
Flir Documentation

Dianomic Systems

Oct 28, 2021

Contents

1	Quick Start Guide	1
1.1	Introduction to FLIR Bridge	1
1.2	HW setup and getting to know the software	1
1.3	Starting and stopping Flir	2
1.4	Troubleshooting Flir	2
1.5	Running the Flir GUI	3
1.6	Adding a Thermal Camera	4
1.7	Managing Data Sources	7
1.8	Viewing Data	10
1.9	Sending Data to Other Systems	14
1.10	PI Web API OMF Endpoint	15
1.11	EDS OMF Endpoint	18
1.12	OCS OMF Endpoint	20
1.13	PI Connector Relay	22
1.14	Number Format Hints	29
1.15	Integer Format Hints	29
1.16	Type Name Hints	29
1.17	Type Hint	29
1.18	Tag Name Hint	30
1.19	Datapoint Specific Hint	30
1.20	Asset Framework Location Hint	30
1.21	Adding OMF Hints	30
1.22	Backing up and Restoring Flir	31
1.23	Updating the software	31
1.24	Troubleshooting and Support Information	32
2	Processing Data	33
2.1	Why Use Filters?	33
2.2	What Can Be Done?	33
2.3	Where Can it Be Done?	34
2.4	Some Useful Filters	44
3	Flir Architecture	47
3.1	Flir Core	48
3.2	Storage Layer	48
3.3	Southbound Microservices	48
3.4	Northbound Microservices	48

3.5	Filters	49
3.6	Event Engine	49
3.7	REST API	49
3.8	Graphical User Interface	49
4	FLIR Bridge Plugins	51
4.1	South Plugins	51
4.2	North Plugins	53
4.3	Filter Plugins	53
4.4	Notification Rule Plugins	55
4.5	Notification Delivery Plugins	55
5	Securing Flir	57
5.1	Enabling HTTPS Encryption	57
5.2	Requiring User Login	59
5.3	User Management	63
5.4	Certificate Store	66
6	Buffering & Storage	69
6.1	Configuring The Storage Plugin	70
6.2	Installing A PostgreSQL server	72
6.3	SQLite Plugin Configuration	73
7	Tuning Flir	75
7.1	South Service Advanced Configuration	75
8	Notifications Service	79
8.1	Notifications	79
8.2	Installing the Notification Service	81
8.3	Starting The Notification Service	82
8.4	Configuring The Notification Service	83
8.5	Using The Notification Service	85
9	Set Point Control	95
9.1	Control Functions	95
9.2	Control Paths	96
10	Plugin Developer Guide	105
10.1	Plugins	105
10.2	Writing and Using Plugins	108
10.3	South Plugins	112
10.4	South Plugins in C	121
10.5	C++ Support Classes	132
10.6	Hybrid Plugins	137
10.7	North Plugins	138
10.8	Storage Plugins	148
10.9	Filter Plugins	151
10.10	Notification Delivery Plugins	165
10.11	Testing Your Plugin	169
11	REST API Developers Guide	177
11.1	The Flir REST API	177
11.2	Administration API Reference	178
11.3	User API Reference	196

12	Version History	201
12.1	Flir v1	201
13	Kerberos authentication	209
13.1	Introduction	209
13.2	PI-Server as the North endpoint	209
13.3	North plugin	209
13.4	Flir server configuration	210
13.5	Kerberos authentication on RedHat/CentOS	212
14	Plugin Documentation	213
14.1	FLIR Bridge South Plugins	213
14.2	FLIR Bridge North Plugins	319
14.3	FLIR Bridge Filter Plugins	356
14.4	FLIR Bridge Notification Rule Plugins	408
14.5	FLIR Bridge Notification Delivery Plugins	413

1.1 Introduction to FLIR Bridge

FLIR Bridge is a sensor-to-cloud HW/SW solution that enables the flow of temperature data from FLIR thermal cameras to 3rd party cloud applications, a multitude of industrial protocols and large asset management software such as PI System. FLIR Bridge is available as a standard version and Pro version, with the pro version having more powerful hardware. The main software difference is that the standard version supports up to 5 sensors, and the Pro version supports unlimited amounts of sensors.

