Design Guide for the Kinetic Edge & Fog Processing Module (EFM)

Last Updated: May 9, 2018

Contents

Introduction

Use the Cisco Edge and Fog Processing Module (EFM) in IoT projects to collect telemetry, transform data, and take action on that data. For example, actions can generate alerts, create reports, or display dashboards. The data can also be used in other applications such as machine learning, Enterprise Resource Management, etc.

This document provides examples of how EFM is used and describes the modules and technology used in the EFM solution. This document also summarizes the major steps used to install the EFM components, gather and transform data, present the data, or take action based on the results.

Tip: See the "EFF Whitepaper" for a full technical description of the Edge and Fog Processing Module system.

How to use this document

The design guide is for the technical architect or other person who implements an IoT project using EFM.

The examples in this document can be reproduced in a development environment to highlight the concepts when building and using the EFM. We will simulate connectivity to series of sensor in a refinery, collect streaming telemetry, perform logic, persist data and publish alerts derived from the sensor values. At the same time, we will give technical details of the product to allow for a better understanding of the product functionality and application to a project.

How does the Edge and Fog Processing Module (EFM) fit into the solution?

The EFM is a scalable software system that sits above the packet network. It is core to the Digital Platform for IoT and delivers data to applications. EFM is a high performance distributed computing system for IoT where computing can be performed anywhere where it is needed, including the edge, fog, datacenter or cloud.

The EFM enables the applications used to leverage this data and perform the necessary business outcomes.

Example outputs when using EFM

Operations for manufacturing, oil and gas, energy and many other industries, are highly automated. The sensors and actuators are part of systems that have been in production for many years. Some of the communication connections allow for modern physical interfaces that support data networking, while others use serial interfaces or some sort of front end controller. In many cases the telemetry is not real-time or is simply not available to the rest of the enterprise. By acquiring the data from sensors and enabling the actuators for use in a modern data platform, applications can leverage this data and take actions.

Examples of outputs from this data using EFM include:

- Collect the streaming telemetry and make it available to one or more applications for consumption anywhere in the enterprise
- Store important telemetry in a time series historian database for forensics, analytics, reporting, etc.
- Query data from the historian database
- Create operational dashboards with basic metrics for use by plant operators and management
- Generate reports from historical data and store the reports in the historian database
- Other outputs

Advanced level outputs include:

- Check thresholds and generate alarms for equipment health monitoring
- Compute statistics and visualization with in graphs such as Bollinger Bands
- Integrate with IT systems such as manufacturing process automation (such as SAP)
- Integrate with Machine Learning microservices that can be consumers of the telemetry (for example, exporting data to IBM Watson or others)

EFM also works with custom built dashboards, including Cisco or third-party applications that provide solutions to some common business problems. For example:

- Preventive Maintenance
- Real-time quality detection
- Asset Tracking and maintenance
- Conditioned based maintenance
- Overall Equipment Efficiency (OEE)
- Remote Monitoring
- Personnel Safety

Summary steps to deploy EFM

An EFM project involves two major platforms:

1. The data network that connects the things to the compute devices, routers, switches, host operating systems, etc. Many Cisco design materials are available to assist with this part.
2. The IoT Data Platform, which we call the Edge Fog Module.

Working with data in an EFM project can be broken down into the following basic steps:

1. Understand your Data
2. Move the data where it is needed
3. Transform and enrich data to suit the target
4. Save the data
5. Present the data (in an application or dashboard)

EFM Terminology

The EFM is based upon the Distributed Services Architecture (DSA), an open source platform and development environment for IoT devices and microservices. DSA presumes data heterogeneity, so all telemetry must be normalized into a common format. This abstracts the applications from the specific communication protocol of the devices. DSA also presumes distributed microservices, allowing the deployment of applications anywhere they are needed.

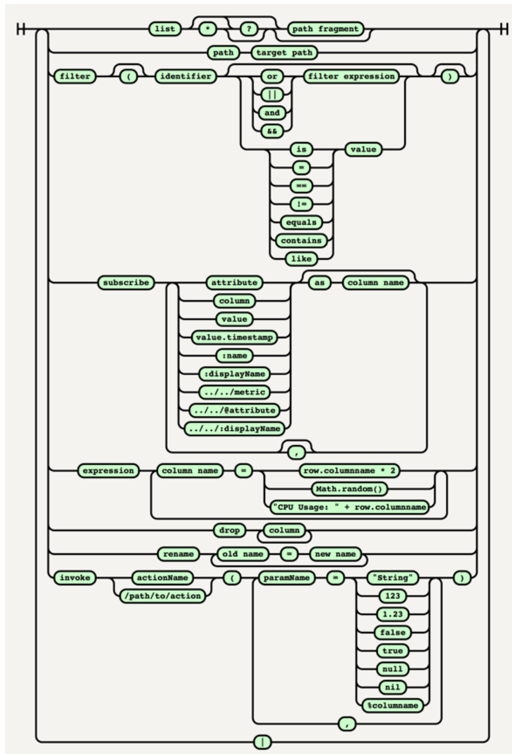
To understand how EFM and DSA work, you need to understand some basic terms:

Table 1. Terminology

Term	Description
Nodes	Everything is a node in the EFM. Nodes have data, expose actions, have a profile and can have children.
EFM Licensed Node	A licensed node is the instance of one or more brokers, links and microservices on a physical or virtual compute platform.
Message Broker or Broker	The broker is a core component to the EFM system. The Broker acts as a message router for incoming and outgoing streams. The Links that are connected to the Broker act as originators of the data streams. All communication between nodes is performed via the message broker, which is based on a publish-subscribe bidirectional message exchange. The broker handles message Quality of Services functions. Connections between brokers form a graph. It also provides introspection capabilities, allowing for a client or application to traverse entire graph and discover all nodes and capabilities. The broker manages subscriptions for listeners; as a consequence, node data is only published through the system if something is subscribed to it. On the publish side: • Updates are trigged via a “set” of a value • The nodes retain the last set value • When there are no subscribers, a set occurs but is not sent to the broker • By default, if the new value is the same as the old, no new message is sent
Broker to Broker communications	To form a scalable and distributed stream processing network the architecture allows for brokers to connect to other brokers. This allows for deployment of brokers, links and microservices anywhere in the system.
Upstream connection	Outbound connections that are created by the initiating node (broke, links or microservice). This enables full-duplex communication traversal of firewalls and use of proxies.
Downstream connection	Inbound connections that are received and accepted by another node (broker,link or microservice. This enables full-duplex communication traversal of firewalls and use of proxies

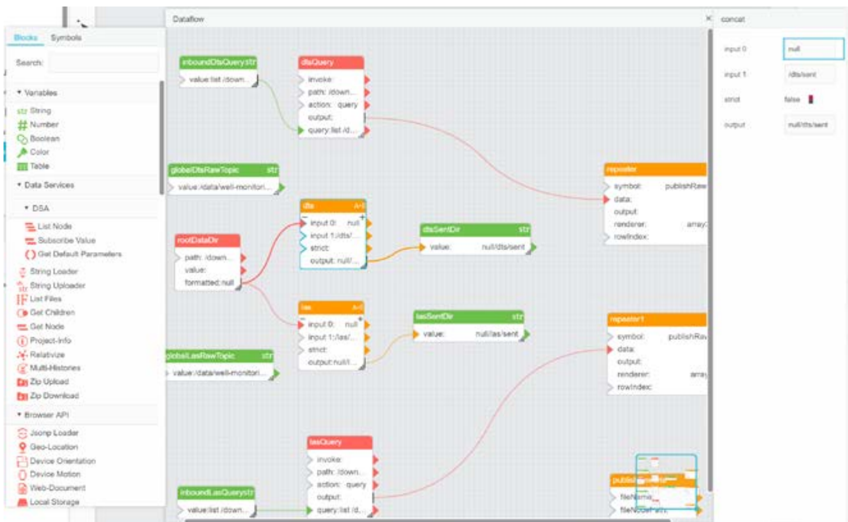
NodeAPI	The nodeAPI is the common communication method for all DSA nodes and facilitates all messaging between entities in a standardized manner. The nodeAPI is responsible for traversing node hierarchies, subscribing to values and streams, and invoking actions on any element within the network. The nodeAPI implements websockets for transport.
DSLink or Link	The link is a domain-specific function that is exposed to the EFM network. The link implements the nodeAPI and enables the microservices. There are 3 types of links: • Device links—Provide connectivity to a specific type of device or protocol (e.g., Modbus, WeMo, etc.) • Bridge links—Enable two way communications with other general-purpose protocols (e.g., MQTT, MTConnect, etc.) • “Engine” links—Contain logic or connect to processes that provide specific functionality (e.g., JDBC, SPARK, Dataflow, etc.)
Quality of Service (QoS) in EFM	The subscription quality of service (QoS) is implemented by the message broker.[1] **Note:** Publishers do not know what QoS will be need and therefore do not determine the QoS level for a particular publication. The QoS values available for EFM 1.5 are: • 0 — **default**. The responder/broker won't cache the value for requester. If the responder's updating speed is faster than the requester's reading speed, the broker only sends the requester the last state, including a rollup of all skipped values. • 1— **queued**. The responder/broker cache values for the requester, but will drop the queue as soon as the requester is disconnected. • 2—**durable**. The responder/broker cache values for the requester, and makes sure it doesn't miss data if the requester's connection is slow, or when requester is offline for a while. • 3— **durable and persist**. Both 1 and 2. The responder/broker will backup the whole cache queue.
EFM Message Broker	A small footprint component working with other brokers to form a message bus. The EFM Message Broker provides reliable and flexible data delivery between devices and micro services. The sources can be devices such as sensors or other micro services. Consumers can be micro services or user applications.

[1] The current definitions can be found at https://github.com/IOT-DSA/docs/wiki/Methods#subscribe

<table>
<tr>
<td>Data Query Language Link</td>
<td>
An engine that provides the capability to subscribe to multiple nodes using a query language. It is a powerful method for selectively subscribing to dynamic collections of nodes. It can filter, transform and invoke actions on nodes as part of the query. Since it subscribes to the EFM messaging system, its output is a continuous query table.
The DQL query language syntax is illustrated below.


For example, using “option traverseBrokers=true | list /data/* | subscribe :name value” as a query returns a table with all nodes and values under the /data path.[2]
</td>
</tr>
</table>

[2] For a more detailed description of the DSA DQL engine querly language, please review https://github.com/IOT-DSA/dslink-dart-dql/blob/master/README.md .

DataFlow Engine	This streaming engine provides event driven data transformation and logic execution capabilities. It is used to build simple to complex algorithms that clean data, build tables, transform, perform mathematical and string functions and can export easily using the built-in functions. It also can be used to trigger and actuate devices as a resulting action. It includes library of “blocks”, which are functions for use in transformations. Flows are created by using the output a functional block as the input to another. It can “glue” together actions from other microservices. A block can subscribe to messages and process them, thereby allowing developers to create “engines” of their own without coding. For example, calculating Bollinger Bands can be performed in a DataFlow. The DataFlow Editor is used to create dataflows. It provides a powerful graphical development environment that not only allows for the management of a dataflow, it allows for block by block output examination for troubleshooting and experimentation.  Creating a DataFlow with the Data Flow Editor graphical editor.
System	This link provides the system metrics of the underlying compute platform or virtual machine (CPU, memory, etc.)

Microservices included with the EFM

ParStream Historian Database - this microservice or application is a high-performance Historian that is used to put the data to rest in a time series oriented data store. It supports structured or unstructured data, and is a critical component in virtually every IoT engagement.

The principal characteristics of the ParStream Historian Database are:

- Small footprint and MPP architecture enables small and large installations
- Patented indexing algorithm for simultaneous high speed ingest and high-speed query
- SQL query support makes data accessible to enterprise data analysts and their tools

Global topic space in the Message Broker

Global topic space in the Broker - the message brokers maintain a separate global topic space on the built-in "/data" node. The /data node allows you to create a canonical data schema to serve as a common publishing space across the system. There are several benefits of using the /data node space:

- Abstracts the subscribers need to understand the device/link specific node hierarchy.
- Serve as a common location for status metrics.
- With the use of DQL, explained earlier, it is possible to query the node namespace and return a dynamic table of nodes and values.
- The last values are stored with the broker, even if the links are no longer operational. The other values are stored in the link.

EFM Use Case: Refinery Simulator

A refinery is a very large operational environment that traditional deploys machines, pumps, valves, storage tanks, gas burners, etc. Because equipment operates continuously during the production cycle, we want to monitor different elements in this environment.

We have created a Refinery Simulator than can be downloaded and installed on the EFM to allow the user implement a basic EFM system and understand the steps we go through to collect, transform, persist and derive outputs to generate alerts.

