?$/,
    exclude: /,(node_modules)/,
    loader: 'babel-loader',
    query: {
        presets: ['es2015', 'react']
    }
},

test: /\.(css$/,
    use: ['style-loader', 'css-loader']
},

test: /\.(scss$/,
    use: [{loader: 'style-loader'}, {loader: 'css-loader'}, {loader: 'sass-loader'}]
],

],
},
plugins: [
    new CopyWebpackPlugin([{
            from: path.resolve(__dirname, './src/resources/')
        }]
    ],
    resolve: {

extensions: ['.js', '.json', '.jsx', '.scss']

2. Now copy the following content into `HelloWorld.jsx` under `src` directory. This contains all the logic for our widget.

```jsx
import React, { Component } from 'react';

class HelloWorld extends Component {
  render() {
    return (<h1>Hello, World!</h1>);
  }
}

global.dashboard.registerWidget('HelloWorld', HelloWorld);
```

Note that we have registered the react component as a widget in the Dashboard portal by calling the `registerWidget` function where the first argument is the ID you want to register your widget, and the second argument is the react component.

3. Dashboard portal requires few meta information regarding the widget in order to identify it. `<WIDGET_ROOT>/src/resources/widget Conf.json` file contains this meta information. When building the widget this configuration file gets copied into the final widget directory. Add the following content into the `widgetConf.json` file.

```json
{
  "name": "HelloWorld",
  "id": "HelloWorld",
  "thumbnailURL": "",
  "configs": {}
}
```

4. Now the source of the widget is complete. To install the dependencies required to build your widget, navigate to the `<WIDGET_ROOT>` directory and issue the following command.

```bash
npm install
```

5. Go to the `<WIDGET_ROOT>` directory and issue the following command to build the widget.

```bash
npm run build
```

6. Once the build is successful the final widget directory is created in the `<WIDGET_ROOT>/dist` directory. Copy the `<WIDGET_ROOT>/dist` directory.
directory into the `<SP_HOME>/wso2/dashboards/deployment/web-ui-apps/portal/extensions/widgets` directory.
7. Restart WSO2 Stream Processor.
8. Now log in to the Dashboard portal and create a new dashboard.

For instructions to access the Dashboard Portal, see **Visualizing Data**.
For instructions to create a new dashboard, see **Creating New Dashboards**.

You can view the newly created `HelloWorld` widget in the widget listing panel of the Dashboard Designer.

Now you understand how to develop a widget and get it rendered in the Dashboard portal.

Let's improve this widget to add more interactive elements to it.

1. Clone the WSO2 `carbon-dashboards` repository to your machine.
2. Copy the `HelloWorld` widget you created to the `carbon-dashboards/samples/widgets` directory.
3. To import the channel manager to access the database, you need to use the `WidgetChannelManager`. To do this, create a separate file named `ExtendedWidget.jsx` as shown below.

```javascript
import React from 'react';
import Widget from '@wso2-dashboards/widget';
import WidgetChannelManager from
'.//./././components/dashboards-web-component/src/utils/dashboard-
channel/WidgetChannelManager';

let widgetChannelManager = null;

export default class ExtendedWidget extends Widget {

  constructor(props) {
    super(props);
  }

  getWidgetChannelManager() {
    if(!widgetChannelManager) {
      widgetChannelManager=new WidgetChannelManager();
    }
    return widgetChannelManager;
  }
}
```

4. Update the `HelloWorld.jsx` file as shown below.

- The `react-vizgrammar` module is a library that is used to create the charts in the widget named `ExtendedWidget` that you previously created. To add the config, metadata and data fields of the table, you can find the required instructions in `react-vizgrammar - Table Chart Samples`.
- Here, `console.info` is the action of the row click. You can use the required function that is related to the second widget as the `handleData` function in your implementation.
import React, { Component } from 'react';
import ExtendedWidget from './ExtendedWidget';
import VizG from 'react-vizgrammar';

class HelloWorld extends ExtendedWidget {
  constructor(props) {
    super(props);

    this.tableConfig = {
      //add table configures
    };

    this.metadata = {
      //add meta data
    };

    this.data = [
      //add table data here
    ];

    this.handleData = this.handleData.bind(this);
  }

  render() {
    return (
      <VizG config={this.tableConfig} data={this.data} metadata={this.metadata} onClick={this.handleData} />
    );
  }

  handleData(row) {
    console.info(row);
  }

  global.dashboard.registerWidget('HelloWorld', HelloWorld);
} // (widgetId, reactComponent)

5. If you consider the publisher-subscriber concept, the widget with the table is a publisher and the widget that triggers the table action with a row click is a subscriber. To define this:
   - Open the widgetConf.json file of the publisher widget and change the pubsub type to publisher as shown below.
Similarly open the widgetConf.json file of the subscriber widget and change the pubsub type to subscriber.

6. Navigate to the <WIDGET_ROOT> directory and issue the following command to build the widget.

   npm run build

   Once the build is successful, the widget is created in the <WIDGET_ROOT>/dist directory.

7. Copy the <WIDGET_ROOT>/dist/HelloWorld directory and place it in the <SP_HOME>/wso2/dashboards/deployment/web-ui-apps/portal/extensions/widgets directory.

The Dashboard portal also provides additional capabilities to widget developers by providing a set of APIs via a base widget component. Please follow the step below to extend your widget from the base widget component.

Extending from base widget component

To extend from the base widget component, follow the steps below:

1. Add base widget as a dependency in your package.json file.

   ```json
   'dependencies': {
      ...
      '@wso2-dashboards/widget': '1.1.1',
      ...
   },
   ```

2. Import base widget in your widget source file.

   ```javascript
   import Widget from '@wso2-dashboards/widget';
   ```

3. Extend the base widget.

   ```javascript
   class MyWidget extends Widget {
      ...
   }
   ```

Base widget component provides following capabilities for a widget author. In order to consume these capabilities from your custom widget, extend your widget from the above Widget class.
Resolving CSS style conflicts

When creating custom widgets there is a possibility of getting styles conflicts since all the widgets are rendered within a single page. There are certain practices a widget developer can follow to overcome this issue. To understand these practices follow “Resolving CSS style conflicts”.

Retrieving the information to be displayed

To retrieve information to be displayed in this widget via an RDBMS data provider, follow the steps below:

1. Access the data provider configurations in `<SP_HOME>/deployment/web-ui-apps/portal/extensions/widgets/<WIDGET_NAME>/widgetConf.json`.
2. Use the base widget API to create the WebSocket connection. For this, you can use the following APIs:
   - Subscribe to the endpoint
     
     ```javascript
     super.getWidgetChannelManager().subscribeWidget(<Widget id>, <callback method to handle data>, <data provider configs>);
     ```

   - Unsubscribe from the endpoint
     
     ```javascript
     super.getWidgetChannelManager().unsubscribeWidget(<Widget ID>);
     ```

Consume user information within a widget

There can be a scenario where a particular widget needs to access the user information of current logged in user. For that, an API has been introduced to the widget developer via the base widget component.

Follow the steps mentioned below to access user information. Please note that this API currently exposes only the username of the current user.

**NOTE: This guide assumes that you already have a widget which is deployable in dashboard portal.**

1. Make sure your widget has extended from the based widget provided by the dashboard portal. If not please refer “Extending from base widget component”.
2. Call the following method to get the user information.

   ```javascript
   const userInfo = super.getCurrentUser();
   ```

A sample widget has been shipped with WSO2 Stream Processor named “UserInfo” to demonstrate this feature. You can find the source code of the sample widget at [https://github.com/wso2/carbon-dashboards/blob/v4.0.38/samples/widgets/UserInfo](https://github.com/wso2/carbon-dashboards/blob/v4.0.38/samples/widgets/UserInfo).

Inter Widget Communication via the PubSub Model

The Dashboard Portal allows inter-widget communication via the publisher/subscriber model. In any publisher/subscriber approach the publisher (i.e., sender) sends messages without specifically targeting a subscriber (i.e., receiver). The receiver indicates that it needs to receive one or more messages published by a particular publisher or a set of publishers.

Inter-widget communication in the Dashboard Portal is based on the same model where the dashboard itself keeps track of its publishers, subscribers, and their subscriptions. When a particular publisher publishes a message, the dashboard routes the message to its subscribers based on the store-and-forward mechanism.
Implementing a publisher/subscriber widgets

Before you begin:
You need to create a widget that can be deployed on the Dashboard Portal. For detailed instructions, see Generating Widgets.

To implement publisher/subscriber widgets follow the steps below.

Step 1: Create a publisher widget

1. Ensure that your widget is extended from the based widget provided by the dashboard portal. If not, follow the instructions in Creating Custom Widgets - Extending from base widget component.
2. To specify the widget as a publisher widget, add the following configuration to the widgetConf.json file of the widget that can be found in the <SP_HOME>/wso2/dashboard/deployment/web-ui-apps/portal/extensions/widgets/<WIDGET_NAME> directory.

```json
configs: {
    ...
    pubsub: {
        types: ['publisher'],
    },
    ...
}
```

3. To publish a message call the following function. The message can be of any type.

```javascript
super.publish('message');
```

To demonstrate how publisher widgets are created and used, a sample widget named Publisher is shipped with WSO2 Stream Processor. To view the source code of this widget, click here.

Step 2: Create a subscriber widget

1. Ensure that your widget is extended from the based widget provided by the dashboard portal. If not, follow the instructions in Creating Custom Widgets - Extending from base widget component.
2. To specify the widget as a subscriber widget, add the following configuration to the widgetConf.json file of the widget that can be found in the <SP_HOME>/wso2/dashboard/deployment/web-ui-apps/portal/extensions/widgets/<WIDGET_NAME> directory.

```json
configs: {
    ...
    pubsub: {
        types: ['subscriber'],
    },
    ...
}
```

3. To subscribe to publishers call the following function.
super.subscribe((message) => {
   // Handle the message here.
});

Wiring publishers and subscribers

Once the publishers and subscribers are implemented and added to the dashboard, the subscribers need to subscribe for input from the required publishers. This is achieved via publisher/subscriber wiring. The following steps demonstrate how the publisher/subscriber wiring is carried out.

The Publisher and Subscriber widgets that are shipped with WSO2 SP by default are used in the steps for the purpose of demonstration.

1. Open the Dashboard designer and drag and drop the Publisher and Subscriber widgets to an existing dashboard. For detailed information, see Generating Widgets.
2. On the Subscriber widget, click the icon in the top right corner (marked in the image below) to open the Subscriber Configurations panel to the right.

   ![Subscriber widget](image)

3. In the Subscriber Configurations panel, select the name of the widget that needs to be considered as the publisher of this widget. In this example, let’s select Publisher as shown below.

   ![Subscriber Configurations panel](image)

As a result, messages can be sent from the Publisher widget to the Subscriber widget. If you want the subscriber widget to unsubscribe, you can clear this check box.

Resolving CSS style conflicts

A widget developer can use CSS styles for styling the widget in different ways, such as inline styles, external stylesheet, etc.

Since a dashboard is a collection of multiple widgets each using its own set of styles, if the styling is not done correctly there is a probability of final dashboard ending up with stylesheet conflicts.

To demonstrate how subscriber widgets are created and used, a sample widget named Subscriber is shipped with WSO2 Stream Processor. To view the source code of this widget, click here.
There are several approaches that we can take in order to prevent this.
Using inline styles

Inline styles are one of the safest approaches to prevent style conflicts. In this method the respective CSS styles are bound with the underline
markup and only affect that specific tag hence there won’t be style conflicts.
Using CSS modules

Inline styles lack in maintainability. Inability to share common styles among elements end up in a huge pile of duplicate code. This can be avoided
by using CSS as classes. But since all the widgets are rendered in the un-sandboxed environment probably there will be style conflicts.
This can be avoided by using CSS modules.
CSS modules allow to style a component via CSS stylesheet like approach, but the class names are scoped to the local ReactJS component.
This is achieved by prefixing each CSS class name with a random string at the build-time. Please follow the following steps to enable CSS
modules.
For implement CSS modules webpack build system, style-loader, and the css-loader need to be installed in within the widget. Use the following
command to install the above modules.
1. Open the webpack.config.js file in the widget root directory.
2. We need to configure the Webpack css-loader to load CSS as CSS modules. Add the following configuration into the webpack.config.js
file.

module.exports = {
...
module: {
loaders: [
...
{
test: /\.css$/,
loader:
'style-loader!css-loader?modules=true&localIndentName=[name]__[local]
___[hash:base64:5]'
},
...
]
...
}
...
}

3. Now open the widget JSX file and use the CSS styles/classes defined in the external CSS file as follows.

...
import styles from './MyComponent.css';
class MyComponent extends React.component {
/* ... */
render() {
return (
<button className={styles.myButton} />
)
}
/* ... */
}


Share dashboards with persisted widget states

A dashboard can have widgets which the user can interact with. For an example, there can be a widget which the user can pick a date range which resulting another set of widgets change there dataset based on the selected date range. In this kind of scenario, there needs to be a way the user can save the current state of the dashboard or share it with someone else. In order to handle this scenario, a set of APIs has been introduced to the widget developer via the base widget component.

Follow the steps mentioned below to implement state persistence.

NOTE: This guide assumes that you already have a widget which is deployable in dashboard portal.

1. Make sure your widget has extended from the based widget provided by the dashboard portal. If not please refer “Extending from base widget component”.
2. Call the following method to store the state of the widget.

   ```javascript
   super.setWidgetState(key, stateObj);
   ```

3. Call the following method to retrieved the saved state of the widget.

   ```javascript
   const state = super.getWidgetState(key);
   ```

A sample widget has been shipped with WSO2 Stream Processor named “WidgetState” to demonstrate this feature. You can find the source code of the sample widget at https://github.com/wso2/carbon-dashboards/tree/v4.0.8/samples/widgets/WidgetState.

Accessing Widget Configuration

This section explains how to access the configuration of a widget in the `<SP_HOME>/deploment/web-ui-apps/portal/extensions/widgets/<WIDGET_NAME>/widgetConf.json` file to understand how data displayed in the widget is fetched.

To access the widget configuration, follow the steps below.

1. Ensure that your widget is extended from the based widget provided by the dashboard portal. If not, follow the instructions in Creating Custom Widgets - Extending from base widget component.

   The base widget version is required to be newer than 1.2.1.

2. Call the following method to get the widget configuration information.

   ```javascript
   super.getWidgetConfiguration(this.props.widgetID)
   .then((message) => {
     //Access widget configuration from message.data
   })
   .catch((error) => {
     // Handle Rest API call failure
   });
   ```

Dashboard Authorization Configuration

There are two levels of permission to use the SP Dashboard Portal to visualize information:

This section explains the levels of permission granted for users of the Dashboard Portal. For granting permission for individual dashboards within the portal, see Securing Dashboards.

- Creator: User roles with this permission level have administrative privileges over the Dashboard Portal. They are allowed to design, view,
**Viewer**: User roles with this permission level are only allowed to view dashboards.

The following topics cover how to configure dashboard authorization:

- Prerequisites
- Configuring permissions

**Prerequisites**

Before configuring Dashboard Portal permissions, the user roles to be assigned permissions must be already defined in the user store with the required user IDs. For detailed instructions, see User Management.

**Configuring permissions**

The following is a sample configuration of user roles for the Dashboard Portal. These user roles are defined in the `<SP_HOME>/conf/dashboard/deployment.yaml` file under the `wso2.dashboard.portal:` namespace as shown below.

```yaml
wso2.dashboard.portal:
  roles:
    creator:
      - name: role1
        id: 1
    viewer:
      - name: role2
        id: 2
```

**Managing Widgets**

- Editing widgets
- Deleting widgets

**Editing widgets**

In WSO2 Stream Processor, widgets are treated as React components, and they are available as compiled, minified codes. Therefore, the source code of widgets is not human-readable. Due to this, they cannot be edited manually. To edit an existing widget, we need to get the source code, edit it and compiled again.

Editing widgets by editing the source code is not recommended because all the generated widgets are served by one central code base. Editing this code affects all the widgets, and this may cause errors when updating the product pack via WUM.

**Deleting widgets**

You can delete an existing widget via the following API.

```
DELETE https://localhost:9643/portal/apis/widgets/{id}
```

**Importing and Exporting Dashboards**

In order to make it possible to deploy a single dashboard in multiple WSO2 product instances (e.g., multiple SP instances, other SP-based Analytics profiles such as EI Analytics, etc.), WSO2 SP allows dashboards to be imported and exported.

- Exporting dashboards
- Importing dashboards
Exporting dashboards

Exporting dashboards is carried out via a REST API that can be invoked via the following curl command. This generates a JSON file that can be imported to another SP-based WSO2 product.

```bash
curl
-u <USERNAME>:<PASSWORD>
-o '<DASHBOARD_ID>.json'
-k
'https://localhost:9643/portal/apis/dashboards/<DASHBOARD_ID>/export'
```

e.g., the following curl exports a dashboard named `ei-analytics`.

```bash
curl
-u admin:admin
-o 'ei-analytics.json'
-k
'https://localhost:9643/portal/apis/dashboards/eianalytics/export'
```

This export generates the following JSON file.

```
{
  dashboard: { ... },
  widgets: {
    generated: [ ... ],
    custom: [ ... ]
  }
}
```

The fields included in the dashboard structure in this generated file are as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dashboard</td>
<td>This field contains the definition of the dashboard, including the dashboard metadata such as the dashboard name, URL and page structures including widgets.</td>
</tr>
<tr>
<td>widgets.generated</td>
<td>This field contains the definitions of widgets generated via the Widget generation wizard.</td>
</tr>
<tr>
<td>widgets.custom</td>
<td>This contains the list of the names of the custom widgets contained in the dashboard.</td>
</tr>
</tbody>
</table>

Importing dashboards

Once a dashboard is exported from one SP-based WSO2 product as explained in Exporting Dashboards, it can be exported into the Dashboard portal of any other SP-based WSO2 product instance. This is done by copying the generated JSON file and placing it in the `<SP_HOME>/wso2/dashboard/resources/dashboards` directory of the required SP-based WSO2 product pack.

e.g., If the `ei-analytics.json` generated in the above example is imported to another WSO2 SP instance, its path in the SP pack is `<SP_HOME>/wso2/dashboard/resources/dashboards/ei-analytics.json`.

When the SP server is started, it picks the JSON files located in the `<SP_HOME>/wso2/dashboard/resources/dashboards` directory and copies those dashboards and widgets into the relevant tables in the underlying `DASHBOARD_DB` database. Once the dashboard is successfully imported, the following messages are logged in a file named `carbon.log` in the `<SP_HOME>/wso2/dashboard/logs` directory.
Working with Business Rules

In stream processing, there are common use cases for analyzing statistics that involve operations such as calculating the average, minimum, maximum etc., for different endpoints. The Business Rules Manager allows you to define templates and generate business rules from them for different scenarios with common requirements.

Watch the following screencast to understand the complete functionality of the WSO2 SP Business Rules feature.

The following topics cover how to create templates for business rules, create business rules from them and how to manage them.

- Creating Business Rules
- Managing Business Rules
- Creating a Business Rule Template
- Business Rules Templates
- Configuring Business Rules Manager Permissions

Creating Business Rules

This section explains how to create a business rule. A business rule can be created from a template or from scratch.

- Prerequisites
- Creating a business rule from a template
- Creating a business rule from scratch

Prerequisites

In order to create a business rule from a template, the following prerequisites must be completed:

- Both the Dashboard and Worker runtimes of the Stream Processor server must be started.
- The templates for business rules must be defined in a template group, and must be configured in the `<SP_HOME>/conf/dashboard/deployment.yaml` file. For detailed instructions, see Business Rules Templates.

Creating a business rule from a template

Creating business rules from an existing template allows you to use sources, sinks and filters already defined and assign variable values to process events.

To create a business rule from a template, follow the procedure below:

1. Go to `<SP_HOME>` from the terminal and start the Dashboard runtime of WSO2 SP with one of the following commands:
   - On Windows: `dashboard.bat --run`
   - On Linux/Mac OS: `./dashboard.sh`
2. Start the Worker runtime of WSO2 SP with one of the following commands:
   - On Windows: `worker.bat --run`
   - On Linux/Mac OS: `./worker.sh`
3. Access the Business Rule Manager via one of the following URLs.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>URL Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>http://&lt;SP_HOST&gt;:&lt;HTTP_PORT&gt;/business-rules</td>
<td><a href="http://0.0.0.0:9090/business-rules">http://0.0.0.0:9090/business-rules</a></td>
</tr>
<tr>
<td>HTTPS</td>
<td>https://&lt;SP_HOST&gt;:&lt;HTTPS_PORT&gt;/business-rules</td>
<td><a href="https://0.0.0.0:9443/business-rules">https://0.0.0.0:9443/business-rules</a></td>
</tr>
</tbody>
</table>
3. This opens the following:

4. Click **Create** to open the following page.

5. Then click **From Template** to open the **Select a Template Group** page where the available templates are displayed.

6. Click on the template group that contains the required template to create a rule from it. In this example, the rule is created based on a template in the **Sweet Factory** template group that is packed with WSO2 SP by default. Therefore, click **Sweet Factory** to open this template group.

7. In the template group, expand the **Rule Template** list as shown below, and click on the required template. For this example, click **Identify Continuous Production Decrease**.
8. If you want to change the rule template from which you want to create the rule, select the required value for the Rule Template field.
9. Enter a name for the business rule in the Business Rule Name field.
10. Enter values for the rest of the fields following the instructions in the UI.

The fields displayed for the rule differ based on the template selected.

11. If you want to save the rule and deploy it later, click Save. If you want to deploy the rule immediately, click Save and Deploy.

Creating a business rule from scratch

Creating a rule from scratch allows you to define the filter logic for the rule at the time of creating instead of using the filter logic already defined in a template. However, you can select the required source and sink configurations from an existing template.

To create a business rule from scratch, follow the procedure below:

1. Start the Dashboard Portal of WSO2 SP with one of the following commands:
   - On Windows: dashboard.bat --run
   - On Linux/Mac OS: sh dashboard.sh
2. Start the Worker runtime of WSO2 SP with one of the following commands:
   - On Windows: worker.bat --run
   - On Linux/Mac OS: ./worker.sh
3. Access the Business Rule Manager via one of the following URLs.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>URL Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>http://&lt;SP_HOST&gt;:&lt;HTTP_PORT&gt;/business-rules</td>
<td><a href="http://0.0.0.9090/business-rules">http://0.0.0.9090/business-rules</a></td>
</tr>
<tr>
<td>HTTPS</td>
<td>https://&lt;SP_HOST&gt;:&lt;HTTPS_PORT&gt;/business-rules</td>
<td><a href="https://0.0.0.9443/business-rules">https://0.0.0.9443/business-rules</a></td>
</tr>
</tbody>
</table>

This opens the following:

4. Click Create to open the following page, and then click From Scratch.
5. This opens the Select a Template Group page where the available template groups are displayed as shown in the example below.

   Select a Template Group

   Stock Exchange
   Domain for stock exchange analytics

6. Click on the template group from which you want to select the required sources and sinks for your business rule. For this example, click Stock Exchange to open that template group as shown below.
7. Click **Input** to expand the **Input** section. Then select the rule template from which the source and input configurations for the business rule must be selected.

This displays the list of available sources and the exposed attributes of the selected template as shown below.
8. Click Filters to expand the Filters section, and click + to add a new filter. A table is displayed as shown below.

9. To define a filter, follow the steps below:
   a. In the Attribute field, select the attribute based on which you want to define the filter condition.
   b. In the Operator field, select an operator.
   c. In the Value/Attribute field, enter the value or the attribute based on which the operator is applied to the attribute you selected in step a.

   e.g., If you want to filter events where the price is less than 100, select values for the fields as follows:
Once you have defined two or more filters, enter the rule logic in the Rule Logic field using OR, AND and NOT conditions. The examples of how you can use these keywords are explained in the table below.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>1 OR 2 returns events that match either filter 1 or 2.</td>
</tr>
<tr>
<td>AND</td>
<td>1 AND 2 returns events that match both filters 1 and 2.</td>
</tr>
<tr>
<td>NOT</td>
<td>NOT 1 returns events that do not match filter 1.</td>
</tr>
</tbody>
</table>

10. Click Output to expand the Output section. Then select the rule template from which the sink and output configurations for the business rule must be selected.

This displays the section for mapping configurations as shown in the example below.
11. Select the relevant attribute names for the **Input** column. When publishing the events to which the rule is applied via the selected predefined sink, each input event you select is published with the corresponding name in the **Output** column.

The output mappings displayed differ based on the output rule template you select.

12. If you want to save the rule and deploy it later, click **Save**. If you want to deploy the rule immediately, click **Save and Deploy**.

**Managing Business Rules**

This section covers how to view, edit, deploy and delete business rules that are created from a template.

- Prerequisites
- Viewing business rules
- Editing business rules
- Deploying business rules
- Deleting business rules

**Prerequisites**

In order to manage business rules, the following prerequisites must be completed:

- One or more business rules must be already created. For instructions to create a rule, see Creating Business Rules.
- The Business Rules Manager should be started and accessed by following the procedure below.
  1. Start the Dashboard Portal of WSO2 SP with one of the following commands:
     - On Windows: `dashboard.bat --run`
     - On Linux/Mac OS: `./dashboard.sh`
2. Start a worker node with one of the following commands:
   - On Windows: `worker.bat --run`
   - On Linux/Mac OS: `./worker.sh`

3. Access the Business Rule Manager via one of the following URLs.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>URL Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>http://&lt;SP_HOST&gt;:&lt;HTTP_PORT&gt;/portal</td>
<td><a href="http://0.0.0.0:9443/business-rules/">http://0.0.0.0:9443/business-rules/</a></td>
</tr>
<tr>
<td>HTTPS</td>
<td>https://&lt;SP_HOST&gt;:&lt;HTTPS_PORT&gt;/portal</td>
<td><a href="https://0.0.0.0:9443/business-rules/">https://0.0.0.0:9443/business-rules/</a></td>
</tr>
</tbody>
</table>

**Viewing business rules**

Once you start and access the Business Rules Manager, the available business rules are displayed as shown in the example below.

![Business Rules Manager](image)

To view a business rule, click the icon for viewing (marked in the above image) for the relevant row. This opens the rule as shown in the example below.
Editing business rules

Once you start and access the Business Rules Manager, the available business rules are displayed as shown in the example below.

To edit a business rule, click the icon for editing (marked in the above image) for the relevant row. This opens the rule as shown in the example below.
Modify values for the parameters displayed as required and click Save.

**Deploying business rules**

To deploy an existing undeployed business rule, follow the procedure below:

<table>
<thead>
<tr>
<th>Business Rule</th>
<th>Status</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Production Alert</td>
<td>Deployed</td>
<td></td>
</tr>
<tr>
<td>Excess Production Alert</td>
<td>Saved</td>
<td></td>
</tr>
</tbody>
</table>

To deploy a business rule, click the icon for deploying (marked in the above image) for the relevant row. As a result, a message appears to inform...
you that the rule is successfully deployed.

If the icon marked in the image above is not displayed for a business rule, it means that the rule is already deployed.

**Deleting business rules**

To delete an existing business rule, follow the procedure below:

To delete a business rule, click the icon for deleting (marked in the above image) for the relevant row. A message appears to confirm whether you want to proceed with the deletion. Click **Delete** in the message. As a result, another message appears to inform you that the rule is successfully deleted.

### Creating a Business Rule Template

To create a business template using the Business Template editor, follow the procedure below:

1. Go to `<SP_HOME>` from the terminal and Access the Stream Processor Studio via the [http://<HOST_NAME>:<EDITOR_PORT>/editor](http://<HOST_NAME>:<EDITOR_PORT>/editor) URL.
   - On Windows: `editor.bat --run`
   - On Linux/Mac OS: `editor.sh`


   The default URL is [http://localhost:9390/template-editor](http://localhost:9390/template-editor)

3. The Template Editor opens as shown below. There are two views from which you can interact and create a template group. **Design view** allows you to visualize a template group and interact with it. **Code view** allows you to interact with a template group by typing content. (For more information about template group structure, see [Business Rules Templates](#).)

   Do not template sensitive information such as passwords in a Siddhi application or expose them directly in a Siddhi application. For detailed instructions to protect sensitive data by obfuscating them, see [Protecting Sensitive Data via the Secure Vault](#).
The following sections explain the two methods of creating a template group. To create a template group:

- Create from Design View
- Create from code view

Create from Design View

To create a business rules template group from the design view, follow the procedure below:

1. Enter a UUID (Universally Unique Identifier), name and a description for the template group as follows.

<table>
<thead>
<tr>
<th>Field</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>UUID</td>
<td>sweet-factory</td>
</tr>
<tr>
<td>Name</td>
<td>Sweet Factory</td>
</tr>
<tr>
<td>Description</td>
<td>Analyzes Sweet Factory scenarios</td>
</tr>
</tbody>
</table>

2. Expand the first rule template that exists by default, and enter the following details. (Note that, you need to configure the deployment nodes as explained in Prerequisites for Business Rules)
2. To include a Siddhi application template, expand the first template that is displayed by default, and enter the following Siddhi application template.
A script is a javascript that can be applied when the inputs provided by the business user who uses the template need to be processed before replacing the values for the template variables. e.g., If the average value is not provided, a function within the script can derive it by calculating it from the minimum value and the maximum value provided by the business user.
5. To specify the attributes that need to be considered as variables, select the relevant check boxes under **Select templated elements**. In this example, you can select the `username` and `timeRange` check boxes to select the attributes with those names as the variables.

Then click **Add Script** to update the script with the selected variables with auto-generated function bodies as shown below.
6. Edit the script to add the required functions. In this example, let's rename `myFunction1(input)` to `getUsername(email)`, and `myFunction2(input)` to `validateTimeRange(number)`. 
```javascript
var username = getUsername('${userInputForUsername}');
var timeRange = validateTimeRange('${userInputForTimeRange}');

/**
 * Extracts the username from given email
 * @returns Extracted username
 * @param email Provided email
 */
function getUsername(email) {
  if (email.match(/\S+@\S+/g)) {
    if (email.match(/\S+@\S+/g)[0] === email) {
      return email.split('@')[0];
    }
    throw 'Invalid email address provided';
  }
  throw 'Invalid email address provided';
}

/**
 * Validates the given value for time range
 * @returns Processed input
 * @param input User given value
 */
function validateTimeRange(number) {
  if (isNaN(number) || number > 0) {
    return number;
  } else {
    throw 'A positive number expected for time range';
  }
}
```
var username = getUsername('${userInputForusername}');
var timeRange = validateTimeRange('${userInputFortimeRange}');

/**
 * Extracts the username from given email
 * @returns Extracted username
 * @param email Provided email
 */
function getUsername(email) {
    if (email.match(/\S+@\S+/g)) {
        if (email.match(/\S+@\S+/g)[0] === email) {
            return email.split('@')[0];
        }
        throw 'Invalid email address provided';
    }
    throw 'Invalid email address provided';
}

/**
 * Validates the given value for time range
 * @returns Processed input
 * @param input User given value
 */
function validateTimeRange(number) {
    if (!isNaN(number) && (number > 0)) {
        return number;
    } else {
        throw 'A positive number expected for time range';
    }
}

7. To generate properties, click Generate against Properties.

Properties

This expands the Properties section as follows.

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>timeInterval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>userInputForusername</td>
<td></td>
<td></td>
</tr>
<tr>
<td>userInputFortimeRange</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8. Enter values for the available properties as follows. For this example, let's enter values as shown in the following table.

A property is defined for each templated attribute (defined in the `${templatedElement}` format) so that it is self descriptive for the business user who uses the template. The values configured for each property is as follows:

- **Field Name**: The name with which the templated attribute is displayed to the business user.
- **Field Description**: A description of the property for the business user to understand its purpose.
- **Default Value**: The value assigned to the property by default. The business user can change this value if required.
- **Options**: This is an optional configuration that allows you to define a set of values for a property so that the business user can select the required value from a list. This is useful when the possible value for the property is a limited set of options.

<table>
<thead>
<tr>
<th>Property</th>
<th>Field Name</th>
<th>Field Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeInterval</td>
<td>Time interval (in seconds)</td>
<td>Production amounts are considered per time interval</td>
<td>6</td>
</tr>
<tr>
<td>userInputForusername</td>
<td>Manager Email ID</td>
<td>Email address to show in greeting</td>
<td><a href="mailto:example@email.com">example@email.com</a></td>
</tr>
<tr>
<td>userInputFortimeRange</td>
<td>Time Range (in milliseconds)</td>
<td>Time period in which, product amounts are analyzed for decrease</td>
<td>5</td>
</tr>
</tbody>
</table>

9. To save the template, click the save icon at the top of the page.

Create from code view

When you use the code view, the same parameters for which you enter values in the design view are represented as JSON keys. For each parameter, you can specify a value against the relevant JSON key as shown in the extract below.
When you update the code view with a valid template group definition, the design view is updated simultaneously as shown below.

However, if the content you enter in the code view is an invalid template group, the design view is not updated, and an error is displayed as follows.

 Error in code view. Design view & saving the template rely on the latest valid configuration

'type' of rule template 'identifying-continuous-production-decrease' should be template/input/output
When an error is detected in the entered template group structure, the **Recover** button is displayed with the error message.

When you click **Recover**, the code view is receted to the latest detected valid template group definition. At any given time, the design view displays information based on the latest detected valid template group definition.

It is not recommended to add Siddhi application templates and scripts using the code view because they need to be provided as a single line, and the possible escape characters should be handled carefully.

**Business Rules Templates**

Rule Templates are used as specifications to gain inputs from users through dynamically generated fields, for creating business rules. A template group is a business domain level grouping. The definition of a template looks as follows.

```json
{
    "templateGroup" : {
        "name" : "<Name of the template group>",
        "uuid" : "<UUID for the template group>",
        "description" : "<(Optional) description for the template group>",
        "ruleTemplates" : [
            {
                "name" : "<Name of the rule template>",
                "uuid" : "<UUID for the rule template>",
                "type" : "template",
                "instanceCount" : "one <or> many",
                "description" : "<(Optional) description for the rule template>",
                "script" : "<(Optional) Javascript with reference to the properties>",
                "templates" : [
                    {
                        "type" : "siddhiApp",
                        "content" : "<SiddhiApp_1 with ${templatedProperty_x}>"
                    },
                    {
                        "type" : "siddhiApp",
                        "content" : "<SiddhiApp_n with ${templatedProperty_y}>"
                    }
                ],
                "properties" : {
                    "templatedProperty_x" : {
                        "fieldName" : "<Field name for the property>",
                        "description" : "<Description for the property>",
                        "defaultValue" : "<Default value for the property>"},
                    "templatedProperty_y" : {
                        "fieldName" : "<Field name for the property>",
                        "description" : "<Description for the property>",
                        "defaultValue" : "<Default value for the property>",
                        "options" : [{"<option_1>"}, "<option_n>"]
                    }
                }
            }
        ]
    }
}
```
null
The following parameters are configured:

**Template Group basic data**

The following parameters are configured under `templateGroup`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A name for the template group</td>
<td>Required</td>
</tr>
<tr>
<td>uuid</td>
<td>A uniquely identifiable id for the template group</td>
<td>Required</td>
</tr>
<tr>
<td>description</td>
<td>A description for the template.</td>
<td>Optional</td>
</tr>
</tbody>
</table>

**Rule Template details**

Multiple rule templates can be defined under a `templateGroup`. For each `ruleTemplate`, the following set of parameters need to be configured:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A name for the rule template</td>
<td>Required</td>
</tr>
<tr>
<td>uuid</td>
<td>A uniquely identifiable id for the rule template</td>
<td>Required</td>
</tr>
<tr>
<td>type</td>
<td>The type of the rule template. Possible values are as follows:</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>* template: Used only to create an entire business rule from template</td>
<td></td>
</tr>
<tr>
<td></td>
<td>* input : Used only in creating a business rule from scratch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>* output : Used only in creating a business rule from scratch</td>
<td></td>
</tr>
<tr>
<td>instanceCount</td>
<td>This specifies whether the business rules derived from the template can be deployed only on one node, or whether they can be deployed on many nodes.</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>Possible values are as follows:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>* one</td>
<td></td>
</tr>
<tr>
<td></td>
<td>* many</td>
<td></td>
</tr>
</tbody>
</table>
The Java script to be executed on the templated fields. Developers can use this script for:

- validating purposes.
- deriving values for a templated parameter by combining some other entered parameters

Each templated element that is going to be derived from entered parameters, has to be mentioned as a variable in the global scope of the javascript.