The FLIR Bridge provides a scalable, secure, robust infrastructure for collecting data from sensors, processing data at the edge and transporting data to historian and other management systems. FLIR Bridge can operate over the unreliable, intermittent and low bandwidth connections often found in IoT applications. FLIR Bridge is implemented as a collection of microservices which include:

Core services, including security, monitoring, and storage Data transformation and alerting services South services: Collect data from sensors and other FLIR Bridge systems North services: Transmit data to historians and other systems Edge data processing applications

1.2 HW setup and getting to know the software

1.2.1 Connecting power

The FLIR Bridge and FLIR Bridge Pro are delivered without a power supply which can be bought separately. To power the bridges; they either need the official FLIR power supplies or a 3rd party power supply fulfilling these requirements:

- FLIR Bridge: 10-30 VDC
- FLIR Bridge Pro: 19-24 VDC

If the FLIR power supplies are used, the white cable should be connected to plus (+) and the black cable to negative (-).

1.2.2 Getting to know the User interface

Once the correct power has been connected, the Bridge could be connected to a display together with a keyboard and mouse.

There should be no login needed, but if root privileges are needed, the following passwords are used at delivery:

- FLIR Bridge: 111111
- FLIR Bridge Pro: MIC-710ai

Once you are logged in to the Ubuntu desktop environment, the Bridge can be setup as any other Ubuntu desktop computer when it comes to Ethernet, WiFi and other software.

The Bridge have a special FLIR software running in the background which has the sole purpose of collecting data from FLIR cameras. To modify this software, open up a web browser, preferably the already installed Chromium browser. In the browser address bar, type “localhost” and press enter.

Now you are in the Bridge GUI which is the control center for adding FLIR cameras and connecting the data to 3rd party systems.

To add a camera, please refer

1.3 Starting and stopping Flir

Flir administration is performed using the “flir” command line utility. You must first ssh into the host system. The Flir utility is installed by default in /usr/local/flir/bin.

The following command options are available:

- **Start:** Start the Flir system
- **Stop:** Stop the Flir system
- **Status:** Lists currently running Flir services and tasks
- **Reset:** Delete all data and configuration and return Flir to factory settings
- **Kill:** Kill Flir services that have not correctly responded to Stop
- **Help:** Describe Flir options

For example, to start the Flir system, open a session to the Flir device and type:

```
/usr/local/flir/bin/flir start
```

1.4 Troubleshooting Flir

Flir logs status and error messages to syslog. To troubleshoot a Flir installation using this information, open a session to the Flir server and type:

```
grep -a 'flir' /var/log/syslog | tail -n 20
```