Our Refinery Simulator streams data on several topics:

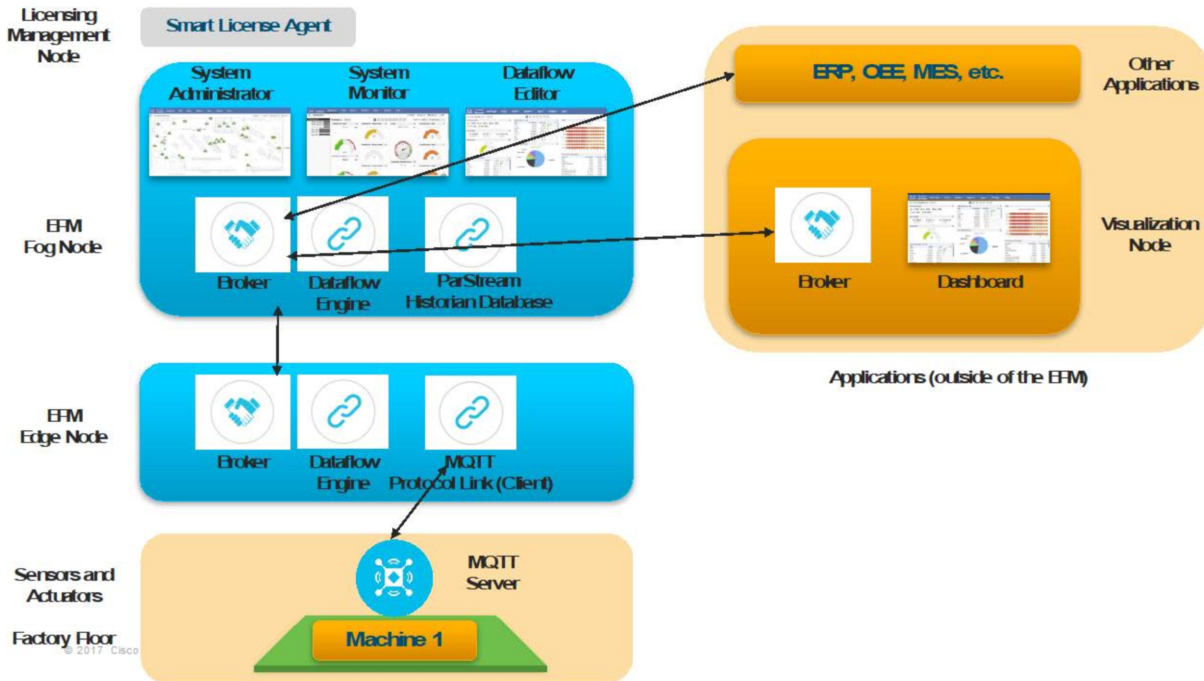
- Mobile gas detectors that are deployed at different locations to monitor toxic gas levels.
- Machine temperature and vibration measurements.
- Valves with their position status (open/closed). Since valves can be actuated, the EFM is a bidirectional communication platform and can change the position-based user input or computational logic.

There are several use cases that this sensor data can allow to solve:

- Preventive Maintenance for machine and other components.
- Safety, to monitor and alert accumulation of toxic gases levels beyond recommend values before they become a personal safety factor or explosive.

For the examples in this document, we will focus on the particular use case of Toxic Gas Levels and generate alerts so a plant operator can take appropriate preventive action. The values used are for educational purposes only and do not reflect real case scenarios.

The following illustration shows the Refinery Simulator allowing for Toxic Gas Level monitoring and alerting



This example can be broken into the following functional blocks:

- Refinery running on the edge node—Where existing equipment is in operation.
- EFM Edge Node—The first compute device that communicates with the factory devices via the simulator micro service protocol link as a client.
- EFM Fog Node—This node stores data, manages and monitors the system. These functions are performed with the EFM System Administrator, System Monitor, DataFlow Engine and Editor, Message Broker and the ParStream Historian Database.
- Licensing Node—This node runs the EFM Smart License Agent and is the only node that needs to communicate to cisco.com for purposes of license validation.

Working with data

Research device communication protocols

Determine the communication protocols that will be used to connect to sensors and actuators.

Start with the most basic concept: we must get the data from the sensors or controllers that interface with these sensors. Many sensors rely on PLCs (Programmable Logic Controllers) for their management. Understanding the communication method and protocol is crucial in allowing the EFM to interface with these sources.

Typical protocols and means of communication in industrial environments are:

- Common Industrial Protocol (CIP)
- MTConnect
- Profinet
- MQTT
- Serial binary streams to legacy equipment

Understand the specific vendor specification

Many of the DSLinks used by the EFM are generic and can be used for a specific protocol while also allowing discovery of the objects used on the device. It is important to understand the communications protocol and how it maps to the data tags to ensure that they are being used correctly.

Understand the sampling frequency

Understand how the sampling frequency might impact the sensor and EFM performance.

The EFM message broker receives data from the DSLink and streams that data to destinations that subscribe to the particular node or object.

The EFM link always preserves the last value until a new value is received, so if a new subscriber requests the value it will always get the latest value in the link. This means that even if many subscribers query the node value, the message broker does not need to re-request it from the sensor.

It is important to understand for each value or set of values:

- How is the value represented and needs to be parsed (for example, a bit map on a serial link or an integer value with a named object)?
- What is the required sampling rate for the specific application (once per second, 20 milliseconds, etc.)?
- What is the sensor's recommended sampling rate? For example, a PLC may require specifying CPU cycles for communications. If too many cycles are assigned for communications, it can degrade its primary function and put operations at risk.
- Understand the valid value ranges to help clean the data later.

Test and verify values for accuracy

It is important to test and verify that telemetry is being received from the sensors or controllers for the following:

- Test for valid inputs. It is possible to have values out of range that need to be cleaned later.
- Verify for accuracy. Verify that the telemetry received is the same value generated by the sensor.

Build a sample system

Select the hardware for each node

We recommend setting up a broker and the communications dslink(s) as close to the sensors as possible. This allows the EFM to:

- Offload data from the sensors and not oversubscribe the sensor or controller hardware
- Perform edge processing
- Stream to subscribers
- Perform QoS for message delivery through the message broker

Before selecting hardware to run the EFM broker, consider the following:

- The environmental requirements for the compute and networking resources (industrialized enclosure, DC power source, Industrial Ethernet, etc.).
- The message rate from the sensors, which will lead to compute platform requirements (see performance recommendation below).
- The physical and logical communications required (for example, Serial Interfaces, Ethernet, Wi-Fi, etc.).
- In addition to collecting the telemetry, will there be a need to do other edge computing?
- What is the compute architecture of the platform? This affects what messaging protocol DSLinks can be supported. See “Installing a messaging protocol DSLink”.
- Where does the equipment need to be located?

See the hardware selection in the following deployment example:

	Edge Node	**Fog Node**
Requirements	Support for: • The message broker • The Data Flow Engine and Editor (requires DART SDK)	Support for: • The message broker • The Data Flow Engine and Editor (requires DART SDK) for data store to the Historian Database • The ParStream Historian Database and ParStream DSLink (Java SDK) • Large disk • Rack mountable off the plant floor

		• Support for the System Administrator and System Monitor
Options	Because of the JAVA SDK and DART SDK requirement, must be an x86_64 platform - IR809, IR829 or UCS Server	• Because of the JAVA SDK and DART SDK requirement, must be an x86_64 platform • System Administrator and System Monitor require Linux platform • ParStream Historian Database require Linux platform
Hardware Selected	IR Virtual Machine on a UCS 220 server with 1 Core CPU, 2 GB RAM, 50GB disk is ideal for the fog node	Virtual Machine on a UCS 220 server with 2 Core CPU, 4 GB RAM, 2TB disk is ideal for the fog node.[3]

	Smart License Agent Node
Requirements	• Linux or Windows • Support for Java 1.8 • Internet connectivity required, optionally can use a connection to a Cisco Licensing Satellite Agent that serves as a proxy.
Hardware Selected	Virtual Machine with 1 core, 1GB RAM, with RedHat Linux. Internet connectivity assumed for the example.

For example, 3 virtual machines can run on a single compute platform. It is common that real deployments will involve nodes to be geographically dispersed.

Dashboard Node, Smart License Agent, Edge and Fog Node:

- Edge Node: 10.88.24.151
- Fog Node: 10.88.24.150
- Smart License Agent: 10.88.24.152

[3] This is a basic historian database configuration, See the release notes for a recommended configuration.

Install the main EFM Broker System Components

This section includes basic steps to install the EFM on the Linux platform. The administrator needs to select the Linux operating system variant, create accounts and configure the networking basics beyond the scope of this document.

Note: See to the EFM Linux Installation Guide for more detailed information, including instructions to configure the networking options.

1. Create a non-root Linux account “efm” and password for installing and operating the edge, fog and licensing nodes. The *adduser* command creates a new user “efm” and a new group “efm”.

   Redhat 7.2/Centos 7.2

   ```
   $ sudo adduser efm -m -p <password>
   ```

   Ubuntu 16.04

   ```
   $ sudo adduser efm
   Enter new UNIX password: <password>
   Retype new UNIX passoword: <password>
   Enter the new value, or press ENTER for the default
           Full Name []: efm
           Room Number []:
           Work Phone []:
           Home Phone []:
           Other []:
   ```

2. Create the EFM root installation directory (all operating systems).

   a. As the host administrator (sudo), create the EFM root installation directory. Unless otherwise defined, this will be /opt/cisco. Also change the owner and group to “efm”.

      i. `$ sudo mkdir /opt/cisco; sudo chown efm /opt/cisco; sudo chgrp efm /opt/cisco`

   b. Log in as user “eff” from the current user.

      ```
      $ su eff
      ```

3. Download the EFM software to the efm home directory.

   a. Go to the https://www.cisco.com/c/en/us/support/cloud-systems-management/edge-fog-fabric/products-user-guide-list.html.

   b. Click **Download Software**.

   c. Download the release for your deployment.

   d. Unzip the image

      ```
      $ unzip EFM_1.1.zip
      ```

   e. Change into unzipped folder

      ```
      $ cd EFM-1.1
      ```

4. Install the EFM Smart License Agent on the Licensing Node.

   ```
   ./efm-linux license
   ```

The Smart Licensing Tool can be found in the /bin directory of the product instance. For example, the directory path for a default install is /opt/cisco/iotdc/efm_licensing/bin.

To start the smart licensing tool, enter the following:

```
# /opt/cisco/iotdc/efm_license/bin/efm-device-licensing.sh
```

To start the smart licensing tool for NODES type the following:

```
# /opt/cisco/iotdc/efm_license/bin/efm-node-licensing.sh
```

Refer to the [Cisco Data Connect EFM Smart Licensing Tool User Guide]() for more information.

Install the EFM components on the fog node (excluding the Licensing Agent)

1. Install the EFM Message Broker and the DQL, System, ParStream and Dataflow engine DSLinks.
   In this example, no global variables are set, all defaults are used.

   ```
   ./efm-linux broker
   ```

2. Install the System Administrator:

   ```
   ./efm-linux admin
   ```

3. Install the System Monitor

   ```
   ./efm-linux standalone_monitor
   ```

4. Install the ParStream Historian Database:

   ```
   ./efm-linux parstream
   ```

5. Set the environment PARSTREAM_HOME, LD_LIBRARY_PATH and PATH as suggested by the install script:

   ```
   export PARSTREAM_HOME=/opt/cisco/iotdc/parstream
   export LD_LIBRARY_PATH=$PARSTREAM_HOME/lib:$LD_LIBRARY_PATH
   export PATH=$PATH:$PARSTREAM_HOME/bin
   ```

6. Configure the EFM Message Broker parameters.

   The /opt/cisco/iotdc/efm_server/server.json file defines the operating parameters for the combined message broker and the web browser on Linux and Windows. The message browser is built in to display the Dataflow Editor, System Monitor and System Administrator.

   The documentation discusses the parameters that can be defined. Make sure that the following values are set for this guide.

   - Use ports 8080 and 8443 for the “efm” user so sudo privileges are not required.
   - The certPassword parameter must be a non-null value (shown with a “*” in this example).
   - If the Internet is not available during the broker operation, then the parameter isAlwaysOffline should be set to “true” for proper operation. Otherwise use the default value “false”.

   For example:

   ```
    "port": 8080,
   "httpsPort": 8443,
   "certPassword": "*",
   "isAlwaysOffline": false,
   ```


Start and stop the EFM Message Broker on the fog and edge nodes

This step starts the EFM Message Broker, EFM Data Flow Engine and Editor and the DSLinks that are installed.

Start

```
/opt/cisco/iotdc/efm_server/bin/daemon.sh start
```

Stop

```
/opt/cisco/iotdc/efm_server/bin/daemon.sh stop
```

Start the EFM ParStream Historian Database

Start an instance of the ParStream Historian Database on the FOG node as single instance in a cluster. Rather than defining a standalone instance, define a single instance of a cluster that allows us to add additional cluster nodes later without having to stop the Historian Database. Otherwise, it is necessary to stop and start again, which can impact the operation of the EFM.

Create the folders for the parstream instance and copy the configuration files used to define the tables.

1. Download the files parstream.ini, alertHistory.sql and sensorTelemtry.sql from this location (under the EFM Training Videos).

2. Log in as the efm user and follow these steps to create the location to store the configurations and the data files for the database instance:

   ```
   $ sudo mkdir /data; sudo chown efm /data; sudo chgrp efm /data
   $ mkdir /data
   $ mkdir /data/training
   $ mkdir /data/training/conf
   $ cp parstream.ini /data/training/conf
   $ mkdir /data/training/sql
   $ cp *.sql /data/training/sql
   ```

   Note: We will use the *.sql files in a later module.

   Note: The parameter userAuthentication is set to “false” in the parstream.ini file for this guide. When we later define the ParStream DSLink we will use dummy values.

3. Start the ParStream instance “parstream1” in this guide.

   ```
   $ cd /data/training (If not defined already the next 3 lines)
   $ export PARSTREAM_HOME=/opt/cisco/iotdc/parstream
   $ export LD_LIBRARY_PATH=$PARSTREAM_HOME/lib:$LD_LIBRARY_PATH
   $ export PATH=$PATH:$PARSTREAM_HOME/bin
   $ nohup /opt/cisco/iotdc/parstream/bin/parstream-server parstream1 &
   $ ps -ef | grep parstream (to verify that parstream is running)
   ```

Tip: For more information about configuring, starting and stopping ParStream Historian Database clusters refer to the documentation found in the "`EFM-1.0/Docs/ParStream 4.4.3`" folder.

Broker to Broker communications

Install the EFM Components on the edge node

The edge node is installed on Linus (as with the fog node).