The entered parameters should be templated in the script itself, and will be later replaced with their respective entered values.

Consider the following script

```javascript
/*
* Validates a number and returns after adding 10 to it
* @throws Error when a non number is entered
*/
function deriveValue(value){
    if( !isNaN(value) ) {
        return value + 10;
    }
    throw "A number is required";
}

var derivedValue = deriveValue(${enteredValue});
```

enteredValue should be defined as a property under properties in order to be filled by the user and replaced later.

The derived value stored in derivedValue will be then used to replace ${derivedValue} in the SiddhiApp template.

<table>
<thead>
<tr>
<th>Description</th>
<th>A brief description of the rule template.</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Templates</td>
<td>These are the artifacts (i.e SiddhiApps) with templated parameters, that will be instantiated with replaced values when a business rule is created.</td>
<td>Required</td>
</tr>
<tr>
<td>Properties</td>
<td>You can add a field name, description, default value and possible values (optional) for the templated parameters.</td>
<td>Required</td>
</tr>
</tbody>
</table>

1. In the BusinessRules section of the `<SP_HOME>/conf/dashboard/deployment.yaml` file, add a configuration for the template you created as shown below.

```yaml
wso2.business.rules.manager:

datasource: <datasourceName>
  - nodeURL1:
    - ruleTemplateUUID1
    - ruleTemplateUUID2
  nodeURL2:
    - ruleTemplateUUID1
    - ruleTemplateUUID2
```
2. Save the template group as a .json file in the <SP_HOME>/wso2/dashboard/resources/businessRules/templates directory.

Configuring Business Rules Manager Permissions

There are two permission levels for a business rules application:

- **Manager**: User roles with this permission level have administrative privileges over business rules. They are allowed to create, view, edit, deploy or delete business rules.
- **Viewer**: User roles with this permission level are only allowed to view business rules.

The following topics cover how to configure Business Rules Manager permissions.

- Prerequisites
- Configuring permissions

Prerequisites

Before configuring Business Rules Manager permissions, the user roles to be assigned permissions must be already defined in the user store with the required user IDs. For detailed instructions, see User Management.

Configuring permissions

Roles related to the Business Rules Manager need to be added under the wso2.business.rules.manager component namespace in the <SP_HOME>/conf/dashboard/deployment.yaml file.

The following is a sample configuration of user roles for the Business Rules Manager.

```
wso2.business.rules.manager:
  roles:
    manager:
    - name: role1
      id: 1
    viewer:
    - name: role2
      id: 2
```

Monitoring Stream Processor

The Status Dashboard of WSO2 SP allows you to monitor the metrics of a stand-alone WSO2 SP instance or a WSO2 SP cluster. This involves monitoring whether all processes of the WSO2 SP setup are working in a healthy manner, monitoring the current status of a SP node, and viewing metrics relating to the history of a node or the cluster. Both JVM level metrics or Siddhi application level metrics can be viewed from the monitoring dashboard.

The following sections cover how to configure the Status Dashboard and analyze statistics relating to your WSO2 SP deployment in it.

- Configuring the Status Dashboard
- Viewing Statistics

Configuring the Status Dashboard

The following sections cover the configurations that need to be done in order to view statistics relating to the performance of your WSO2 SP deployment in the status dashboard.

- Assigning unique carbon IDs to nodes
- Setting up the database
- Configuring metrics
- Configuring cluster credentials
- Configuring permissions

Assigning unique carbon IDs to nodes

Carbon metrics uses the carbon ID as the source ID for metrics. Therefore, all the worker nodes are required to have a unique carbon ID defined
to uniquely identify a server
id: wso2-sp
  # server name
name: WSO2 Stream Processor
  # ports used by this server

Setting up the database

To monitor statistics in the Status Dashboard, you need a shared metrics database that stores the metrics of all the nodes.
Set up a database of the required type by following the steps below. In this section, a MySQL database is created as an example.

1. Download and install the required database type. For this example, let's download and install MySQL Server.
2. Download the required database driver. For this example, download the MySQL JDBC driver.
3. Unzip the database driver you downloaded, copy its JAR file (mysql-connector-java-x.x.xx-bin.jar in this example), and place it in the <SP_HOME>/lib directory.
4. Enter the following command in a terminal/command window, where username is the username you want to use to access the database.
   ```bash
   mysql -u username -p
   ```
   When prompted, specify the password you are using to access the databases.
5. When prompted, specify the password you are using to access the databases with the username you specified.
6. Create two tables named WSO2_METRICS_DB (to store metrics data) and WSO2_STATUS_DASHBOARD_DB (to store statistics). To create MySQL tables for this example, run the following commands.
   ```sql
   mysql> create database WSO2_METRICS_DB;
   mysql> use WSO2_METRICS_DB;
   mysql> source <SP_HOME>/wso2/editor/dbscripts/metrics/mysql.sql;
   mysql> grant all on WSO2_METRICS_DB.* TO username@localhost identified by "password";
   ```
   ```sql
   mysql> create database WSO2_STATUS_DASHBOARD_DB;
   mysql> use WSO2_STATUS_DASHBOARD_DB;
   mysql> source <SP_HOME>/wso2/editor/dbscripts/metrics/mysql.sql;
   mysql> grant all on WSO2_STATUS_DASHBOARD_DB.* TO username@localhost identified by "password";
   ```

- Create two datasources named WSO2_METRICS_DB and WSO2_STATUS_DASHBOARD_DB by adding the following datasource configurations under the wso2.datasources section of the <SP_HOME>/conf/worker/deployment.yaml file.

The names of the datasources must be the same as the names of the database tables you created for metrics and statistics. You need to change the values for the username and password parameters to the username and password that you are using to access the MySQL database.
For detailed information about datasources, see carbon-datasources.
WSO2_METRICS_DB

- name: WSO2_METRICS_DB
description: The datasource used for dashboard feature
jndiConfig:
  name: jdbc/WSO2MetricsDB
definition:
type: RDBMS
configuration:
jdbcUrl:
'jdbc:mysql://localhost/WSO2_METRICS_DB?useSSL=false'
  username: root
  password: root
  driverClassName: com.mysql.jdbc.Driver
  maxPoolSize: 50
  idleTimeout: 60000
  connectionTestQuery: SELECT 1
  validationTimeout: 30000
  isAutoCommit: false

WSO2_STATUS_DASHBOARD_DB

- name: WSO2_STATUS_DASHBOARD_DB
description: The datasource used for dashboard feature
jndiConfig:
  name: jdbc/wso2_status_dashboard
  useJndiReference: true
definition:
type: RDBMS
configuration:
jdbcUrl:
'jdbc:mysql://localhost/WSO2_STATUS_DASHBOARD_DB?useSSL=false'
  username: root
  password: root
  driverClassName: com.mysql.jdbc.Driver
  maxPoolSize: 50
  idleTimeout: 60000
  connectionTestQuery: SELECT 1
  validationTimeout: 30000
  isAutoCommit: false

The following are sample configurations for database tables when you use other database types supported.
Click here to view the sample datasource configurations

<table>
<thead>
<tr>
<th>Database Type</th>
<th>MetricsDatasource</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MSSQL

name: WSO2_METRICS_DB
description: The datasource used for dashboard feature
jndiConfig:

definition:
type: RDBMS
configuration:

jdbcUrl: jdbc:sqlserver://localhost;databaseName=wso2_metrics
username: root
connectionTestQuery: SELECT 1
validationTimeout: 30000
isAutoCommit: false
### Oracle

<table>
<thead>
<tr>
<th>name: WSO2_METRICS_DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>description: The datasource used for dashboard feature</td>
</tr>
<tr>
<td>jndiConfig:</td>
</tr>
<tr>
<td>name: jdbc/WSO2MetricsDB</td>
</tr>
<tr>
<td>definition:</td>
</tr>
<tr>
<td>type: RDBMS</td>
</tr>
<tr>
<td>configuration:</td>
</tr>
<tr>
<td>jdbcUrl: 'jdbc:oracle:thin:@localhost:1521/xe'</td>
</tr>
<tr>
<td>username: root</td>
</tr>
<tr>
<td>password: root</td>
</tr>
<tr>
<td>driverClassName: oracle.jdbc.driver.OracleDriver</td>
</tr>
<tr>
<td>maxPoolSize: 50</td>
</tr>
<tr>
<td>idleTimeout: 60000</td>
</tr>
<tr>
<td>connectionTestQuery: SELECT 1</td>
</tr>
<tr>
<td>validationTimeout: 30000</td>
</tr>
<tr>
<td>isAutoCommit: false</td>
</tr>
</tbody>
</table>

---

**Configuring metrics**

This section explains how to configure metrics for your status dashboard.

**Configuring worker metrics**

To enable metrics and to configure metric-related properties, do the following configurations in the `<SP_HOME>/conf/worker/deployment.yaml` file for the required nodes.

1. **To enable Carbon metrics**, set the `enabled` property to `true` under `wso2.metrics` as shown below.

   ```yaml
   wso2.metrics:
     enabled: true
   ```

2. **To enable JDBC reporting**, set the `Enable JDBC parameter` to `true` in the `wso2.metrics.jdbc: => reporting: subsection as shown below. If JDBC reporting is not enabled, only real-time metrics are displayed in the first page of the Status dashboard, and information relating to metrics history is not displayed in the other pages of the dashboard. To render the first entry of the graph, you need to wait for the time duration specified as the `pollingPeriod`.

   ```yaml
   # Enable JDBC Reporter
   name: JDBC
   enabled: false
   pollingPeriod: 60
   ```
3. Under `wso2.metrics.jdbc`, configure the following properties to clean up the database entries.

```yaml
wso2.metrics.jdbc:
  # Data Source Configurations for JDBC Reporters
dataSource:
    # Default Data Source Configuration
    - &JDBC01
      # JNDI name of the data source to be used by the JDBC Reporter.
      # This data source should be defined in a *-datasources.xml file in conf/datasources directory.
      dataSourceName: java:comp/env/jdbc/WSO2MetricsDB
      # Schedule regular deletion of metrics data older than a set number of days.
      # It is recommended that you enable this job to ensure your metrics tables do not get extremely large.
      scheduledCleanup:
        # Enable scheduled cleanup to delete Metrics data in the database.
        enabled: false
        # The scheduled job will cleanup all data older than the specified days
        daysToKeep: 7
        # This is the period for each cleanup operation in seconds.
        scheduledCleanupPeriod: 86400
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataSource</td>
<td>&amp;JDBC01</td>
<td>The name of the datasource used to store metric data.</td>
</tr>
<tr>
<td>dataSource &gt; dataSourceName</td>
<td>java:comp/env/jdbc/WSO2MetricsDB</td>
<td></td>
</tr>
<tr>
<td>dataSource &gt; scheduledCleanup &gt; enabled</td>
<td>false</td>
<td>If this is set to true, metrics data stored in the database is cleared at a specific time interval as scheduled.</td>
</tr>
<tr>
<td>dataSource &gt; scheduledCleanup &gt; daysToKeep</td>
<td>3</td>
<td>If scheduled clean-up of metric data is enabled, all metric data in the database that are older than the number of days specified in this parameter are deleted.</td>
</tr>
<tr>
<td>dataSource &gt; scheduledCleanup &gt; scheduledCleanupPeriod</td>
<td>86400</td>
<td>This parameter specifies the time interval in seconds at which all metric data stored in the database must be cleared.</td>
</tr>
</tbody>
</table>

4. JVM metrics of which the log level is set to OFF are not measured by default. If you need to monitor one or more of them, add the relevant metric name(s) under the `wso2.metrics: levels` subsection as shown in the extract below. As shown below, you also need to mention log4j mode in which the metrics need to be monitored (i.e., OFF, INFO, DEBUG, TRACE, or ALL).
wso2.metrics:
    # Enable Metrics
    enabled: true
    jmx:
        # Register MBean when initializing Metrics
        registerMBean: true
        # MBean Name
        name: org.wso2.carbon:type=Metrics
        # Metrics Levels are organized from most specific to least:
        # OFF (most specific, no metrics)
        # INFO
        # DEBUG
        # TRACE (least specific, a lot of data)
        # ALL (least specific, all data)
    levels:
        # The root level configured for Metrics
        rootLevel: INFO
        # Metric Levels
        levels:
            jvm.buffers: 'OFF'
            jvm.class-loading: INFO
            jvm.gc: DEBUG
            jvm.memory: INFO

Click here to view the default metric levels supported...

- **Class loading**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jvm.class-loading.loaded.current</td>
<td>INFO</td>
<td>The gauge showing the number of classes currently loaded for the JVM.</td>
</tr>
<tr>
<td>jvm.class-loading.loaded.current</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.class-loading.unloaded.total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Garbage collector**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jvm.gc.PS-MarkSweep.count</td>
<td>DEBUG</td>
<td>The gauge for showing the garbage collection and memory usage. Monitoring this allows you to identify memory leaks and memory trash that have a negative impact on performance.</td>
</tr>
<tr>
<td>jvm.gc.PS-MarkSweep.time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.gc.PS-Scavenge.time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Memory**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metric.level.jvm.gc</td>
<td>INFO</td>
<td>The gauge for showing the used and committed memory in WSO2 SP.</td>
</tr>
<tr>
<td>Property</td>
<td>Default Level</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>jvm.memory.heap.committed</td>
<td>INFO</td>
<td>The gauge for showing the used and committed heap in WSO2 SP.</td>
</tr>
<tr>
<td>jvm.memory.heap.init</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.memory.heap.max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.memory.heap.usage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.memory.non-heap.committed</td>
<td>INFO</td>
<td>The gauge for showing the used code cache and used CMS Perm Gen in WSO2 SP.</td>
</tr>
<tr>
<td>jvm.memory.non-heap.init</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.memory.non-heap.max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.memory.non-heap.usage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.memory.non-heap.used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.memory.total.committed</td>
<td>INFO</td>
<td>The gauge for showing the total memory currently available for the JVM.</td>
</tr>
<tr>
<td>jvm.memory.total.init</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.memory.total.max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.memory.total.used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jvm.memory.pools</td>
<td>OFF</td>
<td></td>
</tr>
<tr>
<td>jvm.memory.pools</td>
<td>OFF</td>
<td></td>
</tr>
</tbody>
</table>

**Operating system load**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jvm.os</td>
<td>INFO</td>
<td>The gauge for showing the current load imposed by the JVM on the operating system.</td>
</tr>
<tr>
<td>jvm.os.cpu.load.process</td>
<td>INFO</td>
<td></td>
</tr>
<tr>
<td>jvm.os.cpu.load.process</td>
<td>INFO</td>
<td></td>
</tr>
<tr>
<td>jvm.os.system.load.average</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Threads**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jvm.threads.count</td>
<td>Debug</td>
<td>The gauge for showing the number of active and idle threads currently available in the JVM thread pool.</td>
</tr>
<tr>
<td>jvm.threads.daemon.count</td>
<td>Debug</td>
<td>The gauge for showing the number of active daemon threads currently available in the JVM thread pool.</td>
</tr>
<tr>
<td>jvm.threads.blocked.count</td>
<td>OFF</td>
<td>The gauge for showing the number of threads that are currently blocked in the JVM thread pool.</td>
</tr>
<tr>
<td>jvm.threads.deadlock.count</td>
<td>OFF</td>
<td>The gauge for showing the number of threads that are currently deadlocked in the JVM thread pool.</td>
</tr>
<tr>
<td>jvm.threads.new.count</td>
<td>OFF</td>
<td>The gauge for showing the number of new threads generated in the JVM thread pool.</td>
</tr>
<tr>
<td>jvm.threads.runnable.count</td>
<td>OFF</td>
<td>The gauge for showing the number of runnable threads currently available in the JVM thread pool.</td>
</tr>
<tr>
<td>jvm.threads.terminated.count</td>
<td>OFF</td>
<td>The gauge for showing the number of threads terminated from the JVM thread pool since user started running the WSO2 API Manager instance.</td>
</tr>
</tbody>
</table>
### File descriptor details

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file.descriptor.max.count</td>
<td>INFO</td>
<td>The gauge for showing the number of open file descriptors and max file descriptors.</td>
</tr>
<tr>
<td>file.descriptor.open.count</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Swap space

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>swap.space.free.size</td>
<td>INFO</td>
<td>The gauge for showing the amount of free and total swap space in bytes.</td>
</tr>
<tr>
<td>swap.space.total.size</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Configuring Siddhi application metrics

To enable Siddhi application level metrics for a Siddhi application, you need to add the `@app:statistics` annotation below the Siddhi application name in the Siddhi file as shown in the example below.

```java
@App:name('TestMetrics')
@app:statistics(reporter = 'jdbc')
define stream TestStream (message string);
```

The following are the metrics measured for a Siddhi application.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Components to which the metric is applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>• Windows (per window.find and window.add)</td>
</tr>
<tr>
<td></td>
<td>• Mappers (per sink mapper, source mapper)</td>
</tr>
<tr>
<td></td>
<td>• Queries (per query)</td>
</tr>
<tr>
<td></td>
<td>• Tables (per table insert, find, update, updateOrInsert, delete, contains)</td>
</tr>
<tr>
<td>Throughput</td>
<td>• Windows (per window.find and window.add)</td>
</tr>
<tr>
<td></td>
<td>• Mappers (per sink mapper, source mapper)</td>
</tr>
<tr>
<td></td>
<td>• Queries (per query)</td>
</tr>
<tr>
<td></td>
<td>• Tables (per table insert, find, update, updateOrInsert, delete, contains)</td>
</tr>
<tr>
<td>Memory</td>
<td>Queries (per query)</td>
</tr>
<tr>
<td>Buffered Events Count</td>
<td>Streams (per stream)</td>
</tr>
<tr>
<td>Number of events at disruptor</td>
<td>Sources (per source)</td>
</tr>
<tr>
<td>Number of events produced/received after restart</td>
<td>Sinks (per sink)</td>
</tr>
</tbody>
</table>

### Configuring cluster credentials

In order to access the nodes in a cluster and derive statistics, you need to maintain and share a user name and a password for each node in a SP cluster. This user name and password must be specified in the `<SP_HOME>/conf/dashboard/deployment.yaml` file. If you want to secure
sensitive information such as the user name and the password, you can encrypt them via WSO2 Secure Vault.

1. To specify the user name and the password to access a node, define them under the `wso2.status.dashboard` section as shown in the following example.

   ```yaml
   wso2.status.dashboard:
       workerAccessCredentials:
           username: 'admin'
           password: 'admin'
   ```

2. To encrypt the user name and the password you defined, define aliases for them as described in Protecting Sensitive Data via the Secure Vault.

   This functionality is currently supported only for single tenant environments.

Configuring permissions

The following are the three levels of permissions that can be granted for the users of WSO2 Stream Processor Status Dashboard.

<table>
<thead>
<tr>
<th>Permission Level</th>
<th>Granted permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SysAdmin</td>
<td>• Enabling/disabling metrics</td>
</tr>
<tr>
<td></td>
<td>• Adding workers</td>
</tr>
<tr>
<td></td>
<td>• Deleting workers</td>
</tr>
<tr>
<td></td>
<td>• Viewing workers</td>
</tr>
<tr>
<td>Developers</td>
<td>• Adding workers</td>
</tr>
<tr>
<td></td>
<td>• Deleting workers</td>
</tr>
<tr>
<td></td>
<td>• Viewing workers</td>
</tr>
<tr>
<td>Viewers</td>
<td>Viewers</td>
</tr>
</tbody>
</table>

The `admin` user in the userstore is assigned the `SysAdmin` permission level by default.

To assign different permission levels to different roles, you can list the required roles under the relevant permission level in the `wso2.status.dashboard` section of the `SP_HOME/conf/dashboard/deployment.yaml` file as shown in the extract below.

```yaml
wso2.status.dashboard:
   sysAdminRoles:
      - role_1
   developerRoles:
      - role_2
   viewerRoles:
      - role_3
```

The display name of the roles given in the configuration must be present in the user store. To configure user store check, User Management.

Viewing Statistics

To view the status dashboard, follow the procedure below:

1. Start the dashboard for your worker node by issuing one of the following commands:
   - For Windows: `dashboard.bat`
   - For Linux: `./dashboard.sh`
2. Access the Status Dashboard via the following URL format.

   ```text
   https://localhost:<SP_DASHBOARD_PORT>/sp-status-dashboard
   ```
e.g., https://0.0.0.0:9643/sp-status-dashboard

After login this opens the Status Dashboard with the nodes that you have already added as shown in the example below.

If no nodes are displayed, add the nodes for which you want to view statistics by following the steps in Worker Overview - Adding a worker to the dashboard.

For a detailed description of each page in this dashboard, see the following topics:

- Node Overview
- Viewing Node-specific Pages
- Viewing Worker History
- Viewing Statistics for Siddhi Applications
- Viewing Statistics for Parent Siddhi Applications
- App Overview

Node Overview

Once you login to the status dashboard, the nodes that are already added to the Status Dashboard are displayed as shown in the following example:

Adding a node to the dashboard

If no nodes are displayed, you can add the nodes for which you want to view the status by following the procedure below:

1. Click **ADD NEW NODE**.
This opens the following dialog box.

Let's add a new node

HOST

HTTPS PORT

ADD NODE  TEST CONNECTION  CANCEL

2. Enter the following information in the dialog box and click **ADD NODE** to add a gadget for the required node in the **Node Overview** page.
   a. In the **Host** parameter, enter the host ID of the node you want to add.
   b. In the **Port** parameter, enter the port number of the node you want to add.

3. If the node you added is currently unreachable, the following dialog box is displayed.

Unreachable Node

Node details you entered is currently unreachable. Please choose the run time environment.

WORKER  MANAGER  CANCEL

Click either **WORKER** or **MANAGER**. If you click **WORKER**, the node is displayed under **Never Reached**. If you click **Manager**, the node is displayed under **Distributed Deployments** as shown below.

Distributed Deployments

Managers

Manager is not reachable!

localhost:9443
Last Updated: N/A

The following basic details are displayed for each node.

- **CPU Usage**: The CPU resources consumed by the SP node out of the available CPU resources in the machine in which it is deployed is expressed as a percentage.
- **Memory Usage**: The memory consumed by the node as a percentage of the total memory available in the system.
- **Load Average**.
- **Siddhi Apps**: The total number of Siddhi applications deployed in the node.
### Viewing status details

The following is a list of sections displayed in the **Node Overview** page to provide information relating to the status of the nodes.

#### Distributed Deployments

<table>
<thead>
<tr>
<th>View (Example)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Distributed Deployments" /></td>
<td>The nodes that are connected in the distributed deployment are displayed under the relevant group ID in the status dashboard (e.g., sp in the above example). Both managers and workers are displayed under separate labels. <strong>Managers:</strong> The active manager node in the cluster is indicated by a green dot that is displayed with the host name and the port of the node. Similarly, a grey dot is displayed for passive manager nodes in the cluster. <strong>Workers:</strong> When you add an active manager node, it automatically retrieves the worker node details that are connected with that particular deployment. If the worker node is already registered in the Status Dashboard, you can view the metrics of that node as follows:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Purpose</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>To determine whether the request load is efficiently allocated between the nodes of a cluster.</td>
<td></td>
</tr>
<tr>
<td>To determine whether the cluster has sufficient resources to handle the load of requests.</td>
<td></td>
</tr>
<tr>
<td>To identify the nodes connected with the particular deployment.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Recommended Action</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>If there is a disparity in the CPU usage and the memory consumption of the nodes, redeploy the Siddhi applications between the nodes to balance out the workload.</td>
<td></td>
</tr>
<tr>
<td>If the CPU and memory are fully used and the request load is increasing, allocate more resources (e.g., more memory, more nodes, etc.).</td>
<td></td>
</tr>
</tbody>
</table>

#### Clustered nodes

<table>
<thead>
<tr>
<th>View (Example)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Clustered nodes" /></td>
<td>The nodes that are clustered together in a high-availability deployment are displayed under the relevant cluster ID in the Status Dashboard (e.g., under WSO2_A_1 in the above example). The active node in the cluster (i.e., the active worker in a minimum HA cluster or the active manager in a fully distributed cluster) are indicated by a green dot that is displayed with the hostname and the port of the node. Similarly, a grey dot is displayed for passive nodes in the cluster.</td>
</tr>
</tbody>
</table>
### Purpose

This allows you to determine the following:

- Whether the request load is efficiently allocated between the nodes of a cluster.
- Whether the cluster has sufficient resources to handle the load of requests.

### Recommended Action

- If there is a disparity in the CPU usage and the memory consumption of the nodes, redeploy the Siddhi applications between the nodes to balance out the workload.
- If the CPU and memory are fully used and the request load is increasing, allocate more resources (e.g., more memory, more nodes, etc.).

### Single nodes

#### View (Example)

#### Description

This section displays statistics for SP servers that operate as single node setups.

#### Purpose

This allows you to compare the performance of single nodes against each other.

#### Recommended Action

- If the CPU usage of a node is too high, investigate the causes for it and take corrective action (e.g., undeploy unnecessary Siddhi applications).
- If any underutilized single nodes are identified, you can either deploy more Siddhi applications that are currently deployed in other nodes with a high request load. Alternatively, you can redeploy the Siddhi applications of the underutilized node to other nodes, and then shut it down.

### Nodes that cannot be reached

#### View (Example)

#### Description

When a node is newly added to the Status dashboard and it is unavailable, it is displayed as shown in the above examples.

#### Purpose

This allows you to identify nodes that cannot be reached at specific hosts and ports.

#### Recommended Action

- Check whether the host and port of the node you added is correct.
- Check whether any authentication errors have occurred for the node.

### Nodes that are currently unavailable

#### View (Example)

#### Description

When a node that could be viewed previously is no longer available, its status is displayed in red as shown in the example above. The status displayed for such nodes is applicable for the last time at which the node had been reachable.

#### Purpose

This allows you to identify previously available nodes that have become unreachable.

#### Recommended Action

- Check whether the node is inactive.
- Check whether any authentication errors have occurred for the node.
### Nodes for which metrics is disabled

<table>
<thead>
<tr>
<th>View (Example)</th>
<th><img src="image1.png" alt="Image" /></th>
</tr>
</thead>
</table>

**Description**
When a node for which metrics is disabled is added to the Status dashboard, you can view the number of active and inactive Siddhi applications deployed in it. However, you cannot view the CPU usage, memory usage and the load average.

**Purpose**
This allows you to identify nodes for which metrics is not enabled.

**Recommended Action**
Enable metrics for the required nodes to view statistics about their status in the Status Dashboard. For instructions to enable metrics, see [Monitoring the Stream Processor - Configuring the Status Dashboard](#).

### Nodes with JMX reporting disabled

<table>
<thead>
<tr>
<th>View (Example)</th>
<th><img src="image2.png" alt="Image" /></th>
</tr>
</thead>
</table>

**Description**
When a node with JMX reporting disabled is added to the Status dashboard, you can view the number of active and inactive Siddhi applications deployed in it. However, you cannot view the CPU usage, memory usage and the load average.

**Purpose**
This allows you to identify nodes for which JMX reporting is disabled.

**Recommended Action**
Enable JMX reporting for the required nodes to view statistics about their status in the Status Dashboard. For instructions to enable JMX reporting, see [Monitoring the Stream Processor - Configuring the Status Dashboard](#).

### Statistics trends

<table>
<thead>
<tr>
<th>View (Example)</th>
<th><img src="image3.png" alt="Image" /></th>
</tr>
</thead>
</table>

**Description**
This displays the change that has taken place in the CPU usage, memory usage and the load average of nodes since the status was last viewed in the status dashboard. Positive changes are indicated in green (e.g., a decrease in the CPU usage in the above example), and negative changes are indicated in red (an increase in the memory usage and the load average in the above example).

**Purpose**
This allows you to view a summary of the performance trends of your SP clusters and single nodes.

**Recommended Action**
Based on the performance trend observed, add more resources to your SP clusters/single nodes to handle more events, or shutdown one or more nodes if there is excess resources.

### Viewing Node-specific Pages

When you open the WSO2 SP Status Dashboard, the **Node Overview** page is displayed by default. To view information specific to a selected worker node, click on the relevant widget. This opens a separate page for the worker node as shown in the example below.
Status indicators

The following gadgets can be viewed for the selected worker.

*Server General Details*
This gadget displays general information relating to the selected worker node.

Purpose
This allows you to understand the distribution of nodes in terms of the location, the time zone, operating system used etc., and to locate them.

Recommended Action
In a distributed set up, you can use this information to evaluate the clustered setup and make changes to optimize the benefits of deploying WSO2 SP as a cluster (e.g., making them physically available in different locations to minimize the risk of all the nodes failing at the same time etc.).

CPU Usage

View (Example)
**Memory Used**

<table>
<thead>
<tr>
<th>Description</th>
<th>This displays the memory usage of the selected node.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>This allows you to observe the memory usage of a selected node over time.</td>
</tr>
<tr>
<td>Recommended Action</td>
<td></td>
</tr>
</tbody>
</table>
  - Identify sudden slumps in the memory usage, and investigate the reasons (e.g., a reduction in the requests received due to system failure).
  - If there are continuous increases in the memory usage, check whether there is an increase in the requests handled, and whether you have enough memory resources to handle the increased demand. If not, add more memory to the node or reallocate some of the Siddhi applications deployed in the node to other nodes. |

**System Load Average**

<table>
<thead>
<tr>
<th>Description</th>
<th>This displays the system load average of the selected node.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>This allows you to observe the system load average of a selected node over time.</td>
</tr>
<tr>
<td>Recommended Action</td>
<td></td>
</tr>
</tbody>
</table>
  - Identify sudden slumps in the CPU usage, and investigate the reasons (e.g., such as authentication errors that result in requests not reaching the SP server).
  - Identify continuous increases in the CPU usage and check whether the node is overloaded. If so, reallocate some of the Siddhi applications deployed in the node. |
### Overall Throughput

<table>
<thead>
<tr>
<th>Description</th>
<th>This displays the overall throughput of the selected node.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>This allows you to assess the performance of the selected node in terms of the throughput over time.</td>
</tr>
<tr>
<td>Recommended Action</td>
<td></td>
</tr>
</tbody>
</table>
- Compare the throughput of the node against that of other nodes with the same amount of CPU and memory resources. If there are significant variations, investigate the causes (e.g., the differences in the number of requests received by different Siddhi applications deployed in the nodes).
- Observe changes in the throughput over time. If there are significant variances, investigate the causes (e.g., whether the node has been unavailable to receive requests during a given time).

### Siddhi Applications

<table>
<thead>
<tr>
<th>View (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Siddhi Applications</td>
</tr>
<tr>
<td>App Name</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>app1</td>
</tr>
<tr>
<td>app2</td>
</tr>
<tr>
<td>app3</td>
</tr>
<tr>
<td>app4</td>
</tr>
<tr>
<td>app5</td>
</tr>
</tbody>
</table>
Description
This table displays the complete list of Siddhi applications deployed in the selected node. The status is displayed in green for active Siddhi applications, and in red for inactive Siddhi applications. In addition, the following is displayed for each Siddhi application:

- **Age**: The age of the Siddhi application in milliseconds.
- **Latency**: The time (in milliseconds) taken by the Siddhi application to process one request.
- **Throughput**: The number of requests processed by the Siddhi application since it has been active.
- **Memory**: The amount of memory consumed by the Siddhi application during its current active session, expressed in milliseconds.

Purpose
This allows you to assess the performance of each Siddhi application deployed in the selected node.

Recommended Action
- Identify the inactive Siddhi applications that are required to be active and take the appropriate corrective action.
- Identify Siddhi applications that consume too much memory, and identify ways in which the memory usage can be optimized (e.g., use incremental processing).

Viewing Worker History
This section explains how to view statistics relating to the performance of a selected node for a specific time interval.

1. Login to the Status Dashboard. For detailed instructions, see Monitoring the Stream Processor - Viewing the Status Dashboard. When you login, the Node Overview page is displayed by default.
2. Click on the required node to view information specific to that node.
3. In the page displayed with node-specific information, click one of the following gadgets to open the Metrics page:
   - CPU Usage
   - Memory Used
   - System Load Average
   - Overall Throughput
4. In the Metrics page, click the required time interval. Then the page displays statistics relating to the performance of the selected node applicable to that time period.
5. If you want to view more details, click More Details.

As a result, the following additional information is displayed for the node for the selected time period.

- **CPU Usage**

- **JVM OS as CPU**
Viewing Statistics for Siddhi Applications

When you open the WSO2 SP Status Dashboard, the Node Overview page is displayed by default.

1. To view information specific to a selected worker node, click on the relevant gadget. This opens the page specific to the worker.
2. To view information specific to a Siddhi application deployed in the Siddhi node, click on the relevant Siddhi application in the Siddhi Applications table. This opens a page with information specific to the selected Siddhi application as shown in the example below.
The following statistics can be viewed for an individual Siddhi Application.

Latency

<table>
<thead>
<tr>
<th>View (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Latency Graph" /></td>
</tr>
</tbody>
</table>

| Description | This displays the latency of the selected Siddhi application. Latency is the time taken to complete processing a single event in the event flow. |
| Purpose | This allows you to assess the performance of the selected Siddhi application. |
| Recommended Action | If the latency of the Siddhi application is too high, check the Siddhi queries and rewrite them to optimise performance. |

Overall Throughput
View
(Example)

| Description | This shows the overall throughput of a selected Siddhi application over time. |
| Purpose | This allows you to assess the performance of the selected Siddhi application. |
| Recommended Action | • If the throughput of a Siddhi application varies greatly overtime, investigate reasons for any slumps in the throughput (e.g., errors in the deployment of the application).  
• If the throughput of the Siddhi application is lower than expected, investigate reasons, and take corrective action to improve the throughput (e.g., check the Siddhi queries in the application and rewrite them with best practices to achieve greater efficiency in the processing of events). |

Memory Used

| Description | This displays the memory usage (in MB) of a selected Siddhi application over time. |
| Purpose | This allows you to monitor the memory consumption of individual Siddhi applications. |
| Recommended Action | If there are major fluctuations in the memory consumption of a Siddhi application, investigate the reasons (e.g., Whether the Siddhi application has been inactive at any point of time). |
### View (Example)

<table>
<thead>
<tr>
<th>Description</th>
<th>This displays the queries defined in the Siddhi file of the application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>This allows you to check the queries of the Siddhi application if any further investigations are needed based on the observations of its latency, throughput and the memory consumption.</td>
</tr>
<tr>
<td>Recommended Action</td>
<td>Edit the Siddhi file if any changes that can improve the performance of the Siddhi application are identified. For detailed instructions to write a Siddhi application, see <a href="#">Creating a Siddhi Application</a>. For detailed information about the Siddhi logic, see the <a href="#">Siddhi Query Guide</a>.</td>
</tr>
</tbody>
</table>

### Design View

<table>
<thead>
<tr>
<th>Description</th>
<th>This displays the graphical view for queries defined in the Siddhi file of the application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>This allows you to check the flow of the queries of the Siddhi application in the graphical way.</td>
</tr>
<tr>
<td>Recommended Action</td>
<td>Edit the Siddhi file if any changes that can improve the performance of the Siddhi application are identified.</td>
</tr>
</tbody>
</table>

### Siddhi App Component Statistics
This table displays performance statistics related to different components within a selected Siddhi application (e.g., queries). The columns displayed are as follows:

- **Type**: The type of the Siddhi application component to which the information displayed in the row applies. The component type can be queries, streams, tables, windows and aggregations. For more information, see Siddhi Application Overview - Common components of a Siddhi application.
- **Name**: The name of the Siddhi component within the application to which the information displayed in the row apply.
- **Metric Type**: The metric type for which the statistics are displayed. This can be either the latency (in milliseconds), throughput (the number of events per second), or the amount of memory consumed (in bytes). The metric types based on which the performance of a Siddhi component is measured depends on the component type.
- **Attribute**: The attribute to which the given value applies.
- **Value**: The value for the metric type given in the row.

Purpose

This allows you to carry out a detailed analysis of the performance of a selected Siddhi application and identify components that have a negative impact on the overall performance of the Siddhi application.

Recommended Action

Identify the components in a Siddhi application that have a negative impact on the performance, and rewrite them to improve performance. To understand Siddhi concepts in order to rewrite the components, see the Siddhi Query Guide.

Viewing Statistics for Parent Siddhi Applications

When you open the WSO2 SP Status Dashboard, the Node Overview page is displayed by default. To view information specific to an active manager, click on the required active manager node in the Distributed Deployments section. This opens a page with parent Siddhi applications deployed in that manager node as shown in the example below.

This page provides a summary of information relating to each parent Siddhi application as described in the table below. If a parent Siddhi application is active, it is indicated with a green dot that appears before the name of the Siddhi application. Similarly, an orange dot is displayed for inactive parent Siddhi applications.
<table>
<thead>
<tr>
<th>Detail</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Groups</strong></td>
<td>This indicates the number of execution groups of the parent Siddhi application. In the above example, the Testing Siddhi application has only one execution group.</td>
</tr>
<tr>
<td><strong>Child Apps</strong></td>
<td>This indicates the number of child applications of the parent Siddhi application. The number of active child applications is displayed in green, and the number of inactive child applications are displayed in red.</td>
</tr>
</tbody>
</table>
| **Worker Nodes** | The number displayed in yellow indicates the total number of worker nodes in the resource cluster. In the above example, there are two worker nodes in the cluster.  
The number displayed in green indicates the number of worker nodes in which the Siddhi application is deployed. In the above example, the Testing parent Siddhi application is deployed only in one worker node although there are two worker nodes in the resource cluster. |

If you click on a parent Siddhi application, detailed information is displayed as shown below.

![Detailed Information](image)

The following are the widgets displayed.

**Code View**

**Description**

This displays the queries defined in the Parent Siddhi file of the application.

This allows you to check the queries of the Siddhi application if any further investigations are needed based on the kafka diagrams and performance.

For detailed instructions to write a Siddhi application, see Converting to a Distributed Streaming Application.

For detailed information about the Siddhi logic, see the Siddhi Query Guide.

**Purpose**

This allows you to check the queries of the Siddhi application if any further investigations are needed based on the observations of the performance of the distributed cluster to which it belongs.
**Recommended Action**

Edit the Siddhi file if any changes that can improve the performance of the Siddhi application are identified. For detailed instructions to write a Siddhi application, see Converting to a Distributed Streaming Application. For detailed information about the Siddhi logic, see the Siddhi Query Guide.

**Distributed Siddhi App Deployment**

<table>
<thead>
<tr>
<th>View (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Distributed Siddhi App Deployment" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a graphical representation of how Kafka topics are connected to the child Siddhi applications of the selected parent Siddhi application. Kafka topics are represented by boxes with red margins, and the child applications are represented by boxes with blue margins.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is displayed for you to understand how the flow of information takes place.</td>
</tr>
</tbody>
</table>

**Child App Details**

<table>
<thead>
<tr>
<th>View (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Child App Details" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>
| This table displays the complete list of child Siddhi applications of the selected parent Siddhi application. The status is displayed in green for active Siddhi applications, and in red for inactive Siddhi applications. In addition, the following is displayed for each Siddhi application:

- **Group Name**: The name of the execution group to which the child application belongs.
- **Child App Status**: This indicates whether the child application is currently active or not.
- **Worker Node**: The HTTPS host and The HTTPS port of the worker node in which the child siddhi application is deployed. |

<table>
<thead>
<tr>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>To identify the currently active child applications.</td>
</tr>
</tbody>
</table>

**App Overview**

When you open the WSO2 SP Status Dashboard, the Node Overview page is displayed by default. If you want to view all the Siddhi applications deployed in your WSO2 SP setup, click on the App View tab (marked in the image below). The App Overview tab opens and all the Siddhi applications that are currently deployed are displayed as shown in the image below.
The status is displayed in green for active Siddhi applications, and in red for inactive Siddhi applications.

If no Siddhi applications are deployed in your WSO2 SP setup, the following message is displayed.

There is no siddhi apps deployed

The Siddhi applications are listed under the deployment mode in which they are deployed (i.e., Single Node Deployment, HA Deployment, and Distributed Deployment).

If your WSO2 SP setup is a distributed deployment, only the parent Siddhi applications are displayed in this tab.

The following information is displayed for each Siddhi application.

- **Siddhi Application**: The name of the Siddhi application.
- **Status**: This indicates whether the Siddhi application is currently active or inactive.
- **Deployed Time**: The time duration that has elapsed since the Siddhi application was deployed in the SP setup.
- **Deployed Node**: The host and the port of the SP node in which the Siddhi application is deployed.

The purpose of this tab is to check the status of all the Siddhi applications that are currently deployed in the SP setup.

If you click on a Siddhi Application under Single Node Deployment or HA Deployment, information specific to that Siddhi application is displayed as explained in Viewing Statistics for Siddhi Applications.

If you click on the parent Siddhi application under Distributed Deployment, information specific to that parent Siddhi application is displayed as explained in Viewing Statistics for Parent Siddhi Applications.

If you click on a deployed node, information specific to that node is displayed as explained in Viewing Node-specific Pages.
Deployment Guide

WSO2 SP is deployed by starting two or more nodes with the same group ID in a cluster set up. The group ID and other configurations required for a cluster set up are edited in the `<SP_HOME>/conf/<worker|manager|editor|dashboard>/deployment.yaml` files.

A node is considered inactive when it fails to return a heartbeat pulse twice at a specified time interval. When an inactive node is identified, it is removed from the cluster. If that node becomes active again and rejoins the cluster, it is assigned a new ID. Therefore, at any given time, a cluster may have one or more active nodes. If all the nodes in a cluster are inactive, the cluster is considered inactive.

When a node with a specific group ID is started in a clustered setup, it registers itself as a member of the cluster identified by that group ID. If no other active node is available in that cluster, the node assigns itself as the leader of the cluster. If the leader of the cluster becomes inactive at any given time, an existing active node is assigned as the leader. The node assigned as the new leader is the one that acquires the database lock first. If a leader node that was inactive and removed from the cluster rejoins the cluster as an active member, it is not reassigned the role of the leader unless there are no other existing active nodes in that cluster (in which case, an existing node should be already functioning as the leader).

For detailed instructions to configure different types of WSO2 SP clusters, see the topics below.

- Minimum High Availability Deployment
- Fully Distributed Deployment
- Multi Datacenter High Availability Deployment

Minimum High Availability Deployment

When streaming events with WSO2 Stream Processor (SP), there is a possibility of the node running the SP instance failing due to several unforeseeable reasons. This leads to a loss in the events being streamed until the node can be restarted. A solution for this is to use the WSO2 SP in a High Availability (HA) environment where the processing of events is not halted at an unexpected failover scenario. The recommended HA deployment for WSO2 SP is the two node minimum HA deployment, where two instances of WSO2 SP are running in parallel as depicted in the diagram below.

In this minimum HA setup, one node is assigned as the active node while the other node is assigned as the passive node. Only the active node processes the incoming events and publishes the outgoing events. Internally, the active node publishes the events to the passive node, but the passive node does not process or send any events outside as mentioned earlier. In a scenario where the active node fails, the passive node is activated, and it starts receiving events and then publishes them from where the active node left off. Once the terminated (previously active) node restarts, it operates in the passive state. In the passive node, the databridge port and other ports related to sources remain closed until the node becomes active.

The ports that are open only in the active node at a given time include the Siddhi Store Query API endpoint to which requests are sent by invoking the Siddhi Store REST API. These ports are configured in the `<SP_HOME>/conf/worker/deployment.yaml` file. For more information about this port configuration, see Managing Stored Data via REST APIs.
For a two-node minimum HA cluster to work, only the active node should receive events. By design, you can only send events to active node. To achieve this, you can use a load balancing mechanism that sends events in failover manner.

**Prerequisites**

In order to configure a minimum HA cluster, the following prerequisites must be completed:

- It is recommended to run this setup with two CPUs. Each CPU should have four cores, and 4GB memory.
- Two binary packs of WSO2 SP must be available.
- A working RDBMS instance to be used for clustering of the 2 nodes.
- Download the MySQL connector from [here](#). Extract and find the `mysql-connector-java-*.jar`. Place this JAR in the `<SP_HOME>/lib` directory of both nodes.
- In order to retrieve the state of the Siddhi Applications deployed in the system in case of a scenario where both the nodes fail, state persistence must be enabled for both nodes by specifying the same `datasource/file` location. For detailed instructions, see Configuring Database and File System State Persistence.
- A load balancer or some other client-side data publishing mechanism that works in a failover manner must be available to publish events to one of the available nodes (i.e., the active node).

**Configuring a minimum HA cluster**

When a failover occurs, the Siddhi Store Query API endpoint configured in node 2 (which becomes the currently active node) is opened, and all the store query traffic is directed to that endpoint.
To configure a minimum HA cluster, follow the steps below:

1. For each node, enter a unique ID for the `id` property under the `wso2.carbon` section. (e.g., `id: wso2-sp`). This is used to identify each node within a cluster.
2. To allow the two nodes to use same persistence storage, you need to configure persistence configuration under the `state.persistence` section. State persistence can be configured to use db-based or file-based. However, the persistence storage needs to be shared between nodes irrespective of the method. The following is a configuration for db-based file persistence.