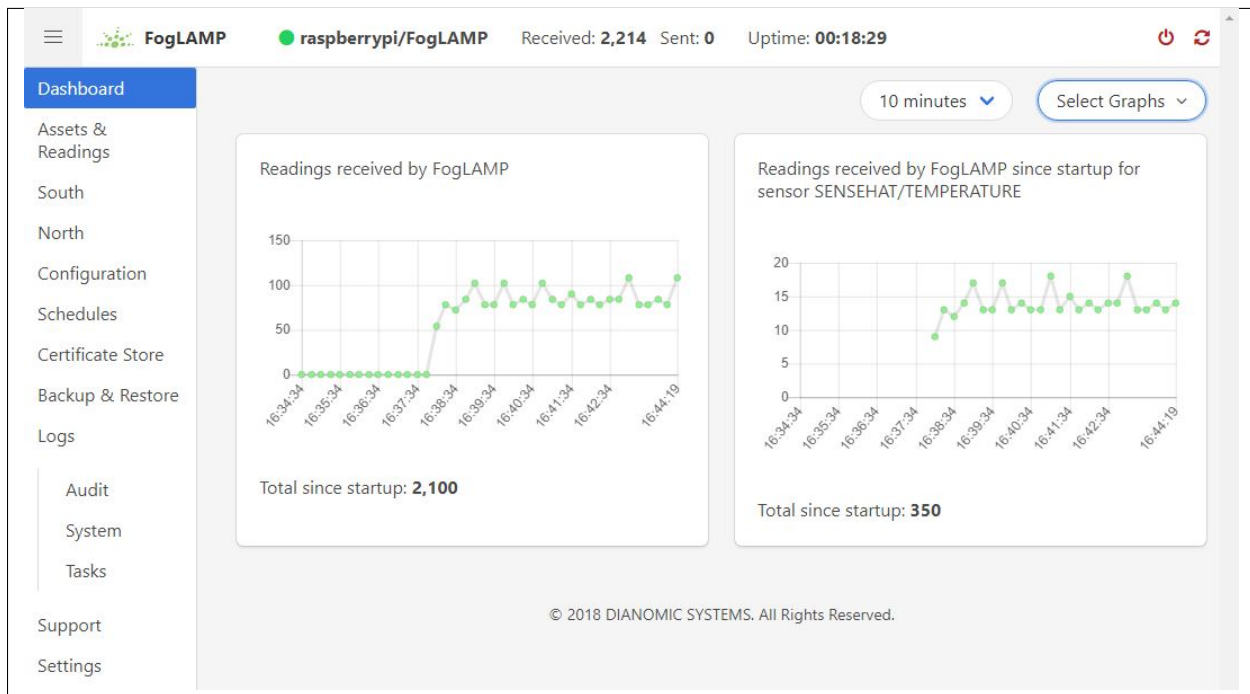
1.5 Running the Flir GUI

Flir offers an easy-to-use, browser-based GUI. To access the GUI, open your browser and enter the IP address of the Flir server into the address bar. This will display the Flir dashboard.

You can easily use the Flir UI to monitor multiple Flir servers. To view and manage a different server, click “Settings” in the left menu bar. In the “Connection Setup” pane, enter the IP address and port number for the new server you wish to manage. Click the “Set the URL & Restart” button to switch the UI to the new server.

If you are managing a very lightweight server or one that is connected via a slow network link, you may want to reduce the UI update frequency to minimize load on the server and network. You can adjust this rate in the “GUI Settings” pane of the Settings screen. While the graph rate and ping rate can be adjusted individually, in general you should set them to the same value.

1.5.1 Flir Dashboard



This screen provides an overview of Flir operations. You can customize the information and time frames displayed on this screen using the drop-down menus in the upper right corner. The information you select will be displayed in a series of graphs.

You can choose to view a graph of any of the sensor reading being collected by the Flir system. In addition, you can view graphs of the following system-wide information:

- **Readings:** The total number of data readings collected by Flir since system boot
- **Buffered:** The number of data readings currently stored by the system
- **Discarded:** Number of data readings discarded before being buffered (due to data errors, for example)
- **Unsent:** Number of data readings that were not sent successfully
- **Purged:** The total number of data readings that have been purged from the system

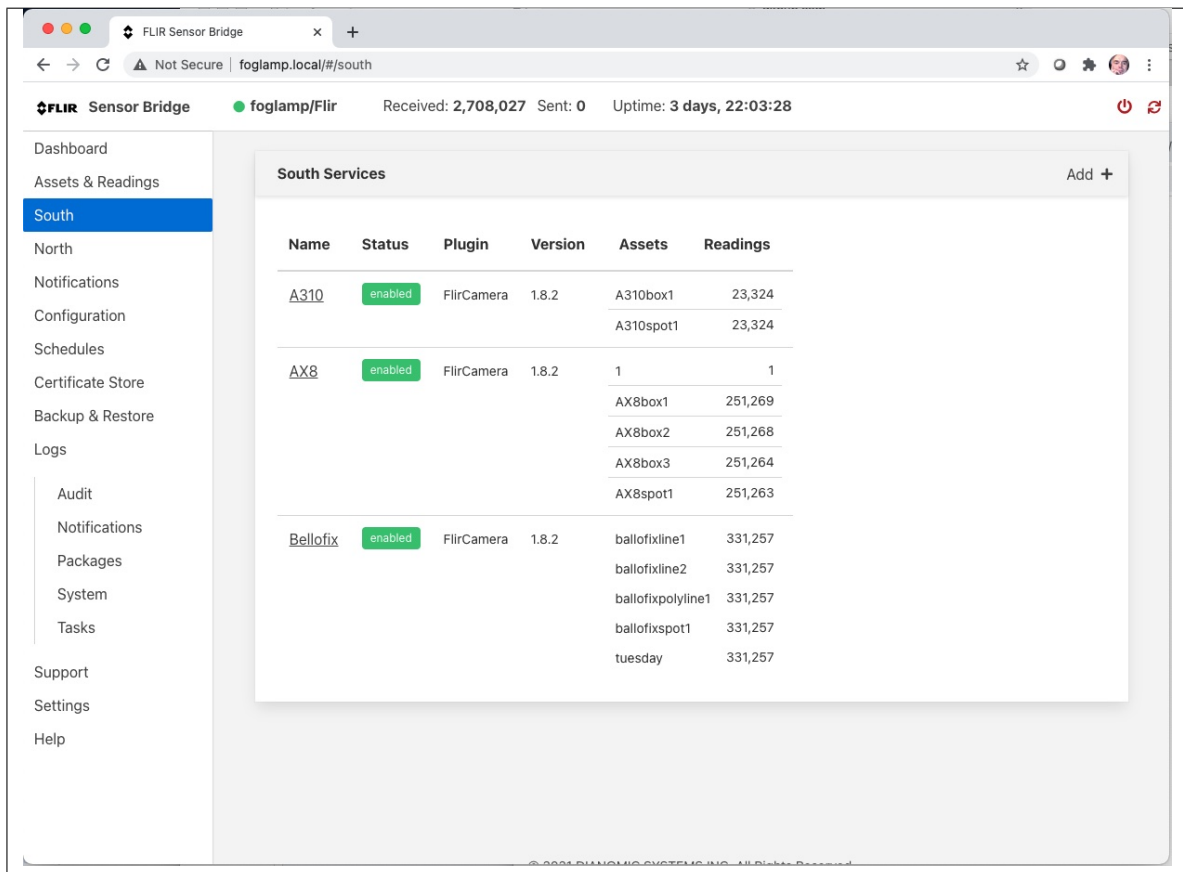
- **Unspurged:** The number of data readings that were purged without being sent to a North service.