The process repeats the steps for installation on the fog node, excluding the System Administrator, System Monitor, Licensing Agent and ParStream Historian Database.

Create a broker to broker connection between the fog node and the edge node

Use the Fog Node System Administrator to create a broker to broker connection between the Fog node and the Edge Node.

1. Connect to the System Administrator installed on the fog node.
2. Enter the following:
   https://10.88.24.151:8443/efm-admin
3. For a new messaging connection, enter minimum a of the following 3 settings:
   - The name of the upstream broker that will be used on this connection (broker names are not global).
   - The URL of the message broker connection that includes the IP address, port and selection of the SSL type. For example, https://10.88.24.151:8443/conn has an IP address of 10.88.24.151, port 8443 and uses https as a websockets transport.
   - The name of the local broker in this connection.

For example, enter the following:

- Name (Remote Broker): EdgeNode
- https://10.88.24.151:443/conn
- Broker Name (This Broker): FogNode

Note: Names only are local to the messaging broker pairs, but new connections must not reuse the same name pair.

Install the Refinery Simulator DSLink (microservice) on the edge node

DSLinks are microservices for which there are 3 general categories:

- Device Links — connect to a specific device protocol. (e.g. Modbus)
- Bridge Links —connect to non-EFM broker brokers (e.g. MQTT), usually bi-directional allowing to publish and subscribe
- Logical Links —perform some processing task on data

Links expose data into a node hierarchy. Links also expose specific actions that can be performed and are presented by the link. Actions that are exposed are Remote Procedure Call (RPC) style, allowing for the DSLink to be placed anywhere and distributed the functionality

For a list of open source DSA links that are available for the ERM, refer to the URL at https://iot-dsa.github.io/links/web/status/ .

DSLinks are built using the DSA SDK, which the EFM is based upon. The SDKs are open sourced and available at http://iot-dsa.org/.

The Refinery Simulator DSLink (microservice) explained

The refinery simulator DSLink simulates sensors/devices for the following three use cases. The simulator instantiates an underlying grid (by default 1000x1000) on top of which it overlays different things to support the three use cases. These are configurable by the parameters for the "create simulator" action.

- **Gas detection**. You can create a configurable number of mobile gas detectors. These measure 4 toxic gas types - H2S, O2, CO and LEL (which is a combination of other toxic gases). Location of the sensor is a separate stream from the gas levels. It is modeled that way because in the actual refinery, the gas detectors that are worn by the employees provide gas level data via OPC over Wi-Fi and we get their position data through Wi-Fi triangulation using Cisco MSE. The sensors will take a randomized walk around the refinery once they start up. They are programmed to have a higher probability of continuing in the same direction for simplicity.

  You can create a simulated gas leak through "Gas Leak" node and associated action. Gas leaks start at the specified coordinates, then grow ("bloom") to the maximum bloom radius while increasing to the given max intensity. The leak then falls off (shrinks) before ending. If any of the mobile gas detectors pass through a simulated leak, you will see the elevated gas levels on the detector nodes.

- **Equipment Maintenance**. The simulator creates a fixed number of "machines" each with a collection of sensors. There are three types of sensors that can be on the machines: pressure, temperature and vibration (which measures both velocity and acceleration). The fixed machines (combination of sensors per machine) are modeled directly on the production refinery, so they cannot be changed. The values the sensors provide move within certain ranges. The pressure sensor simulations are implemented very specifically to simulate a series of pumps that increase the pressure of the liquid stream in a chain. All the values will stay within the aforementioned ranges, so it won't really provide much opportunity for alerting.

- **Valve Configuration**. You can create a configurable number of valves. These simulate the position of valves located throughout the refinery. There is an action on the valve to change its position. These do not change on their own (just like in the real refinery).

Install the Refinery Simulator

Download the Refinery Simulator DSLink from the following folder (it is not included in the EFM software distribution):

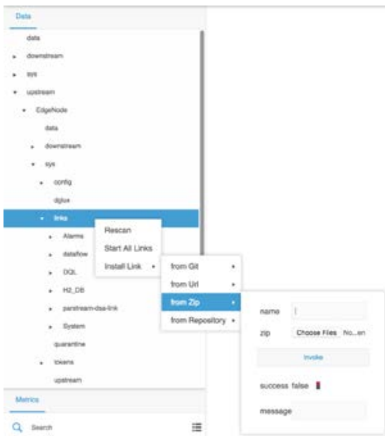
1. Go to the Box folder: https://www.cisco.com/c/en/us/support/cloud-systems-management/edge-fog-fabric/tsd-products-support-series-home.html

2. Open the folder: **EFM Training Videos > Simulator DSLInks**.

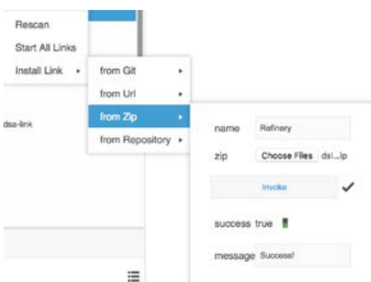
3. Download the file dslink-java-refinery-simulator-0.0.1-SNAPSHOT.zip.

We will use the Dataflow Editor to show another way of installing the edge node, but it can also be installed in the System Administrator.

1. Connect to the Fog Node using a browser.
2. Navigate to **upstream > EdgeNode > sys**.
3. Right click and select **Install Link > from Zip**.



4. In the Name field, type any value (such as "Refinery").
5. Choose the **dslink-java-refinery-simulator-0.0.1-SNAPSHOT.zip** file from your local hard drive.
6. Click **Invoke** and the success indicator should change to "true".[4]
7. Rescan to view the newly installed link:
   a. On the same sys node, right click on **Rescan**.
   b. The new link "Refinery" should appear.



8. Restart the link:
   a. Right click on the **Refinery** link node and select **Start Link**.
   b. In the Metric pane, the link should show: enable: true and status: connected.
9. Verify that the installation was successful:

[4] When installing from a Zip file the broker does not rescan automatically to show a newly installed link. When installing from a repository, this is performed automatically.

a. In the Data Pane, under **upstream > EdgeNode > downstream** a new node “Refinery-Simulator” should appear.

b. Right click the node to view the available actions.

Configure and start the Refinery Simulator

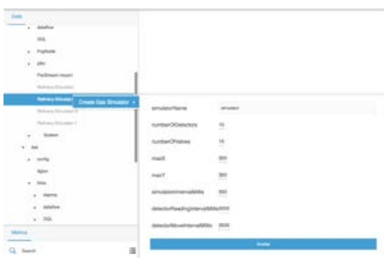
We need to define the size of our simulation.

1. Select **upstream > EdgeNode > downstream**.

2. Right click **Refinery-Simulator** to expose the action “Create Gas Simulator”.

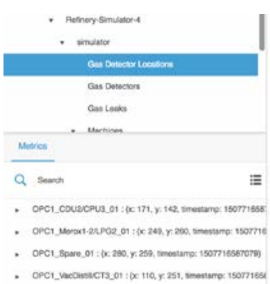
3. Change simulatorName to “simulator”.

4. Change maxX and maxY to 300 and 300 respectively.
Note: Change these values since the default values can be very CPU intensive and are not necessary for this exercise.



5. Under the Refinery-Simulator node, right-click the **simulator** node and select **Start** to start the simulator.

6. Verify that the simulator is properly generating streaming data by selecting each node of the Refinery Simulator in the Data pane. For example:



Use the EFM Dataflow Editor for data investigation

Once telemetry is available via a dslink or a microservice we can use the Dataflow Editor to explore. Remember that everything is a node in the **Data** pane. DSLinks and microservices expose the data into a node hierarchy, publishing the values as children under the path.

1. Open the dataflow editor on the fog node using a web browser on Linux and Windows.
2. Enter http://fog_node_ip_address:8443/dataflow.html.
   For example: http://10.88.24.150:8443/dataflow.html.
3. The browser opens the Dataflow Editor and the Data pane shows a tree structure with node hierarchy.
4. Select a node to display one or more metrics in the Metric pane. The following example shows the Dataflow editor, the data pane and the Metrics pane.

5. Go to the Refinery-Simulator node under **upstream > EdgeNode > downstream > Refinery-Simulator > simulator**.
6. Select from a list of Gas Detector Locations, Gas Detectors, Gas Leaks, Machines and Valves.
   Tip: Some of these nodes expand even more.
   Note: The values in the Metric pane are continuously updated from the streamed source.

The data in the Refinery Simulator appears to be cleaned and normalized. But it is common practice to use an initial dataflow to subscribe to sensor data that we receive from protocol links, clean and then publish into a data structure that can be used for other subscribers in a common fashion.

The Refinery Simulator has organized the data structure tree as follows, where the simulator is at the top.

simulator

- Gas Detector Locations (x,y coordinates of Gas Detectors at any given time)
- Gas Detectors (sensor values of the gas levels at a given time)
- Machines
- Valves

The Gas Detector and Gas Detector Locations are JSON tables. But if you click on the node, as shown above, the table expands to show each row as a discrete node that can be used separately.

Transform the Data

Use the EFM Dataflow editor to create a series of data transformations to create different outcomes, which are common examples is many IoT projects. Our end goal is to create a series of dataflows that will monitor all the Refinery gas sensors and alert us if the H2S (Hydrogen Sulfide) levels exceed certain values. These examples are simple and functional, as in real projects, and build upon each other to create more complex logic. We will also persist the data in the historian database using two different available methods and then query the database for historical data that can be used in a dataflow.

The examples are:

1. Create Dataflows to create a merged stream
2. Create an Alert from the published combinedStream
3. Store data using dataflow to a Historian Watch Group
4. Store data using dataflow to a Defined Schema in the historian database
5. Build a dataflow to query data from the historian database

In most projects, start by ingesting raw data from sensors, and then cleaning and putting the data into a canonical form that has a common name format and value type. As a best practice, we publish the newly derived metrics in the /data path under each broker. As we build more advanced dataflows and queries, we can create continuous subscriptions to metrics in the /data path using wildcards rather than explicitly creating the logic for each specific node, making our logic better for scaling across many sensors.

For example, in the following data hierarchy, each Refinery broker is connected from the Fog Node and appears upstream. The brokers are named Refinery1, Refinery2, Refinery3, etc. until Refinery9. While each Refinery node might have categories for GasDetectors, Machines, Valves, others might have more categories. But the important concept is that it is consistent and easy to understand and navigate, which allows us to search using a pattern.

For example, searching with the path pattern **/upstream/*/data/Machines/*** returns all the nodes for all the machines in all refineries. From the Fog Node, the data hierarchy would look like the following:

/upstream

/Refinery1/data/GasDetectors

/Refinery1/data/Machines

/Refinery2/data/GasDetectors

/Refinery2/data/Machines

...

/Refinery9/data/GasDetectors

/Refinery9/data/Machines

/Refinery9/data/Valves

Data Flow Editor

The Dataflow Editor is a visual data manipulation environment. To build logic, use predefined blocks and bind them together using the outputs of one block to the input of the next block.

Dataflow is event driven: a change in one block that modifies the output will trigger the next block to update.

The Dataflow engine is a microservice that executes the logic defined in the Dataflow Editor. The dataflows are persisted to disk and will continue to function after a restart.

Create Dataflows to create a merged stream

We are going to create two dataflows to achieve our outcome. For now, we will use this dataflow with the metrics from a single gas detector.

We are going to show a basic dataflow **ex1-streamMerge** that:

- Subscribes to two different but related "streams"
- Merges those streams
- Uses a conditional (if)
- Creates a "complex" metric
- Publishes the resulting combined stream

Then we create another dataflow **ex2-combinedAlert** that

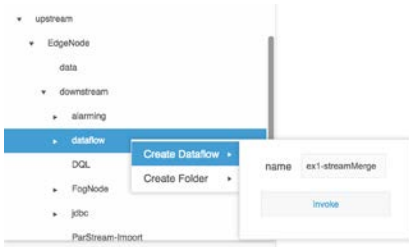
- Subscribes to the combined stream and create a rule for an alert

While it is possible to put all the functionality into a single dataflow, the best practice is to separate the dataflow into functional tasks. This method makes it easier to build the dataflow, troubleshoot and with some experience create reusable dataflows for different projects.

Create a Dataflow to merge related objects

Create a new dataflow named **ex1-streamMerge** in the Dataflow editor on the Edge Node (continue to assume we are connecting from the Fog Node).

For example, create a new Dataflow under upstream/EdgeNode/downstream/dataflow on the edge node called "Example1".

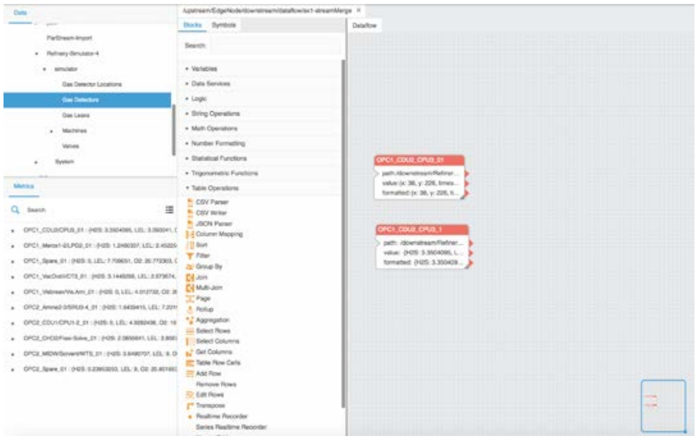


Select the  symbol, and the canvas will open so we can start to populate the logic.