```yaml
state.persistence:
  enabled: true
  intervalInMin: 1
  revisionsToKeep: 2
  persistenceStore:
    org.wso2.carbon.stream.processor.core.persistence.DBPersistenceStore
    config:
      datasource: PERSISTENCE_DB   # A datasource with this name should be defined in wso2.datasources namespace
      table: PERSISTENCE_TABLE
```

The datasource named `PERSISTENCE_DB` in the above configuration can be defined in the `<SP_HOME>/conf/worker/deployment.yaml` file under the `wso2.datasources` section. Following is a sample datasource configuration.

```yaml
- name: PERSISTENCE_DB
  description: The MySQL datasource used for persistence
  jndiConfig:
    name: jdbc/PERSISTENCE_DB
  definition:
    type: RDBMS
    configuration:
      username: root
      password: root
      driverClassName: com.mysql.jdbc.Driver
      maxPoolSize: 50
      idleTimeout: 60000
      connectionTestQuery: SELECT 1
      validationTimeout: 30000
      isAutoCommit: false
```

3. To allow the two nodes in the cluster to coordinate effectively, configure carbon coordination by updating the `cluster.config` section of the `<SP_HOME>/conf/worker/deployment.yaml` as follows:

   a. To enable the cluster mode, set the `enabled` property to `true`.
   b. In order to cluster the two nodes together, enter the same ID as the group ID for both nodes (e.g., `groupId: group-1`).
   c. Enter the ID of the class that defines the coordination strategy for the cluster as shown in the example below.

   ```java
e.g., coordinationStrategyClass:
    org.wso2.carbon.cluster.coordinator.rdbms.RDBMSCoordinationStrategy
```

For more information on how to set up cluster coordination refer Configuring Cluster Coordination.
d. In the **strategyConfig** section, enter information as follows:
   i. For clustering of the two nodes to take place
      ii. Enter the name of the configured datasource shared by the nodes in the cluster as shown in the example below. Data handled by the cluster are persisted here.
         ```
         datasource: WSO2_CLUSTER_DB (A datasource with this name should be configured)
         ```
         Following is a sample datasource configuration for a MySQL datasource that should appear under the dataSources section of the wso2.datasources section in the `<SP_HOME>/conf/worker/deployment.yaml`. For detailed instructions of how to configure a datasource, see Configuring Datasources.

         ```yaml
         Sample MySQL datasource
         
         - name: WSO2_CLUSTER_DB
           description: The MySQL datasource used for Cluster Coordination
           jndiConfig:
             name: jdbc/WSO2ClusterDB
           definition:
             type: RDBMS
           configuration:
             jdbcUrl: 'jdbc:mysql://localhost:3306/WSO2_CLUSTER_DB?useSSL=false'
             username: root
             password: root
             driverClassName: com.mysql.jdbc.Driver
             maxPoolSize: 50
             idleTimeout: 60000
             connectionTestQuery: SELECT 1
             validationTimeout: 30000
             isAutoCommit: false
         ```

   iii. Define the time interval (in milliseconds) at which heartbeat pulse should occur for each node to indicate that it is in an active state as shown in the example below.
       ```
       heartbeatInterval: 1000
       ```

   iv. Define the number of times the heartbeat pulse should be unavailable at the specified time interval in order to consider a node inactive as shown in the example below. A value of two means that if a node fails to send two consecutive heartbeat pulses, it must be identified as inactive and removed from the cluster as a result.
       ```
       heartbeatMaxRetry: 2
       ```

   v. Define the time interval (in milliseconds) at which each node should listen for changes that occur in the cluster as shown in the example below.
       ```
       eventPollingInterval: 1000
       ```

4. Next add the `deployment.config` section to the `<SP_HOME>/conf/worker/deployment.yaml` file with following configurations:
   a. To enable 2 node minimum HA, set the `type` property to `ha` as shown below.
      ```
      type: ha
      ```

   b. To configure the TCP server via which event synchronization is carried out, add a subsection named `eventSyncServer` and enter information as follows:
      i. Set the host and port to enable direct communication between the two nodes via TCP calls as shown in the following example. This ensures that the communication between the nodes is instantaneous.
         ```
         host: localhost
         port: 9893
         ```

      ii. To define the address to which the TCP requests by the active node must be directed, enter the relevant host and port in the `advertisedHost` and `advertisedPort` parameters respectively as shown in the example below. These values are used to map the advertised host and port with the actual host and port defined.
advertisedHost: localhost
advertisedPort: 9893

iii. Define a number of boss threads for the TCP server to handle the connections as shown in the example below. 10 is specified by default.

bossThreads: 10

iv. Define a number of worker threads for the TCP server to handle the connections as shown in the example below. 10 is specified by default.

workerThreads: 10

c. To configure the TCP client via which requests are sent to the SP cluster, add a subsection named `eventSyncClientPool` and add information as follows:

i. Define the maximum number of active connections that must be allowed in the TCP client pool as shown in the example below. 10 is specified by default.

maxActive : 10

ii. Define the maximum number of total connections that must be allowed in the TCP client pool as shown in the example below. 10 is specified by default.

maxTotal : 10

iii. Define the maximum number of idle connections that must be allowed in the TCP client pool as shown in the example below. 10 is specified by default.

maxIdle : 10

iv. Define the number of milliseconds the client pool must wait for an idle object when the connection pool is exhausted. 60000 is specified by default.

maxWait: 60000

v. Define the minimum number of milliseconds an object can sit idle in the pool before it is eligible for eviction. The following is an example. 120000 is specified by default.

minEvictableIdleTimeInMillis : 120000

d. Define the capacity for the buffer queue as shown in the example below. This queue ensures that events are kept in the passive node and the state is synced correctly if a failover takes place. 20000 is specified by default.

minEvictableIdleTimeInMillis : 120000
e. Define the number of threads in the thread pool that extracts bytes from the event byte buffer queue. The following is an example. 5 is specified by default.

```
byterBufferExtractorThreadPoolSize: 5
```

The following extract is an example of the deployment configuration that is defined by following the above steps.

```
deployment.config:
  type: ha
  eventByteBufferQueueCapacity: 20000
  byteBufferExtractorThreadPoolSize: 5
  eventSyncServer:
    host: localhost
    port: 9893
    advertisedHost: localhost
    advertisedPort: 9893
  bossThreads: 10
  workerThreads: 10
  eventSyncClientPool
    maxActive: 10
    maxTotal: 10
    maxIdle: 10
    maxWait: 60000
    minEvictableIdleTimeMillis: 120000
```

### Publishing events to the cluster

The following diagram illustrates how events can be published to a two-node minimum HA cluster that uses sources such as HTTP, JMS queue, JMS topic etc.

The load balancer directs events only to the active SP node in the minimum HA cluster. The active node processes and publishes these events. The events are not published to the passive node. The events are buffered in the passive node so that the cluster can resume processing and publishing events via this node if the currently active node fails.
Starting the cluster

1. Save the required Siddhi applications in the `<SP_HOME>/deployment/siddhi-files` directory in both nodes. In order to ensure that the Siddhi applications are completely synchronized between the active and the passive node, they must be added to the `siddhi-files` directory before the server startup. However, the synchronization can take place effectively even if the Siddhi applications are added while the server is running.

   In deploying Siddhi applications in a two node minimum HA cluster, it is recommended to use a content synchronization mechanism since Siddhi applications must be deployed to both worker nodes. You can use a common shared file system such as Network File System (NFS) or any other shared file system that is available. You need to mount the `<SP_HOME>/deployment/siddhi-files` directory of the two nodes to the shared file system.

2. Start both servers by navigating to `<SP_HOME>/bin` and issuing the following command:
   For Windows: `worker.bat`
   For Linux: `./worker.sh`

   To start two WSO2 SP Nodes in the same machine, the `listenerConfigurations` under the `wso2.transport.http` namespace in the `<SP_HOME>/conf/worker/deployment.yaml` file must be updated to listen to different ports. The offset property under the `ports` section of the `wso2.carbon` section found in the `<SP_HOME>/conf/worker/deployment.yaml` should also be changed in one SP instance to avoid conflicts when starting both servers.

   If the cluster is correctly configured, the following CLI logs can be viewed without any error logs:

   In the active node:

   ```
   [2018-09-09 23:56:54,272] INFO
   {org.wso2.carbon.stream.processor.core.internal.ServiceComponent} - WSO2 Stream Processor Starting in Two Node Minimum HA Deployment
   [2018-09-09 23:56:54,294] INFO
   {org.wso2.carbon.stream.processor.core.ha.HAManager} - HA Deployment: Starting up as Active Node
   ```

   In the passive node:

   ```
   [2018-09-09 23:58:44,178] INFO
   {org.wso2.carbon.stream.processor.core.internal.ServiceComponent} - WSO2 Stream Processor Starting in Two Node Minimum HA Deployment
   [2018-09-09 23:58:44,199] INFO
   {org.wso2.carbon.stream.processor.core.ha.HAManager} - HA Deployment: Starting up as Passive Node
   ```

Fully Distributed Deployment

- Introduction
- Distributed Siddhi applications
- Deployment architecture
- Configuring a distributed cluster

Introduction

The most common deployment pattern for WSO2 SP is the Minimum High Availability Deployment that offers high availability with the minimum amount of resources. However, there are a few user scenarios where the HA (High Availability) deployment is not sufficient to handle the throughput.
The Distributed Deployment pattern is supported so that a high volume of data can be distributed among multiple SP instances instead of having them accumulated at a single point. It is suitable to be used in scenarios where the volume of data handled is too high to be managed in a single SP instance or a minimum high availability deployment.

**Distributed Siddhi applications**

A Siddhi Application is a combination of multiple Siddhi executional elements. A Siddhi executional element can be a Siddhi Query or a Siddhi Partition. In distributed processing perspective, a collection of these executional elements is called an execution group. Execution group is the smallest unit of execution.

Distributed processing of a Siddhi application allows users to execute multiple instances of each execution group in-parallel in multiple SP instances.

Users can specify execution groups and the parallelism to execute them by annotating existing Siddhi applications. Following sample application is annotated in that manner.

```yaml
@App:name('wso2-app')
@info(name = 'query-1')
@dist(execGroup='group-1')
from TempStream#window.time(2 min)
select avg(temp) as avgTemp, roomNo, deviceID
insert all events into AvgTempStream;

@info(name = 'query-2')
@dist(execGroup='group-1')
from every( e1=TempStream ) ->
e2=TempStream[e1.roomNo==roomNo and (e1.temp + 5) <= temp ] within 10 min
select e1.roomNo, e1.temp as initialTemp, e2.temp as finalTemp
insert into AlertStream;

@info(name = 'query-3')
@dist(execGroup='group-2',parallel = '2')
from TempStream [(roomNo >= 100 and roomNo < 110) and temp > 40 ]
select roomNo, temp
insert into HighTempStream;
```

This sample distributed Siddhi application contains two execution groups named group-1 and group-2 (defined via execGroup='GROUP_ID'). group-1 contains two queries named query-1 and query-2. group-2 contains query-3. No specific number of parallel instances are specified for group-1. Therefore, only one instance is created for it at runtime by default. Two parallel instances are specified for group-2.

The following is an illustration of how each parallel instance is created as a separate Siddhi application.
Each Siddhi application is deployed in the available resource nodes of the distributed cluster. All these Siddhi applications communicate with each other using Kafka topics. The system creates Kafka topics representing each stream and configures the Siddhi applications to use these topics as required.

For detailed information, see Converting to a Distributed Streaming Application.

Deployment architecture
WSO2 Stream Processor has a component named Dashboard in the User Interface and Dashboard layer. The Dashboard allows users to view output of analytics in an interactive manner. It also conveys observability information the cluster, status of the list of the applications (i.e., Siddhi applications) currently submitted, and the status of each Stream Processor node. The JVM metrics as well as Siddhi application level metrics can be viewed through this dashboard.

Job Manager nodes handle all the Management layer related functionalities. This layer contains two WSO2 SP Manager instances configured to run in the highly availability mode. Here, the Manager parses the distributed Siddhi application provided by the user, partitions it into multiple Siddhi applications, wires them using Kafka topics, and deploys them in the available worker nodes. Management layer also handles the effects of the worker nodes joining/leaving the distributed cluster by re-distributing the the Siddhi applications accordingly.

The processing layer (also known as the eesource cluster) is represented by multiple WSO2 SP Worker instances that are configured as workers. Each WSO2 SP worker instance in this layer registers itself to the Manager Cluster when it starts. These workers periodically send their heartbeats to the Manager Cluster. This allows the Managers to identify the active worker nodes and the inactive ones. The worker nodes (resource nodes) run the Siddhi applications assigned to them by their Manager nodes. In addition, they are also capable of handling network partitions in a graceful manner as depicted in the following diagram.
As depicted above, a worker node periodically synchronizes its configurations and the Siddhi applications with the manager Node. If the network get partitioned or if the manager becomes unreachable, it undeploys the applications deployed in it. By doing so, it allows the Siddhi applications to be rescheduled in other work nodes that are maintaining their connections with the manager nodes.

It is required to use Apache Kafka and Apache Zookeeper to configure a fully distributed SP cluster. As explained above, a Kafka cluster which is a part of the Persistence layer and the Transport layer holds all the topics that are used for communication by distributed Siddhi applications. Persistence stores of the Persistence layer can be RDBMs databases that store both configuration and system state data. Identity and access management of all the WSO2 Stream Processor nodes are handled by any OAuth Identity provider such as of the WSO2 Identity and Access Management (WSO2 IAM).

There are no restrictions to run WSO2 Stream Processor in the distributed mode on any environment. It can run in the distributed mode on bare metal, VMs, and containers. Here the manager nodes are grouped in a single cluster backed by a database for correlation. Similarly, dashboard nodes can also be deployed in a separate cluster. The worker nodes on the other hand are not aware of each other. They are synchronized with manager nodes from which they receive instructions.

Manager cluster

The manager cluster contains two or more WSO2 SP instances configured to run in the high availability mode. The manager cluster is responsible for parsing a user-defined distributed Siddhi application, dividing it to multiple Siddhi applications, creating the required topics and then deploying them in the available resource nodes. The manager cluster also handles resource nodes that join/leave the distributed cluster, and re-schedules the Siddhi applications accordingly. Since manager nodes are deployed in a high availability mode, if and when the active manager node goes down, another node in the manager will be elected as the cluster to handle the resource cluster.

Resource cluster

A resource cluster contains multiple WSO2 SP instances. Each instance sends a periodic heartbeat to the manager cluster so that the managers at any given time can identify the resource nodes that are active in the cluster. The resource nodes are responsible for running Siddhi applications assigned to them by the manager nodes. A resource node continues to run its Siddhi applications until a manager node undeploys them, or until it is no longer able to reach a manager node to send its heartbeat. If a manager node is unreachable for a specified amount of time, the resource node stops operating, removes its deployed Siddhi applications and waits until it can reach a manager node again.

Deployed Siddhi applications communicate among themselves via Kafka topics.

Kafka cluster

A Kafka cluster holds all the topics used by distributed Siddhi applications. All communications between execution groups take place via Kafka. You can only publish and receive data from distributed Siddhi applications via Kafka. In order to do so, you can either define a Kafka source in the
initial distributed Siddhi application or use the Kafka source created by distributed implementation. Note that installing Kafka and Zookeeper is a prerequisite to configure a distributed deployment.

Configuring a distributed cluster

This section explains how to configure a distributed WSO2 SP cluster.

Prerequisites

In order to configure a fully distributed HA cluster, the following prerequisites must be completed:

- A WSO2 SP binary pack must be available for each node in the cluster.
- Each SP node should have an ID under wso2.carbon in the `<SP_HOME>/conf/manager/deployment.yaml` or `<SP_HOME>/conf/worker/deployment.yaml` file depending on the cluster node being configured.
- A working RDBMS instance to be used for clustering of the manager nodes.
- We currently support only MySQL. Support for other databases will be added soon.
- The datasource to be shared by the nodes in the manager cluster must be already defined in the `<SP_HOME>/conf/manager/deployment.yaml` file.
- For MySQL to work with the Stream Processor, download the MySQL connector from here, extract and find the `mysql-connector-java-5.*.*-bin.jar`. Drop the jar to the `<SP_HOME>/lib` directory in both manager nodes.
- In order to retrieve the state of the Siddhi Applications deployed in the system in case of a scenario where both the nodes fail, state persistence must be enabled for all worker nodes. For detailed instructions, see Configuring Database and File System State Persistence.
- A Zookeeper cluster and Kafka cluster should already be started and hosts and ports should be known.
  - Supported Zookeeper version - 3.4.6
  - Supported Kafka version - 2.11-0.10.0.0
- For all manager and resource nodes to communicate with the Kafka broker the following kafka libs found in `<KAFKA_HOME>/libs` should be converted to OSGI and added to `<SP_HOME>/lib` directory:
  - `kafka_2.11-0.10.0.0.jar`
  - `kafka-clients-0.10.0.0.jar`
  - `metrics-core-2.2.0.jar`
  - `scala-parser-combinators_2.11-1.0.4.jar`
  - `scala-library-2.11.8.jar`
  - `zkclient-0.8.jar`
  - `zookeeper-3.4.6.jar`
- To convert Kafka libs to OSGI,
  - Create the source folder (eg: kafka) and copy the respective Kafka libs to the created folder.
  - Create another folder (eg: kafka-osgi) as the destination folder to which the converted OSGI bundles will be added.
  - Navigate to `<SP_HOME>/bin` and issue the following command
    - For Linux: `./jartobundle.sh <path_to_source_folder> <path_to_destination_folder>`
    - For Windows: `./jartobundle.bat <path_to_source_folder> <path_to_destination_folder>`
  - If converted successfully then for each lib, following messages would be shown on the terminal
    - INFO: Created the OSGi bundle `<kafka-lib-name>.jar` for JAR file `<absolute_path>/kafka/<kafka-lib-name>.jar`
- You can find the osgi converted libs in the destination folder. Copy them to the `<SP_HOME>/lib` directory.

Configuring the cluster

To configure a fully distributed HA cluster, follow the procedure below:

- Configure manager nodes
- Configure resource nodes

Configure manager nodes

To configure a node as a manager node, update the `<SP_HOME>/conf/manager/deployment.yaml` file as follows. The fully distributed cluster can have one or more manager nodes. For more information on how to set up cluster coordination see Configuring Cluster Coordination.
1. In the `cluster.config` section, make the following changes.
   a. To enable the cluster mode, set the `enabled` property to `true`.
   b. In order to cluster all the manager nodes together, enter the same cluster ID as the group ID for all the nodes (e.g., `groupId: group-1`).
   c. Enter the ID of the class that defines the coordination strategy for the cluster as shown in the example below.
      ```
      e.g., `coordinationStrategyClass: org.wso2.carbon.cluster.coordinator.rdbms.RDBMSCoordinationStrategy`
      ```

2. In the `strategyConfig` section of `cluster.config`, enter information for the required parameters as follows.
   a. Enter the ID of the datasource shared by the nodes in the cluster as shown in the example below. Data handled by the cluster are persisted here.
      ```
      datasource: SP_MGT_DB
      ```

   The SP_MGT_DB datasource is configured to an h2 database by default. You must create a MySQL database and then configure this datasource in the `<SP_HOME>/conf/manager/deployment.yaml` file of the required manager. The following is a sample configuration:

   ```
   - name: SP_MGT_DB
     description: The MySQL datasource used for Cluster Coordination
     # JNDI mapping of a data source
     jndiConfig:
       name: jdbc/WSO2ClusterDB
     # data source definition
     definition:
       # data source type
       type: RDBMS
       # data source configuration
       configuration:
         jdbcUrl: 'jdbc:mysql://<host>:<port>/<database_name>?useSSL=false'
         username: '<Username_Here>'
         password: '<Password_Here>'
         driverClassName: com.mysql.jdbc.Driver
         maxPoolSize: 50
         idleTimeout: 60000
         connectionTestQuery: SELECT 1
         validationTimeout: 30000
         isAutoCommit: false
   ```

   b. Specify the time interval (in milliseconds) at which heartbeat pulse should occur within the manager cluster to indicate that a manager is in an active state as shown in the example below.
      ```
      heartbeatInterval: 500
      ```

   c. Specify the number of times the heartbeat pulse must be unavailable at the specified time interval in order to consider a manager node as inactive as shown in the example below. A value of four means that if a manager node fails to send four consecutive heart beat pulses, it will be identified as unresponsive and another manager node will act as the active node in the manager cluster.
      ```
      heartbeatMaxRetry: 4
      ```

   d. Specify the time interval (in milliseconds) at which each node should listen for changes that occur in the cluster as shown in the example below.
      ```
      eventPollingInterval: 1000
      ```

3. In the `deployment.config` section, enter information as follows:
   a. In the `type` field, enter the type of the cluster as `distributed`.
      ```
      type: distributed
      ```
   b. For the `httpsInterface` parameter, specify the host and the port of the node.
      ```
      Host should be the IP of the network interface though which nodes are connected. (i.e LAN IP). Each node should have a separate port if deployed in same physical machine.
      ```
Configure resource nodes

To configure the resource nodes for a fully distributed HA cluster, edit the <SP_HOME>/conf/worker/deployment.yaml file as follows. You have to uncomment (remove the # in front of each line) the section under # Sample of deployment.config for Distributed deployment. Now start performing following steps under deployment.config section.

1. In the type field, enter the type of the cluster as distributed.
   type: distributed
2. For the httpsInterface parameter, specify the host, port and the user credentials of the configuring resource node.

   The host must be the IP of the network interface through which the nodes are connected (i.e., LAN IP). If all the nodes are deployed in the same physical machine, each node must have a separate port.

   e.g., host=localhost, port:9090, username:admin, password:admin

3. In the leaderRetryInterval parameter, enter the number of milliseconds for which the resource node must keep retrying to connect with a manager node. If the time specified for this parameter elapses without the resource node connecting to a manager node, the resource node is shut down.
   e.g., leaderRetryInterval: 5000

4. In the resourceManagers parameter, specify the hosts, ports and user credentials of the manager nodes to which the resource node must try to connect. If there are multiple managers, a sequence must be specified.

   Following is a sample deployment configuration for a resource node.
Starting the cluster

To start the distributed SP cluster, follow the procedure below:

1. Start each manager by navigating to the `<SP_HOME>/bin` directory and issuing the following command:
   - For Windows: `manager.bat`
   - For Linux: `./manager.sh`

2. Start each worker by navigating to the `<SP_HOME>/bin` directory and issuing the following command:
   - For Windows: `worker.bat`
   - For Linux: `./worker.sh`

3. When both manager and resource nodes are successfully started, the following is printed in the log entry:

   ```
   INFO {org.wso2.carbon.kernel.internal.CarbonStartupHandler} - WSO2 Stream Processor started in x sec
   ```

4. Siddhi applications should be deployed to the manager cluster using one of the following methods.
   a. Dropping the `.siddhi` file in to the `<SP_HOME>/wso2/manager/deployment/siddhi-files/` directory before or after starting the manager node.
   b. Sending a "POST" request to `http://<host>:<port>/siddhi-apps`, with the Siddhi App attached as a file in the request as shown in the example below. Refer Stream Processor REST API Guide for more information on using WSO2 Stream Processor APIs.

   ```
   Sample CURL request to deploy Siddhi application
   ```

   ```
   curl -X POST "https://localhost:9543/siddhi-apps" -H "accept: application/json" -H "Content-Type: text/plain" -d @TestSiddhiApp.siddhi -u admin:admin -k
   ```

**Important**

State persistence must be enabled for all worker nodes using a common database. For detailed instructions, see Configuring Database and File System State Persistence

**Important**

To deploy Siddhi applications in Distributed deployment it is recommended to use a content synchronization mechanism since Siddhi applications must be deployed to both manager nodes. You can use a common shared file system such as Network File System (NFS) or any other shared file system that is available. You need to mount the `<SP_HOME>/wso2/manager/deployment/siddhi-files/` directory of the two nodes to the shared file system.
Multi Datacenter High Availability Deployment

A **datacenter** refers to any clustered setup (i.e., a Minimum HA Deployment or a Fully Distributed Deployment) that has a separate message broker configured for it.

This is a WSO2 SP deployment pattern where two identical setups of SP are run in two different datacenters for disaster recovery. The setups are typically maintained at two different physical locations to minimize the risk of both the datacenters failing at the same time.

Out of the two datacenters, only one datacenter publishes events. This datacenter is referred to as the active center, and the other datacenter is referred to as the passive datacenter. If the active datacenter fails, the previously passive datacenter becomes the active datacenter and starts publishing events from where the previous datacenter stopped.

In order to run WSO2 SP in this setup, the following is required:

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Synchronization</td>
<td>In order to enable the passive datacenter to start publishing messages from where the active datacenter stopped before failing, both datacenters should receive the same events and process them simultaneously. Therefore, the publisher that send messages to this WSO2 SP setup must be configured to publish events to both the data centers.</td>
</tr>
<tr>
<td>A message broker</td>
<td>In order to achieve guaranteed delivery and other QoS (Quality of Service) characteristics such as processing each event exactly once, ordering of events, etc., a message broker must be used. Two separate instances of the broker can be deployed in each data center. When sending messages to this WSO2 SP setup, the publisher should send events to both brokers so that both datacenters build their state simultaneously. This ensures that if one datacenter fails, the messages lost as a result can be fetched from the broker of the other datacenter.</td>
</tr>
<tr>
<td>A sequence number for each message</td>
<td>If the active datacenter fails and the currently passive datacenter has to start publishing events, you need to ensure the following:</td>
</tr>
<tr>
<td></td>
<td>- No duplicated events must be created in the recovering datacenter when it fetches the lost events from the broker in the other datacenter. Assigning a unique sequence ID to each event allows the recovering node to identify duplicated events, and ensure that each event is processed only once to avoid an unnecessary system overhead.</td>
</tr>
<tr>
<td></td>
<td>- When the active datacenter fails, the passive datacenter must be able to identify the messages that are not already published. This can be achieved by assigning each event a unique sequence ID.</td>
</tr>
<tr>
<td>A sink</td>
<td>This is configured in order to send the input events to be processed by a Siddhi application Siddhi application to both the brokers in the to WSO2 SP datacenters.</td>
</tr>
<tr>
<td></td>
<td>The following conditions should be met in the event sink configuration:</td>
</tr>
<tr>
<td></td>
<td>- The <code>type</code> parameter must have <code>kafkaMultiDC</code> as the value. This is a sink type that publishes events to brokers configured for datacenters.</td>
</tr>
<tr>
<td></td>
<td>- The hosts of the two message brokers to which the published events should be sent must be listed for the <code>bootstrap.servers</code> parameter.</td>
</tr>
<tr>
<td></td>
<td>- A sequence ID must be configured so that if the active datacenter fails, the passive datacenter can identify the events that were already published and avoid publishing them again. The sequence number assigned to each event is always greater by one than the previous event even if the previous event is not successfully published.</td>
</tr>
<tr>
<td></td>
<td>e.g.,</td>
</tr>
</tbody>
</table>

```
@sink(type='kafkaMultiDC', topic='myTopic', bootstrap.servers='host1:9092, host2:9092', sequence.id='sink1', @map(type='xml'))
Define stream BarStream (symbol string, price float, volume long);
```

For more information see [kafkaMultiDC extension](https://docs.wso2.com/display/SP47/Minimum+HA+Deployment).
A source

This is configured for each node in each datacenter to receive events. Each node receives the events sent to the message broker of the datacenter to which it belongs.

The following conditions must be met in this configuration:

- The type parameter must have kafkaMultiDC as the value.
- Two hosts to which the received events should be sent must be listed for the bootstrap.servers parameter.

For example:

```java
@source(type=kafkaMultiDC, bootstrap.servers='host1:9092, host2:9092')
define stream Foo (attribute1 string, attribute2 int);
```

For more information see [kafkaMultiDC extension](#).

Samples

This section provides an overview of the samples shipped with WSO2 SP to help you understand the capabilities of WSO2 SP and well as WSO2 Siddhi. These samples can be accessed and run via the WSO2 Stream Processor Studio.

Sample list

To view the complete list of WSO2 samples, click [here](#).

Running a sample

To access and run a sample from the WSO2 Stream Processor Studio, follow the steps below.

1. Navigate to the `<SP_HOME>/bin` directory and issue the following command to start the WSO2 Stream Processor Studio.
   - For Windows: `editor.bat`
   - For Linux: `sh editor.sh`

All the available samples can be accessed from the welcome-page.

3. Click on the required sample to open its Siddhi application.

If the sample you want to try out is not visible in the default view, click **More Samples**. This opens the **Import Sample** dialog box where the complete list of available samples are displayed. Click on the required sample to expand it, and then click on its Siddhi file as shown below. As a result, the Siddhi application is entered in the **Sample Name** field. Click **Open** to open the sample and proceed to execute it.
4. To save the Siddhi application in your workspace so that you can run it, click File => Save. This opens the Save to Workspace dialog box. Click Save.

5. To run the Siddhi application, click Run => Run. If the application has successfully started, a message similar to the following example is displayed in the output console.