1.6 Adding a Thermal Camera

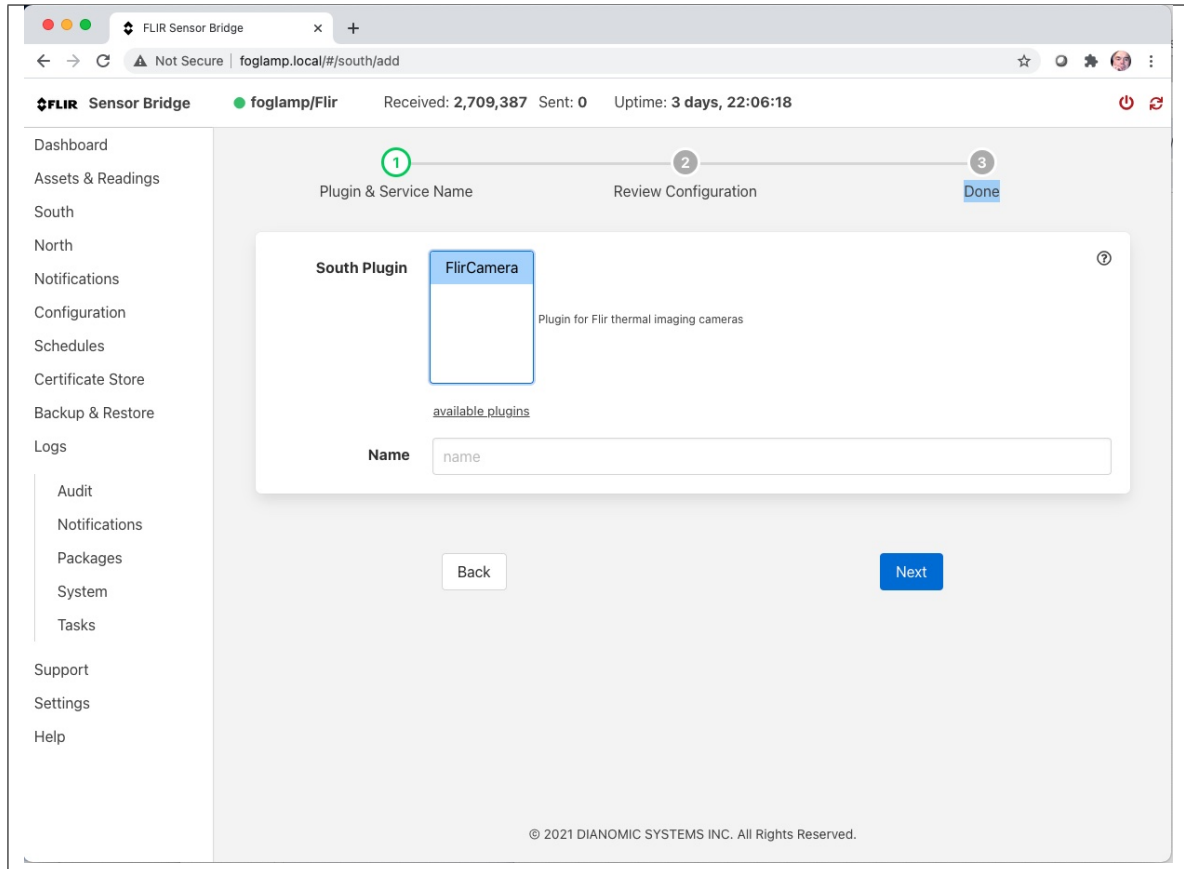
The FLIR Bridge allows thermal data to be captured for several FLIR cameras, processed and sent to data historians, cloud services and industrial systems. The cameras and other sensors that the bridge supports are connected via south services, each south service is configured with a plugin that supports the device from which data is collected, in the case of the FLIR thermal cameras this plugin is known as the *FlirCamera* plugin.