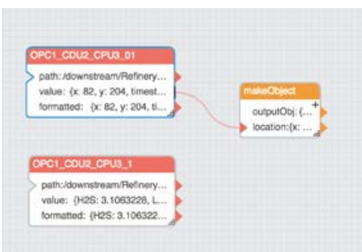
Now find the metrics we are interested in transforming. Go to the Refinery Simulator and under simulator there are 2 metrics: Gas Detector Locations and the Gas Detectors. What we really want to see is the gas levels at a specific location at any given time. In order to accomplish this, we will merge two specific metrics Gas Detector Location OPC1_CDU2/CPU3_01 and Gas Detectors OPC1_CDU2/CPU3_01, each representing values as JSON tables, together.

Right click the **Gas Detector Location OPC1_CDU2/CPU3_01** in the Metrics pane and drag it into the canvas. Do the same for **Gas Detectors OPC1_CDU2/CPU3_01**.

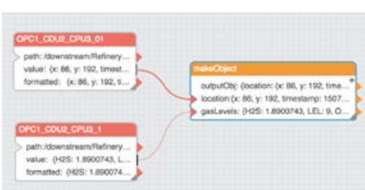
Note that the values inside the Dataflow blocks are updated at the same time the values stream in Metrics pane. If we look at the values, we can see that they show the x, y location and timestamp for one and the other has HS2, LEL, O2, CO gas levels and the timestamp. These timestamps can be different since they are streamed differently. We have now subscribed to the two streams as input into the Dataflow and the following should appear.



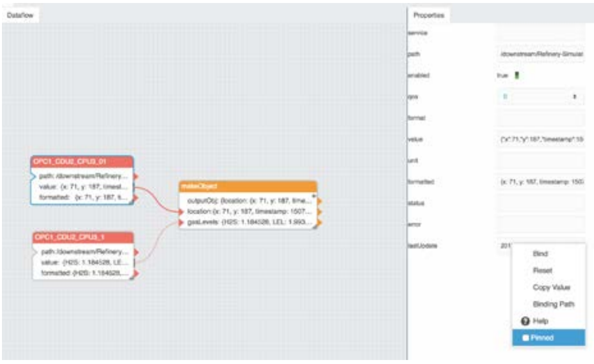
Since we are going to merge or combine the 2 objects that are JSON objects, the Dataflow editor has a block, under Logic, called “makeObject”. We are going to create a composite JSON and to make it work drag over the “makeObject” and click the “+” to allow for additional parameters (for input). Let us call the first one “location”, since it is JSON, set to type “dynamic”. Let us bind the location from the Gas Detector Location OPC1_CDU2/CPU3_01 by right clicking the mouse and dragging the value to the input of location as shown below:



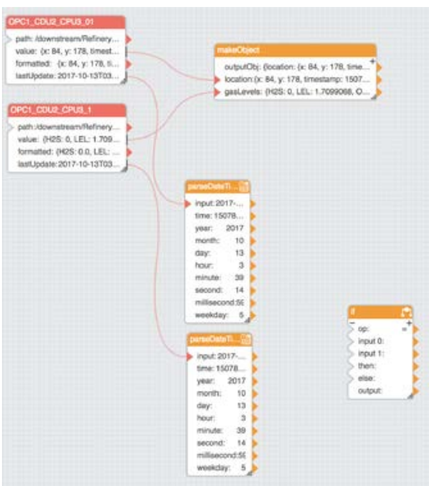
Now add another parameter to the “makeObject” block by clicking on the “**+**” again. This will be called “gasLevels” and set the type “dynamic”. Bind the location from the **Gas Detectors OPC1_CDU2/CPU3_**01 by right clicking the mouse and dragging the value to the input of location as shown below:



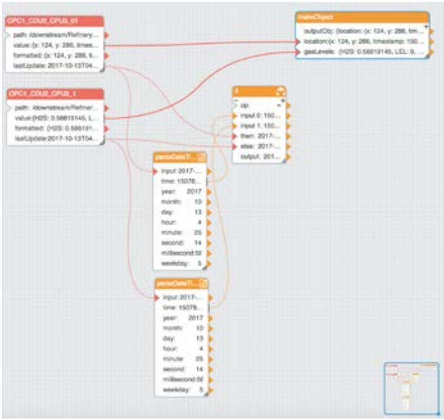
We want to add a timestamp that represents the last updated time from the combination. Because their respective timestamps are publishing at a different frequency, we want to get the last updated time for each individual subscription. The lastUpdate property is not shown on the input blocks, but select the block to view it in the bottom of the right Properties pane. Any property can be made visible, if they are not in the default, by left clicking the blue dot on the right and selecting “pinned”.



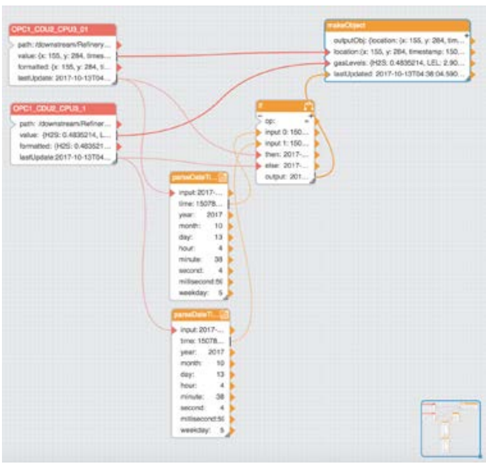
But we only want one value that represents the greatest value of the two. For this we will use a conditional "if" block" from the Logic/Operations section. We can't use the lastUpdate values directly because they are strings. We need to pre-process this with a "parseDateTime" block. Let us bind the "lastUpdate" from the Gas Detector Location OPC1_CDU2/CPU3_01 to the first parseDateTime block and then the "lastUpdate" from Gas Detectors OPC1_CDU2/CPU3_01 to the second parseDateTime block as shown:



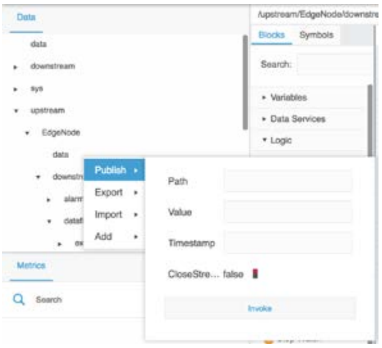
Now we can use the time field that is in milliseconds. Let us bind the "time" fields from each corresponding parseDateTime block into the "if" block input 0 and input 1. The "if" block will compare the two values and use the values from "then" and "else" for the output. We want to use the string values from the original blocks, not in milliseconds, as output. So we will bind the lastUpdate values from Gas Detector Location OPC1_CDU2/CPU3_01 and Gas Detectors OPC1_CDU2/CPU3_01 to the "then" and "else" respectively. So now the output will be a string.



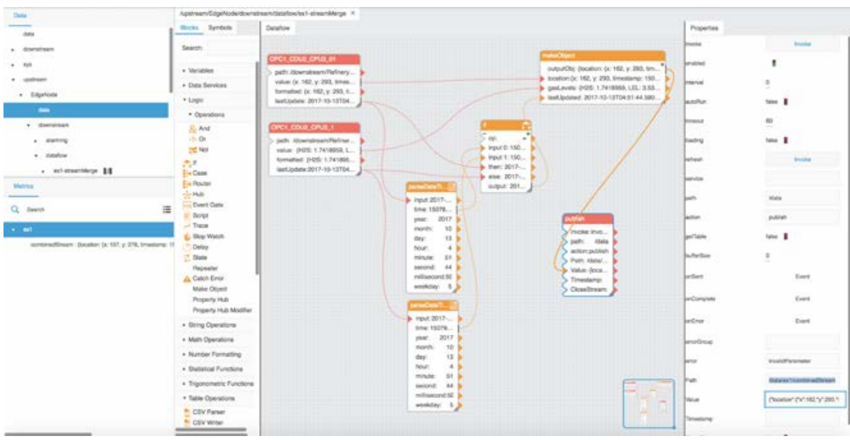
Now add a new parameter to the makeObject block called lastUpdated. We now bind the output of the “if” block.



But now we want to publish the outcome to the /data/ex1/combinedStream. In order to “publish”, this is not a Dataflow block, but an action on the “data” node (this is the “data” node under the EdgeNode). By right clicking on the “data” node the actions Publish, Export, Import and Add appear. Left click the “Publish” and drag into the Dataflow Editor canvas.



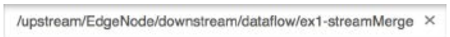
We need to bind the "outputObj" parameter from the makeObject to the Value parameter in the publish block. In the "Path" parameter type "/data/ex1/combinedStream". To test this, click on the **Invoke** button at the top. This will run one instance of the block.



But in order for the block to update for any change to an input, we must change the "autoRun" parameter to "true".

Now the value in the Metric pane is updating as the inputs are streamed.

Close this Dataflow by clicking the "x" at the top of the canvas.



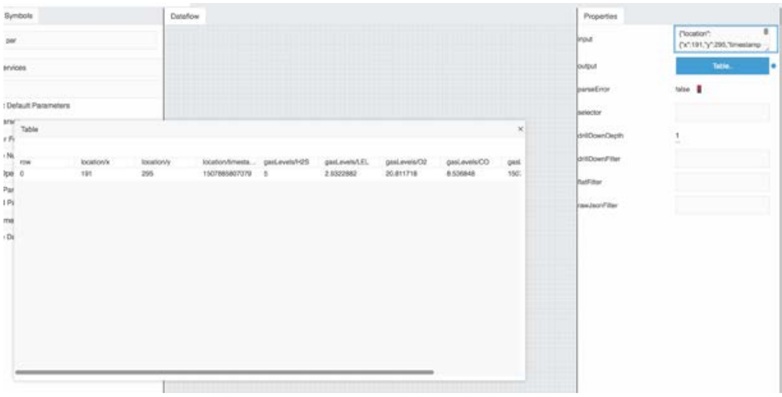
Create an alert from the published combinedStream

Create a new Dataflow called "ex2-combinedAlert".

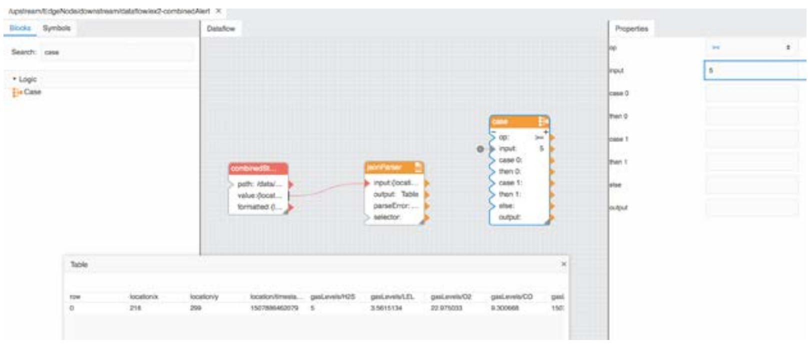
Select the  symbol, and the canvas will open so we can start to populate the logic. Use the ex1-streamMerge output called "combinedStream" as input. First select the data node (under EdgeNode), then left click on the Metric plane "combinedStream" value and drag it into the canvas. We have now subscribed to the "combinedStream" in this dataflow what was published in the previous dataflow.

We want to have a block that allows us to parse JSON. The "JSON Parser" block. This will take a JSON input and output a table value. When it outputs a table value, you can see in the output field it indicates "Table..." and in the properties window you can click on the output button "Table" to view the content in a pop-up.

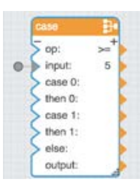
Clicking the "Table..." button will display the following:



We want to have different levels of severity for the alert we want to create. If H2S is above 3, then severity is 1, if above 5 then severity is 2. For anything else it is 0. Since we have multiple levels of comparison is a “case” block. We select the case block and drag into the canvas. We select the “case” block and we select the op parameter to “>=”. But we want to use the value of the gasLevel/H2S in the table. With the JSON table still open, we click on the cell with the data we want and drag it to the “case” input. It can also be dragged into the input onto the properties cell. The grey circle indicates the binding is not visible on the canvas, because when we close the JSON pop-up window, the binding must continue.



The grey circle indicates the binding is not visible on the canvas, because when we close the JSON pop-up window, the binding must continue.



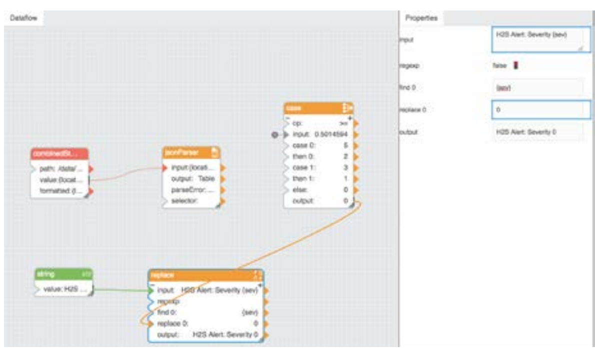
Populate the case values for case 0 >=5 then 2 and case 1 >= 3 then 2 else 0. The output should automatically start to update.

Now that we have calculated the alert, we need to publish it. But first create a string to add to the canvas and create a Template for what we want our alert string to look like. Templates are used to replace any value rather than each occurrence (we will see more of this later).

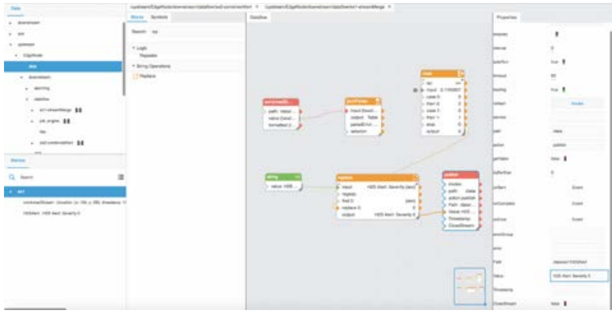
Create a string block with the template value. Find the string block and drag it into the canvas. In the Properties pane, change the value to “H2S Alert: Severity {sev}”. Replace the “{sev}” string with a dynamic value. Note that creating a string input box is a good way to easily identify where an input change might be needed (rather than searching in the replace box).