```
```

6. The testing steps differ from one sample to another. Therefore, to test the Siddhi application of the required sample, follow the instructions provided at the top of the relevant Siddhi file as comments. The following is an example.

```
Purpose:
This application demonstrates how to receive events via TCP transport and carry out data pre-processing with numerous Siddhi extensions (e.g. string extension, time extension). For more information on Siddhi extensions please refer to "https://wso2.github.io/siddhi/extensions/". In this sample, a composite ID is obtained using string concatenation and the time format of the incoming event is altered from "yyyyMMddHHmmss" to "dd-mmm-yyyy	timestamp"

Prerequisites:
1) Save this sample

Executing the Sample:
1) Start the Siddhi application by clicking on ‘Run’
2) If the Siddhi application starts successfully, the following messages would be shown on the console
   - Tcp Server started in 0.0.0.0:8082
   - DataPreprocessing.siddhi - Started Successfully!

Notes:
If you edit the code while it’s running, stop the application => Save => Start.
```

Once test a sample following the given instructions, you can view the results in the output console.

Stream Processor Solutions

This section covers the solutions designed by WSO2 with the WSO2 Stream Processor. These solutions are by default, shipped with the Stream processor.

The following are the available solutions.

- Analytics Solutions
- Distributed Message Tracer
- HTTP Analytics
- Twitter Analytics

Analytics Solutions

Analytics Solutions refer to WSO2 SP solutions pre-designed to work with other WSO2 products. The following solutions are currently supported.

- EI-Analytics
- API-Analytics
- IS-Analytics
At a given time, you can run one or more of these solutions in your WSO2 SP server. To specify the Analytics Solutions you want to configure, edit the `<SP_HOME>/conf/worker/deployment.yaml` file as follows:

1. Under the `# Carbon Configuration Parameters` comment and `wso2.carbon` section, specify the server type. The value depends on the solutions you want to enable in your SP server. Possible values are as follows:
   - `wso2-sp`: This allows you to enable multiple Analytics Solutions in your SP server. In addition, you can also deploy custom Siddhi applications and other custom SP artifacts to implement your own SP solutions.
   - `wso2-apim-analytics`: This only allows you to run the APIM Analytics solution in your SP server. When this value is specified, all the artifacts related to the APIM Analytics Solution are deployed and you cannot deploy any other Siddhi application in the SP server.
   - `wso2-ei-analytics`: This only allows you to run the EI Analytics solution in your SP server. When this value is specified, all the artifacts related to the EI Analytics Solution are deployed and you cannot deploy any other Siddhi application in the SP server.
   - `wso2-is-analytics`: This only allows you to run the IS Analytics solution in your SP server. When this value is specified, all the artifacts related to the IS Analytics Solution are deployed and you cannot deploy any other Siddhi application in the SP server.

   e.g., If you want to run multiple Analytics Solutions in your SP server, this parameter can be configured as follows:

   ```yaml
   # Carbon Configuration Parameters
   wso2.carbon:
     # server type
     type: wso2-sp
   ```

2. If you selected `wso2-sp` as the server type, enable the required solutions by adding the names of the required solutions as uncommented text in the `analytics.solutions` section under the `# Carbon Configuration Parameters` comment.

   ```yaml
   # Carbon Configuration Parameters
   wso2.carbon:
     # server type
     type: wso2-sp
   
   analytics.solutions:
     IS-analytics.enabled: false
     APIM-analytics.enabled: true
     EI-analytics.enabled: true
   ```

   In this example, only the APIM Analytics is enabled by setting the `APIM-analytics.enabled` parameter to `true`.

### Distributed Message Tracer

The Distributed Message Tracer solution allows you to trace the list of events that belong to a message and search through its results. A message is a single user activity that may invoke one or more services that independently generate events that are sent to WSO2 SP for processing. In addition, the activity itself generates an event that is considered the parent events.

e.g., When a transaction being processed passes through many subsystems, you can search through events collected from different subsystems to check whether the transaction is completed, the subsystem at which it is currently being processed etc.

This solution allows you to set up the Message Tracers to match different scenarios.

The Distributed Message Tracer solution is packed with WSO2 SP by default. To use this solution, follow the sections below:

To try out the Distributed Message Tracer, you can run the `Message Tracer` sample shipped with WSO2 Stream Processor. For detailed instructions to run a sample, see [Samples](#).

- Setting up the Message Tracer
- Tracing messages in the dashboard
- Using the WSO2 SP Message Trace client

Setting up the Message Tracer
This step involves configuring a Siddhi application in a way that allows the required details relating to activities can be captured and processed as required. To do this, let’s define a business rule via the WSO2 SP Business Rules Manager as follows. For more information, see Creating Business Rules.

1. In your terminal, navigate to the `<SP_HOME>/bin` directory. Then issue one of the following commands to start the dashboard server.
   - On Windows: `dashboard.bat --run`
   - On Linux/Mac OS: `./dashboard.sh`

2. Start a WSO2 SP worker runtime by issuing one of the following commands from the `<SP_HOME>/bin` directory.
   - On Windows: `worker.bat --run`
   - On Linux/Mac OS: `./worker.sh`

3. Access the Business Rules Manager via one of the following URLs.
   - For HTTP: `http://<SP_HOST>:<HTTP_PORT>/business-rules`
   - For HTTPS: `https://<SP_HOST>:<HTTPS_PORT>/business-rules`

4. If you do not have any business rules that are already created, click Create. If there are existing rules, click + to create a new rule.

5. Click Template to create the business rule from the business template that is already available in the Distributed Message Tracer solution.

6. Click on Message Tracer to create your business rule from the Message Tracer template group.

This opens the Message Tracer page.
7. To configure the Siddhi application, select **Message Tracer App Template** in the **Rule Template** field.

![Activity Explorer](image)

This opens the **Message Tracer App Template** business rules template.

8. Enter the required information in the fields of the **Message Tracer App Template** business rules template as follows.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Rule Name</td>
<td>A unique name for the business rule.</td>
</tr>
<tr>
<td>Span Stream Definition</td>
<td>This defines the source and the stream configuration to capture the required information.</td>
</tr>
<tr>
<td>Component Name</td>
<td>The module that generates the associated span. This can be constant or an attribute you included in the <strong>Span Stream Definition</strong> field.</td>
</tr>
<tr>
<td>Trace ID</td>
<td>The name of the attribute in the <strong>Span Stream Definition</strong> that captures the ID of the trace. The trace refers to the complete tracing carried out for the activity flow. A message flow may span over multiple services. The trace ID is unique to each message flow.</td>
</tr>
<tr>
<td>Span ID</td>
<td>The name of the attribute in the <strong>Span Stream Definition</strong> that captures the ID of the span. When the message flow spans over multiple services, the extent of the flow that relates to one of the services is identified as a single span with a unique ID.</td>
</tr>
<tr>
<td>Baggage Items</td>
<td>The name of the attribute in the <strong>Span Stream Definition</strong> to capture baggage items. Baggage items are common payloads that move across the trace.</td>
</tr>
<tr>
<td>Parent ID</td>
<td>The name of the attribute in the <strong>Span Stream Definition</strong> that captures the parent ID of the span. This is relevant when an invoked service in the message flow in turn invokes another service. In this scenario, the span of the first service is the parent of the span of the second service.</td>
</tr>
<tr>
<td>Service Name</td>
<td>The name of the attribute in the <strong>Span Stream Definition</strong> to capture the name of the service that generates the event. The span ID displayed in the <strong>Message Tracer</strong> dashboard to visualize the trace is derived from the service name. Therefore, each service needs to have a unique name. You can enter an concatenation to generate a unique service name, e.g., The default value <code>str:concat(correlation_activity_id, timestamp)</code> generates the service name by combining the value for the <strong>start time</strong> attribute (i.e., timestamp) and the message ID (i.e., trace ID).</td>
</tr>
<tr>
<td>Start Time</td>
<td>The name of the attribute in the <strong>Span Stream Definition</strong> to capture the start time of the span in milliseconds.</td>
</tr>
<tr>
<td>End Time</td>
<td>The name of the attribute in the <strong>Span Stream Definition</strong> to capture the end time of the span in milliseconds.</td>
</tr>
<tr>
<td>Tags</td>
<td>The name of the attribute in the <strong>Span Stream Definition</strong> to capture tags. A tag is a JSON payload specific to a span. You can use a concatenation via <code>str:concat</code>.</td>
</tr>
<tr>
<td>Span References</td>
<td>The name of the attribute in the <strong>Span Stream Definition</strong> to capture span references.</td>
</tr>
<tr>
<td>Parent Span is Defined</td>
<td>This field specifies whether the <strong>Span Stream Definition</strong> needs to capture the ID of the parent span or not. Is this is set to <strong>false</strong> the parent ID is generated and assigned by the system.</td>
</tr>
</tbody>
</table>
9. Click SAVE & DEPLOY. The rule you create is added as shown in the example below.

![Business Rules](image)

10. Click + and enter another business rule from the Message Tracer Source Template template with the following information.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Rule Name</td>
<td>A unique name for the business rule.</td>
</tr>
<tr>
<td>Source Type</td>
<td>The type of the source via which events relating to activities are received.</td>
</tr>
<tr>
<td>Map Type</td>
<td>The format in which events relating to activities are received.</td>
</tr>
<tr>
<td>Source Options</td>
<td>Here, you can enter optional properties to configure the source. The properties entered depends on the source type you selected. e.g., if the source type is kafka, you can enter topic.list='kafka_topic', partition.no.list='0', threading.option='single.thread.'</td>
</tr>
</tbody>
</table>

11. Click SAVE & DEPLOY.
Tracing messages in the dashboard

To trace activities via the **Message Tracer** dashboard, follow the procedure below:

1. Access the Dashboard Portal via one of the following URLs.
   - http://<SP_HOST>:<HTTP_PORT>/portal (e.g., http://localhost:9290/portal)
   - https://<SP_HOST>:<HTTPS_PORT>/portal (e.g., https://localhost:9643/portal)

2. Log in with your credentials. The **Distributed Message Tracer** dashboard is included in the dashboard listing as shown below.

3. Click on the **Distributed Message Tracer** dashboard to open the dashboard. The widgets in the dashboard are displayed as follows.

4. To filter the information you want to view in the dashboard, do the following:
   a. In the **Service** field, select the service for which you want to view message statistics.
   b. Click on the required date range in the **Date Range** widget to view message statistics for the required time period. If the time period for which you want to view statistics is not displayed in this widget, you can click **Custom** and select a specific time interval.

   Once you enter information to filter message statistics to be displayed, click **SEARCH**. The available traces are displayed in the **Tracing List** section as shown in the example below.
Once you click on the required trace ID, information related to the trace is displayed as shown in the following example.

Using the WSO2 SP Message Trace client

To publish messages to be traced WSO2 SP Message Trace client, you need to wrap the WSO2 SP Message Trace client as shown below once you develop the service, and get a WSO2 SP Tracer instance.

```java
// createStreamProcessorTracerClient instance
StreamProcessorTracerClient streamProcessorTracerClient = new StreamProcessorTracerClient();
Properties tracerProperties = new Properties();

// read configuration file and populate tracerProperties.
/*/
* reporter.wso2sp.publisher.type: thrift
* reporter.wso2sp.publisher.username: admin
* reporter.wso2sp.publisher.password: admin
* reporter.wso2sp.publisher.url: tcp://localhost:7611
* reporter.wso2sp.publisher.authUrl: ssl://localhost:7711
* reporter.wso2sp.publisher.service.name: WSO2 MSF4J
* trace.name: wso2sp
* javax.net.ssl.trustStorePassword: wso2carbon
*/
tracerProperties.setProperty(WSO2SP_REPORTER_PUBLISHER_TYPE, configRegistry.getConfigOrDefault(getFullQualifiedConfig(WSO2SP_REPORTER_PUBLISHER_TYPE), DEFAULT_WSO2SP_REPORTER_PUBLISHER_TYPE_CONFIG));
tracerProperties.setProperty(WSO2SP_REPORTER_USERNAME, configRegistry.getConfigOrDefault(getFullQualifiedConfig(WSO2SP_REPORTER_USERNAME), DEFAULT_WSO2SP_REPORTER_USERNAME));
tracerProperties.setProperty(WSO2SP_REPORTER_PASSWORD, configRegistry.getConfigOrDefault(getFullQualifiedConfig(WSO2SP_REPORTER_PASSWORD), null));
tracerProperties.setProperty(WSO2SP_REPORTER_URL, configRegistry.getConfigOrDefault(getFullQualifiedConfig(WSO2SP_REPORTER_URL), DEFAULT_WSO2SP_REPORTER_URL));
tracerProperties.setProperty(WSO2SP_REPORTER_AUTHURL, configRegistry.getConfigOrDefault(getFullQualifiedConfig(WSO2SP_REPORTER_AUTHURL), DEFAULT_WSO2SP_REPORTER_AUTHURL));
tracerProperties.setProperty(WSO2SP_REPORTER_DATABRIDGE_AGENT_CONFIG, configRegistry.getConfigOrDefault(getFullQualifiedConfig(WSO2SP_REPORTER_DATABRIDGE_AGENT_CONFIG), null));
tracerProperties.setProperty(WSO2SP_REPORTER_TRUSTSTORE, configRegistry.getConfigOrDefault(getFullQualifiedConfig(WSO2SP_REPORTER_TRUSTSTORE), null));
tracerProperties.setProperty(WSO2SP_REPORTER_TRUSTSTORE_PASSWORD, configRegistry.getConfigOrDefault(getFullQualifiedConfig(WSO2SP_REPORTER_TRUSTSTORE_PASSWORD), null));
tracerProperties.setProperty(TRACER_NAME, TRACER_VALUE);
streamProcessorTracerClient.init(tracerProperties);
Using the WSO2 SP Message Trace Client with Ballerina

This section covers a sample setup to run the SP Message Trace Client with Ballerina.

Before you begin:
Download Ballerina from here.
Install it by following the instructions in the Ballerina Installation Guide.

To run the Message Trace Client with Ballerina, follow the steps below:

1. Clone the ballerina-observability repository.
2. To build the ballerina-sp-extension component, navigate to the ballerina-observability/tracing-extensions/modules/ballerina-sp-extension directory and issue the following command:
   
   mvn clean install

   This creates JAR files in the target of the component.
3. Copy the JAR files created in the ballerina-sp-extension component, and place them in the bre/lib directory of the Ballerina distribution.
4. Create a ballerina sample service as follows, and save it as HelloWorldService.bal in a preferred location in your machine.

```
import ballerina/http;
import ballerina/log;

service<http:Service> hello bind { port: 9099 } {
    sayHello(endpoint caller, http:Request req) {
        http:Response res = new;
        res.setPayload("Hello, World!");
        caller->respond(res) but { error e => log:printError("Error sending response", err = e) };
    }
}
```

5. Download the resources from here and unzip.

   You need to change the file path destinations to the resource files accordingly.

6. Add the following to ballerina.conf file. You need to change the file path destinations to the javax.net.ssl.trustStore.
Testing the Distributed Message Tracer with Ballerina

After setting up the Message Tracer to run with Ballerina as specified in the previous section, you can test the Distributed Message Tracer with Ballerina by following the steps below:

1. Copy the ballerinatracer_0.siddhi file from the resources you downloaded and copy them to `<SP_HOME>/wso2/worker/deployment/siddhifiles` directory of your SP worker.
2. Start the SP worker by issuing one of the following commands.
   - On Windows: `worker.bat --run`
   - On Linux/Mac OS: `./worker.sh`
3. Execute a GET command with the following URL.
   ```
   http://localhost:9099/hello/sayHello
   ```
4. Access the Dashboard Portal and log in by entering admin as both the username and the password. Then click on the Distributed Message Tracer dashboard to open it. For more detailed information about accessing the Distributed Message Tracer dashboard, see Tracing messages in dashboard.
5. Click Search to list all the Trace IDs, and click on the available Trace IDs on the Tracing List widget.

The Ballerina service message trace is displayed as follows:

```
[b7a.observability.tracing]
enabled=true
name="wso2sp"

[b7a.observability.tracing.wso2sp]
reporter.wso2sp.publisher.type="thrift"
reporter.wso2sp.publisher.username="admin"
reporter.wso2sp.publisher.password="admin"
reporter.wso2sp.publisher.url="tcp://localhost:7611"
reporter.wso2sp.publisher.authUrl="ssl://localhost:7711"
javax.net.ssl.trustStorePassword="wso2carbon"
reporter.wso2sp.publisher.service.name="ballerina_hello_world"
```
HTTP Analytics

HTTP Analytics is an analytics solution powered by WSO2 Stream Processor. It is packed with WSO2 Stream Processor by default. It allows you to monitor your HTTP analysis data by visualizing them in a preconfigured dashboard.

To use this solution to view HTTP analytics data in the HTTP Analytics dashboard, follow the steps below.

- **Step 1: Set up HTTP Analytics**
- **Step 2: View HTTP Analysis data**

**Step 1: Set up HTTP Analytics**

This step involves defining a source to specify how the HTTP request events analyzed via this solution need to be captured and defining a store to store the events to be analyzed as per your business requirement. To do this, let's define a business rule via the WSO2 SP Business Rules Manager as follows. For more information, see [Creating Business Rules](#).

1. In your terminal, navigate to the `<SP_HOME>/bin` directory. Then issue one of the following commands to start the dashboard server.
   - On Windows: `dashboard.bat --run`
   - On Linux/Mac OS: `./dashboard.sh`

2. Start a WSO2 SP worker runtime by issuing one of the following commands from the `<SP_HOME>/bin` directory.
   - On Windows: `worker.bat --run`
   - On Linux/Mac OS: `./worker.sh`

3. Access the Business Rules Manager via one of the following URLs.
   - For HTTP: `http://<SP_HOST>:<HTTP_PORT>/business-rules (e.g., http://0.0.0.0:9090/business-rules)`
   - For HTTPS: `https://<SP_HOST>:<HTTPS_PORT>/business-rules (e.g., https://0.0.0.0:9443/business-rules)`

4. If you do not have any business rules that are already created, click **Create**. If there are existing rules, click + to create a new rule.

5. Click **Template** to create the business rule from the business template that is already available in the HTTP Analytics solution.
6. Click on HTTP-Analytics to create your business rule from the HTTP Analytics template group.

This opens the HTTP-Analytics page. Select the HTTP-Analytics-Processing template in the RuleTemplate field.

This opens the predefined fields of the HTTP-Analytics-Processing template.

7. To define the business rule, enter information in the fields displayed as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Rule Name</td>
<td>A unique name for the business rule.</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### Step 2: View HTTP Analysis data

**Before you begin:**
In order to view information in the HTTP Analytics dashboard, issue some sample cURL commands to generate events. The cURL commands can be of the following format.

```bash
curl -v -X POST \
   http://<HOST>:8280/<NAME_OF_CREATED_BUSINESS_RULE>_0/RequestsStream \
   -H 'content-type: application/json' \
   -d '{
   "event": {
   "timestamp": <CURRENT_TIMESTAMP_IN_MILLISECOND>*,
   "serverName": "localhost",
   "serviceName": "A",
   "serviceMethod": "GET",
   "responseTime": 1000.00,
   "httpResponseCode": 200,
   "userAgent": "Mozilla/5.0 (iPad; U; CPU OS 3_2_1 like Mac OS X; en-us) AppleWebKit/531.21.10 (KHTML, like Gecko) Mobile/7B405",
   "requestIP": "127.0.0.1"
   }
}'
```

- When sending multiple requests to ensure that timestamp is changed so that the later requests have higher timestamp value.
- If Yes is selected for the **Use system time for aggregation** field when deploying the siddhi application, the events do not need to have valid values for the timestamp attribute because this value is ignored. cURL commands can be passed with 0 as the timestamp value.

8. Click SAVE & DEPLOY.
curl -v -X POST \\
http://localhost:8280/http-analytics_rule_one_0/RequestsStream \\
-H 'content-type: application/json' \\
-d '{
  "event": {
    "timestamp": 1528795987,
    "serverName": "localhost",
    "serviceName": "A",
    "serviceMethod": "GET",
    "responseTime": 1000.00,
    "httpResponseCode": 200,
    "userAgent": "Mozilla/5.0 (iPad; U; CPU OS 3_2_1 like Mac OS X; en-us) AppleWebKit/531.21.10 (KHTML, like Gecko) Mobile/7B405",
    "requestIP": "127.0.0.1"
  }
}'

curl -v -X POST \\
http://localhost:8280/http-analytics_rule_one_0/RequestsStream \\
-H 'content-type: application/json' \\
-d '{
  "event": {
    "timestamp": 1528796173,
    "serverName": "localhost",
    "serviceName": "A",
    "serviceMethod": "GET",
    "responseTime": 1000.00,
    "httpResponseCode": 200,
    "userAgent": "Mozilla/5.0 (iPad; U; CPU OS 3_2_1 like Mac OS X; en-us) AppleWebKit/531.21.10 (KHTML, like Gecko) Mobile/7B405",
    "requestIP": "127.0.0.1"
  }
}'
To view information in the HTTP Analytics dashboard, follow the procedure below:

1. Access the Dashboard Portal via one of the following URLs:
   - http://<SP_HOST>:<HTTP_PORT>/portal (e.g., http://localhost:9290/portal)
   - https://<SP_HOST>:<HTTPS_PORT>/portal (e.g., https://localhost:9643/portal)
2. Log in with your credentials. The **HTTP Analytics** dashboard is included in the dashboard listing as shown below.
3. Click on the **HTTP Analytics** dashboard to open it. Information is displayed in the widgets as shown below.
Twitter Analytics

Twitter Analytics solution allows you to create a Twitter application that can be integrated into your service, and monitor the Tweets generated from it. This solution is packed in WSO2 SP by default. To use this solution, follow the steps below:

- Step 1: Set up the Twitter Analytics solution
- Step 2: Monitor Twitter activity

Step 1: Set up the Twitter Analytics solution

This step involves setting up the Twitter Analytics solutions by defining how the execution needs to be done using information specific to your Twitter application. In order to do this, you need to create a business rule using the predefined business rules template, Popular Tweets Analysis shipped with this solution. For more information about Business Rules, see Working with Business Rules.

Before you begin:
Create a Twitter application and collect the required information as follows:

1. Create your Twitter Application in Twitter Application Management.
2. Open the created application, and click on the Permissions tab. Select the Read and Write option and click Update Settings.
3. Click on the Keys And Access Tokens tab. Generate a new access token by clicking Create My Access Token.
4. Collect following values from the Keys and Access Tokens tab:
   - Consumer Key
   - Consumer Secret
   - Access Token
   - Access Token Secret

To create a business rule to set up the Twitter Analytics solution, follow the procedure given below:

1. In your terminal, navigate to the `<SP_HOME>/bin` directory. Then issue one of the following commands to start the dashboard server.
   - On Windows: `dashboard.bat --run`
   - On Linux/Mac OS: `./dashboard.sh`

2. Start a WSO2 SP worker runtime by issuing one of the following commands from the `<SP_HOME>/bin` directory.
   - On Windows: `worker.bat --run`
   - On Linux/Mac OS: `./worker.sh`

3. Access the Business Rules Manager via one of the following URLs.
   - For HTTP: `http://<SP_HOST>:<HTTP_PORT>/business-rules` (e.g., `http://0.0.0.0:9090/business-rules`)
   - For HTTPS: `https://<SP_HOST>:<HTTPS_PORT>/business-rules` (e.g., `https://0.0.0.0:9443/business-rules`)
4. If you do not have any business rules that are already created, click Create. If there are existing rules, click + to create a new rule.
5. Click **Template** to create the business rule from the business template that is already available in the Twitter Analytics solution.

6. Click on **Twitter Analytics** to create your business rule from the template.

This opens the **Twitter Analytics** page.

7. Select **Popular Tweets Analysis** in the Rule Template field.
The **Popular Tweets Analysis** template opens with the template-specific fields.

8. To specify how the tweets generated by your application needs to be processed, enter information in the fields displayed as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Rule Name</td>
<td>A unique name for the business rule.</td>
</tr>
<tr>
<td>Consumer Key</td>
<td>The API key to access your Twitter application.</td>
</tr>
<tr>
<td>Consumer Secret</td>
<td>The API secret to access your Twitter application.</td>
</tr>
<tr>
<td>Access Token</td>
<td>The access token via which Tweets are generated from your Twitter account.</td>
</tr>
</tbody>
</table>
| Access Token Secret    | The access token secret to secure the Tweets generated via the access token from your Twitter account.
| Filtering Keywords     | The Tweets are filtered by the keywords entered in this field.             |
| Filtering Language     | The Tweets are filtered by the language entered in this field.             |
| Query                  | The query that defines how the Tweets need to be filtered.                 |
| Count                  | The maximum number of Tweets that should be displayed on a page at a given time. |
| Polling Interval       | The time interval (in seconds) at which the Tweets need to be polled.      |

9. Click **Save and Deploy**.

**Step 2: Monitor Twitter activity**

To monitor Twitter activity via the **Twitter Analytics** dashboard, follow the procedure below:

1. Access the Dashboard Portal via one of the following URLs.
   - http://<SP_HOST>:<HTTP_PORT>/portal (e.g., http://localhost:9290/portal)
   - https://<SP_HOST>:<HTTPS_PORT>/portal (e.g., https://localhost:9643/portal)
2. Log in with your credentials. The **Twitter Analytics** dashboard is included in the dashboard listing as shown below.
3. Click on the **Twitter Analytics** dashboard to open it. Information is displayed in the widgets as shown in the example below.
Admin Guide

The following topics explore various product deployment scenarios and other information useful for system administrators.

- Supporting Different Transports
- Protecting Sensitive Data via the Secure Vault
- Configuring Business Rule Deployment
- Configuring Database and File System State Persistence
- Configuring Cluster Coordination
- User Management
- Working with Databases
- Authorization & Permission Model
- General Data Protection Regulations (GDPR) for WSO2 Stream Processor
Supporting Different Transports

Follow the relevant section for the steps that need to be carried out before using the required transport to receive and publish events via WSO2 SP.

- Kafka transport
- JMS transport
- MQTT transport
- RabbitMQ transport

Kafka transport

To enable WSO2 SP to receive and publish events via the Kafka transport, follow the steps below:

1. Download the Kafka broker from here.
2. Convert and copy the Kafka client jars from the <KAFKA_HOME>/libs directory to the <SP_HOME>/lib directory as follows.
   a. Create a directory in a preferred location in your machine and copy the following JARs to it from the <KAFKA_HOME>/libs directory.

   This directory will be referred to as the SOURCE_DIRECTORY in the next steps.

   - kafka_2.11-0.10.2.1.jar
   - kafka-clients-0.10.2.1.jar
   - metrics-core-2.2.0.jar
   - scala-library-2.11.8.jar
   - scala-parser-combinators_2.11-1.0.4.jar
   - zkclient-0.10.jar
   - zookeeper-3.4.9.jar

   b. Create another directory in a preferred location in your machine.

   This directory will be referred to as the DESTINATION_DIRECTORY in the next steps.

   c. To convert all the Kafka jars you copied into the <SOURCE_DIRECTORY>, issue the following command.
      - For Windows: %SP_HOME%/bin/jartobundle.bat <SOURCE_DIRECTORY_PATH> <DESTINATION_DIRECTORY_PATH>
      - For Linux: %SP_HOME%/bin/jartobundle.sh <SOURCE_DIRECTORY_PATH> <DESTINATION_DIRECTORY_PATH>

   d. Copy the converted files from the <DESTINATION_DIRECTORY> to the <SP_HOME>/lib directory.
   e. Copy the jars that are not converted from the <SOURCE_DIRECTORY> to the <SP_HOME>/samples/sample-clients/lib directory.

3. The Kafka server should be started before sending events from WSO2 SP to a Kafka consumer.

JMS transport

Follow the steps to configure the Apache ActiveMQ message broker:

1. Install Apache ActiveMQ JMS.

   This guide uses ActiveMQ versions 5.7.0 - 5.9.0. If you want to use a later version, for instructions on the necessary changes to the configuration steps, go to Apache ActiveMQ Documentation.

2. Download the activemq-client-5.x.x.jar from here.
3. Register the InitialContextFactory implementation according to the OSGi JNDI spec and copy the client jar to the <SP_HOME>/lib directory as follows.
   a. Navigate to the <SP_HOME>/bin directory and issue the following command.
      - For Linux: ./icf-provider.sh org.apache.activemq.jndi.ActiveMQInitialContextFactory <Downloaded Jar Path>/activemq-client-5.x.x.jar <Output Jar Path>
      - For Windows: ./icf-provider.bat org.apache.activemq.jndi.ActiveMQInitialContextFactory <Downloaded Jar Path>/activemq-client-5.x.x.jar <Output Jar Path>
Once the client jar is successfully converted, the `activemq-client-5.x.x` directory is created. This directory contains the following:

- `activemq-client-5.x.x.jar` (original jar)
- `activemq-client-5.x.x_1.0.0.jar` (OSGi-converted jar)

In addition, the following messages are logged in the terminal.

```
INFO: Executing 'jar uf <absolute_path>/activemq-client-5.x.x/activemq-client-5.x.x.jar -C <absolute_path>/activemq-client-5.x.x/internal/CustomBundleActivator.class' [timestamp]
org.wso2.carbon.tools.spi.ICFProviderTool
addBundleActivatorHeader - INFO: Running jar to bundle conversion [timestamp]
org.wso2.carbon.tools.converter.utils.BundleGeneratorUtils
convertFromJarToBundle - INFO: Created the OSGi bundle activemq_client_5.x.x_1.0.0.jar for JAR file
<absolute_path>/activemq-client-5.x.x/activemq-client-5.x.x.jar
```

b. Copy `activemq-client-5.x.x/activemq-client-5.x.x.jar` and place it in the `<SP_HOME>/samples/sample-clients` directory.

c. Copy `activemq-client-5.x.x/activemq_client_5.x.x_1.0.0.jar` and place it in the `<SP_HOME>/lib` directory.

4. Create a directory in a preferred location in your machine and copy the following JARs to it from the `<ActiveMQ_HOME>/libs` directory.

- `hawtbuf-1.9.jar`
- `geronimo-jms_1.1_spec-1.1.1.jar`
- `geronimo-j2ee-management_1.1_spec-1.0.1.jar`

5. Create another directory in a preferred location in your machine.

   This directory will be referred to as the `DESTINATION_DIRECTORY` in the next steps.

   - hawtbuf-1.9.jar
   - geronimo-jms_1.1_spec-1.1.1.jar
   - geronimo-j2ee-management_1.1_spec-1.0.1.jar

6. To convert all the Kafka jars you copied into the `<SOURCE_DIRECTORY>`, issue the following command.

   - For Windows: `<SP_HOME>/bin/jartobundle.bat <SOURCE_DIRECTORY_PATH> <DESTINATION_DIRECTORY_PATH>`
   - For Linux: `<SP_HOME>/bin/jartobundle.sh <SOURCE_DIRECTORY_PATH> <DESTINATION_DIRECTORY_PATH>`

7. Copy the converted files from the `<DESTINATION_DIRECTORY>` to the `<SP_HOME>/lib` directory.

8. Copy the jars that are not converted from the `<SOURCE_DIRECTORY>` to the `<SP_HOME>/samples/sample-clients/lib` directory.

MQTT transport

Follow the steps to configure the MQTT message broker:

1. Download the `org.eclipse.paho.client.mqttv3-1.1.1.jar` file from here.
2. Place the file you downloaded in the `<SP_HOME>/lib` directory.

RabbitMQ transport

Follow the steps below to configure the RabbitMQ message broker:

1. Download RabbitMQ from here.
2. Create a directory in a preferred location in your machine. This directory is referred to as the `<SOURCE_DIRECTORY>` in the rest of the procedure.
3. Copy the following files from the `<RabbitMQ_HOME>/plugins` directory to the `<SOURCE_DIRECTORY>` you created.
4. Create another directory in a preferred location in your machine. This directory is referred to as the `<DESTINATION_DIRECTORY>` in this
Protecting Sensitive Data via the Secure Vault

WSO2 SP uses several artifacts for its functionality including deployment configurations for tuning its operation as well as deployable artifacts for extending its functionality. In each of these scenarios, there can be situations where the data specified is of a sensitive nature e.g., access tokens, passwords, etc.

The following topics cover how sensitive data can be obfuscated in both scenarios to avoid scrutiny,

- Securing sensitive data in deployment configurations
- Protecting sensitive data in Siddhi applications

Securing sensitive data in deployment configurations

WSO2 SP offers the Cipher tool to encrypt sensitive data in deployment configurations. This tool works in conjunction with WSO2 Secure Vault to replace sensitive data that is in plain text with an alias. The actual value is then encrypted and securely stored in the SecureVault. At runtime, the actual value is retrieved from the alias and used. For more information, see WSO2 Secure Vault.

To encrypt sensitive data to avoid scrutiny, follow the procedure below:

1. Open the `<SP_HOME>/conf/<RUNTIME>/secrets.properties` file and enter the following information.
   a. Enter the required sensitive data element with the value in plain text as shown in the example below.
      
      ```
      wso2.sample.password1=plainText ABC@123
      ```
   b. Enter the alias to be used in the required configuration file instead of the actual value of sensitive data you specified in the previous step as shown in the example below.
      
      ```
      password: ${sec:wso2.sample.password1}
      ```

2. To encrypt the sensitive data element and store it in the secure vault, run the Cipher tool by issuing a command similar to the following example for the required profile (i.e., worker, manager, dashboard, or editor).
   
   ```
   sh <SP_HOME>/bin/ciphertool.sh -runtime worker
   ```

Protecting sensitive data in Siddhi applications

A parameter named ref is used to secure sensitive information in Siddhi applications that are deployed in WSO2 SP. For Siddhi applications that use storage technologies supported by Carbon Datasources, it is also possible to use Carbon datasources instead of specifying the connection parameters directly on the Siddhi file.

Using the ref parameter

Siddhi 4.0 supports the ref parameter that enables the user to specify parts of their definition outside the Siddhi App. Extensions that support this functionality include:

- Stores
- Sources
- Sinks

This method of securing sensitive data involves defining the store parameters required by a connection instance in the `<SP_HOME>/conf/<RUNTIME>/deployment.yaml` file, and referring to those from Siddhi applications via the ref parameter.

Example

In the `<SP_HOME>/conf/<RUNTIME>/deployment.yaml` file, some connection parameters are defined for a store named store1 as follows:
The Siddhi application includes the following configuration:

```java
@Store(ref='store1')
@PrimaryKey('id')
@Index('houseId')
define table SmartHomeTable (id string, value float, property bool, plugId int, householdId int, houseId int, currentTime string);
```

Here `@Store(ref='store1')` refers to `store1` defined in the `deployment.yaml` file, and as a result, the properties defined for this store is applicable to the Siddhi application when it is connected to the store.

### Using carbon datasources

Currently, Carbon Datasources only support relational datasource definitions. You can also define RDBMS Store artifacts using Carbon Datasources or JNDI instead of directly specifying the connection parameters. Then the datasource definitions defined in the `<SP_HOME>/conf/<RUNTIME>/deployment.yaml` file can be secured via the process described under Securing sensitive data in deployment configurations.