In order to add a camera as a data source using the GUI of the bridge:

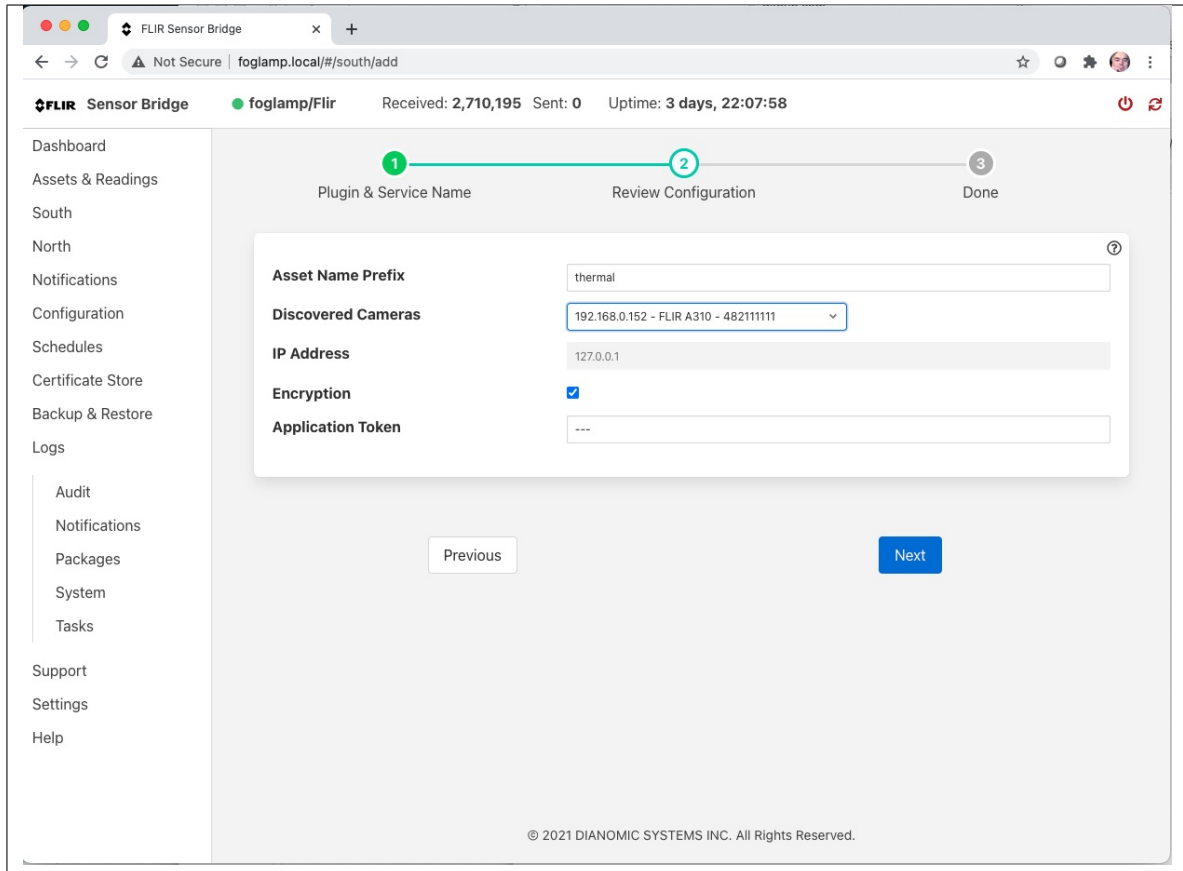
- Select the *South* menu item from the menu pane on the left side of the screen.