Find the “replace” block and drag into the canvas. Bind the output of the previous “string” box to the input of “replace” box.

Select the “replace” box and under Properties, change “find 0” to “{sev}”. But we will use the output of the case block. So, we will bind the output of the “case” block to the replace 0 of the “replace” block.



Now publish the alert. As we did with the previous dataflow, right click the “data” node to display the actions Publish, Export, Import and Add. Left click **Publish** and drag it into the Dataflow Editor canvas. We will bind output of the “replace” block into the value parameters of the “publish” block. Under the Properties pane, change the Path to “/data/ex1/H2SAlert” and set autoRun to “true”.



Now we have created and published a working dataflow that publishes an alert. Any application can subscribe to the alert and will receive it if it becomes available.

Store the data in a historian database

ParStream is the historian database that is included with the EFM, but the EFM supports a pluggable model for historians. We use the ParStream historian to persist data that will be used for applications. Persisting the data from a node causes a subscription to occur and therefore the message broker generates traffic across the network.

Background on the ParStream Historian Database

ParStream is a time-series database that is:

- Optimized for INSERT operation and for complex analytical queries
- At the same time, ParStream is a non-transactional database: it supports INSERT and QUERY only. No UPDATE or DELETE functions are defined for performance reasons

Architecturally the ParStream Historian Database is:

- Massively parallel, shared-nothing
- Optimized for high volume
- Optimized for extremely complex analytical queries
- Designed to run on a cluster of commodity servers
- Supports high availability and replication mechanism by defining a replication factor across multiple nodes in the cluster

Some of the key features are:

- Fast
    - o Installation, configuration and data migration

    - Transformation and indexing
    - Inserts and queries
    - High volume ingests
- Advanced and extensible analytics
- Schema-based
- SQL queries
- Infrastructure and platform independent
- Software-only product

The placement of the historian database varies from project to project, depending on geographical diversity, network latency and support. It is not uncommon to use a historian database in a regional node to support a month of telemetry, but also to feed to the larger datacenter historian that will become the Data Lake to support longer term analytics.

Database design, clustering and performance optimization

To optimally use the ParStream Historian Database, you should understand how it operates with large amounts of data produce high performance results.

Note: As with all databases, the optimal configuration is a topic beyond the scope of this document. Also, there is much more functionality than is possible to describe in this guide. Please refer to the ParStream documentation for more details and to explore the parameter options.

Storing data using the EFM ParStream Historian

The ParStream Historian allows us to persist data to a database. There are two methods of creating the databases :

- Watch group - a watch group creates a table that stores a metric value it will be subscribed to. This is easy, fast and simple for single metrics. The ParStream Link will automatically assist in creating the database based upon detection of the metric path and value it is to subscribed to.
- Pre-defined schema - in this model one or more user defined tables will be used.

Each of these methods serves a purpose and we will explore how they are used in projects. Pre-defined schemas are usually used for more advanced users that desire a richer model models or optimizations for query performance. For example, rather than storing a JSON table with many values, these can be parsed and stored in separate columns for faster query by the historian.

Configuring ParStream for our examples for schema based tables

In this guide, we have defined the use of a single instance of a cluster. This was started in the Starting the EFM ParStream Historian Database section earlier as well as the placement of the *.sql files in the /data/training/sql directory.

While we have started the parstream server, we have not created any tables nor have we linked them to the ParStream DSLink for use. That is our next task. To give some context, we typically run the ParStream Historian Database on fog and data center nodes that support more CPU performance and storage capacity.

Building tables to persist data in defined schema tables

Let us build the sensorTelemetry and alertHistory tables that we will use to persist data later on in this guide. As a reminder, our parstream server is running on the fog node with the port 23456.

As the efm user, type the following[5]:

```
$ cd /data/training/sql
$ cat sensorTelemetry.sql | /opt/cisco/iotdc/parstream/bin/pnc -p 23456
$ cat alertHistory.sql | /opt/cisco/iotdc/parstream/bin/pnc -p 23456
```

These tables have been defined with a specific structure consistent on what we plan to persist and can allow us to query the data. It is possible to create tables with a single sensor entry per row or many columns that represent different sensors in a row. How we design the table depends on the on the frequency of each sensor. If it makes sense to aggregate, and if the aggregation allows us to query the data as needed later on.

The sensorTelemtry has the following structure:

Column	Description
sensorId	varstring(64)
metricName	varstring(128)
metricValue	Float
eventTime	timestamp csv_format 'YYYY-MM-DD HH24:MI:SS'
etlEventTime	timestamp (defined to speed up queries in Parstream and partitioning)

The alertHistory has the following structure:

Column	Description
alertID	varstring(128)
siteID	varstring(64)
sensorID	varstring(64)
sensorType	varstring(64)

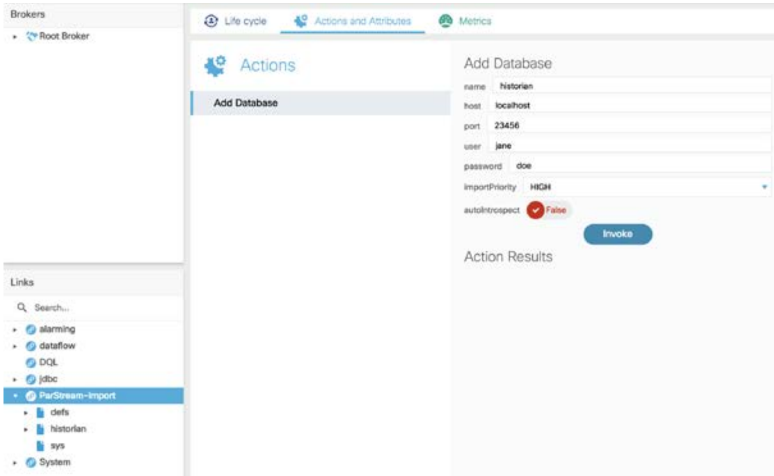
[5] Use the pnc tool to connect to the ParStream Historian Database from in a Linux shell. This tool is included in the EFM software distribution in the ParStream /bin folder after installation.

sensorValue	float
alertType	varstring(64)
eventTime	timestamp
etlEventTime	timestamp (defined to speed up queries in Parstream and partitioning)

Configuring the ParStream DSLink

Using the EFM System Administrator on the Fog node, configure the ParStream-Import DSLink to use the newly created tables. This is done by the following:

1. Open in a browser **https://Fog_node_IP_address:8443/efm-admin**.
2. Click on **Root Broker** in the Brokers list.
3. Click on the **Management** tab.
4. Expand the **links** node in the tree.
5. Under the “**Links**” pane at the bottom left, select **ParStream-Import**.
6. Select the **Actions** tab. **Add Database** will already be selected for you. Populate the following fields:
    - Name: historian
    - Host: localhost (dslink is the same host as the parstream server)
    - Port: 23456
    - User: jane (dummy value - user authentication set to false)
    - Password: doe (dummy value - user authentication set to false)
    - ImportPriority: HIGH
    - autoIntrospect: True (tells it to collect metadata on the instance at startup)
7. Click **Invoke**.

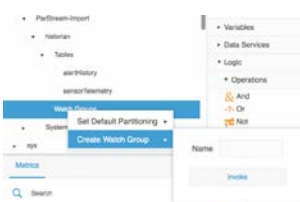


8. After invoking, the historian node appears under the ParStream-Import link.

9. Expand the historian and tables nodes. The alertHistory and sensorTelemetry tables appear.

Historian Watch Groups

The Watch Groups node is automatically created under the historian node (under **ParStream-Import**). It is a historian method that allows us to create simple tables to store metric values (without using to the method described previously using the Linux shell interface). This method stores a metric into a table. For every metric that is added, a new column is added and can be queried.



Store data using dataflow to a Historian Watch Group

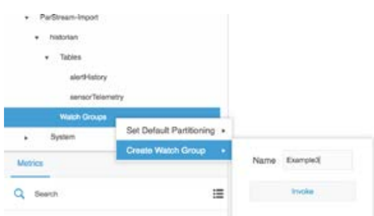
Create a new dataflow example called “ex3-insertTelemetry”. But in this case, create it on the Fog node where the ParStream Historian Database is located and have it subscribe to the data source on the remote Edge node. This is the more common scenario in most IoT projects.

Let use a browser to open the Dataflow Editor on the fog node, as in examples above, by typing http://fog_node_ip_address:8443/dataflow.html.

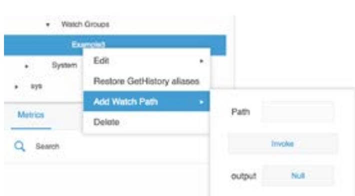
In this example, we will accomplish the following:

- Create a Historian database (table) under the Watch Group section to store the published the combined stream, under the Edge node path “/data/ex1/combinedStream”. (Reminder, the /data path is local to each, so when referencing it is important to make sure the proper location in the hierarchy is referenced)
- Store the metric using the internal path name (Different than the display path name, similar to URIs, certain characters are replaced or eliminated)
- Perform a simple query

Start by creating a new historian table in the Watch Group called “Example3”. Right click on “Watch Groups”, move to the “Create Watch Group” and type in the name “Example3”. Create the table by left clicking the mouse and selecting “Invoke”.



We need to add the metric in which we want the Watch Group Example3 to persist. But before doing so, first find the correct internal name of the metric.



Create a new Dataflow called “Example3” on the Fog node. Right click under downstream/dataflow to create the dataflow named “example3” and “Invoke”. Next, navigate the dataflow by clicking on the symbol next to Example3. A blank canvas should appear.

Since we want to persist the Edge node path “/data/ex1/combinedStream”, navigate the Data pane and find the data node.

In the Metrics pane, expand the “ex1” node so that “combinedStream” appears.

Drag the combinedStream metric into the blank dataflow canvas by left clicking it and moving it to the canvas. Now select the “combinedStream” box that is in the dataflow editor. The Properties pane will appear and we will use that to determine the internal path name for this metric.

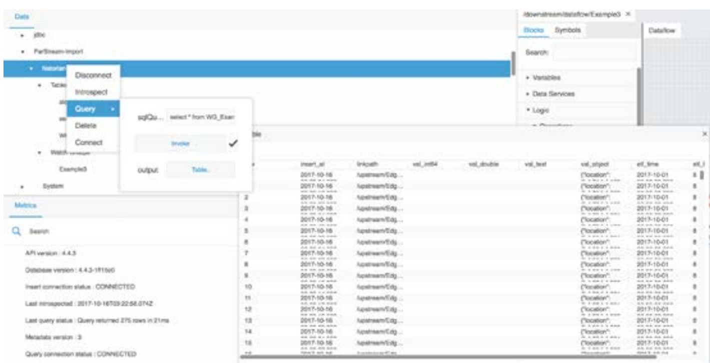
If we view the Properties pane, the “path” value is “/upstream/EdgeNode/data/ex1/combinedStream”. Since we didn’t use spaces, it has stayed the same. Delete the “combinedStream” block in the dataflow since it was only used for the purposes of obtaining the internal path.

Cut this value to the Clipboard in the local browser and go to the historian Watch Group Example3. Let us right click to “Add Watch Path” and paste “/upstream/EdgeNode/data/ex1/combinedStream”, then “Invoke”. At this moment, the table is created and a new node in the Metrics pane appears with the symbol to indicate it is being persisted in the historian.

We can verify that the historian is persisting the data by performing a query.

The ParStream-Import link does not automatically add the newly created Watch Group Example3 table to the list of Tables. We need to select the “historian” node and then left click “Introspect” for this to appear. Now we can see a new table called “WG_Example3”.

Select the “historian” node, right click and move to query box to type “select * from WG_Example3;” and left click and select Invoke. A check mark should appear next to the Invoke box to show completion. Left click on the output “Table..” button. A pop-up window should appear showing rows that have been inserted.



We have now persisted and queried. Like any other action, we can also invoke it into a dataflow (by dragging the metric from the Metric pane).

Store data using dataflow to a Defined Schema in the historian database

In this example, we will use the “combinedStream” metric on the Fog node. This is a JSON table of several gas levels. The telemetry that we will insert are the 4 different gas levels and the time they were measured. We will build a functional dataflow that serves only this purpose; the goal is to build the logic, but make it resusable if we want to turn it into a subrouting (see Symbols later).

In this example, create a new dataflow on the Fog node called “ex4-insertTelemtry” that accomplishes the following:

- Use the “combinedStream” metric, but use the individual gas levels and the gas time
- Demonstrate how renaming blocks assist in self-documentation (with use of Control-Enter)

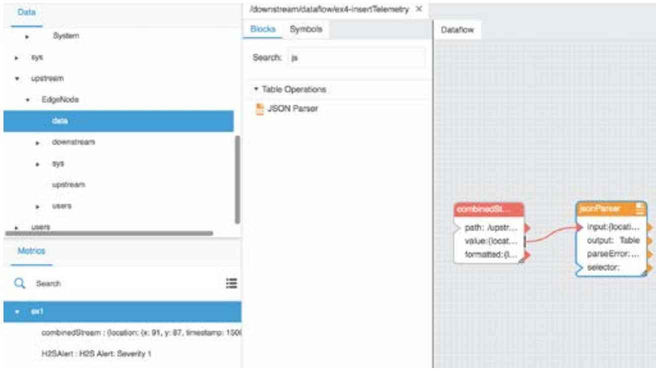
- Use the 5 separate metrics in a normalized generic schema with 4 inserts, 4 gas values with a the original gas time to the event Time of each
- Define new metrics that are more generic and include SiteID and SensorID

To begin, create a new Dataflow called “ex4-insertTelemetry” on the Fog node. Right click under downstream/dataflow to create the dataflow named “ex4-insertTelemetry” and “Invoke”. Navigate the dataflow by clicking on the symbol next to ex4-insertTelemetry. A blank canvas should appear.