The following is an example of an RDBMS store definition using Carbon datasources for looking up connection parameters.

```java
@Store(type='rdbms', datasource='MY_DS')
@PrimaryKey('id')
@Index('houseId')
define table SmartHomeTable (id string, value float, property bool, plugId int, householdId int, houseId int, currentTime string);
```

This can also be done via JNDI lookup as shown below:

```java
@Store(type='rdbms', jndi.resource='jdbc/MyDS')
@PrimaryKey('id')
@Index('houseId')
define table SmartHomeTable (id string, value float, property bool, plugId int, householdId int, houseId int, currentTime string);
```

### Configuring Business Rule Deployment
WSO2 Business Rules Manager uses ruleTemplates within templateGroups, to derive business rules from. Each ruleTemplate will have a UUID - that is used to uniquely identify itself. When a ruleTemplate is specified under a worker node; SiddhiApps derived in business rules created out of that ruleTemplate will be deployed in the specified worker node.

1. Go to the deployment.yaml file under `<WSO2SP_HOME>/conf/dashboard`
2. Deployment configurations for the Business Rules Manager are specified under `wso2.business.rules.manager`
3. Provide the URL(s) of worker node(s) that is/are available to deploy SiddhiApps, under `deployment_configs` - in the format `<HOST_NAME>:<PORT>`

```yaml
deployment_configs:
- <NODE1_HOST_NAME>:<NODE1_PORT>
- <NODE2_HOST_NAME>:<NODE2_PORT>
```

Eg:

```yaml
deployment_configs:
- localhost:9090
- 10.100.4.140:9090
```

4. List down the UUIDs of required ruleTemplates under each node such that: when a ruleTemplate is specified under a worker node, SiddhiApps derived in the business rules created out of that ruleTemplate will be deployed in that specific node.

```yaml
deployment_configs:
- <NODE1_HOST_NAME>:<NODE1_PORT>:
  - ruleTemplate1_UUID
  - ruleTemplate2_UUID
  - ruleTemplate3_UUID
```

Eg:

```yaml
deployment_configs:
- localhost:9090:
  - sweet-production-kpi-analysis
    - stock-exchange-input
    - stock-exchange-output
```

5. A rule template can be configured under more than one node, as long as its instanceCount is many

```yaml
deployment_configs:
- <NODE1_HOST_NAME>:<NODE1_PORT>:
  - ruleTemplate1_UUID
  - ruleTemplate2_UUID
  - ruleTemplate3_UUID
- <NODE2_HOST_NAME>:<NODE2_PORT>:
  - ruleTemplate1_UUID
  - ruleTemplate3_UUID
- <NODE3_HOST_NAME>:<NODE3_PORT>:
  - ruleTemplate2_UUID
  - ruleTemplate3_UUID
  - ruleTemplate4_UUID
```
Eg:

```
deployment_configs:
  - localhost:9090:
    - sweet-production-kpi-analysis
    - stock-exchange-input
    - stock-exchange-output
  - 10.100.40.169:9090:
    - identifying-continuous-production-decrease
    - sweet-production-kpi-analysis
```

Notice that the rule template with UUID sweet-production-kpi-analysis has been configured under two worker nodes. So that, if a business rule is derived from sweet-production-kpi-analysis, the belonging SiddhiApps will be deployed in both the nodes.

6. Specify the username & password, that are common for all the worker nodes

```
username: admin
password: admin
```

7. The complete deployment configuration for Business Rules will look like the following

```
wso2.business.rules.manager:
  datasource: BUSINESS_RULES_DB
  deployment_configs:
    - localhost:9090:
      - stock-data-analysis
      - stock-exchange-input
      - stock-exchange-output
      - identifying-continuous-production-decrease
      - sweet-production-kpi-analysis
    username: admin
    password: admin
```

**Configuring Database and File System State Persistence**

This section explains how to prevent the loss of data that can result from a system failure by persisting the state of WSO2 SP periodically either into a database system or into the file system.

**Prerequisites**

Before configuring database persistence, the following prerequisites must be completed.

- One or more Siddhi Applications must be running in the WSO2 SP server.
- A working RDBMS instance that can be used for data persistence must exist.
- The requirements of the datasource must be already defined.
- Database persistence involves updating the databases connected to WSO2 Steam Processor with the latest information relating to the events that are being processed by WSO2 SP at a given time.

### Configuring database system persistence

The supported databases are H2, MySQL, Postgres, MSSQL and Oracle. The relevant JDBC driver jar should be downloaded and added to the `<P_HOME>/lib` directory to prior to using database system persistence.

To configure periodic data persistence, update the `<SP_HOME>/conf/worker/deployment.yaml` file under `state.persistence` as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
<th>Required Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>This enables data persistence.</td>
<td>true</td>
</tr>
<tr>
<td>intervalInMin</td>
<td>The time interval in minutes that defines the interval in which state of Siddhi applications should be persisted</td>
<td>1</td>
</tr>
<tr>
<td>revisionsToKeep</td>
<td>The number of revisions to keep in the system. When a new persist takes place, the old revisions are removed.</td>
<td>3</td>
</tr>
<tr>
<td>persistenceStore</td>
<td>The persistence store.</td>
<td>org.wso2.carbon.stream.processor.core.persistence.DBPersistenceStore</td>
</tr>
<tr>
<td>config &gt; datasource</td>
<td>The datasource to be used in persisting the state. The provided datasource should be properly defined in the deployment.yaml. For detailed instructions of how to configure a datasource, see Configuring Datasources.</td>
<td>WSO2_PERSISTENCE_DB (Datasource with this name should be defined in wso2.datasources)</td>
</tr>
<tr>
<td>config &gt; table</td>
<td>The table that should be created and used for the persisting of the state.</td>
<td>PERSISTENCE_TABLE</td>
</tr>
</tbody>
</table>

The following is a sample segment of the required configurations in the `<SP_HOME>/conf/worker/deployment.yaml` file to configure file system persistence.
Configuring file system persistence

This section explains how to persist the states of Siddhi applications during a required time interval in the file system in order to maintain back-ups. To configure state persistence, update the `<SP_HOME>/conf/worker/deployment.yaml` file under `state.persistence` as follows:

```
state.persistence:
  enabled: true
  intervalInMin: 1
  revisionsToKeep: 3
  persistenceStore:
    org.wso2.carbon.stream.processor.core.persistence.DBPersistenceStore
    config:
      datasource: <DATASOURCE NAME>  # A datasource with this name should be defined in wso2.datasources namespace
      table: <TABLE NAME>
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
<th>Required Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>This enables data persistence.</td>
<td>true</td>
</tr>
<tr>
<td>intervalInMin</td>
<td>The time interval in minutes that defines the interval in which state of Siddhi applications should be persisted</td>
<td>1</td>
</tr>
<tr>
<td>revisionsToKeep</td>
<td>The number of revisions to keep in the system. When a new persist takes place, the old revisions are removed.</td>
<td>3</td>
</tr>
<tr>
<td>persistenceStore</td>
<td>The persistence store.</td>
<td>org.wso2.carbon.stream.processor.core.persistence.FileSystemPersistenceStore</td>
</tr>
<tr>
<td>config &gt; location</td>
<td>A fully qualified folder location to where the revision files should be persisted.</td>
<td>siddhi-app-persistence</td>
</tr>
</tbody>
</table>
The following is a sample segment of the required configurations in the `<SP_HOME>/conf/worker/deployment.yaml` file to configure file system persistence.

### Sample deployment.yaml segment

```yaml
state.persistence:
  enabled: true
  intervalInMin: 1
  revisionsToKeep: 2
  persistenceStore:
    org.wso2.carbon.stream.processor.core.persistence.FileSystemPersistenceStore
    config:
      location: siddhi-app-persistence
```

### Configuring Cluster Coordination

Multiple WSO2 SP nodes can be configured to work together by configuring a cluster coordination strategy that is used in various deployments such as the Minimum HA Deployment and Fully Distributed Deployment. At present, cluster coordination is supported via an RDBMS instance using and RDBMS coordination strategy. Support for cluster coordination via a Zookeeper instance will be supported in the near future.

At any given time, there is a leader in an SP cluster that is arbitrarily selected among the members of the cluster. The RDBMS coordination strategy that is used for cluster coordination works on the concept of heartbeats where the members of the cluster periodically send heartbeat signals via the datasource to the leader of the cluster. If the leader node does not detect a pre configured consecutive number of heartbeats from a specific node, the relevant node is removed from the cluster. Similarly, if the leader node fails to update its heartbeat, the cluster re-elects a new leader.

### Prerequisites

In order to configure a cluster, the following prerequisites must be completed:

- A minimum of two binary packs of WSO2 SP must be available.
- A working RDBMS instance must be available to be shared among the nodes of the cluster.

Currently, we only support MySQL. Support for other databases will be added soon.

### Configuring the Cluster with the RDBMS coordination strategy

To configure a cluster for several nodes, the `cluster.config` section of the `<SP_HOME>/conf/<worker|manager>/deployment.yaml` should be configured for all the nodes as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
<th>Sample Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>Set this value to <code>true</code> to enable cluster coordination for the node.</td>
<td><code>true/false</code></td>
</tr>
<tr>
<td>groupId</td>
<td>The group ID is used to identify the cluster to which the node belongs.</td>
<td>group-1</td>
</tr>
<tr>
<td>coordinationStrategyClass</td>
<td>The clustering class to be used.</td>
<td>org.wso2.carbon.cluster.coordinator.rdbms.RDBMSCoordinationSta</td>
</tr>
<tr>
<td>Configuration Key</td>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>strategyConfig &gt; datasource</td>
<td>The shared datasource to be used in the cluster. The datasource specified must be properly configured in the deployment.yaml file. For detailed instructions to configure a datasource, see Configuring Datasources.</td>
<td>WSO2_CARBON_DB</td>
</tr>
<tr>
<td>strategyConfig &gt; heartbeatInterval</td>
<td>This value defines the time interval in milliseconds between heartbeat pulses sent by nodes to indicate that they are still alive within the cluster.</td>
<td>1000</td>
</tr>
<tr>
<td>strategyConfig &gt; heartbeatMaxRetry</td>
<td>The number of times the heartbeat pulse can be unavailable until a node is identified as unresponsive. If a node fails to send its heartbeat pulse to the leader of the cluster after a number of retries equal to the number specified here, that node is removed from the cluster.</td>
<td>2</td>
</tr>
<tr>
<td>strategyConfig &gt; eventPollingInterval</td>
<td>The time interval in milliseconds at which a node listens to identify the changes happening within the cluster. The changes may include a new node joining the cluster, a node being removed from the cluster and the coordinator changed event.</td>
<td>1000</td>
</tr>
</tbody>
</table>

Following is a sample segment of the configurations needed for RDBMS coordination in the deployment.yaml: 
User Management

User management in Stream Processor has the following features,

- The concept of single user store, which is either local or external.
- File based user store as the default embedded store.
- Ability to connect to an external Identity Provider using SCIM2 and OAuth2 protocols.
- Ability to extend user authentication as per the scenario

Following sections include further information and configuration steps for the user management in SP,

- Introduction to User Management
- User Management via the IdP Client Interface

Introduction to User Management

User management is a mechanism which involves defining and managing users, roles and their access levels in a system. A user management dashboard or console provides system administrators a holistic view of a system's active user sessions, their log-in statuses, the privileges of each user and their activity in the system, enabling the system administrators to make business-critical, real-time security decisions. A typical user management implementation involves a wide range of functionality such as adding/deleting users, controlling user activity through permissions, managing user roles, defining authentication policies, managing external user stores, manual/automatic log-out, resetting user passwords etc.

Any user management system has users, roles, user stores and user permissions as its basic components.

Users

Users are consumers who interact with your organizational applications, databases or any other systems. These users can be a person, a device or another application/program within or outside of the organization's network. Since these users interact with internal systems and access data, the need to define which user is allowed to do what is critical to most security-conscious organizations. This is how the concept of user management developed.

Permission

A permission is a 'delegation of authority' or a 'right' assigned to a user or a group of users to perform an action on a system. Permissions can be granted to or revoked from a user/user group/user role automatically or by a system administrator. For example, if a user has the permission to log-in to a system, then the permission to log-out is automatically implied without the need of granting it specifically.

User Roles

A user role is a consolidation of several permissions. Instead of associating permissions with a user, administrator can associate permissions with a user role and assign the role to users. User roles can be reused throughout the system and prevents the overhead of granting multiple permissions to each and every user individually.

User Store

---

Sample deployment.yaml segment

```yaml
cluster.config:
  enabled: true
  groupId: <GROUP ID>
  coordinationStrategyClass:
    org.wso2.carbon.cluster.coordinator.rdbms.RDBMSCoordinationStrategy
strategyConfig:
  datasource: <DATASOURCE NAME>
  heartbeatInterval: 1000
  heartbeatMaxRetry: 2
  eventPollingInterval: 1000
```
A user store is a persistent storage where information of the users and/or user roles is stored. User information includes log-in name, password, first name, last name, e-mail etc. It can be either file based or a database maintained within SP or externally to it. User stores used in SP differs based on the interface(IdP Client) used to interact with the user store. By default, a file based user store maintained in the `<SP_HOME>/conf/<PROFILE>/deployment.yaml` file interfaced through 'Local' IdP Client is enabled.

### User Management via the IdP Client Interface

In WSO2 Stream Processor, user management is carried out through the Identity Provider Client (IdP Client) interface that can be switched as required for the user scenario. Furthermore, a custom IdP Client can be written to encompass the required user store connection and authentication.

IdP clients can be switched by specifying the required IdP client in the `auth.configs:` section in the `<SP_HOME>/conf/<PROFILE>/deployment.yaml` file.

```yaml
auth.configs:
  # Type of the IdP Client used for the user authentication
  type: local
```

The active IdP client is `local` by default.

Following are the IdP Clients available for WSO2 SP:

- Local IdP Client
- External IdP Client

**Local IdP Client**

The local IdP Client interacts with the file-based user store that is defined in the `<SP_HOME>/conf/<PROFILE>/deployment.yaml` file under a `auth.configs` namespace as follows:

```yaml
auth.configs:
  type: 'local'
  userManager:
    adminRole: admin
  userStore:
    users:
      - user:
          username: admin
          password: YWRtaW4=
          roles: 1
          roles:
            - role:
              id: 1
              displayName: admin
```

The above user and role is added by default.

**Parameters**

The parameters used in the above configurations are as follows:

- `adminRole`, `username`, `password`, `roles` and `displayName` are the required parameters for the `userStore` configuration.
- If new users/roles are added and the above default user and role are also needed, the following parameters must be added to the user store along with the added user/role.
### Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userManager &gt; adminRole</td>
<td>admin</td>
<td>The name of the role that has administration privileges.</td>
</tr>
<tr>
<td>userManager &gt; userStore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>users &gt; user &gt; username</td>
<td>admin</td>
<td>The username of the user.</td>
</tr>
<tr>
<td>userManager &gt; userStore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>users &gt; user &gt; password</td>
<td>YWRtaW4=</td>
<td>The encrypted password of the user.</td>
</tr>
<tr>
<td>userManager &gt; userStore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>users &gt; user &gt; roles</td>
<td>1</td>
<td>A comma separated list of the IDs of the roles assigned to the user.</td>
</tr>
<tr>
<td>userManager &gt; userStore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>roles &gt; role &gt; id</td>
<td>1</td>
<td>The unique ID for the role.</td>
</tr>
<tr>
<td>userManager &gt; userStore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>roles &gt; role &gt; admin</td>
<td>admin</td>
<td>The name of the role.</td>
</tr>
</tbody>
</table>

Furthermore, Local IdP Client functionality can be controlled via the properties defined in the `<SP_HOME>/conf/<PROFILE>/deployment.yaml` file under the auth.configs namespace as shown below.

```yaml
auth.configs:
  type: local
  properties:
    sessionTimeout: 3600
    refreshSessionTimeout: 86400
```

The following are the properties that can be configured for the local IdP provider:

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>properties &gt; sessiontimeout</td>
<td>3600</td>
<td>The number of seconds for which the session is valid once the user logs in.</td>
</tr>
<tr>
<td>properties &gt; refreshSessionTimeout</td>
<td>86400</td>
<td>The number of seconds for which the refresh token used to extend the session is valid.</td>
</tr>
</tbody>
</table>

The complete default configuration of the Local IdP Client is as follows:
auth.configs:
  type: 'local'
  properties:
    sessionTimeout: 3600
    refreshSessionTimeout: 86400
  userManager:
    adminRole: admin
  userStore:
    users:
      -
        user:
          username: admin
          password: YWRtaW4=
          roles: 1
          roles:
            -
              role:
                id: 1
                displayName: admin

External IdP Client

External IdP Client authenticates users by interacting with an external identity provider via OAuth2 and SCIM2 protocols. The user store is maintained by the external identity provider. WSO2 SP authenticates by requesting an access token from the identity provider using the password grant type.

The identity provider with which WSO2 SP interacts with to authenticate users must be started before the SP server.

The auth manager must be configured under the auth.configs namespace as shown below:

auth.configs:
  type: external
  authManager:
    adminRole: admin

The parameters used in the above configurations areas follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userManager &gt; adminRole</td>
<td>admin</td>
<td>The name of the role that has administration privilages.</td>
</tr>
</tbody>
</table>

Furthermore, external IdP client functionality can be controlled via the properties defined in the <SP_HOME>/conf/<PROFILE>/deployment.yaml file under the auth.configs namespace as shown below.
The following are the properties that can be configured for the external IdP provider:

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmDcrUrl</td>
<td><a href="https://localhost:9443/identity/connect/register">https://localhost:9443/identity/connect/register</a></td>
<td>The Dynamic Client Registration (DCR) endpoint of the key manager in the IdP.</td>
</tr>
<tr>
<td>dcrAppOwner</td>
<td>kmUsername</td>
<td></td>
</tr>
<tr>
<td>kmTokenUrl</td>
<td><a href="https://localhost:9443/oauth2">https://localhost:9443/oauth2</a></td>
<td>The token endpoint of the key manager in the IdP.</td>
</tr>
<tr>
<td>kmUsername</td>
<td>admin</td>
<td>The username for the key manager in the IdP.</td>
</tr>
<tr>
<td>kmPassword</td>
<td>admin</td>
<td>The password for the key manager in the IdP.</td>
</tr>
<tr>
<td>idpBaseUrl</td>
<td><a href="https://localhost:9443/scim2">https://localhost:9443/scim2</a></td>
<td>The SCIM2 endpoint of the IdP.</td>
</tr>
<tr>
<td>idpUsername</td>
<td>admin</td>
<td>The username for the IdP.</td>
</tr>
<tr>
<td>idpPassword</td>
<td>admin</td>
<td>The password for the IdP.</td>
</tr>
<tr>
<td>portalAppContext</td>
<td>portal</td>
<td>The application context of the Dashboard Portal application in WSO2 SP.</td>
</tr>
<tr>
<td>statusDashboardAppContext</td>
<td>monitoring</td>
<td>The application context of the Status Dashboard application in WSO2 SP.</td>
</tr>
<tr>
<td>businessRulesAppContext</td>
<td>business-rules</td>
<td>The application context of the Business Rules application in WSO2 SP.</td>
</tr>
<tr>
<td>databaseName</td>
<td>WSO2_OAUTH_APP_DB</td>
<td>The name of the wso2.datasource used to store the OAuth application credentials</td>
</tr>
<tr>
<td>cacheTimeout</td>
<td>900</td>
<td>The cache timeout for the validity period of the token in seconds.</td>
</tr>
</tbody>
</table>
Writing custom IdP Client

When writing a custom IdP client, the following two interfaces must be implemented:

- **IdPClientFactory**: This is a factory OSGi service that initializes the custom IdP client using the properties from IdPClientConfiguration.
- **IdPClient**: An interface with functions to provide user authentication and retrieval by the other services.

Working with Databases

WSO2 SP stores product-specific data in the H2 database located in the `<SP_HOME>/wso2/<RunTime>/database` directory by default.

This embedded H2 database is suitable for development, testing, and for some production environments. For most production environments, however, we recommend you to use an industry-standard RDBMS such as Oracle, PostgreSQL, MySQL, MS SQL. Most table schemas are self-generated by the feature itself. For others, you can use the scripts provided with WSO2 SP (in the `<SP_HOME>/wso2/<RunTime>/dbscripts` directory) to install and configure databases, including Oracle, PostgreSQL, MySQL and MS SQL.

The following sections explain how to change the default databases and queries:

- Configuring Datasources
- Configuring Database Queries

Configuring Datasources

In WSO2 SP, there are datasources specific to each runtime (i.e., worker, editor, manager, and dashboard runtimes). The datasources of each runtime are defined in the `<SP_HOME>/conf/<runtime>/deployment.yaml` file. e.g., To configure a datasource in the worker runtime, the relevant configurations need to be added in the `<SP_HOME>/conf/worker/deployment.yaml` file.

To view a sample datasource configuration for each database type supported, click on the following links:

If the database driver is not an OSGi bundle, then it should be converted to OSGi (using jartobundle.sh) before placing it in the `<SP_HOME>/lib` directory. For detailed instructions, see Adding Third Party Non OSGi Libraries.

e.g., sh WSO2_SP_HOME/bin/jartobundle.sh ojdbc6.jar WSO2_SP_HOME/lib/

MySQL
wso2.datasources:
dataSources:
  - name: WSO2_TEST_DB
description: The datasource used for test database
jndiConfig:
  - name: jdbc/WSO2_TEST_DB
definition:
  - type: RDBMS
configuration:
  - jdbcUrl: jdbc:mysql://hostname:port/testdb
  - username: root
  - password: root
  - driverClassName: com.mysql.jdbc.Driver
  - maxPoolSize: 50
  - idleTimeout: 60000
  - connectionTestQuery: SELECT 1
  - validationTimeout: 30000
  - isAutoCommit: false

POSTGRES

wso2.datasources:
dataSources:
  - name: WSO2_TEST_DB
description: The datasource used for test database
jndiConfig:
  - name: jdbc/WSO2_TEST_DB
definition:
  - type: RDBMS
configuration:
  - jdbcUrl: jdbc:postgresql://hostname:port/testdb
  - username: root
  - password: root
  - driverClassName: org.postgresql.Driver
  - maxPoolSize: 10
  - idleTimeout: 60000
  - connectionTestQuery: SELECT 1
  - validationTimeout: 30000
  - isAutoCommit: false

Oracle

There are two ways to configure this database type. If you have a System Identifier (SID), use this (older) format:

jdbc:oracle:thin:@[HOST]:[PORT]:SID
wso2.datasources:
  dataSources:
  - name: WSO2_TEST_DB
    description: The datasource used for test database
    jndiConfig:
      name: jdbc/WSO2_TEST_DB
    definition:
      type: RDBMS
    configuration:
      jdbcUrl: jdbc:oracle:thin:@hostname:port:SID
      username: testdb
      password: root
      driverClassName: oracle.jdbc.driver.OracleDriver
      maxPoolSize: 50
      idleTimeout: 60000
      connectionTestQuery: SELECT 1
      validationTimeout: 30000
      isAutoCommit: false

If you have an Oracle service name, use this (newer) format:
jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE

wso2.datasources:
  dataSources:
  - name: WSO2_TEST_DB
    description: The datasource used for test database
    jndiConfig:
      name: jdbc/WSO2_TEST_DB
    definition:
      type: RDBMS
    configuration:
      jdbcUrl: jdbc:oracle:thin:@hostname:port/SERVICE
      username: testdb
      password: root
      driverClassName: oracle.jdbc.driver.OracleDriver
      maxPoolSize: 50
      idleTimeout: 60000
      connectionTestQuery: SELECT 1
      validationTimeout: 30000
      isAutoCommit: false

The Oracle driver need to be converted to OSGi (using `jartobundle.sh`) before put into `WSO2_SP_HOME/lib` directory. For detailed instructions, see Adding Third Party Non OSGi Libraries.

MSSQL
The following sections explain the default datasources configured in various WSO2 SP components for different purposes, and how to change them.

- **RDBMS data provider**
- **Carbon coordination**
- **Stream Processor core - persistence**
- **Stream Processor - Status Dashboard**
- **Siddhi RDBMS store**
- **Carbon Dashboards**
- **Business Rules**
- **IdP client**
- **Permission provider**

### RDBMS data provider

<table>
<thead>
<tr>
<th>Database Access Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The RDBMS provider publishes records from RDBMS tables into generated widgets. It can also be configured to purge records in tables. In order to carry out these actions, this provider requires access to read and delete records in user defined tables of the database. For more information about the RDBMS data provider, see <a href="#">Generating Widgets</a>.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is required if you select a datasource when generating the widget or use existing widgets that connect to the RDBMS data provider when you run the dashboard profile of WSO2 SP.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Default Datasource Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLE_DB</td>
<td>The default H2 database location is <code>&lt;SP_HOME&gt;/wso2/dashboard/database/SAMPLE_DB</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Default Database</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The default database shipped with a sample table named TRANSACTION_TABLE.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schemas and Queries</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The schema for the sample table is <code>TRANSACTIONS_TABLE (creditCardNo VARCHAR(50), country VARCHAR(50), transaction VARCHAR(50), amount INT)</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tested Database Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2, MySQL, Postgres, Mssql, Oracle 11g</td>
<td>The default queries can be viewed <a href="#">here</a>.</td>
</tr>
</tbody>
</table>
### Database Access Requirement

<table>
<thead>
<tr>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>This is required.</strong></td>
<td>Carbon coordination supports zookeeper and RDBMS based coordination. In RDBMS coordination, database access is required for updating the heartbeats of the nodes. In addition, database access is required to update the coordinator and the other members in the cluster. For more information, see Configuring Cluster Coordination.</td>
</tr>
<tr>
<td><strong>This is required.</strong></td>
<td>For more information, see Configuring Cluster Coordination.</td>
</tr>
<tr>
<td><strong>This is optional.</strong></td>
<td>This is required. However, you can also use Zookeeper coordination instead of RDBMS.</td>
</tr>
<tr>
<td><strong>Default Datasource Name</strong></td>
<td>Carbon datasources is used and the name of the default datasource is WSO2_CARBON_DB.</td>
</tr>
<tr>
<td><strong>Tables</strong></td>
<td>LEADER_STATUS_TABLE, MEMBERSHIP_EVENT_TABLE, REMOVED_MEMBERS_TABLE, CLUSTER_NODE_STATUS_TABLE</td>
</tr>
<tr>
<td><strong>Schemas and Queries</strong></td>
<td>Information about the default queries and the schema can be viewed here.</td>
</tr>
<tr>
<td><strong>Tested Database Types</strong></td>
<td>MySQL, Postgres, Mssql, Oracle 11g</td>
</tr>
</tbody>
</table>

### Stream Processor core - persistence

<table>
<thead>
<tr>
<th>Database Access Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>This involves persisting the state of Siddhi Applications periodically in the database. State persistence is enabled by selecting the org.wso2.carbon.stream.processor.core.persistence.DBPersistenceStore class in the state.persistence section of the &lt;SP_HOME&gt;/conf/&lt;worker/manager&gt;/deployment.yaml file. For more information, see Configuring Database and File System State Persistence.</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Required/Optional</strong></td>
<td>This is optional. WSO2 is configured to persist the state of Siddhi applications by default.</td>
</tr>
<tr>
<td><strong>Default Datasource Name</strong></td>
<td>N/A. If state persistence is required, you need to configure the datasource in the &lt;SP_HOME&gt;/conf/&lt;worker/manager&gt;/deployment.yaml file under state.persistence &gt; config &gt; datasource.</td>
</tr>
<tr>
<td><strong>Tables</strong></td>
<td>N/A. If state persistence is required, you need to specify the table name to be used when persisting the state in the &lt;SP_HOME&gt;/conf/&lt;worker/manager&gt;/deployment.yaml file under state.persistence &gt; config &gt; table.</td>
</tr>
<tr>
<td><strong>Schemas and Queries</strong></td>
<td>Information about the default queries and schema can be viewed here.</td>
</tr>
<tr>
<td><strong>Tested Database Types</strong></td>
<td>H2, MySQL, Postgres, Mssql, Oracle 11g</td>
</tr>
</tbody>
</table>

### Stream Processor - Status Dashboard

<table>
<thead>
<tr>
<th>Database Access Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>To display information relating to the status of your SP deployment, the Status Dashboard needs to retrieve carbon metrics data, registered SP worker details and authentication details within the cluster from the database. For more information, see Monitoring Stream Processor.</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Required/Optional</strong></td>
<td>Required</td>
</tr>
<tr>
<td><strong>Default Datasource Name</strong></td>
<td>WSO2_STATUS_DASHBOARD_DB, WSO2_METRICS_DB</td>
</tr>
<tr>
<td><strong>Tables</strong></td>
<td>METRIC_COUNTER, METRIC_GAUGE, METRIC_HISTOGRAM, METRIC_METER, METRIC_TIMER, WORKERS_CONFIGURATIONS, WORKERS_DETAILS</td>
</tr>
<tr>
<td><strong>Schemas and Queries</strong></td>
<td>Information about the default queries and schema: <a href="https://github.com/wso2/carbon-analytics/blob/v2.0.250/components/org.wso2.carbon.status.dashboard.core/src/main/resources/queries.yaml">https://github.com/wso2/carbon-analytics/blob/v2.0.250/components/org.wso2.carbon.status.dashboard.core/src/main/resources/queries.yaml</a></td>
</tr>
<tr>
<td><strong>Tested Database Types</strong></td>
<td>H2, MySQL, Mssql, Oracle 11g (Postgres is tested with Carbon-Metrics only)</td>
</tr>
</tbody>
</table>
## Siddhi RDBMS store

<table>
<thead>
<tr>
<th>Database Access Requirement</th>
<th>It gives the capability of creating the tables at the siddhi app runtime and access the existing tables if a user defined carbon datasource or JNDI property in a siddhi app. Documentation can be found in <a href="https://wso2-extensions.github.io/siddhi-store-rdbms/api/4.0.15/">https://wso2-extensions.github.io/siddhi-store-rdbms/api/4.0.15/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required/Optional</strong></td>
<td>Optional</td>
</tr>
<tr>
<td>Default Datasource Name</td>
<td>No such default Datasource. User has to create the datasource in the Siddhi app</td>
</tr>
<tr>
<td>Tables</td>
<td>No such default tables. User has to define the tables</td>
</tr>
<tr>
<td><strong>Tested Database Types</strong></td>
<td>H2, MySQL, Mssql, Oracle 11g, DB2, PostgreSQL</td>
</tr>
</tbody>
</table>

## Carbon Dashboards

<table>
<thead>
<tr>
<th>Database Access Requirement</th>
<th>Carbon Dashboard feature uses its datasource to persist the dashboard related information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required/Optional</strong></td>
<td>Optional</td>
</tr>
<tr>
<td>Default Datasource Name</td>
<td>WSO2_DASHBOARD_DB</td>
</tr>
<tr>
<td>Tables</td>
<td>DASHBOARD_RESOURCE</td>
</tr>
<tr>
<td><strong>Tested Database Types</strong></td>
<td>H2, MySQL</td>
</tr>
</tbody>
</table>

## Business Rules

<table>
<thead>
<tr>
<th>Database Access Requirement</th>
<th>Business Rules feature uses database to persist the derived business rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required/Optional</strong></td>
<td>Mandatory</td>
</tr>
<tr>
<td>Default Datasource Name</td>
<td>BUSINESS_RULES_DB</td>
</tr>
<tr>
<td>Tables</td>
<td>BUSINESS_RULES, RULESTEMPLATES</td>
</tr>
<tr>
<td><strong>Tested Database Types</strong></td>
<td>H2, MySQL, Oracle 11g</td>
</tr>
<tr>
<td>Database Access Requirement</td>
<td>IdP client access the DB layer to persist the client id and the client secret of dynamic client registration</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Required/Optional</td>
<td>Mandatory for external IdP clients</td>
</tr>
<tr>
<td>Default Datasource Name</td>
<td>DB_AUTH_DB</td>
</tr>
<tr>
<td>Tables</td>
<td>OAUTH_APPS</td>
</tr>
<tr>
<td>Tested Database Types</td>
<td>H2</td>
</tr>
</tbody>
</table>

| Permission provider        | Permission provider will access the DB to persist permissions and role - permission mappings.          |
|----------------------------|-------------------------------------------------------------------------------------------------
| Database Access Requirement|                                                                                                   |
| Required/Optional          | Mandatory, default is in H2                                                                       |
| Default Datasource Name    | PERMISSIONS_DB                                                                                      |
| Tables                     | PERMISSIONS, ROLE_PERMISIONS                                                                        |
| Tested Database Types      | H2, MySQL, Mssql, Oracle 11g                                                                         |

### Configuring Database Queries

Database queries used in the following features and can configured for new database or override the default queries for the new database version. The relevant configurations should be added to the `<SP_Home>/conf/<Runtime>/deployment.yaml` file.

- Business Rules
- Status Dashboard
- Dashboard
- RDBMS Data Provider

The following are sample database query configurations that override the MySQL default version queries in the RDBMS Data Provider Feature by MySQL new version 5.7.0 and adding new queries for new database type MariaDB with version 10.0.20 respectiveley:
wso2.rdbms.data.provider:
  
  mappings:
    record_delete: "DELETE FROM {{TABLE_NAME}} ORDER BY {{INCREMENTAL_COLUMN}} ASC LIMIT {{LIMIT_VALUE}}"
    total_record_count: "SELECT COUNT(*) FROM {{TABLE_NAME}}"
    record_limit: "{{CUSTOM_QUERY}} ORDER BY {{INCREMENTAL_COLUMN}} DESC LIMIT {{LIMIT_VALUE}}"
    record_greater_than: "SELECT * FROM {{CUSTOM_QUERY}} ORDER BY {{INCREMENTAL_COLUMN}} DESC ) AS A_TABLE WHERE {{INCREMENTAL_COLUMN}} > {{LAST_RECORD_VALUE}}"
  
  type: MySQL
  version: 5.7.0

Business Rules

To add or override database queries for Business Rules you need to configure the `<SP_Home>/conf/dashboard/deployment.yaml` file as indicated in the above example:

2. Copy the queries with the above structure under the `'wso2.business.rules.manager'` namespace.

Status Dashboard

To add or override database queries for Status Dashboard you need to configure the `<SP_Home>/conf/dashboard/deployment.yaml` file as indicated in the above example:

2. Copy the queries with the above structure under the `'wso2.status.dashboard'` namespace.

Dashboard

To add or override database queries for Dashboard you need to configure the `<SP_Home>/conf/dashboard/deployment.yaml` file as indicated in the above example:

2. Copy the queries with the above structure under the `'wso2.dashboard'` namespace.

RDBMS Data Provider

To add or override database queries for RDBMS Data Provider you need to configure the `<SP_Home>/conf/dashboard/deployment.yaml` file as in
Authorization & Permission Model

This page will provide information about the permission model of REST APIs available in each runtime of WSO2 Stream Processor. If you want the complete set of REST APIs which available in WSO2 SP, please find it here. You can find the REST APIs for each runtime and its permission model in the below pages.

- Worker Runtime - REST APIs Permission Model
- Manager Runtime - REST APIs Permission Model

Worker Runtime - REST APIs Permission Model

There are two sets of REST APIs available in worker runtime. Stream Processor APIs and Event Simulator APIs have following permission model. You need to have appropriate permission to invoke these APIs.

Stream Processor APIs

<table>
<thead>
<tr>
<th>Method</th>
<th>API Context</th>
<th>Required Permission</th>
</tr>
</thead>
</table>
| POST   | /siddhi-apps | PermissionString - siddhiApp.manage  
AppName - SAPP |
| PUT    | /siddhi-apps | PermissionString - siddhiApp.manage  
AppName - SAPP |
| DELETE | /siddhi-apps/{appName} | PermissionString - siddhiApp.manage  
AppName - SAPP |
| GET    | /siddhi-apps | PermissionString - siddhiApp.manage or siddhiApp.view  
AppName - SAPP |
| GET    | /siddhi-apps/{appName} | PermissionString - siddhiApp.manage or siddhiApp.view  
AppName - SAPP |
| GET    | /siddhi-apps/{appName}/status | PermissionString - siddhiApp.manage or siddhiApp.view  
AppName - SAPP |
| POST   | /siddhi-apps/{appName}/backup | PermissionString - siddhiApp.manage  
AppName - SAPP |
| POST   | /siddhi-apps/{appName}/restore | PermissionString - siddhiApp.manage  
AppName - SAPP |
| GET    | /statistics | PermissionString - siddhiApp.manage or siddhiApp.view  
AppName - SAPP |
| PUT    | /statistics | PermissionString - siddhiApp.manage  
AppName - SAPP |
| GET    | /system-details | PermissionString - siddhiApp.manage or siddhiApp.view  
AppName - SAPP |
<table>
<thead>
<tr>
<th>Method</th>
<th>API Context</th>
<th>Required Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/siddhi-apps/statistics</td>
<td>PermissionString - siddhiApp.manage or siddhiApp.view&lt;br&gt;AppName - SAPP</td>
</tr>
<tr>
<td>PUT</td>
<td>/siddhi-apps/{appName}/statistics</td>
<td>PermissionString - siddhiApp.manage&lt;br&gt;AppName - SAPP</td>
</tr>
<tr>
<td>PUT</td>
<td>/siddhi-apps/statistics</td>
<td>PermissionString - siddhiApp.manage&lt;br&gt;AppName - SAPP</td>
</tr>
</tbody>
</table>

Event Simulator APIs

<table>
<thead>
<tr>
<th>Method</th>
<th>API Context</th>
<th>Required Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/simulation/single</td>
<td>PermissionString - simulator.manage&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>POST</td>
<td>/simulation/feed</td>
<td>PermissionString - simulator.manage&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>GET</td>
<td>/simulation/feed</td>
<td>PermissionString - simulator.manage or simulator.view&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>PUT</td>
<td>/simulation/feed/{simulationName}</td>
<td>PermissionString - simulator.manage&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>GET</td>
<td>/simulation/feed/{simulationName}</td>
<td>PermissionString - simulator.manage or simulator.view&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>DELETE</td>
<td>/simulation/feed/{simulationName}</td>
<td>PermissionString - simulator.manage&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>POST</td>
<td>/simulation/feed/{simulationName}?action=run</td>
<td>PermissionString - simulator.manage&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>POST</td>
<td>/simulation/feed/{simulationName}?action=pause</td>
<td>PermissionString - simulator.manage&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>POST</td>
<td>/simulation/feed/{simulationName}?action=resume</td>
<td>PermissionString - simulator.manage&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>POST</td>
<td>/simulation/feed/{simulationName}?action=stop</td>
<td>PermissionString - simulator.manage&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>GET</td>
<td>/simulation/feed/{simulationName}/status</td>
<td>PermissionString - simulator.manage or simulator.view&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>POST</td>
<td>/simulation/files</td>
<td>PermissionString - simulator.manage&lt;br&gt;AppName - SIM</td>
</tr>
<tr>
<td>GET</td>
<td>/simulation/files</td>
<td>PermissionString - simulator.manage or simulator.view&lt;br&gt;AppName - SIM</td>
</tr>
</tbody>
</table>
Manager Runtime - REST APIs Permission Model

Manager runtime only have Stream Processor(Siddhi APIs) APIs and have following permission model. You need to have appropriate permission to invoke these APIs.

<table>
<thead>
<tr>
<th>Method</th>
<th>API Context</th>
<th>Required Permission</th>
</tr>
</thead>
</table>
| POST   | /siddhi-apps | PermissionString - siddhiApp.manage  
              AppName - SAPP |
| PUT    | /siddhi-apps | PermissionString - siddhiApp.manage  
              AppName - SAPP |
| DELETE | /siddhi-apps/{appName} | PermissionString - siddhiApp.manage  
                                             AppName - SAPP |
| GET    | /siddhi-apps | PermissionString - siddhiApp.manage or siddhiApp.view  
                                             AppName - SAPP |
| GET    | /siddhi-apps/{appName} | PermissionString - siddhiApp.manage or siddhiApp.view  
                                             AppName - SAPP |
| GET    | /siddhi-apps/{appName}/status | PermissionString - siddhiApp.manage or siddhiApp.view  
                                             AppName - SAPP |
| POST   | /siddhi-apps/{appName}/backup | PermissionString - siddhiApp.manage  
                                             AppName - SAPP |
| POST   | /siddhi-apps/{appName}/restore | PermissionString - siddhiApp.manage  
                                             AppName - SAPP |
| GET    | /statistics | PermissionString - siddhiApp.manage or siddhiApp.view  
                                             AppName - SAPP |
| PUT    | /statistics | PermissionString - siddhiApp.manage  
                                             AppName - SAPP |
| GET    | /system-details | PermissionString - siddhiApp.manage or siddhiApp.view  
                                             AppName - SAPP |
GET /siddhi-apps/statistics
PermissionString - siddhiApp.manage or siddhiApp.view
AppName - SAPP

PUT /siddhi-apps/{appName}/statistics
PermissionString - siddhiApp.manage
AppName - SAPP

PUT /siddhi-apps/statistics
PermissionString - siddhiApp.manage
AppName - SAPP

General Data Protection Regulations (GDPR) for WSO2 Stream Processor

The General Data Protection Regulation (GDPR) is a new legal framework formalized by the European Union (EU) in 2016. This regulation comes into effect from 28 May 2018, and can affect any organization that processes Personally Identifiable Information (PII) of individuals who live in Europe. Organizations that fail to demonstrate GDPR compliance are subjected to financial penalties.

The following topics cover how GDPR-compliancy is achieved in WSO2 Stream Processor.

**Removing Personally Identifiable Information via the Forget-me Tool**

In WSO2 SP, event streams specify the schema for events to be selected into the SP event flow to be processed. This schema can include user IDs and other PII (Personally Identifiable Information) that you want to delete from log files and such. This can be done via the Forget-me Tool.

- **Step 1:** Configure the config.json file
- **Step 2:** Execute the Forget-me tool

**Step 1: Configure the config.json file**

The `<SP_HOME>/wso2/tools/identity-anonymization-tool-x.x.x/conf/config.json` file specifies the locations from which persisted data need to be removed.

The log-file processor is specified in the configuration file of the Forget-Me tool as shown in the sample below in order to remove data with PII from the logs. If you have configured logs with PII to be saved in another location, you can add it to this list of processors.

Do you want to learn more about GDPR?
If you are new to GDPR, we recommend that you take a look at our article series on **Creating a Winning GDPR Strategy**.

- Part 1 - Introduction to GDPR
- Part 2 - 7 Steps for GDPR Compliance
- Part 3 - Identity and Access Management to the Rescue
- Part 4 - GDPR Compliant Consent Design

For more resources on GDPR, see the white papers, case studies, solution briefs, webinars, and talks published on our WSO2 GDPR homepage. You can also find the original GDPR legal text here.
This extract shows the default configuration of WSO2 SP. SP only saves PII in log files by default. Therefore, this configuration allows the Forget-me tool to delete these logs that are saved in `<SP_HOME>/wso2/<PROFILE>/logs` directory.

**Step 2: Execute the Forget-me tool**

To execute the Forget-me tool, issue the following command pointing to the `<SP_HOME>` directory.