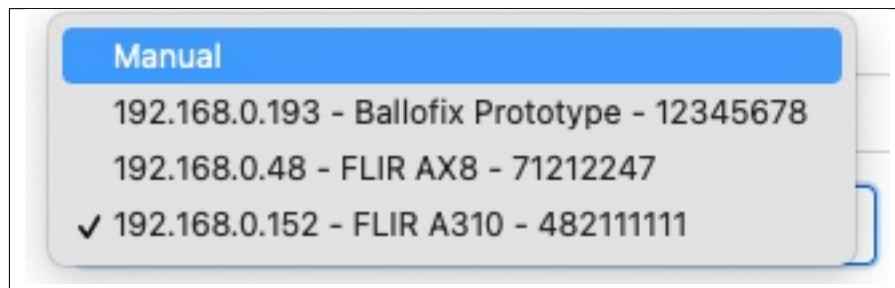
- Click on the *Add +* link in the top right corner. You will be presented with a list of south plugins to choose from. The exact contents of that list will be dependent on the plugins that you have installed on your FLIR Bridge.



- Select the *FlirCamera* entry from the list of South Plugins
- Enter a unique name for your camera device and click on *Next*
- You will be presented with a configuration screen for your south plugin



- The entries on this screen should now be populated
 - **Asset Name Prefix:** This is the name used to prefix data that is collected from the camera. For example if you collect spot1 from your camera and the asset prefix is *tx14* then the asset name used for spot 1's data will be *tx14spot1*. Each camera should have a unique asset prefix.
 - **Discovered Cameras:** The plugin will attempt to automatically discover all the FLIR thermal cameras on your network. These will be presented in this list, you can select the camera you wish to use for this south service. Selecting the entry *Manual* from this list will allow you to enter an IP address manually and can be used if the camera could not be discovered.

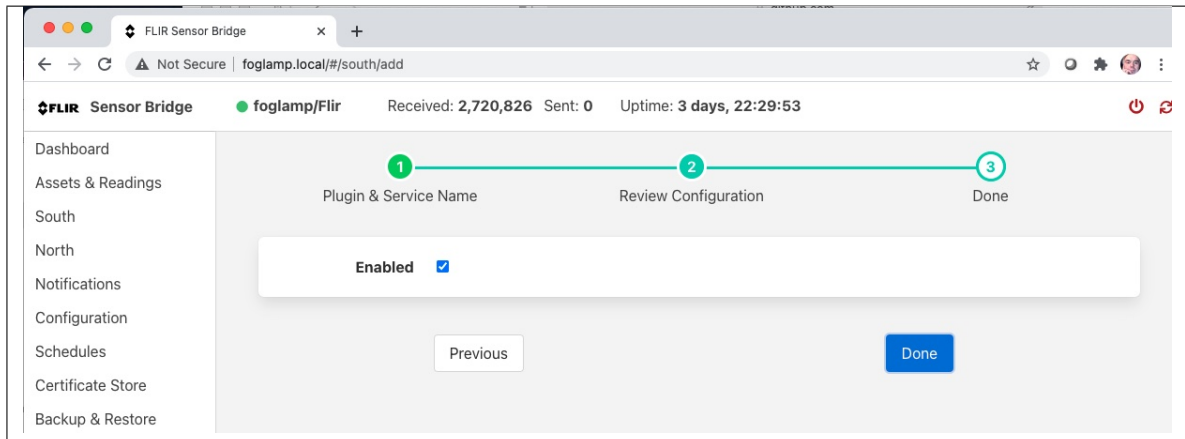


- **IP Address:** This is only used if the option to manually enter an IP address has been selected in the *Discovered Cameras* list. The user should manually enter the address of the camera to use.
- **Encryption:** This toggle control allows the user to control if encryption should be attempted when retrieving data from the camera. Selecting it will cause encryption to be used if the camera supports encryption.

If the camera does not support encryption then the data will be retrieved without encrypting it. Turning this toggle off will cause the data to be always retrieved without encryption.