Subscribe to our input metric “combinedStream” again. Navigate the Data pane until we find the data node /upstream/EdgeNode/data and left click.

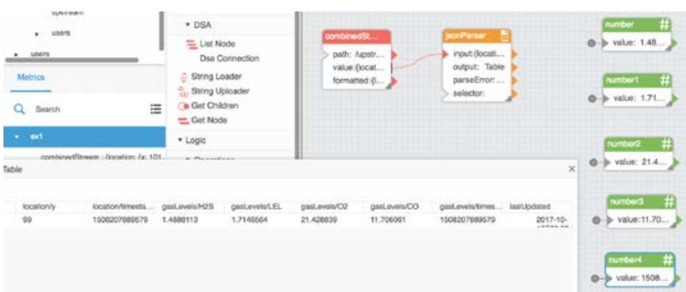
In the Metrics pane, let us expand the “ex1” node so that “combinedStream” will appear.

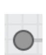
Let us drag the combinedStream metric into the blank dataflow canvas by left clicking it and moving it to the canvas. We are going to parse the metrics in each row of the combinedStream JSON table. We will use the jsonParse block and bind the value from the combinedStream block to the input of the jsonParse block. In the Properties pane we can click on the output Table to view the pop-up with the table and metrics in columns.



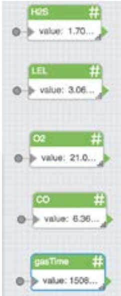
The output Table from the jsonParse of combinedStream block.

Since we want to use the specific gas levels and the event time, create 5 number blocks and drag the metric value from the gas levels and gasLevels/timestamp input (drag the numeric value in the first row, not the column headers).



Note that a symbol  appears as input. This is a source that cannot be displayed directly with an arrow.

But the block names number, number1, number2, number3 and number4 make it hard to understand what the content represents. Let us rename the blocks by double-clicking on the block name “number” and replacing the work with “H2” and hit Control-Enter (Control Enter is good practice to change the display name and actual name. Enter only changes the display name). We rename the number2, number3 and number4 to “LEL”, “O2”, “CO” and “gasTime” respectively.

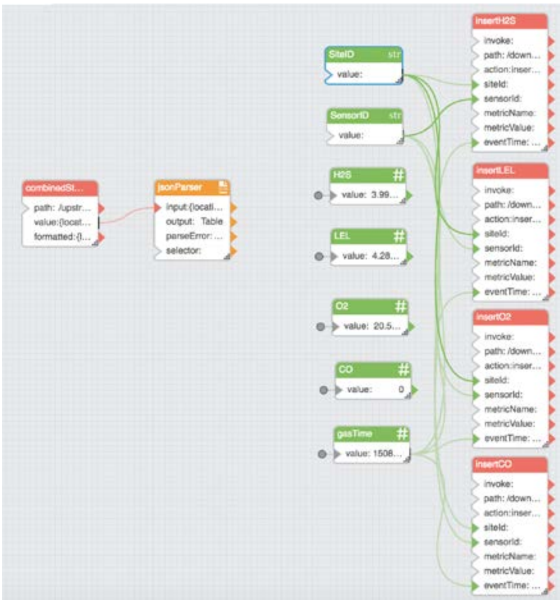


Let us define two additional inputs we will need to persist the metric into the ParStream Historian sensorTelemetry table. Let is create two new string blocks and rename them “SiteID” and “SensorID”. Normally the SiteID and SensorID are dynamic, but in this example, they will be hardcoded. Let is assign the value of “Refinery-FogNode” for SiteID (the broker name) and “OPC_CDU2/CPU3_01” for SensorID that is the Gas Detector name in the properties pane.

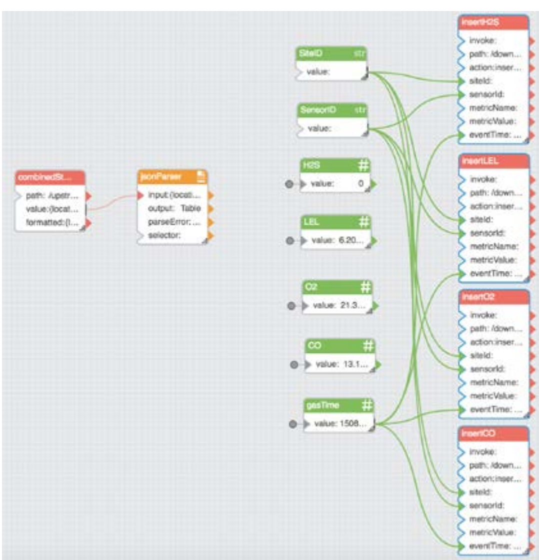
Let us grab a publish action block for the sensorTelemetry table. Find the sensorTelemetry node under “downstream/ParStream-Import/historian/Tables/sensorTelemetry”, right click the mouse over “sensorTelemetry” and move the mouse “Insert Row”; left click to drag the action into the canvas. To make adding more copies of the same action, select the insertRow block just added and hit Control-D three times. Move the blocks to make it easier to see.

Let us also make it easier to understand the function of the block and renamed each block “insertH2S”, “insertLEL”, “insertO2” and “insertCO” respectively with Control-Enter.

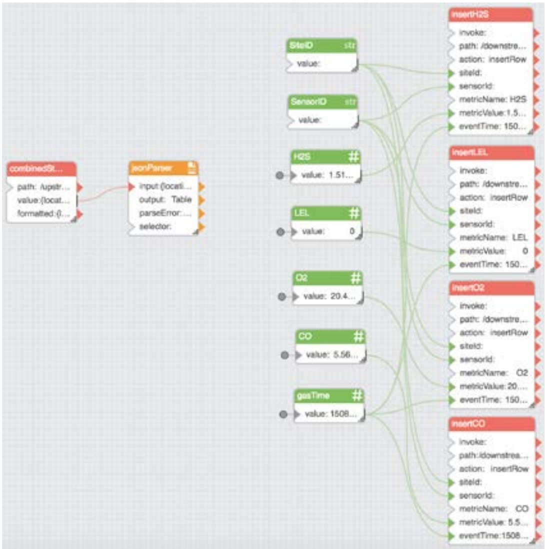
The insertRow has siteID, sensorID, metricName, metricValue and eventTime as inputs we can provide. Let us bind siteID, sensorID and eventTime to each insert block from the SiteID, SensorID and gasTime block respectively.



Let us now bind the H2S, LEL, O2 and CO block's value respectively to the metricValue of each insertRow.

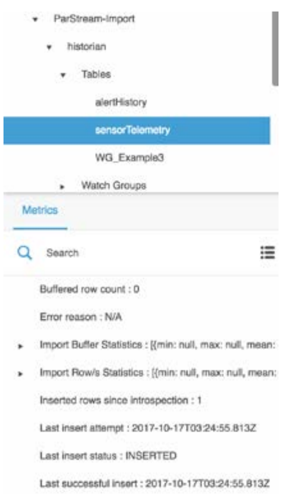


Because we don't have the metricName in a string block, which can easily be done, let us define them in the properties pane for each "insertH2S", "insertLEL", "insertO2" and "insertCO" respectively with "H2S", "LEL", "O2" and "CO". The dataflow should now look like this:

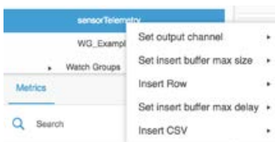


Like any other action in Dataflow, the insertRow does not automatically insert rows. Before we set autoRun to true, let us test a single row insertion by selecting the insertH2S block and under the Properties pane clicking the “Invoke” button right of the “Invoke” parameter. This will insert a single instance and should return a Status below of “INSERTED”.

We can see that not only does it return the INSERTED status, we can verify in the Metrics pane that a row was inserted.



The ParStream Historian has some architectural limits in the design about inserting rows into the database. While it can insert a large number of rows per second, it can only perform a few commits per second as it writes across partitions. Therefore, the ParStream DSLink uses a definable buffer on each table that reduces the number of commits per second and allows the number of rows we can insert per second. This is controlled by right clicking on the table name and setting the values for “Set Insert buffer max size” (rows) and “Set Insert buffer max delay”. Whichever comes first will cause the ParStream DSLink to commit.



We only insert 4 rows per second with gas levels, so we suggest setting the Set Insert max delay in addition to the Set Insert buffer max size. Let us set “Set Insert buffer max size” and “Set Insert max delay” to 10000 and 1000 (for 10,000 rows and 1 second) respectively.

Now let us activate the autoRun on the insertRow blocks. If rows are buffered in the Metrics pane, the “Buffered row count” will be greater than zero.

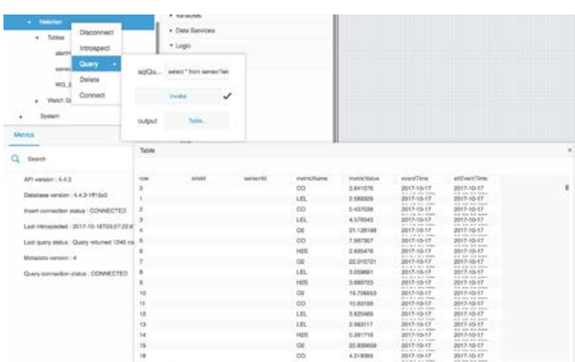
Querying Data from a historian database

Much like inserting into a historian database, querying a database can be achieved through Dataflow or the broker. We show an example of a query that returns rows from the sensorTelemetry table.

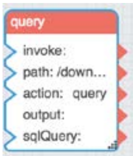
As mentioned earlier, the ParStream historian queries are performed using a SQL style syntax. For many database users, this will easy to learn. For all the SQL functionality, refer to the ParStream documentation.

Let us create a new Dataflow called “ex5-queryTable” on the Fog node. Right click under downstream/dataflow to create the dataflow named “ex5-queryTable” and “Invoke”. Let us navigate the dataflow by clicking on the symbol next to ex5-queryTable. A blank canvas should appear.

The Query action is exposed by selecting the historian (downstream/ParStream-Import/historian in the Data pane) and right-clicking to show actions. We can test a query by inputing into the sqlQuery box the following “select * from sensorTelemetry;” and clicking “Invoke”. When finished next to the output parameter, a “Table..” button will appear. By clicking on the button, a pop-up of the query should be as follows:



In order to use data from a database we can also drag the query action into a Dataflow. Let us expose the query action by selecting the historian (downstream/ParStream-Import/historian in the Data pane) and right-clicking to show actions. Let us left-click the Query action and drag into the blank canvas for ex5-queryTable. Click on the query box so we can input the sqlQuery in the Properties pane.



Under Properties, next to the sqlQuery parameter, input "select * from sensorTelemetry;" and hit Enter. But to test this, next to the "invoke" parameter, click "Invoke". If the error parameter is empty, then the output parameter will show a "Table.." button. Click on the button to view a pop-up window with the query results that are the same as the query above.

As with all actions, this not updating since autoRun is false by default. We can set the interval to 10 and turn autoRun on and new content will be updated every 10 seconds.

One of the most common uses of querying telemetry and alerts from tables is for displaying metrics on a dashboard.

EFM Projects with Advanced Dataflow features

When we build projects with the EFM, there are a large number of sensors across many edge brokers. We have seen examples of building outcomes for a single metric or gas detector. But in reality, we need to monitor more than one sensor at a time in a scalable manner. For this, the EFM relies on the DQL microservice.

We have also discussed the creation of functional dataflows that have clear set of inputs, some transformational logic and how to publish the outcomes. What we did not introduce is the concept of dataflow subroutines or symbols. Once a single instance of a dataflow has been tested, we often convert that into a generic subroutine that can be reused more easily across projects.

Subscribe to more than one node or dynamic query

The DQL is a Distributed Query Language that is implemented as an EFM microservice. DSL uses wildcards to subscribe to more than one node at a time. These queries are continuous and are updated in real time. It requires the DQL DSLink to be installed and is included with the EFM.

The benefits of using DQL include:

- Makes it easy to subscribe to a large number of nodes without creating individual subscriptions
- It has the ability to span the entire data hierarchy, across all message brokers and nodes

- Allows for dynamically query will join a subscription as they become available (as long as you have a pattern that it can understand the node hierarchy and naming convention)
- It can filter, transform and invoke actions on nodes as part of a query

The DQL query returns a table with a list of nodes and their values.

The DQL has the following clauses:

- Option - high level query parameters
- List - define the nodes to participate in the query (as a Path)
- Filter - allows you to subset what comes back from the list
- Subscribe - analogous to SQL projection
- Expression - data manipulations capabilities

Use the pipe symbol "|" to join individual operators together. The operators "*" and "?" are wildcards in the path, but when used at the end it can also be used for string completion.