```
forget-me -U <USERNAME> -d <CONF_DIR> -carbon <SP_HOME>
```

**Removing References to Deleted User Identities**

This section covers how to remove references to deleted user identities in WSO2 SP by running the Forget-me tool.

**Before you begin**

- Note that this tool is designed to run in offline mode (i.e., the server should be shut down or run on another machine) in order to prevent unnecessary load to the server. If this tool runs in online mode (i.e., when the server is running), DB lock situations on the H2 databases may occur.
- If you have configured any JDBC database other than the H2 database provided by default, copy the relevant JDBC driver to the `<SP_HOME>/wso2/tools/identity-anonymization-tool/lib` directory.

1. Open a new terminal window and navigate to the `<SP_HOME>/bin` directory.
2. Execute one of the following commands depending on your operating system:
   - On Linux/Mac OS: `/forgetme.sh -U <username>`
   - On Windows: `forgetme.bat -U <username>

**Note**

The commands specified above use only the `-U <username>` option, which is the only required option to run the tool. There are several other optional command line options that you can specify based on your requirement. The supported options are described in detail below.

<table>
<thead>
<tr>
<th>Command Line Option</th>
<th>Description</th>
<th>Required</th>
<th>Sample Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>The name of the user whose identity references you want to remove.</td>
<td>Yes</td>
<td><code>-U john.doe</code></td>
</tr>
<tr>
<td>d</td>
<td>The configuration directory to use when the tool is run. If you do not specify a value for this option, the <code>&lt;SP_HOME&gt;/wso2/tools/identity-anonymization-tool-x.x.x/conf</code> directory (which is the default configuration directory of the tool) is used.</td>
<td>No</td>
<td><code>-d &lt;TOOL_HOME&gt;/conf</code></td>
</tr>
</tbody>
</table>
Forget-me Tool Overview

The Forget-me tool is shipped with WSO2 SP by default in the `<SP_HOME>/wso2/tools/identity-anonymization-tool-x.x.x` directory. If required, you can change the default location of the configurations of this tool or make changes to the default configurations. You can also run the Forget-me tool in the standalone mode.

- Changing the default configurations location
- Changing the default configurations of the tool
- Running the Forget-me tool in the standalone mode

Changing the default configurations location

You can change the default location of the tool configurations if desired. You may want to do this if you are working with a multi-product environment where you want to manage configurations in a single location for ease of use. Note that this is optional.

To change the default configurations location for the embedded tool, do the following:

1. Open the `forgetme.sh` file found inside the `<SP_HOME>/bin` directory.
2. The location path is the value given after `-d` within the following line. Modify the value after `-d` to change the location.

   The default location path is `$CARBON_HOME/repository/components/tools/forget-me/conf`.

   ```sh
   sh
   $CARBON_HOME/repository/components/tools/identity-anonymization-tool/bin/forget-me -d
   $CARBON_HOME/repository/components/tools/identity-anonymization-tool/conf -carbon $CARBON_HOME $@
   ```

Changing the default configurations of the tool

All configurations related to this tool can be found inside the `<SP_HOME>/wso2/tools/identity-anonymization-tool/conf` directory. The default configurations are set up as follows:

- **Read Logs**: `<SP_HOME>/wso2/<PROFILE>/logs`
- **Read Datasource**: `<SP_HOME>/conf/<PROFILE>deployment.yaml`
**Default datasources:** WSO2_CARBON_DB, WSO2_METRICS_DB, WSO2_PERMISSIONS_DB, WSO2_DASHBOARD_DB, BUSINESS_RULES_DB, SAMPLE_DB, WSO2_STATUS_DASHBOARD_DB

**Log file name regex:** The regex patterns defined in all the files in the `<SP_HOME>/wso2/tools/identity-anonymization-tool/conf/log-config` directory are considered.

For information on changing these configurations, see Configuring the `config.json` file in the Product Administration Guide.

Running the Forget-me tool in the standalone mode

This tool can run standalone and therefore cater to multiple products. This means that if you are using multiple WSO2 products and need to delete the user's identity from all products at once, you can do so by running the tool in standalone mode.

For information on how to build and run the Forget-Me tool, see Removing References to Deleted User Identities in WSO2 Products in the WSO2 Administration Guide.

**Creating GDPR Compliant Siddhi Applications**

The obfuscation/removal of such PII (Personally Identifiable Information) can be handled in WSO2 Stream Processor by Siddhi Applications that can either modify or remove records that contain the PII. These Siddhi Applications can be written in a way to match the original queries that captured data for persistence so that the same data can be modified or removed as required. For more information about writing Siddhi Queries, see Siddhi Query Guide.

The following sections explain how obfuscation/deletion of sensitive data can be managed via Siddhi queries in a custom Siddhi application developed based on a specific user case.

- **Obfuscating PII**
- **Deleting PII**

**Obfuscating PII**

Let's consider a Siddhi application that includes the following store definition to persist streaming data.

```siddhi
define table customerTable (customerId string, customerName string, entryVector int);
```

In this example, the customer ID is considered PII, and a customer with the `XXX` ID wants that ID to be hidden in the system so that he/she cannot be personally identified with it. Therefore, you need to obfuscate the value for the `customerId` attribute. This can be done by creating an algorithm to create a hashed value or a pseudonym to replace a specific value for the `customerId` attribute.

Let's consider that such an algorithm exists (e.g., as a function named `anonymize`). To invoke this function, you need to add a new query to the Siddhi application as shown in the sample below.

```siddhi
define table customerTable (customerId string, customerName string, entryVector int);

define stream UpdateStream (customerId string);

from UpdateStream
select *
update customerTable
set customerTable.customerName = anonymize(customerTable.customerName)
on customerTable.customerId == XXX;
```

In the above Siddhi application, the query in bold is triggered when a new event is received in the `UpdateStream` stream where the value for the `customerId` attribute is `XXX`. Once it is triggered, the `XXX` customer ID is replaced with a pseudonym.

For more information about writing custom functions, see Siddhi Query Guide - Writing Custom Extensions.

**Deleting PII**

Let's assume that the customer ID in the scenario described above needs to be deleted. To do this, you can write a Siddhi query to delete the value for the `customerId` attribute when is equal to `XXX` as shown below.

```siddhi
define table customerTable (customerId string, customerName string, entryVector int);

define stream UpdateStream (customerId string);

from UpdateStream
select *
update customerTable
set customerTable.customerName = anonymize(customerTable.customerName)
on customerTable.customerId == XXX;
```
define table customerTable (customerId string, customerName string, entryVector int);
define stream DeleteStream (customerId string);

from DeleteStream
delete customerTable
on customerTable.customerId == customerId;

In the above Siddhi application, the query in bold is triggered when a new event is received in the DeleteStream stream where the value for the customerId attribute is XXX. Once it is triggered, the XXX customer ID is deleted.

For more information about the Delete operator used here, see Siddhi Query Guide - Delete.

Adding Third Party Non OSGi Libraries

WSO2 SP is an OSGi-based product. Therefore, when you integrate third party products such as Oracle with WSO2 SP, you need to check whether the libraries you need to add to WSO2 SP are OSGi based. If they are not, you need to convert them to OSGi bundles before adding them to the <SP_HOME>/lib directory.

To convert jar files to OSGi bundles, follow the procedure given below:

1. Download the non-OSGi jar for the required third party product, and save it in a preferred directory in your machine.
2. In your CLI, navigate to the <SP_HOME>/bin directory. Then issue the following command.

   ```
   .jartobundle.sh <PATH_TO_NON-OSGi_JAR> ../lib
   ```

   This generates the converted file in the <SP_HOME>/lib directory.
3. Restart the WSO2 SP server.

Configuring Logging for WSO2 SP

The following sections cover how to set up logging in WSO2 SP.

- Configuring the logging framework

Configuring the logging framework

WSO2 SP the logging framework is implemented by using PAX Logging as the underlying library. High performing log4j 2.0 is used as the logging backend with this framework. This framework supports a number of logging APIs that allows you to use any logging API in the components you develop.

For more information about enabling logging for WSO2 SP, see the following topics.

- Configuring Carbon Logs

Configuring Carbon Logs

This section explains how to enable logging at different levels for WSO2 SP. This is done by adding the required configurations in <SP_HOME>/conf<PROFILE>/log4j2.xml file.
<Configuration>
  <Appenders>
    <Console name="CARBON_CONSOLE" target="SYSTEM_OUT">
      <PatternLayout pattern="[%d] %5p {%c} - %m%ex%n"/>
    </Console>
    <RollingFile name="CARBON_LOGFILE"
      fileName="${sys:carbon.home}/logs/carbon.log"
      filePattern="${sys:carbon.home}/logs/carbon-%d{MM-dd-yyyy}.log">
      <PatternLayout pattern="[%d] %5p {%c} - %m%ex%n"/>
      <Policies>
        <TimeBasedTriggeringPolicy/>
      </Policies>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="CARBON_CONSOLE"/>
      <AppenderRef ref="CARBON_LOGFILE"/>
    </Root>
  </Loggers>
</Configuration>

The above configuration contains the following elements:

**Appenders**

The logs you enable can be printed in multiple destinations known as appenders. The following are the appenders that are currently supported.

- **Carbon Console**
  Carbon console is the console in which WSO2 SP is run.
- **Log file**

**Loggers**

e.g., To enable debug logs on the `org.wso2.carbon.kernel.runtime` package, you can add the following entry to the `log4j2.xml` file.

```
<Logger name="org.wso2.carbon.kernel.runtime" level="debug"/>
```

**Viewing logs**

Logs are published to the console. In addition, when you enable one or more logs for a specific profile, those logs are printed in the `carbon.log` file that is automatically created in the `<SP_HOME>/wso2/<PROFILE>/log` directory.

**Reference Guide**

Reference Guide contains information on the following sub topics.

- Stream Processor REST API Guide
- Performance Analysis Results
- Configuring Default Ports
- Working with Data Providers

**Stream Processor REST API Guide**
The following topics cover information relating to the public APIs exposed from WSO2 Stream Processor.

- Public APIs
- HTTP Status Codes

Public APIs

The following topics list the APIs supported for WSO2 Stream processor from its Worker, Manager, Editor and Dashboard runtimes.

- Siddhi Application Management APIs
- Event Simulation APIs
- Status Monitoring APIs
- Dashboard APIs
- Authentication APIs
- Permission APIs
- Business Rules APIs
- Store APIs

Siddhi Application Management APIs

- Updating a Siddhi Application
- Deleting a Siddhi application
- Listing all active Siddhi applications
- Retrieving a specific Siddhi application
- Fetching the status of a Siddhi Application
- Taking a snapshot of a Siddhi Application
- Restoring a Siddhi Application via a snapshot
- Returning real-time statistics of a worker
- Enabling/disabling worker statistics
- Returning general details of a worker
- Returning detailed statistics of all Siddhi applications
- Enabling/disabling the statistics of a specific Siddhi application
- Enabling/disabling the statistics of all Siddhi applications

Creating a Siddhi application

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Creates a new Siddhi Application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/siddhi-apps</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>Request: text/plain Response: application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>worker/manager</td>
</tr>
</tbody>
</table>

Curl command syntax

```bash
curl -X POST "https://localhost:9443/siddhi-apps" -H "accept: application/json" -H "Content-Type: text/plain" -d @TestSiddhiApp.siddhi -u admin:admin -k
```

Sample curl command
curl -X POST "https://localhost:9443/siddhi-apps" -H "accept: application/json" -H "Content-Type: text/plain" -d @TestSiddhiApp.siddhi -u admin:admin -k

Sample output

The response for the sample curl command given above can be one of the following.

- If API request is valid and there is no existing Siddhi application with the given name, a response similar to the following is generated with response code 201. This response contains a location header with the path of the newly created file from product root home.

- If the API request is valid, but a Siddhi application with the given name already exists, a response similar to the following is generated with response code 409.

  ```json
  {
    "type": "conflict",
    "message": "There is a Siddhi App already exists with same name"
  }
  ```

- If the API request is invalid due to invalid content in the Siddhi queries you have included in the request body, a response similar to the following is generated with response code 400.

  ```json
  {
    "code": 800101,
    "type": "validation error",
    "message": "You have an error in your SiddhiQL at line 8:8, missing INTO at 'BarStream'"
  }
  ```

- If the API request is valid, but an exception occurred during file processing or saving, the following response is generated with response code 500.

  ```json
  {
    "code": 800102,
    "type": "file processing error",
    "message": <error-message>
  }
  ```

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Possible codes are 201, 409, 400, and 500. For descriptions of the HTTP status codes, see HTTP Status Codes.</th>
</tr>
</thead>
</table>

Updating a Siddhi Application

Overview

| Description       |
|-------------------|-------------------------------------------------|
| Updates a Siddhi Application. |
**API Context**
/siddhi-apps

**HTTP Method**
PUT

**Request/Response format**

<table>
<thead>
<tr>
<th>Request</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>text/plain</td>
<td>application/json</td>
</tr>
</tbody>
</table>

**Authentication**
Basic

**Username**
admin

**Password**
admin

**Runtime**
worker/manager

**curl command syntax**

```
curl -X PUT "http://localhost:9090/siddhi-apps" -H "accept: application/json" -H "Content-Type: text/plain" -d @<SIDDHI_APPLICATION_NAME>.siddhi -u admin:admin -k
```

**Sample curl command**

```
curl -X PUT "https://localhost:9443/siddhi-apps" -H "accept: application/json" -H "Content-Type: text/plain" -d @TestSiddhiApp.siddhi -u admin:admin -k
```

**Sample output**

- If the API request is invalid due to invalid content in the Siddhi query, a response similar to the following is returned with response code 400.

  ```json
  {
  "code": 800101,
  "type": "validation error",
  "message": "You have an error in your SiddhiQL at line 8:8, missing INTO at 'BarStream'"
  }
  ```

- If the API request is valid, but an exception occurred when saving or processing files, a response similar to the following is returned with response code 500.

  ```json
  {
  "code": 800102,
  "type": "file processing error",
  "message": <error-message>
  }
  ```

**Response**

**HTTP Status Code**

| Possible codes are 200, 201, 400, and 500. |
| For descriptions of the HTTP status codes, see [HTTP Status Codes](#)|
Deleting a Siddhi application

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Sends the name of a Siddhi application as a URL parameter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/siddhi-apps/{appName}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>DELETE</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>worker/manager</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{appName}</td>
<td>The name of the Siddhi application to be deleted.</td>
</tr>
</tbody>
</table>

curl command syntax


Sample curl command


Sample output

The response for the sample curl command given above can be one of the following:

- If the API request is valid and a Siddhi application with the given name exists, the following response is received with response code 200.

  http://localhost:9090/siddhi-apps/TestExecutionPlan1

- If the API request is valid, but a Siddhi application with the given name is not deployed, the following response is received with response code 404.
If the API request is valid, but an exception occured when deleting the given Siddhi application, the following response is received with response code 500.

```json
{
   "code": 800102,
   "type": "file processing error",
   "message": "<error-message>
}
```

If the API request is valid, but there are restricted characters in the given Siddhi application name, the following response is received with response code 400.

```json
{
   "code": 800101,
   "type": "validation error",
   "message": "File name contains restricted path elements . : 
../../siddhiApp2"
}
```

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200, 404, 500 or 400.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

**Listing all active Siddhi applications**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Lists all the currently active Siddhi applications.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If the <code>isActive=true</code> parameter is set, all the active Siddhi Applications are listed. If not, all the inactive Siddhi applications are listed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>API Context</th>
<th>/siddhi-apps</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HTTP Method</strong></td>
<td>GET</td>
</tr>
<tr>
<td><strong>Request/Response format</strong></td>
<td>Request content type: any</td>
</tr>
<tr>
<td></td>
<td>Response content type: application/json</td>
</tr>
<tr>
<td><strong>Authentication</strong></td>
<td>Basic</td>
</tr>
<tr>
<td><strong>Username</strong></td>
<td>admin</td>
</tr>
<tr>
<td><strong>Password</strong></td>
<td>admin</td>
</tr>
<tr>
<td><strong>Runtime</strong></td>
<td>worker/manager</td>
</tr>
</tbody>
</table>
curl command syntax

```
```

Sample curl command

```
```

Sample output

Possible responses are as follows:

- If the API request is valid and there are Siddhi applications deployed in your SP setup, a response similar to the following is returned with response code 200.

  ```
  ["TestExecutionPlan3", "TestExecutionPlan4"]
  ```

- If the API request is valid, there are Siddhi applications deployed in your SP setup, and a query parameter is defined in the request, a response similar to the following is returned with response code 200. This response only contains Siddhi applications that are active.

  ```
  If these conditions are met, but the isActive parameter is set to false, the response contains only inactive Siddhi applications.
  ```

  ```
  ["TestExecutionPlan3"]
  ```

- If the API request is valid, but there are no Siddhi applications deployed in your SP setup, the following response is returned.

  ```
  []
  ```

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>200</td>
</tr>
</tbody>
</table>

For descriptions of the HTTP status codes, see [HTTP Status Codes](#).

Retrieving a specific Siddhi application

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Retrieves the given Siddhi application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/siddhi-apps/{appName}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
</tbody>
</table>
Password | admin
--- | ---
Runtime | worker/manager

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{app\Name}</td>
<td>The name of the Siddhi application to be retrieved.</td>
</tr>
</tbody>
</table>

**curl command syntax**

```
```

**Sample curl command**

```
```

**Sample output**

The possible outputs are as follows:

- If the API request is valid and a Siddhi application of the given name exists, a response similar to the following is returned with response code 200.

```json
{
    "content": "\nPlan:name('TestExecutionPlan')
define stream FooStream (symbol string, price float, volume long);
source(type='inMemory', topic='symbol', @map(type='passThrough'))Define stream BarStream (symbol string, price float, volume long);
from FooStream
select symbol, price, volume
insert into BarStream;
}
```

- If the API request is valid, but a Siddhi application of the given name is not deployed, a response similar to the following is returned with response code 404.

```json
{
    "type": "not found",
    "message": "There is no Siddhi App exist with provided name : TestExecutionPlan1"
}
```
Fetching the status of a Siddhi Application

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>This fetches the status of the specified Siddhi application</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/siddhi-apps/{appName}/status</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>worker/manager</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{appName}</td>
<td>The name of the Siddhi application of which the status needs to be fetched.</td>
</tr>
</tbody>
</table>

Curl command syntax

```bash
```

Sample curl command

```bash
```

Sample output

- If the Siddhi application is active, the following is returned with response code 200.
  ```json
  {"status":"active"}
  ```

- If the Siddhi application is inactive, the following is returned with response code 200.
  ```json
  {"status":"inactive"}
  ```

- If the Siddhi application does not exist, but the REST API call is valid, the following is returned with the response code 404.
There is no Siddhi App exist with provided name : TestExecutionPlan1

Taking a snapshot of a Siddhi Application

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>This takes a snapshot of the specific Siddhi application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/siddhi-apps/{appName}/backup</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>worker/manager</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{appName}</td>
<td>The name of the Siddhi application of which a snapshot needs to be taken.</td>
</tr>
</tbody>
</table>

curl command syntax

```
curl -X POST "http://localhost:9090/siddhi-apps/{appName}/backup" -H "accept: application/json" -u admin:admin -k
```

Sample curl command

```
```

Sample output

The output can be one of the following:
- If the API request is valid and a Siddhi application exists with the given name, an output similar to the following (i.e., with the snapshot revision number) is returned with response code 201.

```json
{"revision": "89489242494242"}
```

- If the API request is valid, but no Siddhi application with the given name is deployed, an output similar to the following is returned with response code 404.

```json
{
   "type": "not found",
   "message": "There is no Siddhi App exist with provided name : TestExecutionPlan1"
}
```

- If the API request is valid, but an exception has occurred when backing up the state at Siddhi level, an output similar to the following is returned with response code 500.

```json
{
   "code": 800102,
   "type": "file processing error",
   "message": <error-message>
}
```

**Response**

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>201, 404, or 500.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

**Restoring a Siddhi Application via a snapshot**

In order to call this API, you need to have already taken a snapshot of the Siddhi application to be restored. For more information about the API via which the snapshot is taken, see [Taking a snapshot of a Siddhi application](#).

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>This restores a Siddhi application using a snapshot of the same that you have previously taken.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>• To restore without considering the version: /siddhi-apps/{appName}/restore</td>
</tr>
<tr>
<td></td>
<td>• To restore a specific version: /siddhi-apps/{appName}/restore?version=</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>worker/manager</td>
</tr>
</tbody>
</table>

**Parameter Description**
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{appName}</td>
<td>The name of the Siddhi application that needs to be restored.</td>
</tr>
</tbody>
</table>

curl command syntax

```
curl -X POST "http://localhost:9090/siddhi-apps/{appName}/restore" -H "accept: application/json" -u admin:admin -k
```

Sample curl command

```
curl -X POST
```

Sample output

The above sample curl command can generate either one of the following responses:

- If the API request is valid, a Siddhi application with the given name exists, and no revision information is passed as a query parameter, the following response is returned with response code 200.

```
{
   "type": "success",
   "message": "State restored to last revision for Siddhi App :TestExecutionPlan"
}
```

- If the API request is valid, a Siddhi application with the given name exists, and revision information is passed as a query parameter, the following response is returned with response code 200. In this scenario, the Siddhi snapshot is created in the file system.

```
{
   "type": "success",
   "message": "State restored to revision 1234563 for Siddhi App :TestExecutionPlan"
}
```

- If the API request is valid, but no Siddhi application is deployed with the given name, the following response is returned with response code 404.

```
{
   "type": "not found",
   "message": "There is no Siddhi App exist with provided name : TestExecutionPlan1"
}
```

- If the API request is valid, but an exception occurred when restoring the state at Siddhi level, the following response is returned with response code 500.
### Response

| HTTP Status Code | 200, 404 or 500. For descriptions of the HTTP status codes, see [HTTP Status Codes](#). |

#### Returning real-time statistics of a worker

<table>
<thead>
<tr>
<th><strong>Overview</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>API Context</strong></td>
</tr>
<tr>
<td><strong>HTTP Method</strong></td>
</tr>
<tr>
<td><strong>Request/Response format</strong></td>
</tr>
<tr>
<td><strong>Authentication</strong></td>
</tr>
<tr>
<td><strong>Username</strong></td>
</tr>
<tr>
<td><strong>Password</strong></td>
</tr>
<tr>
<td><strong>Runtime</strong></td>
</tr>
</tbody>
</table>

#### Parameter Description

<table>
<thead>
<tr>
<th>curl command syntax</th>
</tr>
</thead>
</table>

#### Sample output

<table>
<thead>
<tr>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HTTP Status Code</strong></td>
</tr>
</tbody>
</table>

#### Enabling/disabling worker statistics

<table>
<thead>
<tr>
<th><strong>Overview</strong></th>
</tr>
</thead>
</table>

---

```json
{
    "code": 800102,
    "type": "file processing error",
    "message": <error-message>
}
```
<table>
<thead>
<tr>
<th>Description</th>
<th>Enables/disables generating statistics for worker nodes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/statistics</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>PUT</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>worker/manager</td>
</tr>
</tbody>
</table>

**Parameter Description**

**curl command syntax**

```bash
```

**Sample output**

**Response**

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

**Returning general details of a worker**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns general details of a worker.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/system-details</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>worker/manager</td>
</tr>
</tbody>
</table>

**Parameter Description**

**curl command syntax**
Sample curl command

```bash
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

**Returning detailed statistics of all Siddhi applications**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the detailed statistics of all the Siddhi applications currently deployed in the SP setup.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/siddhi-apps/statistics</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>worker/manager</td>
</tr>
</tbody>
</table>

**Parameter Description**

curl command syntax

```bash
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>
Enabling/disabling the statistics of a specific Siddhi application

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Enables/disables statistics for a specified Siddhi application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/siddhi-apps/{appName}/statistics</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>PUT</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>worker/manager</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>appName</td>
<td>The name of the Siddhi application for which the Siddhi applications need to be enabled/disabled.</td>
</tr>
</tbody>
</table>

curl command syntax

Sample curl command

```
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td>For descriptions of the HTTP status codes, see HTTP Status Codes.</td>
<td></td>
</tr>
</tbody>
</table>

Enabling/disabling the statistics of all Siddhi applications

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Enables/disables statistics for all the Siddhi applications.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/siddhi-apps/statistics</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>PUT</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
</tbody>
</table>
Password | admin  
---|---  
Runtime | worker/manager  

**Parameter Description**

curl command syntax

---

Sample curl command

```
```

Sample output

**Response**

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
</table>
For descriptions of the HTTP status codes, see [HTTP Status Codes](#).

**Event Simulation APIs**

- Sending a single event for simulation  
- Uploading a feed simulation configuration  
- Retrieving all feed simulation configurations  
- Updating a feed simulation configuration  
- Retrieving a specific feed simulation configuration  
- Deleting a feed simulation configuration  
- Running a feed simulation configuration  
- Pausing a feed simulation  
- Resuming a feed simulation  
- Stopping a feed simulation  
- Retrieving a simulation configuration status by name  
- Uploading a CSV file  
- Fetching CSV file names  
- Updating a CSV file  
- Deleting a CSV file  
- Testing a database connection  
- Retrieving database tables  
- Retrieving database table columns

**Sending a single event for simulation**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Sends a single event for simulation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/simulation/single</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
</tbody>
</table>
| Request/Response format | Request format: text/plain  
Response format: application/json |
| Authentication | Basic |
curl command syntax

Sample curl command


Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
</table>
| For descriptions of the HTTP status codes, see **HTTP Status Codes**.