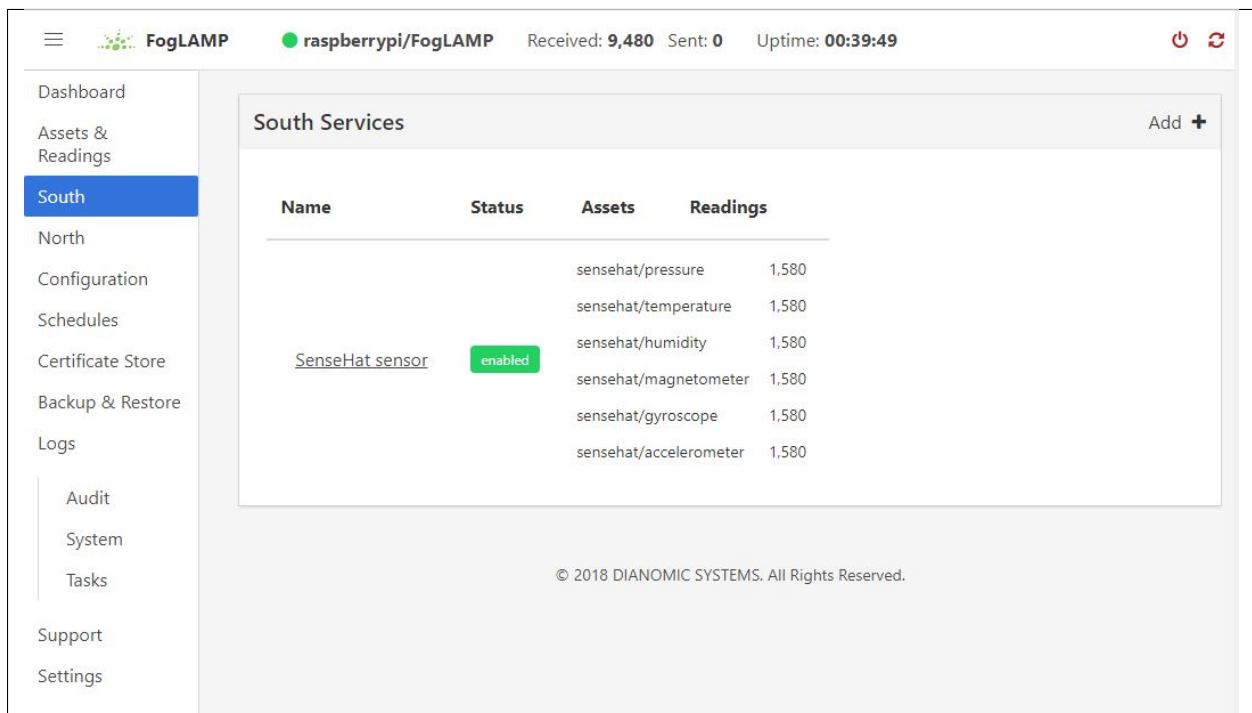
- **Application Token:** Recent cameras allow for the added security of using an application token in order to allow data to be retrieved from the camera. See the section below on creating application tokens. If your camera does not support application tokens then this entry can be left empty.

- Once you have entered the required data in these entry fields click on *Next*



- If you wish to immediately start collecting data then leave the *Enabled* tick box selected and click on *Done*. Deselecting the *Enabled* box will cause the south service to be created but not started, this can be useful if you wish to add some application filters to the data stream.

1.7 Managing Data Sources



Data sources are managed from the South Services screen. To access this screen, click on “South” from the menu bar on the left side of any screen.

The South Services screen displays the status of all data sources in the Flir system. Each data source will display its status, the data assets it is providing, and the number of readings that have been collected.

1.7.1 Adding Data Sources

To add a data source, you will first need to install the plugin for that sensor type. If you have not already done this, open a terminal session to your Flir server. Download the package for the plugin and enter:

```
sudo apt -y install PackageName
```

Once the plugin is installed return to the Flir GUI and click on “Add+” in the upper right of the South Services screen. Flir will display a series of 3 screens to add the data source:

1. The first screen will ask you to select the plugin for the data source from the list of installed plugins. If you do not see the plugin you need, refer to the Installing Flir section of this manual. In addition, this screen allows you to specify a display name for the data source.
2. The second screen allows you to configure the plugin and the data assets it will provide.


Note: Every data asset in Flir must have a unique name. If you have multiple sensors using the same plugin, modify the asset names on this screen so they are unique.

Some plugins allow you to specify an asset name prefix that will apply to all the asset names for that sensor. Refer to the individual plugin documentation for descriptions of the fields on this screen.

3. If you modify any of the configuration fields, click on the “save” button to save them.
4. The final screen allows you to specify whether the service will be enabled immediately for data collection or await enabling in the future.