List, filter, subscribe and expression examples

List examples:	
list /downstream/System/CPU* | subscribe :name value	Specific node only. This returns a table, with name portion of the path and value, with all the nodes under the path that begin with CPU. In reality there is only one, but it could be more than one.
option traverseBrokers=true | list /downstream/*/downstream/System/CPU* | subscribe :name value	Cross Broker tiers. This returns a table, with name portion of the path and value, with all the nodes under the path ALL LEVELS down (using the *) that begin with CPU across ALL Brokers. In reality there is only one, but it could be more than one.
option traverseBrokers=true | list /downstream/?/downstream/System/CPU* | subscribe :name value	Only one level. This returns a table, with name portion of the path and value, with all the nodes under the path ONE LEVEL down (using the ?) that begin with CPU across ALL Brokers. In reality there is only one, but it could be more than one.
option qos=2 |path /downstream/System | subscribe Memory_Usage	Create a subscription with a QoS option 2 to the explicit path and subscribe to Memory_Usage

Filter examples:	
list * | filter $type	Returns all types.
list * | filter $type="int"	Returns only int types.
list * | filter @unit	Returns items with @unit attribute

list * | filter @unit=md	Returns items with @unit attribute = md

Subscribe examples:	
list * | filter $type=”number” | subscribe	Subscribe to all numeric data
list * | filter @unit=”md” | subscribe	Subscribe to all numeric data with attribute=”md”
list /downstream/System/* | subscribe :name value	Subscribe to all names and values for the node; create a table of names and values
path /downstream/System | subscribe Memory_Usage	Explicit path instead of list

Expression examples:	
list * | filter @unit=”md” | subscribe | expression billCustomerJustKidding=”row.value *”	Subscribe to all numeric data with attribute=”md” and calculating billCustomerJustKidding the value of ….
list * | filter @unit=”md” | subscribe | expression msg=”’Value: ‘+row.value”	Subscribe to all numeric data with attribute=”md” concatenating the string “Value:” with the row.value
list * | filter @unit=”md” | subscribe | expression threshold=”Math.max(5000,row.value)”	Subscribe to all numeric data with attribute=”md” calculating the value from a function Math.max using the row.value

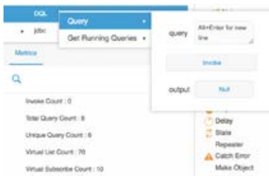
option examples:	
option qos=2 |path /downstream/System | subscribe Memory_Usage	Explicit path instead of list
	Subscribe to all numeric data with attribute=”md” concatenating the string “Value:” with the row.value
list * | filter @unit=”md” | subscribe | expression threshold=”Math.max(5000,row.value)”	Subscribe to all numeric data with attribute=”md” calculating the value from a function Math.max using the row.value

Note: DQL will NOT traverse queries across brokers UNLESS you override the default. This is to prevent that a DQL user from accidentally crossing brokers and protect against accidental rogue queries. To perform queries across brokers, prepend the query with “option traverseBrokers=true”.

Create a Query with Dataflow

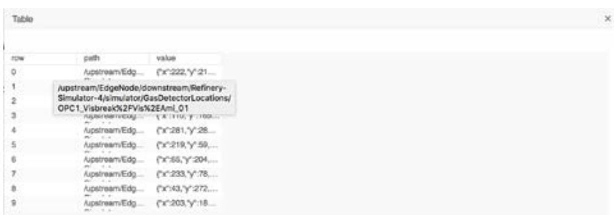
Create a new Dataflow called “ex6-dqlQuery” on the Fog node that will find all the Gas Detector Locations. Right click under downstream/dataflow to create the dataflow named “ex6-dqlQuery” and select “Invoke”. Navigate the dataflow by clicking on the symbol next to ex4-insertTelemetry. A blank canvas should appear.

Next, find the DQL node under /downstream/DQL, select with the mouse, right-click to show the Query action, left-click the Query action to drag into the canvas. A DQL box will appear in the canvas.



Change the name of the block to “getAllLocations” and hit Control-Enter.

Rather than write the query directly, let us use a string variable to define query input. Let us define the string variable block, change the name to “getAllLocationsDQL”, and in the Properties pane change the value to “list /upstream/EdgeNode/downstream/Refinery-Simulator/simulator/GasDetectorLocations/? | subscribe”. Let us bind the “getAllLocationsDQL” to the “getAllLocations” DQL query block to get all the Gas Detector Locations with one level deep of children.[6] Under the properties pane, let us “Invoke” and click on the output “Table..”. A table should appear with the subscribed values.



For the “getAllLocations” DQL query block, change the Property autoRun to “true”.

Now, if a new Gas Detector Locations does come online or go offline, the table updates because it is a continuous query.

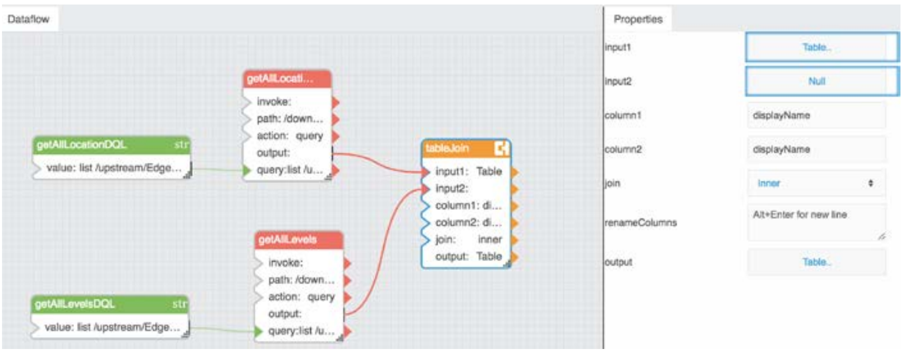
Change the query to “list /upstream/EdgeNode/downstream/Refinery-Simulator/simulator/GasDetectorLocations/? | subscribe :displayName as displayName,value as locations”[7]. Now we have a table with the path (with internal name), displayName and the value.

Next, duplicate the “getAllLocationsDQL” string variable block by hitting Control-D, change the name of the new block to “getAllLevelsDQL”; change the value to “list /upstream/EdgeNode/downstream/Refinery-Simulator-4/simulator/GasDetectors/? | subscribe :displayName as displayName,value as levels”. Drag a new DQL query block, bind “getAllLevelsDQL” as the query and set the autoRun property to “true”. Rename the DQL query block to “getAllLevels”. This now returns a table with the path (with internal name), displayName and the value (that is a JSON table of gas levels).

[6] Note that the node path /upstream/EdgeNode/downstream/Refinery-Simulator/simulator/GasDetectorLocations has no spaces in GasDetectorLocations even though the display name does. It is necessary to use the internal name for a functional query.

[7] As a best practice, we eliminate the colon symbol by renaming :displayName column header to displayName using “as” in the DQL query

Let us merge the streams, which are really two tables. We are going to do a table tableJoin. We need to input the two tables, the column in which we joining and then the type of join we doing. Let us bind the output from the "getAllLocations" DQL query block to the input1; bind output from "getAllLevels" DQL query block to input2. We want to join based upon the displayName column. In the Properties pane type for column1 "displayName", column2 "displayName" and join "inner".



We now have a joined continuous queried table with the path, displayName, locations and JSON values as show below.

Table

row	path	displayName	locations	value
0	/upstream/Edg...	OPC1_Spare_01	{"x":191,"y":22...	{"H2S":3.41947...
1	/upstream/Edg...	OPC1_Visbrea...	{"x":79,"y":184,...	{"H2S":3.13606...
2	/upstream/Edg...	OPC1_Merox1-	{"x":159,"y":15...	{"H2S":1.13434...
3	/upstream/Edg...	OPC2_CHD2/F...	{"x":109,"y":119...	{"H2S":3.12227...
4	/upstream/Edg...	OPC2_MIDW/...	{"x":227,"y":29...	{"H2S":2.26038...
5	/upstream/Edg...	OPC2_Amine2-	{"x":174,"y":92,...	{"H2S":3.05045...
6	/upstream/Edg...	OPC1_VacDisti...	{"x":49,"y":219,...	{"H2S":4.45469...
7	/upstream/Edg...	OPC2_Spare_01	{"x":276,"y":13...	{"H2S":2.70786...
8	/upstream/Edg...	OPC1_CDU2/...	{"x":150,"y":27...	{"H2S":0.24380...
9	/upstream/Edg...	OPC2_CDU1/...	{"x":251,"y":10...	{"H2S":1.72668...

Create Dataflow Symbols (subroutines)

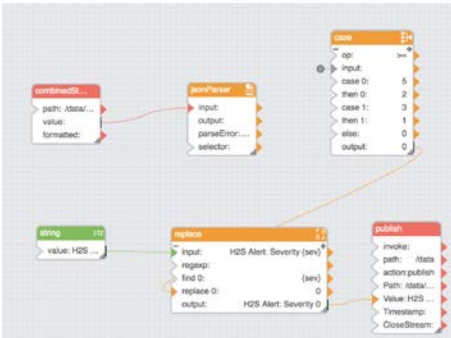
In previous examples, we have operated on individual metrics or using DQL to get back a continuous query table. Let us introduce the concept of symbols, or a version of dataflow subroutines. A symbol is a collection of blocks that perform a function placed into a single container. Symbols have input properties and can create parameters from inside the symbol that are outputs.

This makes dataflows much easier to understand, less complicated and allows reuse of common blocks.

The usual way to start creating a symbol is by creating the logic to operate on a single instance of a metric and then convert it into a symbol. Then we turn the input as a property we can bind inside the symbol.

We are going to create a new dataflow called "ex7-Symbol" on the Fog Node. Let us take the existing dataflow from the edge node "ex2-combinedAlert" and copy it by selecting it, right-clicking to show the action of "Export Dataflow" and

"Invoke". In the output box, we get a JSON representation of the dataflow. Let us select all of it and copy it. Select the "ex7-Symbol" and select "Import Dataflow", paste in the text box in the copied content and "Invoke". Open the dataflow and see that we have a functioning dataflow (note that the data paths have changed, so it might not be fully functional).

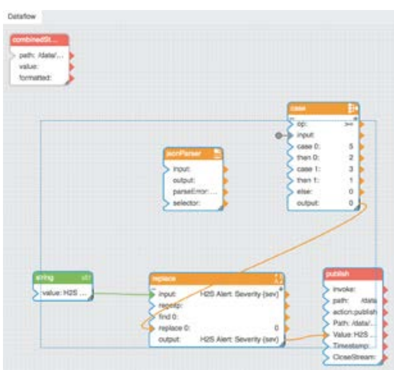


Let us separate the input from the logic, in this case the "combineStream" metric.

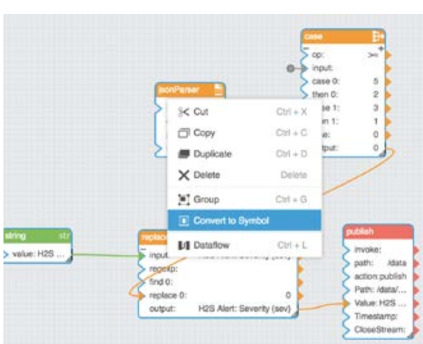
Now look at the dataflow container and separate it into two sections: the inputs and the logic components. For this subroutine, we want to keep the logic section and we will redefine inputs into the new subroutine container.

Do the following to create the symbol:

1. Unbind the output of "combinedStream" from the input of the jsonParser.
2. Select all the blocks except for "combinedStream".

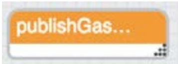


3. Right-click over a selected block and select "Convert to Symbol".



4. Type the name "publishGasAlert", select "OK".

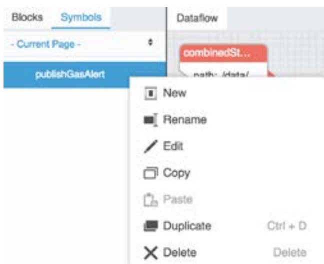
5. The following symbol block appears. This represents an instance of the symbol and if right-clicked, it can be edited.



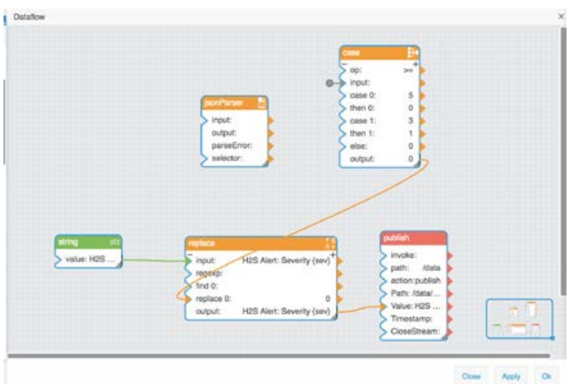
6. Go to left pane in the dataflow canvas, select “Symbols”.

7. Now delete the symbol “publishGasAlert” block in the canvas.

8. We want to edit the symbol subroutine definition, not the instance. Select the “publishGasAlert”, right-click and select Edit.

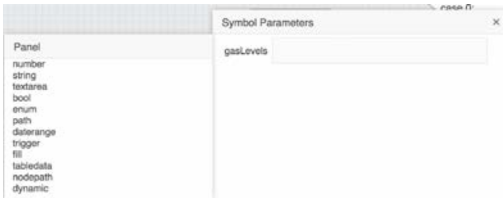


9. A new pop-window will appear with three buttons at the bottom. Symbols have Close, Apply and Ok. Close loses any changes without saving, Apply saves and executes the change and Ok does an apply and close while saving. Normal Dataflow editing applies as you are working



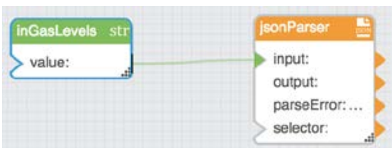
10. Create a JSON input to the symbol. To accomplish this, right-click on the blank canvas and select “Symbol Properties” or a parameter. This lets us define input and output properties. The difference is that for input parameters are bound from sources outside of symbol, the output parameters are bound from blocks or sources inside the symbol.

11. A JSON object is our input. Drag “dynamic” into the “Symbol Parameters” pane. We can select the name and change it to gasLevels.



As a recommendation, we don’t directly bind the Symbol Property gasLevels into the jsonParser block, but we will create a string variable block to show that it makes it easier to follow the dataflow. The inputs from symbol properties become more apparent versus internal bindings.

Create a string variable block, change the name to “inGasLevels” and bind output to the input of jsonParser.