**Uploading a feed simulation configuration**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Uploads a feed simulation configuration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/simulation/feed</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
</tbody>
</table>
| Request/Response format | **Request format**: text/plain  
**Response format**: application/json |
| Authentication | Basic                              |
| Username | admin                                 |
| Password | admin                                 |
| Runtime | • Worker (https port: 9443, http port:9090)  
• Editor |
curl -X POST "http://localhost:9390/simulation/feed" -H "accept: application/json" -H "content-type: text/plain" -d "{"properties":{"simulationName":"TestFeedSimulation","startTimestamp":"1500319950003","endTimestamp":"1500319950009","noOfEvents":"100","description":"Test feed simulator","timeInterval":"1000"},"sources": [{"siddhiAppName":"TestSiddhiApp","streamName":"FooStream","timestampInterval":"1000","simulationType":"CSV_SIMULATION","fileName":"foostream.csv","delimiter":",","isOrdered":true,"indices":"0,1,2"}]"
Updating a feed simulation configuration

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Updates a feed simulation configuration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/simulation/feed/{simulationName}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>PUT</td>
</tr>
<tr>
<td>Request/Response format</td>
<td></td>
</tr>
<tr>
<td>Request format</td>
<td>text/plain</td>
</tr>
<tr>
<td>Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
</tbody>
</table>
| Runtime           | • Worker (https port: 9443, http port:9090)  
|                   | • Editor                                |

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{simulationName}</td>
<td>The name of the simulation configuration that needs to be updated.</td>
</tr>
</tbody>
</table>

curl command syntax

```
curl -X PUT "http://localhost:9390/simulation/feed/TestFeedSimulation" -H "accept: application/json" -H "content-type: text/plain" -d "{"properties":{"simulationName":"TestFeedSimulation","startTimestamp":"","endTimestamp":"","noOfEvents":"100","description":"Test feed simulator","timeInterval":"1000"},"sources":[{"siddhiAppName":"TestSiddhiApp","streamName":"BarStream","timestampInterval":"1000","simulationType":"CSV_SIMULATION","fileName":"foostream.csv","delimiter":"","isOrdered":true,"indices":"0,1,2"}]"
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td>For descriptions of the HTTP status codes, see HTTP Status Codes.</td>
<td></td>
</tr>
</tbody>
</table>

Retrieving a specific feed simulation configuration

Overview
Retrieves a specific feed simulation configuration.

API Context
/simulation/feed/{simulationName}

HTTP Method
GET

Request/Response format
application/json

Authentication
Basic

Username
admin

Password
admin

Runtime
- Worker (https port: 9443, http port:9090)
- Editor

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{simulationName}</td>
<td>The name of the feed simulation configuration that needs to be retrieved.</td>
</tr>
</tbody>
</table>

**curl command syntax**

```
curl -X GET "http://localhost:9390/simulation/feed/TestFeedSimulation" -H "accept: application/json"
```

**Sample output**

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 or 404</td>
<td>For descriptions of the HTTP status codes, see HTTP Status Codes.</td>
</tr>
</tbody>
</table>

**Deleting a feed simulation configuration**

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Deletes a feed simulation configuration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/simulation/feed/{simulationName}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>DELETE</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>- Worker (https port: 9443, http port:9090)</td>
</tr>
<tr>
<td></td>
<td>- Editor</td>
</tr>
</tbody>
</table>
**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{simulationName}</code></td>
<td>The name of the simulation configuration that needs to be deleted.</td>
</tr>
</tbody>
</table>

**curl command syntax**

```bash
curl -X DELETE "http://localhost:9390/simulation/feed/TestFeedSimulation" -H "accept: application/json"
```

**Sample output**

**Response**

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 or 404</td>
<td></td>
</tr>
</tbody>
</table>

For descriptions of the HTTP status codes, see [HTTP Status Codes](#).

---

**Running a feed simulation configuration**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Runs a feed simulation configuration.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>API Context</th>
<th><code>/simulation/feed/{simulationName}?action=run</code></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>HTTP Method</th>
<th>POST</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Request/Response format</th>
<th>application/json</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Authentication</th>
<th>Basic</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Username</th>
<th>admin</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Password</th>
<th>admin</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Worker (https port: 9443, http port:9090)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Editor</td>
</tr>
</tbody>
</table>

---

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{simulationName}</code></td>
<td>The name of the simulation configuration that needs to be run.</td>
</tr>
</tbody>
</table>

**curl command syntax**

```bash
curl -X POST "http://localhost:9390/simulation/feed/{simulationName}?action=run" -H "content-type: application/json"
```

**Sample curl command**

```bash
```
Pausing a feed simulation

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Pauses a currently active feed simulation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td><code>/simulation/feed/{simulationName}?action=pause</code></td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>• Worker (https port: 9443, http port:9090)</td>
</tr>
<tr>
<td></td>
<td>• Editor</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{simulationName}</td>
<td>The name of the simulation configuration that needs to be run.</td>
</tr>
</tbody>
</table>

curl command syntax

curl -X POST "http://localhost:9390/simulation/feed/TestFeedSimulation/?action=pause" -H "accept: application/json"

Sample curl command

curl -X POST "http://localhost:9390/simulation/feed/TestFeedSimulation/?action=pause" -H "accept: application/json"

Sample output

Response
Resuming a feed simulation

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Resumes a paused feed simulation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/simulation/feed/{simulationName}?action=resume</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
</tbody>
</table>
| Runtime           | • Worker (https port: 9443, http port:9090)  
|                   | • Editor                          |

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{simulationName}</td>
<td>The name of the simulation configuration that needs to be resumed.</td>
</tr>
</tbody>
</table>

curl command syntax

```bash
curl -X POST
"http://localhost:9390/simulation/feed/TestFeedSimulation/?action=resume"
-H "accept: application/json"
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200, 403, 404, or 409</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

Stopping a feed simulation

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Stops a currently active feed simulation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/simulation/feed/{simulationName}?action=stop</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
</tbody>
</table>
Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{simulationName}</td>
<td>The name of the simulation configuration that needs to be stopped.</td>
</tr>
</tbody>
</table>

curl command syntax

```
curl -X POST "http://localhost:9390/simulation/feed/TestFeedSimulation/?action=stop" -H "accept: application/json"
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200, 404, or 409</td>
<td>For descriptions of the HTTP status codes, see HTTP Status Codes.</td>
</tr>
</tbody>
</table>

Retrieving a simulation configuration status by name

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Retrieves the status of a given simulation configuration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/simulation/feed/{simulationName}/status</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
</tbody>
</table>
| Runtime | • Worker (https port: 9443, http port:9090)  
  • Editor                                |
**Parameter** | **Description**
---|---
{simulationName} | The name of the simulation configuration of which the status needs to be checked.

**curl command syntax**

```
curl -X POST
"http://localhost:9390/simulation/feed/TestFeedSimulation/?action=resume"
-H "accept: application/json"
```

**Sample curl command**

```
curl -X GET
"http://localhost:9390/simulation/feed/TestFeedSimulation/status" -H
"accept: application/json"
```

**Sample output**

**Response**

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
<td></td>
</tr>
</tbody>
</table>

**Uploading a CSV file**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Uploads a CSV file for feed simulation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/simulation/files</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
</tbody>
</table>
| Runtime | • Worker (https port: 9443, http port:9090)  
• Editor |
HTTP Status Code | 200 or 404  
---|---
For descriptions of the HTTP status codes, see [HTTP Status Codes](#).

### Fetching CSV file names

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Fetches the names of CSV files that are currently uploaded in the system.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/simulation/files</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
</tbody>
</table>
| Runtime | • Worker (https port: 9443, http port:9090)  
• Editor |

**curl command syntax**

```bash
```

**Sample output**

**Response**

| HTTP Status Code | 200 or 404  
---|---
For descriptions of the HTTP status codes, see [HTTP Status Codes](#).

### Updating a CSV file

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Updates a CSV file that is already uploaded in the system.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/simulation/files/{fileName}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>PUT</td>
</tr>
<tr>
<td>Request/Response format</td>
<td></td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
</tbody>
</table>

Runtime

- Worker (https port: 9443, http port:9090)
- Editor

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{fileName}</td>
<td>The name of the CSV file that needs to be updated.</td>
</tr>
</tbody>
</table>

curl command syntax

curl -X PUT -F 'file=@foostream.csv'

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 or 404</td>
<td></td>
</tr>
</tbody>
</table>

For descriptions of the HTTP status codes, see [HTTP Status Codes](#).

Deleting a CSV file

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Deletes the specified CSV file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/simulation/files/{fileName}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>DELETE</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Worker (https port: 9443, http port:9090) Editor</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{fileName}</td>
<td>The name of the CSV file that needs to be deleted.</td>
</tr>
</tbody>
</table>

curl command syntax

curl -X DELETE -F 'file=@foostream.csv'
Sample curl command

```
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

**Testing a database connection**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Tests a database connection.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td><code>/simulation/connectToDatabase</code></td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response format</td>
<td><code>application/json</code></td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
</tbody>
</table>
| Runtime       | • Worker (https port: 9443, http port:9090)  
|               | • Editor                     |

**curl command syntax**

```
```

Sample curl command

```
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

**Retrieving database tables**
## Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Retrieves database tables.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td><code>/simulation/connectToDatabase/retrieveTableNames</code></td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response format</td>
<td><code>application/json</code></td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
</tbody>
</table>
| Runtime | • Worker (https port: 9443, http port:9090)  
• Editor |

### curl command syntax

```
curl -X POST  
"http://localhost:9090/simulation/connectToDatabase/retrieveTableNames" -H  
"accept: application/json" -H "content-type: application/json" -d "{  
"dataSourceLocation": "jdbc:mysql://localhost:3306/DatabaseFeedSimulation",  
"driver": "com.mysql.jdbc.Driver",  
"username": "root",  
"password": "password"}"
```

### Sample curl command

```
curl -X POST  
"http://localhost:9090/simulation/connectToDatabase/retrieveTableNames" -H  
"accept: application/json" -H "content-type: application/json" -d "{  
"dataSourceLocation": "jdbc:mysql://localhost:3306/DatabaseFeedSimulation",  
"driver": "com.mysql.jdbc.Driver",  
"username": "root",  
"password": "password"}"
```

### Sample output

#### Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
</table>

For descriptions of the HTTP status codes, see [HTTP Status Codes](#).

---

## Retrieving database table columns

### Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Retrieves database table columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td><code>/simulation/connectToDatabase/{tableName}/retrieveColumnNames</code></td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response format</td>
<td><code>application/JSON</code></td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
</tbody>
</table>
| Runtime | • Worker (https port: 9443, http port:9090)  
• Editor |
Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{tableName}</td>
<td>The name of the database table of which the columns need to be retrieved.</td>
</tr>
</tbody>
</table>

curl command syntax

Sample curl command

```
curl -X POST
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

Status Monitoring APIs

- Listing all registered workers
- Adding a worker
- Deleting a worker
- Fetching the general details of a worker
- Testing the connection of a worker
- Fetching the history statistics details of a worker
- Fetching the details of all the Siddhi applications of a specific worker
- Getting the text view and the flow of a Siddhi application
- Enabling/disabling Siddhi application statistics
- Fetching the history statistics details of a Siddhi application
- Fetching the component list and the component current metrics
- Fetching the history statistics details of a Siddhi application component
- Fetching the user roles of a specific user
- Reading dashboard configuration details of the currently running dashboard server
- Fetching the HA status

**Listing all registered workers**

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Lists all the workers registered in the Status Dashboard.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/monitoring/apis/workers</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
</tbody>
</table>
### Adding a worker

**Overview**

- **Description**: Adds a worker node to the Status Dashboard.
- **API Context**: /monitoring/apis/workers
- **HTTP Method**: POST
- **Request/Response format**: application/json
- **Authentication**: Basic
- **Username**: admin
- **Password**: admin
- **Runtime**: Dashboard

#### curl command syntax

```bash
  "host": "localhost",
  "port": "9443"
}" -u admin:admin -k
```

#### Sample curl command

```bash
```
Deleting a worker

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Deletes the specified worker from the Status Dashboard.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/monitoring/apis/workers/{workerId}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>DELETE</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{workerId}</td>
<td>The ID of the worker node that needs to be removed from the Status Dashboard.</td>
</tr>
</tbody>
</table>

curl command syntax

curl -X DELETE

Sample output

Response

Fetching the general details of a worker

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Fetches the general details of the specified worker.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/monitoring/apis/workers/{workerId}/system-details</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>
**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{workerId}</td>
<td>The ID of the worker node of which the general details need to be fetched.</td>
</tr>
</tbody>
</table>

**curl command syntax**

Sample curl command

```bash
```

**Sample output**

Response

**Testing the connection of a worker**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Tests the connection of the specified user.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/monitoring/apis/workers/{workerId}/status</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{workerId}</td>
<td>The ID of the worker node of which the connection needs to be tested.</td>
</tr>
</tbody>
</table>

**curl command syntax**

Sample curl command

```bash
```
curl -X POST  
"https://localhost:9643/monitoring/apis/workers/localhost_9444/status" -H  
"accept: application/json" -u admin:admin -k

Sample output

Response

*Fetching the history statistics details of a worker*

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Fetches the history statistics details of the specified worker.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>API Context</strong></td>
<td>/monitoring/apis/workers/{workerId}/history?period='2 hr',type=cpu,memory,_latency</td>
</tr>
<tr>
<td><strong>HTTP Method</strong></td>
<td>GET</td>
</tr>
<tr>
<td><strong>Request/Response format</strong></td>
<td>application/json</td>
</tr>
<tr>
<td><strong>Authentication</strong></td>
<td>Basic</td>
</tr>
<tr>
<td><strong>Username</strong></td>
<td>admin</td>
</tr>
<tr>
<td><strong>Password</strong></td>
<td>admin</td>
</tr>
<tr>
<td><strong>Runtime</strong></td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{workerId}</td>
<td>The ID of the worker node of which the history statistics details need to be fetched.</td>
</tr>
</tbody>
</table>

**curl command syntax**

Sample curl command

```
curl -X GET  
```

Sample output

Response

*Fetching the details of all the Siddhi applications of a specific worker*

**Overview**

| Description | Fetches details relating to all the Siddhi applications of a specific worker. |
API Context | /monitoring/apis/workers/{workerId}/siddhi-apps
---|---
HTTP Method | GET
Request/Response format | application/json
Authentication | Basic
Username | admin
Password | admin
Runtime | Dashboard

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{workerId}</td>
<td>The ID of the worker node for which the details of the Siddhi applications assigned need to be fetched.</td>
</tr>
</tbody>
</table>

curl command syntax


**Sample output**

**Response**

*Getting the text view and the flow of a Siddhi application*

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Fetches the text view and the flow of a specified Siddhi application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/monitoring/apis/workers/{workerId}/siddhi-apps/{appName}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

**Parameter Description**

curl command syntax
Sample curl command

```bash
curl -X GET
```

Sample output

Response

**Enabling/disabling Siddhi application statistics**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Enables/disables statistics for the Siddhi applications of a specified worker node.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/monitoring/apis/workers/{workerId}/siddhi-apps</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>PUT</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{workerId}</code></td>
<td>The ID of the worker node with the Siddhi applications for which statistics need to be enabled/disabled.</td>
</tr>
</tbody>
</table>

**curl command syntax**

```bash
curl -X PUT
```

Sample output

Response
### Fetching the history statistics details of a Siddhi application

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Fetches the history statistics details of a specified Siddhi application.</th>
</tr>
</thead>
</table>

**API Context**

```
/monitoring/apis/workers/{workerId}/siddhi-apps/{appName}/statistics
```

<table>
<thead>
<tr>
<th>HTTP Method</th>
<th>GET</th>
</tr>
</thead>
</table>

| Request/Response format | application/json |

<table>
<thead>
<tr>
<th>Authentication</th>
<th>Basic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{workerId}</code></td>
<td>The ID of the worker node with the Siddhi application of which the history statistics details need to be fetched.</td>
</tr>
<tr>
<td><code>{appName}</code></td>
<td>The name of the Siddhi application of which the history statistics details need to be fetched.</td>
</tr>
</tbody>
</table>

**curl command syntax**

```
curl -X GET 
```

**Sample output**

**Response**

### Fetching the component list and the component current metrics

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Fetches the component list and component current metrics of a specified Siddhi application.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>API Context</th>
</tr>
</thead>
</table>

```
/monitoring/apis/workers/{workerId}/siddhi-apps/{appName}/components
```

<table>
<thead>
<tr>
<th>HTTP Method</th>
<th>GET</th>
</tr>
</thead>
</table>

| Request/Response format | application/json |

<table>
<thead>
<tr>
<th>Authentication</th>
<th>Basic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
</tbody>
</table>

---
Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{workerId}</td>
<td>The ID of the worker node with the Siddhi application of which the component list and component current metrics need to be fetched.</td>
</tr>
<tr>
<td>{appName}</td>
<td>The Siddhi application of which the component list and component current metrics need to be fetched.</td>
</tr>
</tbody>
</table>

curl command syntax

Sample curl command

```bash
```

Sample output

Response

*Fetching the history statistics details of a Siddhi application component*

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Fetches the history statistics details for a specified component of a specified Siddhi application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/monitoring/apis/workers/{workerId}/siddhi-apps/{appName}/components/{componentType}/{componentId}/history?period='2hr',type=cpu,...</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{workerId}</td>
<td>The ID of the worker node with the Siddhi application for which the history statistic details of a specific component needs to be fetched.</td>
</tr>
</tbody>
</table>
The Siddhi application with the component of which history statistics details need to be fetched.

The type of the Siddhi component of which history statistics details need to be fetched. The component type can be queries, streams, tables, windows and aggregations. For more information, see Siddhi Application Overview - Common components of a Siddhi application.

The ID of the Siddhi component of which history statistics details need to be fetched.

curl command syntax

Sample curl command


Sample output

Overview

**Fetching the user roles of a specific user**

<table>
<thead>
<tr>
<th>Description</th>
<th>Fetches the user roles of a specific user.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/monitoring/apis/workers/roles?permissionSuffix=manager</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>


curl command syntax

Sample curl command


Sample output
### Reading dashboard configuration details of the currently running dashboard server

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Reads the dashboard configuration details of the currently running dashboard server.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/monitoring/apis/workers/config</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

**curl command syntax**

Sample curl command

```
```

**Sample output**

### Fetching the HA status

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the status of the given node in the HA cluster in which it is deployed (i.e., whether the node is the active node or the passive node).</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/monitoring/apis/workers/{workerId}/ha-status</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
The ID of the worker node of which the HA status needs to be checked.

curl command syntax


Sample curl command

Response

Dashboard APIs

- List all dashboards
- Search for dashboards
- Create dashboards
- Update dashboards
- Export Dashboards
- Delete dashboards
- List metadata of widgets

List all dashboards

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Lists all the available dashboards in the current SP setup.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/portal/apis/dashboards</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

curl command syntax

Sample curl command


Sample output

Response
Search for dashboards

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Searches for the dashboard with the given ID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/portal/apis/dashboards/{dashboard-id}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

Parameter description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{dashboard-id}</td>
<td>The ID of the dashboard to be fetched.</td>
</tr>
</tbody>
</table>

curl command syntax

Sample curl command


Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200, 404, 500 or 400.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see HTTP Status Codes.</td>
</tr>
</tbody>
</table>

Create dashboards

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Creates a dashboard.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/portal/apis/dashboards</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
</tbody>
</table>
## curl command syntax

**Sample curl command**

```bash
curl -X POST "https://localhost:9643/portal/apis/dashboards" -H "accept: application/json" -H "Content-Type: application/json" -d "{"url":"sampledashboard","name":"Sample Dashboard","description":"This is a sample dashboard","landingPage":"overview","parentId":"1","owner":"test","pages":[]}" -u admin:admin -k
```

**Sample output**

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200, 404, 500 or 400.</td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

## Update dashboards

### Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Updates the specified dashboard.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td><code>/portal/apis/dashboards/{dashboard-id}</code></td>
</tr>
<tr>
<td>HTTP Method</td>
<td>PUT</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td><code>application/json</code></td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

### Parameter description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{dashboard-id}</code></td>
<td>The ID of the dashboard to be updated.</td>
</tr>
</tbody>
</table>

Sample output

Response

HTTP Status Code | 200, 404, 500 or 400. For descriptions of the HTTP status codes, see HTTP Status Codes.
--- | ---

Export Dashboards

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Export the dashboard with the specified URL.</th>
</tr>
</thead>
</table>

API Context

```
/portal/apis/dashboards/{DASHBOARD_URL}/export
```

HTTP Method

POST

Request/Response Format

application/json

Authentication

Basic

Username

admin

Password

admin

Runtime

Dashboard

Parameter description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{DASHBOARD_URL}</td>
<td>The URL of the dashboard to be exported.</td>
</tr>
</tbody>
</table>

curl command syntax

```
curl -X GET -u <USERNAME>:<PASSWORD> -o '<DASHBOARD_ID>.json' '<DASHBOARD_RUNTIME_URL>/portal/apis/dashboards/<DASHBOARD_ID>/export' -k
```

Sample curl command
Sample output

```json
{
    "dashboard": {
        "url": "httpanalytics",
        "owner": "admin",
        "name": "HTTP Analytics",
        "description": "This is a dashboard which is designed to view the HTTP Analytics",
        "landingPage": "home",
        "parentId": "0",
        "pages": [
            {
                "id": "home",
                "name": "Home",
                "content": [
                    {
                        "type": "column",
                        "isClosable": true,
                        "title": "",
                        "content": [
                            {
                                "type": "row",
                                "isClosable": true,
                                "title": "",
                                "height": 18.672606054009275,
                                "content": [
                                    {
                                        "type": "stack",
                                        "isClosable": true,
                                        "title": "",
                                        "height": 50.0,
                                        "width": 39.220055710306404,
                                        "content": [
                                            {
                                                "title": "Request Count Perspective",
                                                "type": "component",
                                                "component": "HTTPAnalyticsRequestCountFilter",
                                                "props": {
                                                    "id": "0fd7222f-09d8-9977-dfd2-df653c2612e5",
                                                    "configs": {
                                                        "pubsub": {
                                                            "types": ["publisher"]
                                                        }
                                                    }
                                                }
                                            }
                                        ]
                                    }
                                ]
                            }
                        ]
                    }
                ]
            }
        ]
    }
}
```
Analytics Request Count Comparison

"config": {
  "id": "8cd2e8db-b6df-cf3d-2866-19f0a9e98920",
  "types": ["subscriber"],
  "publishers": [
    "0fd7222f-09d8-9977-dfd2-df653c2612e5",
    "0fd7222f-09d8-9977-dfd2-df653c2612e5",
    "5decb725-079a-e17f-120c-3b511d92af07",
    "0fd7222f-09d8-9977-dfd2-df653c2612e5",
    "5decb725-079a-e17f-120c-3b511d92af07"
  ],
  "isGenerated": false
},

"widgetID": "HTTPAnalyticsRequestCountComparison"

false,

"show": true },

"lm-react-component"
52.85412262156448,

Analytics Latency Comparison",
"component",
"HTTPAnalyticsLatencyComparison",
"9863e7f7-e0a7-170e-a7a9-e2ab938379c3",

"types": ["subscriber"],

"publishers": [
"0fd7222f-09d8-9977-dfd2-df653c2612e5",
"5decb725-079a-e17f-120c-3b511d92af07",
"0fd7222f-09d8-9977-dfd2-df653c2612e5",
"5decb725-079a-e17f-120c-3b511d92af07",
"0fd7222f-09d8-9977-dfd2-df653c2612e5",
"5decb725-079a-e17f-120c-3b511d92af07"
]

"isGenerated": false

"HTTPRequestAnalyticsLatencyComparison"

false,

"show": true },

"lm-react-component"}
}
47.35729388692177,
"content": [
    {
        "title": "HTTP Analytics Request Count Over Time",
        "component": "HTTPAnalyticsRequestCountOverTime",
        "props": {
            "id": "2cce5a39-17fb-b8f7-7286-0af658d96f3c",
            "configs": {
                "pubsub": {
                    "types": ["subscriber"],
                    "publishers": [
                        "0fd7222f-09d8-9977-dfd2-df653c2612e5",
                        "5decb725-079a-e17f-120c-3b511d92af07"
                    ]
                }
            }
        }
    },
    {
        "isGenerated": false
    }
],
"isClosable": false,
"show": true,
"lm-react-component"
HTTP Analytics Request Statistics

HTTPAnalyticsRequestStatistics

props: {
   id: "08a7c802-3420-d93c-2bb0-3ae5e0923050",
   configs: {
      pubsub: {
         types: ["subscriber"],
         publishers: [
            "0fd7222f-09d8-9977-dfd2-df653c2612e5",
            "5decb725-079a-e17f-120c-3b511d92af07"
         ]
      },
      isGenerated: false
   },
   widgetID: "HTTPAnalyticsRequestStatistics"
},
isClosable: false,
header: {
   show: true
},
componentName: "lm-react-component"

Response Code Analysis

column

Response Code Analysis

column

row

Response Code Analysis
40.98360655737705,

40.98360655737705,

40.98360655737705,

40.98360655737705,

40.98360655737705,

40.98360655737705,

40.98360655737705,

40.98360655737705,

40.98360655737705,

40.98360655737705,
"component",
"HTTPAnalyticsRequestCountComparison",
"props": {
  "id":
  "configs": {
    "pubsub": {
      "types": ["subscriber"],
      "publishers": [
      "998ef264-906d-2f2f-1cbb-562a3307071a",
      "7573c9e6-35a4-c99c-58bb-0c4d749d8e77"
      ],
    }
  },
  "isGenerated": false
},
"widgetID": "HTTPAnalyticsRequestCountComparison"
"isClosable": false,
"show": true },
"lm-react-component"
"type": "column",
"isClosable": true,
"title": "",
"width": 50.0,
"content": [
  {
    "type": "stack",
    "isClosable": true,
    "title": "",
    "width": 50.0,
    "height":
    41.31147540983607,
    "content": [
      {
        "title": "Date Time Range",
        "type":"
"component",
"DateRangePicker",
"props": {
"id": "7573c9e6-35a4-c99c-58bb-0c4d749dde77",
"configs": {
"pubsub": {
"types": ["publisher"]
},
"isGenerated": false
},
"widgetID": "DateRangePicker"
},
"isClosable": false,
"header": {
"show": true
},
"lm-react-component"
],
[
"type": "stack",
"isClosable": true,
"title": "",
"height": 58.68852459016394,
"content": [
{
"title": "HTTP Analytics Request Count Over Time",
"component",
"HTTPAnalyticsRequestCountOverTime",
"props": {
"id": "f0a0f9ef-5d8f-241d-8fdc-be797f43564a",
"configs": {
"pubsub": {
"types": ["subscriber"],
"publishers": [
"998ef264-906d-2f2f-1cbb-562a3307071a",
"7573c9e6-35a4-c99c-58bb-0c4d749dde77"
]
},
"types": ["subscriber"],
"publishers": [
"998ef264-906d-2f2f-1cbb-562a3307071a",
"7573c9e6-35a4-c99c-58bb-0c4d749dde77"
]}}]
"isGenerated": false

"HTTPAnalyticsRequestCountOverTime"

false,

"show": true ),

"lm-react-component"

"header": {

"isClosable": false,

"show": true },

"componentName": "lm-react-component"

"componentName": "lm-react-component"

"isClosable": true,

"title": "HTTP Analytics Request Statistics",

"type": "component",

"component": "HTTPAnalyticsRequestStatistics",

"props": {

"id": "f79dd8e4-b90c-54db-2c77-9368af8fe405",

"configs": {

"pubsub": {

"types": ["subscriber"],

"publishers": ["998ef264-906d-2f2f-1cbb-562a3307071a", "7573c9e6-35a4-c99c-58bb-0c4d749dde77"]

"isGenerated": false

"widgetID": "HTTPAnalyticsRequestStatistics"

"isClosable": false,

"header": { "show": true },

"componentName": "lm-react-component"
"lm-react-component"
        
      ]
    
    ]
  
  ],
  "hasOwnerPermission": false,
  "hasDesignerPermission": false,
  "hasViewerPermission": false
},
  "widgets": {
    "generated": [],
    "custom": [
      "DateRangePicker",
      "HTTPAnalyticsLatencyComparison",
      "HTTPAnalyticsRequestStatistics",
      "HTTPAnalyticsRequestCountOverTime",
      "HTTPAnalyticsResponseCodeFilter",
      "HTTPAnalyticsRequestCountFilter",
      "HTTPAnalyticsRequestCountComparison",
      "HTTPAnalyticsLatencyOverTime"
Delete dashboards

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Deletes the specified dashboard.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/portal/apis/dashboards/{dashboard-id}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>DELETE</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

Parameter description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{dashboard-id}</td>
<td>The ID of the dashboard to be deleted.</td>
</tr>
</tbody>
</table>

curl command syntax

Sample curl command

```
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200, 404, 500 or 400. For descriptions of the HTTP status codes, see HTTP Status Codes.</th>
</tr>
</thead>
</table>

List metadata of widgets

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Lists the metadata of the widgets.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/portal/apis/widgets/</td>
</tr>
</tbody>
</table>
HTTP Method | GET  
Request/Response Format | application/json  
Authentication | Basic  
Username | admin  
Password | admin  
Runtime | Dashboard  

**curl command syntax**

Sample curl command

```bash
```

Sample output

Response

| HTTP Status Code | 200, 404, 500 or 400. | For descriptions of the HTTP status codes, see [HTTP Status Codes](#). |

Authentication APIs

- Log in to a dashboard application
- Log out of the dashboard application
- Redirect URL for login using authorization grant type

**Log in to a dashboard application**

Overview

<table>
<thead>
<tr>
<th>Overview</th>
<th>Logs in to the apps in dashboard runtime such as portal, monitoring or business-rules app.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/login/{appName}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>application/x-www-form-urlencoded</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

**Parameter description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>appName</td>
<td>Path param</td>
<td>The application to which you need to log in.</td>
<td>portal/monitoring/business-rules</td>
</tr>
<tr>
<td>username</td>
<td>Body param</td>
<td>Username for the login</td>
<td></td>
</tr>
<tr>
<td>password</td>
<td>Body param</td>
<td>Password for the login</td>
<td></td>
</tr>
</tbody>
</table>
### grantType

**Body param**

Grant type used for the login

**password/refresh_token**

**authorization_code**

### rememberMe

**Body param**

Whether remember me function enabled

false/true

---

**curl command syntax**

```bash
curl -X POST "https://analytics.wso2.com/login/{appName}" -H "accept: application/json" -H "Content-Type: application/x-www-form-urlencoded" -d "username={username}&password={password}&grantType={grantType}&rememberMe={rememberMe}"
```

**Sample curl command**

```bash
curl -X POST "https://localhost:9643/login/portal" -H "Content-Type: application/x-www-form-urlencoded" -d "username=admin&password=admin&grantType=password"
```

**Sample output**

```json
{"authUser":"admin","pID":"71368eff-cc71-44ef","lID":"a60c1098-3de0-42fb","validityPeriod":3600}
```

**Response**

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

---

**Log out of the dashboard application**

**Overview**

<table>
<thead>
<tr>
<th>Overview</th>
<th>Logs out of the dashboard application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/logout/{appName}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>application/json</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

---

**curl command syntax**

```bash
```
curl -X POST "https://analytics.wso2.com/logout/{appName}" -H "accept: application/json" -H "Authorization: Bearer {access token}"

Sample curl command


Sample output

N/A

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td>For descriptions of the HTTP status codes, see HTTP Status Codes.</td>
<td></td>
</tr>
</tbody>
</table>

Redirect URL for login using authorization grant type

Overview

<table>
<thead>
<tr>
<th>Overview</th>
<th>Redirects URL by the IS in authorization grant type - OAuth2.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/login/callback/{appName}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>JSON</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

Parameter description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{appName}</td>
<td>The application of which the URL needs to be redirected.</td>
</tr>
</tbody>
</table>

curl command syntax

Sample curl command

curl -X GET "https://localhost:9643/login/callback/portal"
Permission APIs

- Add a permission string
- Get permission ID for permission string
- Check whether a specific user role is granted a specific permission
- Delete a permission string
- List roles with a specific permission
- Revoke a specific permission for all roles
- Revoke a specific permission for a specific role

Add a permission string

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Adds a new permission string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/permissions</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard/Worker</td>
</tr>
</tbody>
</table>

**curl command syntax**

```
curl -X POST https://localhost:9443/permissions/ -H 'content-type: application/json' -d '{ "appName":"MON", "permissionString":"MON.manager"}' -k
```

**Sample output**

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
<td></td>
</tr>
</tbody>
</table>
Get permission ID for permission string

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the permission ID for a given permission string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/permissions/app/{appName}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard/Worker</td>
</tr>
</tbody>
</table>

**Parameter description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{appName}</td>
<td></td>
</tr>
</tbody>
</table>

**curl command syntax**

```
curl -X GET https://localhost:9443/permissions/app/MON
```

**Sample output**

```
```

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td>For descriptions of the HTTP status codes, see HTTP Status Codes.</td>
<td></td>
</tr>
</tbody>
</table>

Check whether a specific user role is granted a specific permission

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Checks whether the specified user role is granted a specific permission.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>permissions/auth/{permissionID}/{roleName}</td>
</tr>
</tbody>
</table>
HTTP Method: GET

Request/Response Format

Authentication

Username
Password

Runtime: Dashboard/Worker

Parameter description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{permissionID}</code></td>
<td>The ID of a specific permission. The API checks whether this permission is granted to the specified user role.</td>
</tr>
<tr>
<td><code>{roleName}</code></td>
<td>The ID of a specific user role. The API checks whether this user role is granted the specified permission ID.</td>
</tr>
</tbody>
</table>

curl command syntax

curl -X GET https://localhost:9443/permissions/auth/8dc31fec-8364-3082-9f88-c7ca7d979873/admin

Sample output

Response

HTTP Status Code: 200 or 404

For descriptions of the HTTP status codes, see HTTP Status Codes.

Delete a permission string

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Deletes the specified permission string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td><code>/permissions/{permissionID}</code></td>
</tr>
<tr>
<td>HTTP Method</td>
<td>DELETE</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td></td>
</tr>
<tr>
<td>Authentication</td>
<td></td>
</tr>
<tr>
<td>Username</td>
<td></td>
</tr>
</tbody>
</table>
Password

Runtime: Dashboard/Worker
**Parameter description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{permissionID}</code></td>
<td>The ID of the permission string to be deleted.</td>
</tr>
</tbody>
</table>

**curl command syntax**

```
curl -X DELETE https://localhost:9443/permissions/e9687c6f-b5b2-3216-b3bd-82e7a8e14367
```

**Sample curl command**

```
curl -X DELETE https://localhost:9443/permissions/e9687c6f-b5b2-3216-b3bd-82e7a8e14367
```

**Sample output**

```
Response

HTTP Status Code: 200 or 404
For descriptions of the HTTP status codes, see [HTTP Status Codes](#).
```

**List roles with a specific permission**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Lists the user roles that are currently granted the specified user role.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td><code>/permissions/{permissionsID}/roles</code></td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td></td>
</tr>
<tr>
<td>Authentication</td>
<td></td>
</tr>
<tr>
<td>Username</td>
<td></td>
</tr>
<tr>
<td>Password</td>
<td></td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard/Worker</td>
</tr>
</tbody>
</table>

**Parameter description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{permissionsID}</code></td>
<td>The ID of the permission for which the user roles need to be listed.</td>
</tr>
</tbody>
</table>

**curl command syntax**

```
Sample curl command

curl -X GET
https://localhost:9443/permissions/8dc31fec-8364-3082-9f88-c7ca7d979873/roles

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

**Revoke a specific permission for all roles**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Revokes the specified permission for all the user roles.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/permissions/revoke/{permissionID}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td></td>
</tr>
<tr>
<td>Authentication</td>
<td></td>
</tr>
<tr>
<td>Username</td>
<td></td>
</tr>
<tr>
<td>Password</td>
<td></td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard/Worker</td>
</tr>
</tbody>
</table>

**Parameter description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{permissionID}</td>
<td>The ID of the permission that needs to be revoked for all user roles.</td>
</tr>
</tbody>
</table>

curl command syntax

curl -X POST
https://localhost:9443/permissions/revoke/8dc31fec-8364-3082-9f88-c7ca7d979873

Sample output
Revoke a specific permission for a specific role

Overview

Description
Grants or revokes a permission for the specified user role. The permission is passed as an array in the body of the request.

API Context
/permissions/roles/{roleID}?action=revoke/grant

HTTP Method
POST

Request/Response Format
application/json

Authentication
Username
Password

Runtime
Dashboard/Worker

Parameter description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{roleID}</td>
<td>The ID of the user role for which the permission given in the request body needs to be granted or revoked.</td>
</tr>
</tbody>
</table>

Sample curl command

```
curl -X POST 'https://localhost:9443/permissions/roles/admin?action=revoke' -H 'content-type: application/json' -d '{ "appName":"MON", "permissionString":"MON.manager"}'
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

Business Rules APIs
Lists available business rule instances

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the list of business rule instances that are currently available.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/business-rules/instances</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td></td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

curl command syntax

Sample curl command

```
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see HTTP Status Codes.</td>
</tr>
</tbody>
</table>

Delete business rule with given UUID

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Deletes the business rule with the given UUID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/business-rules/instances/{businessRuleInstanceID}?force-delete=false</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>DELETE</td>
</tr>
</tbody>
</table>
### Request/Response Format

<table>
<thead>
<tr>
<th>Request/Response Format</th>
<th>application/json</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

### Parameter description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{businessRuleInstanceID}</td>
<td>The UUID (Uniquely Identifiable ID) of the business rules instance to be deleted.</td>
</tr>
</tbody>
</table>

### curl command syntax

Sample curl command

```bash
```

### Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

**Fetch template group with the given UUID**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the template group that has the given UUID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/business-rules/template-groups/{templateGroupID}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td></td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>
### Parameter description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{templateGroupID}</td>
<td>The UUID of the template group of which the rule templates need to be fetched.</td>
</tr>
</tbody>
</table>

#### curl command syntax

Sample curl command

```
```

### Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

**Fetch rule templates of the template group with given UUID**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the rule templates of the template group with the given UUID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/business-rules/template-groups/{templateGroupID}/templates</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
</tbody>
</table>

**Request/Response Format**

<table>
<thead>
<tr>
<th>Authentication</th>
<th>Basic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

**Parameter description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{templateGroupID}</td>
<td>The UUID of the template group of which the rule templates need to be fetched.</td>
</tr>
</tbody>
</table>

Sample curl command

Sample curl command
curl -X GET

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
<td></td>
</tr>
</tbody>
</table>

**Fetch rule template of specific UUID available under a template group with specific UUID**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the rule template with the specified UUID that is defined under the template group with the specified UUID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/business-rules/template-groups/{templateGroupID}/templates/{ruleTemplateID}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td></td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

**Parameter description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{templateGroupID}</td>
<td>The UUID of the template group from which the specified rule template needs to be retrieved.</td>
</tr>
<tr>
<td>{ruleTemplateID}</td>
<td>The UUID of the rule template that needs to be retrieved from the specified template group.</td>
</tr>
</tbody>
</table>

**curl command syntax**

Sample curl command

curl -X GET

Sample output
Fetch available template groups

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns all the template groups that are currently available in the SP setup.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/business-rules/template-groups</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>GET</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

curl command syntax


Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see HTTP Status Codes.</td>
</tr>
</tbody>
</table>

Fetch business rule instance with given UUID

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the business rule instance with the given UUID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/business-rules/instances/{businessRuleInstanceID}</td>
</tr>
</tbody>
</table>
HTTP Method | GET
---|---
Request/Response Format | application/json
Authentication | Basic
Username | admin
Password | admin
Runtime | Dashboard

**Parameter description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(businessRuleInstanceID)</td>
<td>The UUID of the business rules instance to be fetched.</td>
</tr>
</tbody>
</table>

**curl command syntax**


**Sample output**

**Response**

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 or 404</td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

**Create and save a business rule**

**Overview**

<table>
<thead>
<tr>
<th>Description</th>
<th>Creates and saves a business rule.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/business-rules /instances?deploy={deploymentStatus}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>
Parameter description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{deploymentStatus}</td>
<td></td>
</tr>
</tbody>
</table>

curl command syntax

-H "accept: application/json" -H "content-type: multipart/form-data" -F 'businessRule={"name":"Business Rule 5","uuid":"business-rule-5","type":"template","templateGroupUUID":"sweet-factory","ruleTemplateUUID":"identifying-continuous-production-decrease","properties":{"timeInterval":"6","timeRangeInput":5","email":"example@email.com"}}' 
-u admin:admin -k

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see <a href="#">HTTP Status Codes</a>.</td>
</tr>
</tbody>
</table>

Update business rules instance with given UUID

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Updates the business rules instance with the given UUID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/business-rules /instances/{businessRuleInstanceID}?deploy={deploymentStatus}</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>PUT</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Dashboard</td>
</tr>
</tbody>
</table>

Parameter description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{businessRuleInstanceID}</td>
<td>The UUID of the business rules instance to be updated.</td>
</tr>
<tr>
<td>{deploymentStatus}</td>
<td></td>
</tr>
</tbody>
</table>
curl command syntax

Sample curl command

```bash
```

Sample output

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>200 or 404</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For descriptions of the HTTP status codes, see HTTP Status Codes.</td>
</tr>
</tbody>
</table>

Store APIs

- Query records in Siddhi store

*Query records in Siddhi store*

Overview

<table>
<thead>
<tr>
<th>Description</th>
<th>Queries records in the Siddhi store. For more information, see Managing Stored Data via REST APIs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Context</td>
<td>/stores/query</td>
</tr>
<tr>
<td>HTTP Method</td>
<td>POST</td>
</tr>
<tr>
<td>Request/Response Format</td>
<td>application/json</td>
</tr>
<tr>
<td>Authentication</td>
<td>Basic</td>
</tr>
<tr>
<td>Username</td>
<td>admin</td>
</tr>
<tr>
<td>Password</td>
<td>admin</td>
</tr>
<tr>
<td>Runtime</td>
<td>Worker</td>
</tr>
</tbody>
</table>
Sample curl command

```bash
curl -X POST https://localhost:7443/stores/query -H "content-type: application/json" -u "admin:admin" -d '{"appName": "AggregationTest", "query": "from stockAggregation select *" }' -k
```

Sample output

```bash
curl -X POST https://localhost:7443/stores/query -H "content-type: application/json" -u "admin:admin" -d '{"appName": "RoomService", "query": "select 10 as roomNumber, 1 as arrival update RoomTypeTable set RoomTypeTable.people = RoomTypeTable.people + arrival on RoomTypeTable.roomNo == roomNumber;" }' -k
```

Response

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 or 404</td>
<td></td>
</tr>
</tbody>
</table>

For descriptions of the HTTP status codes, see [HTTP Status Codes](#).