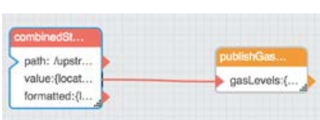


On the blank canvas, right-click and select Symbol Properties. The pop-up will appear again. We are going to bind the gasLevels to the value of the inGasLevels string block. Select the blue dot on the right of the text block and drag to the value of inGasLevels. A small circle will appear to show that it is bound.

One more thing, when creating a symbol, sometimes the bindings from the output of blocks to the input of blocks do not always work. We want to bind the table cell we selected from the jsonParser as input to the case block again. Let us unbind the existing input first.

Now we have a problem, we cannot open Table with subscribed data as we did in the original example because we don’t have a subscription as of yet. Since we don’t know the binding path, let us perform an alternative workaround in the original dataflow ex7-symbol canvas. Click “Ok” on the symbol pop-up window to save and close.

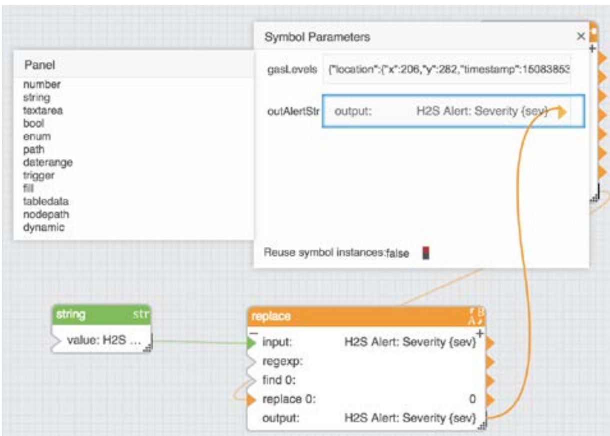
The workaround is to provide an input to the symbol to allow us to use the parsed JSON table for binding. From the left pane, drag the symbol “publishGasLevels” into the canvas. Erase the existing “combinedStream” box because it should not be functional after copying over the dataflow from the Edge node example - the path reference was local to that broker path, but we are on the Fog node now, so the subscription path is incorrect. Go to the Data path /upstream/EdgeNode/data/ex1 and drag in a new “combinedStream” block with the correct relative path on this broker (the value should be updating if functioning properly). Bind the value from “combinedStream” to the gasLevels input property of the “publishGasAlert” symbol.



Now select the “publishGasAlert” symbol block and right-click and select “Edit Symbol”. The pop-up will appear. We can now observe that the jsonParser now has data and the parseError is false.

Again, we want to use the value of the gasLevel/H2S in the table. Open the JSON table from the jsonParse block, click on the cell with the data under gasLevels/H2S and drag it to the “case” input. Now we are publishing data to the Fog broker /data/ex1/H2SAlert.

But rather than publishing directly, make the symbol more generic and create an output parameter call “outAlertStr”. We right-click on the blank canvas and select “Symbol Properties”, from the left drag the string text under Symbol Properties, select the string name and change to “outAlertStr”. We now have created a symbol subroutine that receives a JSON table as input of a Gas Detector instance and calculates the output outAlertStr if certain gas levels are met for H2S.



Use the Dataflow Repeater (block) with symbols

We now would like to watch an input table that can be used as input to a symbols subroutine. When we created the “publishGasAlert” symbol, we are only watching a single instance of a Gas Detector. Let us add functionality to the existing “ex7-Symbol” the “publishGasAlert”symbol to allow for monitoring of an table input that represents a list of Gas Detectors, run the “publishGasAlert” symbol logic to determine the H2S Alert Severity and publish to the /data path based upon the gas detector display name.

We will introduce the use of a repeater block that takes a JSON table as input and looks to see if a value has changed. If it has changed, the output updates and triggers the next blocks to recalculate.

As in the “ex6-dqlQuery” dataflow, we are going to create a continuous query JSON table with list of gas detectors and all gas levels as input to the dataflow logic.

Let us generate the input query. As a best practice, we create a string variable to define query input and change its name to “getAllLevelsDQL”, and in the Properties pane change the value to “list /upstream/EdgeNode/downstream/Refinery-Simulator-4/simulator/GasDetectors/? | subscribe :displayName as displayName,value as levels”. Next, let us find the DQL node under /downstream/DQL, select with the mouse, right-click to show the Query action, left-click the Query action to drag into the canvas. A DQL box will appear in the canvas. Change the name of the block to “getAllLevels” and hit Control-Enter; also change property autoRun to true.

Let us find a new block called a Repeater, under Logic, and drag it into the canvas right of the “getAllLevels” block (see the documentation for more information on Repeater options). There are several input properties we will need to define.

First, the symbol property: on the left pane in the Dataflow Editor, under the Symbols tab, drag the “publishGasAlert” to the property box. Let us bind the output from the “getAllLevels” DQL query block to the repeater data input. This tells the repeater what input table to monitor. Click off the repeater and on the repeater block to expose the update properties table (in particular the renderer properties will expand based upon the symbol properties).

Now we need to bind data to the render properties input called “gasLevels”. In the Properties pane, next to the data property select “Table..”. A pop-up window table will appear. Let us drag the column header “value” to the renderer “gasLevels” property.

Note that the output table from the repeater is only generating Alert Severity 0 values. We need to troubleshoot and fix the repeater. Select the repeater block, right click and Edit Symbol 0 (this number represents the row from the table, so desirably you can quickly see if row 0’s H2S value is above 3, if not select another row). What happens sometimes is that the binding from the jsonParser H2S cell to the case input is not working and the case block is showing no value for the input. Rebind it by selecting the jsonParser, click on the output Table.., drag the H2S cell for row 0 to the case input. In the symbol editor, click “Ok”. We can look again at the repeater output table and see that alert severity levels are being properly generated.

Table

row	gasLevels	outAlertStr
0	{"H2S":3.42522…	H2S Alert: Severity 1
1	{"H2S":1.76327…	H2S Alert: Severity 0
2	{"H2S":2.86397…	H2S Alert: Severity 0
3	{"H2S":0.69057…	H2S Alert: Severity 0
4	{"H2S":2.05949…	H2S Alert: Severity 0
5	{"H2S":0.90813…	H2S Alert: Severity 0
6	{"H2S":0.28834…	H2S Alert: Severity 0
7	{"H2S":2.50084…	H2S Alert: Severity 0
8	{"H2S":0.57946…	H2S Alert: Severity 0
9	{"H2S":4.62985…	H2S Alert: Severity 1

Now modify the publishGasAlert symbol to publish at a different path for every sensor, rather than overwrite each other. Select the repeater block, right click and Edit Symbol 0. We are going to create a new string symbol property called “sensorID”. Right-click on the blank canvas and select “Symbol Properties”. Drag the string name between gasLevels and outAlertStr so we keep inputs together before the outputs. Rename the string to “sensorId”. As before, add a string block to bind the sensorId property. Drag a string block into the canvas, rename it “inSensorId” and from the Symbol Parameters drag the sensorId box to the “inSensorId” input value.

Close the symbol by selecting “Ok”.

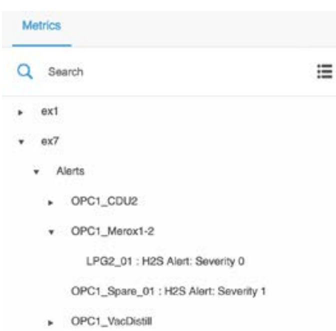
Back to the “ex7-symbol” dataflow, select the repeater, select Properties data “Table..” and the pop-up table will appear. Bind it by dragging the displayName header from the table to the repeater Properties sensorId box.

Select the repeater block, right click and Edit Symbol 0. We are going to create a new string symbol property called “pubPathTemplate”. We are going to use this as a template for the alert path depending on the sensor name. Change the value of the “pubPathTemplate” to “/data/ex7/Alerts/{sensorId}”.

As with the Alert String, add a replace block after the “pubPathTemplate” string block. Bind the value output from “pubPathTemplate” to the replace input. Edit the replace property find 0 with “{sensorId}”, bind the “inSensorId” string to the replace 0 input.

Finally, let us bind the output of the Publish “Path” input (replacing the static value). Then Select “Ok” to close the symbol.

Under the Metric pane we can see alerts appearing.

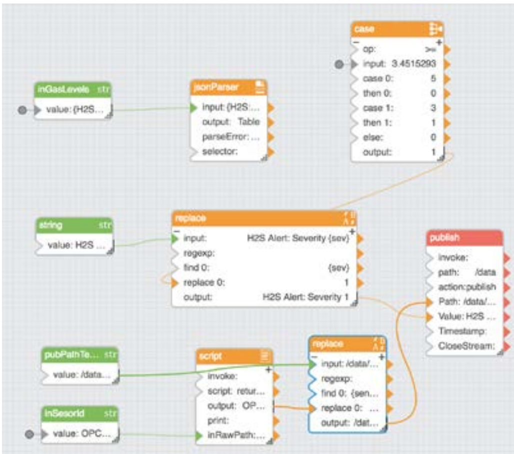


But they are not a single list: the displayName contains characters like “/” that can cause the node to be placed under another node. Let us use another block to correct this.

Select the repeater block and right click and Edit Symbol 0. Remove the binding between publish replace block and the publish Path parameter. Add a new “script” block between inSensorId and the publish replace[8]. On the script block, click “+” to provide an input parameter string called “inRawPath”. Bind the inSensorId value to the script input “inRawPath”. Under the Properties pane, edit the script box by select the “X” symbol in the box; type the JavaScript

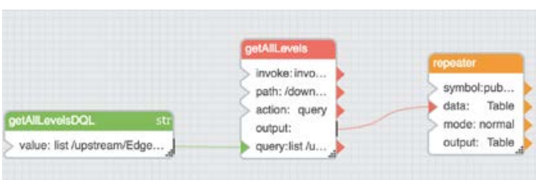
[8] The script block allows us to run a JavaScript function against an input and provides some output. For more details on the set of the supported JavaScript scripting functions, refer to http://wiki.dglogik.com/dglux5_wiki:dgscript:home . We are using the string function str.encodeUriComponent.

"return @.inRawPath.encodeUriComponent();"; then turn autorun property to true. Bind the script output to the replace block replace 0 input. Bind the replace output to the publish Path Input. Select "Ok" to save and close.



Final publishAlertLevels symbol dataflow

Now we have a functioning ex7-symbol dataflow that creates a continuous query table that feeds into the publishGasLevels subroutine to determine the severity of a H2S level at each gas detector and if greater than zero it publishes and alert to the /data/Alerts path.



Final ex7-symbol dataflow to create a continuous query and publish alerts on gas levels.

Presenting the data

Presenting the data is the most important outcome of an EFM system. This can be in the form of dashboards, reports, or as input to other applications. The tools to create dashboards and reporting are not included in the EFM, but are worth providing insight to allow for choice selection.

There are a number of third party visualization applications that provide dashboards and reporting. The ones described below do not indicate a preference, but rather how they interface with the EFM to obtain data.

As we have discussed, the EFM is designed to stream data to any subscriber. A visualization application can also be the subscriber to the telemetry, derived outputs or historical data in the Historian Database. We can separate into two categories the outcomes that are produced to best help make a selection.

Using historical data only

When you only need visualization or business analytics that are not consuming streaming data from the EFM, there are many tools on the market. One example that has been tested to function with the EFM and Parstream Historian is Qlik. More information can be found at https://www.qlik.com.

Stream data with or without historical data

If there is a need to create a dashboard that visualizes streaming data, historical data or a combination of both, we recommend partner applications that can subscribe to the IoT-DSA architecture that the EFM is based upon. Acuity Brand's Project Builder, used to be called DGLux5, has some differentiation when built for the IoT DSA architecture. These benefits are:

- Can subscribe to any node in an existing EFM system
- Installs an IoT DSA message broker that connects to an EFM message broker. This allows for all pub/sub models and QoS. It also allows for easy introspection into the Data hierarchy for the use of metrics
- Includes DQL and dataflow links to allow for advanced data transformation logic inside the visualization tool and metric calculation

A trial license can be downloaded at http://iot-dsa.org/get-started/download-dsa.

Summary

In summary, we use the EFM in IoT projects to collect telemetry, transform that data and take action upon it. There are many possible actions that such as alerts, reports, dashboards or allowing the data to be used in other applications. Applications include machine learning, Enterprise Resource Management, etc.

This document summarized the major steps for an EFM project, beyond the basic installation and setup, would be the following:

- Install a DSLink or microservice to acquire data. Since the EFM does not communicate directly to sensors, the use of dslinks allows us to communicate with other protocols.
- Define a series of basic data transformational outcomes after data acquisition.
    - Transform the raw data into a canonical form (derived) and publish to the /data on the broker. This is common because the raw input is commonly presented in a format that is not desired for later processing.
    - Subscribe to the derived data as input to allow for aggregation, filtering and persisting, if desired.
- Deploy microservices, including Dataflows and the ParStream Historian, where needed. This allows for edge, fog and datacenter processing.
- Present the data. There are several common options:
    - Use the Dataflow engine to publish data to an application
    - Create a custom microservice that subscribes to the data in the EFM
    - Use a generic link, such as ODBC, to query telemetry in the EFM system
    - Use third party applications to visualize the system data

Obtaining documentation and submitting a service request

For information on obtaining documentation, submitting a service request, and gathering additional information, see the monthly *What's New in Cisco Product Documentation*, which also lists all new and revised Cisco technical documentation, at:

http://www.cisco.com/en/US/docs/general/whatsnew/whatsnew.html

Subscribe to the *What's New in Cisco Product Documentation* as a Really Simple Syndication (RSS) feed and set content to be delivered directly to your desktop using a reader application. The RSS feeds are a free service and Cisco currently supports RSS Version 2.0.