### HTTP Status Codes

When REST API requests are sent to carryout various actions, various HTTP status codes will be returned based on the state of the action (success or failure) and the HTTP method (POST, GET, PUT, DELETE) executed. The following are the definitions of the various HTTP status codes that are returned.

- **Success HTTP status codes**
- **Error HTTP status codes**

#### Success HTTP status codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Code Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Ok</td>
<td>HTTP request was successful. The output corresponding to the HTTP request will be returned. Generally used as a response to a successful GET and PUT REST API HTTP methods.</td>
</tr>
<tr>
<td>201</td>
<td>Created</td>
<td>HTTP request was successfully processed and a new resource was created. Generally used as a response to a successful POST REST API HTTP method.</td>
</tr>
<tr>
<td>204</td>
<td>No content</td>
<td>HTTP request was successfully processed. No content will be returned. Generally used as a response to a successful DELETE REST API HTTP method.</td>
</tr>
<tr>
<td>202</td>
<td>Accepted</td>
<td>HTTP request was accepted for processing, but the processing has not been completed. This generally occurs when your successful in trying to undeploy an application.</td>
</tr>
</tbody>
</table>

#### Error HTTP status codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Code Summary</th>
<th>Description</th>
</tr>
</thead>
</table>

For descriptions of the HTTP status codes, see [HTTP Status Codes](#).
404  Not found  Requested resource not found. Generally used as a response for unsuccessful GET and PUT REST API HTTP methods.

409  Conflict  Request could not be processed because of conflict in the request. This generally occurs when you are trying to add a resource that already exists. For example, when trying to add an auto-scaling policy that has an already existing ID.

500  Internal server error  Server error occurred.

**Performance Analysis Results**

This section lists performance analysis experiments conducted focusing on stream query processing and event ingestion with persistence.

- **Summary**
- **Stream query processing**
- **Ingesting events with persistence**

**Summary**

The following table shows a complete summary of the content presented in this section.

<table>
<thead>
<tr>
<th>Tested Item</th>
<th>Data Store</th>
<th>Query Type</th>
<th>Amount of events processed</th>
<th>Average Throughput (events per second)</th>
<th>Average Latency (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Pass-through</td>
<td>None</td>
<td>None</td>
<td>30 million</td>
<td>900K</td>
<td>0.9</td>
</tr>
<tr>
<td>Filter</td>
<td>None</td>
<td>Filter out all the events</td>
<td>30 million</td>
<td>900K</td>
<td>1.5</td>
</tr>
<tr>
<td>Window-small (1 second)</td>
<td>None</td>
<td>Sliding time window</td>
<td>30 million</td>
<td>100K</td>
<td>48</td>
</tr>
<tr>
<td>Window - Large (1 minute)</td>
<td>None</td>
<td>Sliding time window</td>
<td>30 million</td>
<td>100K</td>
<td>130</td>
</tr>
<tr>
<td>Patterns</td>
<td>None</td>
<td>Temporal event sequence patterns</td>
<td>1250 million</td>
<td>500K</td>
<td>550</td>
</tr>
<tr>
<td>Event Ingestion with Persistence</td>
<td>Oracle Event Store</td>
<td>Insert</td>
<td>252 million</td>
<td>70K</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update</td>
<td>75 million</td>
<td>20K</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>MS SQL Event Store</td>
<td>Insert</td>
<td>198 million</td>
<td>55K</td>
<td>44.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update</td>
<td>3.6 million</td>
<td>1K</td>
<td>4.6</td>
</tr>
<tr>
<td></td>
<td>MySQL Event Store</td>
<td>Insert</td>
<td>12.2 million</td>
<td>3.4K</td>
<td>2.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update</td>
<td>3 million</td>
<td>500</td>
<td>0.5</td>
</tr>
</tbody>
</table>

**Notes**
- Event ingestion with persistence tests were conducted using the default Amazon RDS configurations.
- All the tests for event ingestion with persistence were conducted for 1 hour.
- Performance results were aggregated in 5000K event windows.
- The above table had an input rate of 1000K events per second during the first four tests.
- All the tests were conducted using TCP transport.

- **Summary**
- **Stream query processing**
  - Scenario: Running multiple Siddhi queries
  - Scenario: Running Siddhi Pattern query on debs-2013-grand-challenge-soccer-monitoring dataset
- **Ingesting events with persistence**
  - Oracle event store
    - Scenario 1: Insert query - Persisting 252 million events of process monitoring events in Oracle
    - Scenario 2: Update Query - Updating 10 million events in Oracle Data store
  - Microsoft SQL server event store
    - Scenario 1: Insert Query - Persisting 198 million process monitoring events in MS SQL
    - Scenario 2: Update Query - Updating 10 million events in MS SQL data store
  - MySQL event store
Scenario 1: Insert Query - Persisting 12.2 million process monitoring events in MySQL
Scenario 2: Update Query - Updating 100K events in MySQL data store

Stream query processing

**TCP Event Publisher** → TCP → **TCP Event Receiver**

**EC2: c4.2xlarge**

**Stream Processor**

**EC2: c4.2xlarge**

**Scenario: Running multiple Siddhi queries**

**Infrastructure used**
- The experiments were carried out in two c4.2xlarge (8 vCPU, 16GB RAM, EBS storage with 1000 Mbps max dedicated bandwidth) Amazon EC2 instances.
- Linux kernel 4.44, java version "1.8.0_131", JVM flags -Xmx4g -Xms2g
- One node operated as a client.
- Another node operated as a Stream Processor node.
- Experiments were carried out using TCP as the transport.

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Sample Query</th>
<th>Amount of Events Processed</th>
<th>Average Throughput (events per second)</th>
<th>Latency (ms)</th>
</tr>
</thead>
</table>
| Simple Passthrough | `@App:name("TCP_Benchmark")
@source(type = 'tcp',
context='inputStream',@map(type='binary'))
define stream inputStream (iijtimestamp long,value float);
from inputStream
select iijtimestamp,value
insert into tempStream;`  | 30 million                                                                 | 900K                      | 0.9                                     |
| Filter          | `@App:name("TCP_Benchmark")
@source(type = 'tcp',
context='inputStream',@map(type='binary'))
define stream inputStream (iijtimestamp long,value float);
from inputStream[value<=1]
select iijtimestamp,value
insert into tempStream;`  | 30 million                                                                 | 900K                      | 1.5                                     |
### Scenario: Running Siddhi Pattern query on debs-2013-grand-challenge-soccer-monitoring dataset

#### Infrastructure used
- The experiments were carried out in two c4.2xlarge (8 vCPU, 16GB RAM, EBS storage with 1000 Mbps max dedicated bandwidth) Amazon EC2 instances.
- Linux kernel 4.4, java version “1.8.0_131”, JVM flags : -Xmx4g -Xms2g
- One node operated as a client.
- Another node operated as a Stream Processor node.
- Experiments were carried out using TCP as the transport.

#### Dataset
The data used in 2013 DEBS Grand Challenge is collected by the Real-Time Locating System deployed on a football field of the Nuremberg Stadium in Germany. Data originates from sensors located near the players’ shoes (1 sensor per leg) and in the ball (1 sensor). The goalkeeper is equipped with two additional sensors in each hand. The sensors in the players’ shoes and hands produce data at a frequency of 200Hz, while the sensor in the ball produces data at a frequency of 2000Hz. The total data rate reaches roughly 15 position events per second. Every position event describes the position of a given sensor in a three-dimensional coordinate system. The center of the playing field is at coordinate (0, 0, 0) for the dimensions of the playing field and the coordinates of the kickoff.

For more details about the dataset, see [DEBS 2013 Grand Challenge: Soccer monitoring](#).

#### Pattern used
We created patterns for goal scoring scenario. In this scenario, the pattern matching query involves two events referred to as $e_1$ and $e_2$ that

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Window</th>
<th>Rows</th>
<th>Throughput</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>@App:name(&quot;TCP_Benchmark&quot;) @source(type = 'tcp', context='inputStream',@map(type='binary')) define stream inputStream (iijtimestamp long,value float); from inputStream[value&lt;=0.5] select iijtimestamp,value insert into tempStream;</td>
<td>30 million</td>
<td>450K</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>@App:name(&quot;TCP_Benchmark&quot;) @source(type = 'tcp', context='inputStream',@map(type='binary')) define stream inputStream (iijtimestamp long,value float); from inputStream[value&lt;=0.25] select iijtimestamp,value insert into tempStream;</td>
<td>30 million</td>
<td>226K</td>
<td>0.6</td>
<td></td>
</tr>
<tr>
<td>@App:name(&quot;TCP_Benchmark&quot;) @source(type='tcp',context='inputStream', @map(type='binary')) define stream inputStream (iijtimestamp long,value float); from inputStream#window.time(1 sec) select iijtimestamp,value insert into tempStream;</td>
<td>30 million</td>
<td>100K</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>@App:name(&quot;TCP_Benchmark&quot;) @source(type='tcp',context='inputStream', @map(type='binary')) define stream inputStream (iijtimestamp long,value float); from inputStream#window.time(1 min) select iijtimestamp,value insert into tempStream;</td>
<td>30 million</td>
<td>100K</td>
<td>130</td>
<td></td>
</tr>
</tbody>
</table>
should occur one after the other with some preconditions being satisfied. These preconditions include the position (denoted by \(x, y, \) and \(z\)) and the acceleration of the ball (denoted by \(a_{\text{abs}}\)). The numerical constants (such as 29880, 22560, etc.) in the sample pattern matching query with which the values for \(x, y, \) and \(z\) are compared correspond to the boundary points of the goal reg.

**Sample Siddhi App**

```siddhi
@App:name("TCP_Benchmark")

@source(type = 'tcp', context='inputStream',@map(type='binary'))
define stream innerStream(iij_timestamp long,sid int, eventtt long, x double, y, double, z int, v_abs double, a_abs int, vx int, vy int, vz int, ax int, ay int, az int);

from e1=innerStream[(x>29880 or x<22560) and y>-33968 and y<33965 and (sid==4 or sid ==12 or sid==10 or sid==8)]
-> e2=innerStream[(x<=29898 and x>22579) and y<=-33968 and z<2440 and a_abs>=55000 and (sid==4 or sid ==12 or sid==10 or sid==8)]

select
    e2.sid as sid, e2.eventtt as eventtt, e2.x as x,e2.y as y, e2.z as z,e2.v_abs as v_abs,e2.a_abs as a_abs, e2.vx as vx,e2.vy as vy, e2.vz as vz, e2.ax as ax,e2.ay as ay, e2.az as az, e1.iij_timestamp as iij_timestamp
insert into outputStream;
```

**Summary Results**

<table>
<thead>
<tr>
<th>Throughput (events per second)</th>
<th>500,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency (ms)</td>
<td>550</td>
</tr>
</tbody>
</table>

**Ingesting events with persistence**

All the event ingestion with persistence tests were conducted using the following deployment configuration.

**Oracle event store**

**Infrastructure used**

The following infrastructure was used in both scenarios:

- c4.2xlarge (8 vCPU, 16GB RAM, EBS storage with 1000 Mbps max dedicated bandwidth) Amazon EC2 instances operated as the SP node and the TCP client.
- Linux kernel 4.44, java version "1.8.0_131", JVM flags : -Xmx4g -Xms2g
- db.m4.2xlarge (8 vCPU, 32 GB RAM, EBS-optimized storage with 100 Mbps max dedicated bandwidth) Amazon RDS instance with Oracle operated as the database node.
Customized TCP client operated as the data publisher (TCP producer found in samples).
Experiments were carried out using Oracle 12g.

**Scenario 1: Insert query - Persisting 252 million events of process monitoring events in Oracle**

This test involved persisting process monitoring events of approximately 180 bytes each. The test injected 252 million events into WSO2 Stream Processor with a publishing TPS of 70,000 events per second during a time period of one hour.

Throughput Graph

`Throughput in this window (events/second)` `Entire throughput for the run (events/second)`

Latency Graph
Summary Results

<table>
<thead>
<tr>
<th>Throughput (events per second)</th>
<th>70,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency (ms)</td>
<td>42</td>
</tr>
</tbody>
</table>

**Scenario 2: Update Query - Updating 10 million events in Oracle Data store**

The test injected 10 million events into the properly indexed Oracle Database. The test involved persisting process monitoring events of approximately 180 bytes each. 75 million update queries were performed. The publishing throughput was 20,000 events per second during a time period of one hour.

Throughput Graph
Latency Graph

Summary Results
Microsoft SQL server event store

Infrastructure used

- c4.2xlarge (8 vCPU, 16GB RAM, EBS storage with 1000 Mbps max dedicated bandwidth) Amazon EC2 instance operated as the SP node.
- Linux kernel 4.44, java version "1.8.0_131", JVM flags : -Xmx4g -Xms2g
- db.m4.2xlarge (8 vCPU, 32 GB RAM, EBS-optimized storage with 100 Mbps max dedicated bandwidth) Amazon RDS instance with MS SQL Enterprise Edition 2016 operated as the database node
- Customized TCP client operated as the data publisher (Sample TCP client found in samples).

Scenario 1: Insert Query - Persisting 198 million process monitoring events in MS SQL

This test involved persisting process monitoring events of approximately 180 bytes each. The test injected 198 million events into WSO2 Stream Processor with a publishing throughput of 55,000 events per second during a time period of one hour.

Throughput Graph

Latency Graph
Summary Results

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (event per second)</td>
<td>55,000</td>
</tr>
<tr>
<td>Latency (ms)</td>
<td>44.2</td>
</tr>
</tbody>
</table>

**Scenario 2: Update Query - Updating 10 million events in MS SQL data store**

This test injected 10 million events into the properly indexed MS SQL Database. This test involved persisting process monitoring events of approximately 180 bytes each. 3.6 million update queries were performed with a publishing throughput of 1000 events per second during a time period of one hour.
Summary Results

<table>
<thead>
<tr>
<th>Number Of Persisted Events</th>
<th>10 million</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (events per second)</td>
<td>1000</td>
</tr>
<tr>
<td>Latency (ms)</td>
<td>4.6</td>
</tr>
</tbody>
</table>

MySQL event store

Infrastructure Used

- c4.2xlarge (8 vCPU, 16GB RAM, EBS storage with 1000 Mbps max dedicated bandwidth) Amazon EC2 instances operated as the SP node and the TCP client.
- Linux kernel 4.44, java version "1.8.0_131", JVM flags : -Xmx4g -Xms2g
- db.m4.2xlarge (8 vCPU, 32 GB RAM, EBS-optimized storage with 100 Mbps max dedicated bandwidth) Amazon RDS instance with MySQL Community Edition version 5.7 operated as the database node.
- Customized TCP client operated as the data publisher (TCP producer found in samples).
- Experiments were carried out using 5.7.19 MySQL Community Server.

Scenario 1: Insert Query - Persisting 12.2 million process monitoring events in MySQL

This test involved persisting process monitoring events of approximately 180 bytes each. The test injected 12.2 million events into WSO2 Stream Processor with a publishing throughput of 3400 events per second during a time period of one hour.

Throughput Graph
Latency Graph

Summary Results
Throughput (events per second) 3400
Latency (ms) 2.14

MySQL Upper Limit
After about 12.2 million events are published, a sudden drop can be observed in the receiver performance. This number can be considered as the upper limit of MySQL event store with default settings. In order to continue receiving events without a major performance degradation, data should be purged periodically from the event store before it reaches the upper limit.

Scenario 2: Update Query - Updating 100K events in MySQL data store
The test injected 100,000 events into the properly indexed MySQL Database. This test involved persisting process monitoring events of approximately 180 bytes each. Three million update queries were performed with a publishing throughput of 500 events per second during a time period of one hour.

Throughput Graph

Latency Graph
Summary Results

<table>
<thead>
<tr>
<th>Number Of Persisted Events</th>
<th>100K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (events per second)</td>
<td>500</td>
</tr>
<tr>
<td>Latency (ms)</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Conclusion

The performance test results indicate that the event persistence performance of WSO2 Stream Processor is characterized by the event store database performance.

Configuring Default Ports

This page describes the default ports that are used for each runtime when the port offset is 0.

- **Common Ports**
- **Worker Runtime**
- **Manager Runtime**
- **Dashboard Runtime**
- **Editor Runtime**
- **Clustering Ports**
  - Distributed deployment:
    - Manager Node:
    - Worker Node(Resource Node):
  - Minimum High Availability (HA) Deployment:
    - Worker Node:
  - Multi Datacenter High Availability Deployment

**Common Ports**

The following ports are common to all runtimes:

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7611</td>
<td>Thrift TCP port to receive events from clients</td>
</tr>
<tr>
<td>Port</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 7711  | Thrift SSL port for secure transport, where the client is authenticated | deployment.yaml:
|       |                                                                  | databridge.config:
|       |                                                                  |   dataReceivers:
|       |                                                                  |     -
|       |                                                                  |       dataReceiver:
|       |                                                                  |         type: Thrift
|       |                                                                  |         properties:
|       |                                                                  |           tcpPort: '7611'
|       |                                                                  |           sslPort: '7711'
| 9611  | Binary TCP port to receive events from clients                   | To offset the default port modify the following configuration in the deployment.yaml:
|       |                                                                  | databridge.config:
|       |                                                                  |   dataReceivers:
|       |                                                                  |     -
|       |                                                                  |       dataReceiver:
|       |                                                                  |         type: Binary
|       |                                                                  |         properties:
|       |                                                                  |           tcpPort: '9611'
|       |                                                                  |           sslPort: '9711'
| 9711  | Binary SSL port for secure transport, where the client is authenticated | |

**Worker Runtime**

- **9090** HTTP netty transport
- **9443** HTTPS netty transport

**Manager Runtime**

- **9190** HTTP netty transport
- **9543** HTTPS netty transport

**Dashboard Runtime**

- **9290** HTTP netty transport
- **9643** HTTPS netty transport

**Editor Runtime**

- **9390** HTTP netty transport
- **9743** HTTPS netty transport

**Note**

To override the default netty port, modify the following properties in the deployment.yaml file (example given for editor runtime):
### Clustering Ports

Ports which are required for clustering deployment:

**Distributed deployment:**

**Manager Node:**

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9190</td>
<td>HTTP netty transport</td>
</tr>
<tr>
<td>9543</td>
<td>HTTPS netty transport</td>
</tr>
<tr>
<td>2181</td>
<td>Zookeeper Server zooKeeperURLs Port, default port where the external zookeeper server start and manager communicate.</td>
</tr>
</tbody>
</table>

**Worker Node (Resource Node):**

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9090</td>
<td>HTTP netty transport</td>
</tr>
<tr>
<td>9191</td>
<td>For the <code>resourceManagers</code> parameter in the <code>cluster.config</code> section, specify the port of the resource manager. (default value taken as http netty transport port of the manager node)</td>
</tr>
<tr>
<td>9443</td>
<td>HTTPS netty transport</td>
</tr>
<tr>
<td></td>
<td>Minimum High Availability (HA) Deployment:</td>
</tr>
</tbody>
</table>

**Worker Node:**

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9090</td>
<td>HTTP netty transport</td>
</tr>
<tr>
<td>9443</td>
<td>HTTPS netty transport</td>
</tr>
</tbody>
</table>

**Multi Datacenter High Availability Deployment**

Other than the ports used in clustering setup (i.e., a Minimum HA Deployment or a Fully Distributed Deployment), the following is required:

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9092</td>
<td>Ports of the two separate instances of the broker deployed in each data center, (ex: `bootstrap.servers='host1:9092, host2:9092', default 9092 where the external kafka servers start)</td>
</tr>
</tbody>
</table>
Working with Data Providers

Data providers are the sources from which information is fetched to be displayed in widgets. This section describes how to configure the data providers that are currently supported in WSO2 Stream Processor as follows. These configurations determine the parameters that are available to be configured for each data provider when creating widgets via the Widget Generation wizard. For more information, see Generating Widgets.

- RDBMS Batch Data Provider
- RDBMS Streaming Data Provider
- Siddhi Store Data Provider
- Web Socket Provider

RDBMS Batch Data Provider

This data provider queries static tables. The following configuration is an example of an RDBMS Batch Data Provider:

```
"config": {
  "datasourceName": "Twitter_Analytics",
  "queryData": {
    "query": "select type as Sentiment, count(TweetID) as Rate from sentiment where PARSEDATETIME(timestamp, 'yyyy-mm-dd hh:mm:ss','en') > CURRENT_TIMESTAMP()-86400 group by type"
  },
  "tableName": "sentiment",
  "incrementalColumn": "Sentiment",
  "publishingInterval": 60
}
```

RDBMS Streaming Data Provider

This data provider queries dynamic tables. Here, the newer records are published as soon as the table is updated. The following configuration is an example of an RDBMS Streaming Data Provider:

```
"configs": {
  "type": "RDBMSStreamingDataProvider",
  "config": {
    "datasourceName": "Twitter_Analytics",
    "queryData": {
      "query": "select id,TweetID from sentiment"
    },
    "tableName": "sentiment",
    "incrementalColumn": "id",
    "publishingInterval": 5,
    "publishingLimit": 5,
    "purgingInterval": 6,
    "purgingLimit": 6,
    "isPurgingEnable": false
  }
}
```

Siddhi Store Data Provider
This data provider runs a siddhi store query. The following configuration is an example of a Siddhi Store Data Provider:

```json
"configs": {
  "type": "SiddhiStoreDataProvider",
  "config": {
    "siddhiApp": "@App:name("HTTPAnalytics") define stream ProcessedRequestsStream(timestamp long, serverName string, serviceName string, serviceMethod string, responseTime double, httpRespGroup string, userAgent string, requestIP string); define aggregation RequestAggregation from ProcessedRequestsStream select serverName, serviceName, serviceMethod, httpRespGroup, count() as numRequests, avg(responseTime) as avgRespTime group by serverName, serviceName, serviceMethod, httpRespGroup aggregate by timestamp every sec...year;",
    "queryData": {
      "query": "from RequestAggregation within "2018-**-** **:**:**" per "days" select AGG_TIMESTAMP, serverName, avg(avgRespTime) as avgRespTime",
    },
    "publishingInterval": 60,
    "timeColumns": "AGG_TIMESTAMP"
  }
}
```

**Web Socket Provider**

This data provider utilizes web siddhi-io-web socket sink to provide data to the clients. It creates endpoints as follows for the web socket sinks to connect and publish information.

```
wss://host:port/websocket-provider/{topic}
```

The host and port will be the host and port of the Portal Web application

The following configuration is an example of a web socket data provider.

```json
{
  configs: {
    type: 'WebSocketProvider',
    config: {
      subscriberTopic: 'sampleStream',
      mapType: 'json'
    }
  }
}
```
Extending WSO2 Stream Processor

The following topics cover the ways in which WSO2 Stream Processor can be extended.

- Supported Extensions
- Writing Custom Siddhi Extensions
- Configuring System Parameters for Siddhi Extensions

Supported Extensions

The siddhi extensions supported by default for WSO2 Stream Processor can be downloaded from the Analytics Extensions Store.

Writing Custom Siddhi Extensions

Custom extensions can be written in order to apply use case specific logic that is not available in Siddhi out of the box or as an existing extension.

There are five types of Siddhi extensions that you can write to cater your specific use cases. These extension archetypes are listed below with their related maven archetypes. You can use these archetypes to generate maven projects for each extension type.

- siddhi-execution
- siddhi-io
- siddhi-map
- siddhi-script
- siddhi-store

siddhi-execution

Siddhi-execution provides following extension types:

- Function
- Aggregate Function
- Stream Function
- Stream Processor
- Window

You can use one or more from above mentioned extension types and implement according to your requirement. For more information about these extension types, see Siddhi Query Guide - Extensions.

To install and implement the siddhi-io extension archetype, follow the procedure below:

1. Issue the following command from your CLI.

   ```
   mvn archetype:generate
   -DarchetypeGroupId=org.wso2.siddhi.extension.archetype
   -DarchetypeArtifactId=siddhi-archetype-execution
   -DarchetypeVersion=1.0.1
   -DgroupId=org.wso2.extension.siddhi.execution
   -Dversion=1.0.0-SNAPSHOT
   ```

2. Enter the required execution name in the message that pops up as shown in the example below.
   Define value for property 'executionType': ML

3. To confirm that all property values are correct, type y in the console. If not, press N.

4. Once you perform the above steps, a skeleton source code is created. You need to update this with the relevant extension logic. Then build the source code and place the build extension jar in the `<SP_HOME>/lib` directory.

siddhi-io

Siddhi-io provides following extension types:

- sink
- source

You can use one or more from above mentioned extension types and implement according to your requirement. siddhi-io is generally used to
work with IO operations as follows:

- The Source extension type gets inputs to your Siddhi application.
- The Sink extension publishes outputs from your Siddhi application.

For more information about these extension types, see Siddhi Query Guide - Extensions.

To implement the siddhi-io extension archetype, follow the procedure below:

1. Issue the following command from your CLI.

   ```
   mvn archetype:generate
   -DarchetypeGroupId=org.wso2.siddhi.extension.archetype
   -DarchetypeArtifactId=siddhi-archetype-io
   -DarchetypeVersion=1.0.1
   -DgroupId=org.wso2.extension.siddhi.io
   -Dversion=1.0.0-SNAPSHOT
   ```

2. Enter the required execution name in the message that pops up as shown in the example below.

   Define value for property 'typeOf_IO': http

3. To confirm that all property values are correct, type y in the console. If not, press n.

4. Once you perform the above steps, a skeleton source code is created. You need to update this with the relevant extension logic. Then build the source code and place the build extension jar in the `<SP_HOME>/lib` directory.

siddhi-map

Siddhi-map provides following extension types,

- Sink Mapper
- Source Mapper

You can use one or more from above mentioned extension types and implement according to your requirement as follows.

- The Source Mapper maps events to a predefined data format (such as XML, JSON, binary, etc), and publishes them to external endpoints (such as E-mail, TCP, Kafka, HTTP, etc).
- The Sink Mapper also maps events to a predefined data format, but it does it at the time of publishing events from a Siddhi application.

For more information about these extension types, see Siddhi Query Guide - Extensions.

To implement the siddhi-map extension archetype, follow the procedure below:

1. Issue the following command from your CLI.

   ```
   mvn archetype:generate
   -DarchetypeGroupId=org.wso2.siddhi.extension.archetype
   -DarchetypeArtifactId=siddhi-archetype-map
   -DarchetypeVersion=1.0.1
   -DgroupId=org.wso2.extension.siddhi.map
   -Dversion=1.0.0-SNAPSHOT
   ```

2. Enter the required execution name in the message that pops up as shown in the example below.

   Define value for property 'typeOf_IO': http

3. To confirm that all property values are correct, type y in the console. If not, press n.

4. Once you perform the above steps, a skeleton source code is created. You need to update this with the relevant extension logic. Then build the source code and place the build extension jar in the `<SP_HOME>/lib` directory.

siddhi-script

Siddhi-script provides the Script extension type.

The script extension type allows you to write functions in other programming languages and execute them within Siddhi queries. Functions
defined via scripts can be accessed in queries similar to any other inbuilt function.

For more information about these extension types, see Siddhi Query Guide - Extensions.

To implement the siddhi-script extension archetype, follow the procedure below:

1. Issue the following command from your CLI.

   ```
   mvn archetype:generate
   -DarchetypeGroupId=org.wso2.siddhi.extension.archetype
   -DarchetypeArtifactId=siddhi-archetype-script
   -DarchetypeVersion=1.0.1
   -DgroupId=org.wso2.extension.siddhi.script
   -Dversion=1.0.0-SNAPSHOT
   ```

2. Enter the required execution name in the message that pops up as shown in the example below.

   Define value for property 'typeOfScript':

3. To confirm that all property values are correct, type Y in the console. If not, press N.

4. Once you perform the above steps, a skeleton source code is created. You need to update this with the relevant extension logic. Then build the source code and place the build extension jar in the `<SP_HOME>/lib` directory.

**siddhi-store**

Siddhi-store provides the Store extension type.

The Store extension type allows you to work with data/events stored in various data stores through the table abstraction.

For more information about these extension types, see Siddhi Query Guide - Extensions.

To implement the siddhi-store extension archetype, follow the procedure below:

1. Issue the following command from your CLI.

   ```
   mvn archetype:generate
   -DarchetypeGroupId=org.wso2.siddhi.extension.archetype
   -DarchetypeArtifactId=siddhi-archetype-store
   -DarchetypeVersion=1.0.1
   -DgroupId=org.wso2.extension.siddhi.store
   -Dversion=1.0.0-SNAPSHOT
   ```

2. Enter the required execution name in the message that pops up as shown in the example below.

   Define value for property 'storeType': RDBMS

3. To confirm that all property values are correct, type Y in the console. If not, press N.

4. Once you perform the above steps, a skeleton source code is created. You need to update this with the relevant extension logic. Then build the source code and place the build extension jar in the `<SP_HOME>/lib` directory.

**Configuring System Parameters for Siddhi Extensions**

The pre-written Siddhi extensions supported by WSO2 Stream Processor are configured with default values for system parameters. If you need to override those values, you can refer to those extensions from the `<SP_HOME>/conf/<RUNTIME>/deployment.yaml` file and add the system parameters with the required values as key-value pairs. To do this, follow the procedure below:

1. Open the `<SP_HOME>/conf/<RUNTIME>/deployment.yaml` file.

   The `<RUNTIME>` can be worker or editor where Siddhi is run.

2. The extensions belong to the Siddhi component. Therefore, to edit the Siddhi component, add a main section to the file named `siddhi`. Then add a subsection named `extensions` to indicate that the configurations related to Siddhi extensions as shown below.
For each separate extension you want to configure, add a sub-section named `extension` under the `extensions` subsection.

Under each `extension` subsection, add two key-value pairs as follows.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the extension. e.g., tcp</td>
</tr>
<tr>
<td>namespace</td>
<td>The archetype of the extension. e.g., source</td>
</tr>
</tbody>
</table>

The archetypes of extensions supported are `source`, `sink`, `execution`, `io`, `map`, `script` and `store`.

Add a subsection named `properties` to override the system properties. Then add the system properties with the required values as key-value pairs. as shown below.

Following are examples for overriding default values for system properties.

**Example 1: Defining host and port for TCP**

```yaml
siddhi:
  extensions:
    - extension:
        name: tcp
        namespace: source
        properties:
          host: 0.0.0.0
          port: 5511
```

**Example 2: Overwriting the default RDBMS configuration**
siddhi:
  extensions:
    - extension:
        name: rdbms
        namespace: store
        properties:
          mysql.batchEnable: true
          mysql.batchSize: 1000
          mysql.indexCreateQuery: "CREATE INDEX {{TABLE_NAME}}_INDEX ON {{TABLE_NAME}} ({{INDEX_COLUMNS}})"
          mysql.recordDeleteQuery: "DELETE FROM {{TABLE_NAME}}{{CONDITION}}"
          mysql.recordExistsQuery: "SELECT 1 FROM {{TABLE_NAME}}{{CONDITION}} LIMIT 1"

FAQ

- What are the core differences between WSO2 Stream Processor and WSO2 Complex Event Processor?
- Can we detect events that have not occurred using Siddhi?

What are the core differences between WSO2 Stream Processor and WSO2 Complex Event Processor?

WSO2 SP can do everything that WSO2 CEP can do. WSO2 SP also has some additional features as explained in the following table.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Sub Attribute</th>
<th>WSO2 Complex Event Processor</th>
<th>WSO2 Stream Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td></td>
<td>WSO2 CEP is powered by Siddhi 3.x.</td>
<td>WSO2 SP is powered by Siddhi 4.x. This version is more stable than Siddhi 3.x with improved performance and bug fixes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WSO2 CEP is based on WSO2 Carbon 4.x.</td>
<td>WSO2 SP is based on WSO2 Carbon 5.x, and therefore it is more lightweight.</td>
</tr>
<tr>
<td>Working in containerized environments</td>
<td></td>
<td>WSO2 CEP encounters certain challenges when working in containerized environments.</td>
<td>WSO2 SP is designed to be container friendly and can carry out native distributed processing.</td>
</tr>
<tr>
<td>Configuration</td>
<td></td>
<td>The main configurations (i.e., configurations that define parts of the event flow such as collecting information, processing logic and publishing results etc.) are done in separate files/UIs.</td>
<td>All the configurations related to an event flow are contained within a single Siddhi application. For more information about configuring the event flow in a single file, see Creating a Siddhi Application.</td>
</tr>
<tr>
<td>Incremental Analysis</td>
<td></td>
<td>WSO2 CEP focuses mainly on real-time processing and does not perform time series aggregations.</td>
<td>WSO2 SP uses the incremental analysis capabilities of Siddhi 4.x to perform time series aggregations without having to integrate with third party platforms such as Apache Spark. Incremental analysis smoothly federates real-time analytics with batch analytics by allowing both forms of analytics to be done in the same message flow. For more information, see Incremental Analysis.</td>
</tr>
<tr>
<td>Distributed Deployment</td>
<td>Scalability</td>
<td>WSO2 CEP is less scalable compared to WSO2 SP due to its limited container-friendliness.</td>
<td>WSO2 SP is highly scalable with its container friendliness.</td>
</tr>
<tr>
<td></td>
<td>Distributed Architecture</td>
<td>The distributed architecture of WSO2 CEP is based on Apache Storm.</td>
<td>The distributed architecture of WSO2 SP is based on Kafka. As a result, the distributed deployment of WSO2 SP is more fault tolerant and it supports exactly-once processing. For more information about distributed deployment of SP, see the Deployment Guide.</td>
</tr>
</tbody>
</table>
| Multi Data Center Deployment | WSO2 CEP does not have built-in support for multi data center deployment. | WSO2 SP has built-in support for multi data center deployment.  
For more information about multi data center deployment, see Multi Datacenter High Availability Deployment. |
|-----------------------------|--------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| Tooling                     | Query editing                                                            | WSO2 CEP Management Console has a UI that supports only query editing.  
WSO2 SP offers a richer query editing tool that also supports auto completion, event simulation and debugging for Siddhi queries and more.  
For more information, see Understanding the Development Environment. |
| Status monitoring            | WSO2 CEP has Carbon metric support to view only JVM analytics.            | The Status Dashboard of WSO2 SP allows you to monitor your SP deployment with a complete set of statistics related to performance, resource consumption etc., of Siddhi applications and JVM.  
For more information, see Monitoring Stream Processor. |
| Business Rules              | WSO2 CEP does not include a feature to allow business users to create rules out of business templates. | WSO2 SP allows you to create business rules from templates.  
For more information, see Working with Business Rules. |

Can we detect events that have not occurred using Siddhi?

This can be achieved via Siddhi. The `not <condition> for <time period>` key word in Siddhi allows you to detect the non-occurrence of events.

For more information about this key word, see Siddhi Query Guide - Logical Patterns.

For a demonstration of how non-occurring events can be detected in a real world user scenario, see Correlating Events for Complex Event Processing - User Scenario 2.