1.7.2 Configuring Data Sources

SenseHat sensor South Service

pollInterval	<input type="text" value="1000"/>
assetNamePrefix	<input type="text" value="sensehat/"/> 
pressureSensor	<input checked="" type="checkbox"/>
pressureSensorName	<input type="text" value="pressure"/>
temperatureSensor	<input checked="" type="checkbox"/>
temperatureSensorName	<input type="text" value="temperature"/>
humiditySensor	<input checked="" type="checkbox"/>
humiditySensorName	<input type="text" value="humidity"/>
gyroscopeSensor	<input checked="" type="checkbox"/>
gyroscopeSensorName	<input type="text" value="gyroscope"/>
accelerometerSensor	<input checked="" type="checkbox"/>
accelerometerSensorName	<input type="text" value="accelerometer"/>
magnetometerSensor	<input checked="" type="checkbox"/>
magnetometerSensorName	<input type="text" value="magnetometer"/>
joystickSensor	<input checked="" type="checkbox"/>
joystickSensorName	<input type="text" value="joystick"/>
Enabled	<input checked="" type="checkbox"/>
Service Info	<input type="text" value="http://0.0.0.0:39557"/>

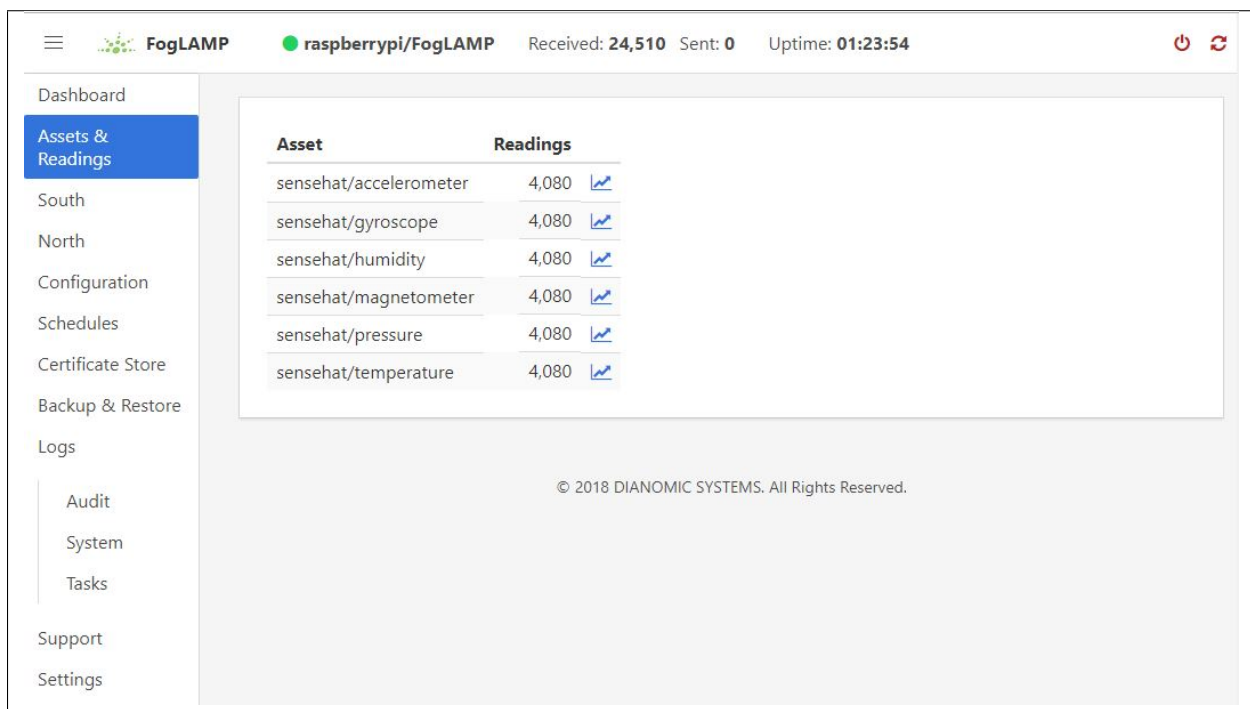
Cancel
Save

To modify the configuration of a data source, click on its name in the South Services screen. This will display a list of all parameters available for that data source. If you make any changes, click on the “save” button in the top panel to save the new configuration. Click on the “x” button in the upper right corner to return to the South Services screen.

1.7.3 Enabling and Disabling Data Sources

To enable or disable a data source, click on its name in the South Services screen. Under the list of data source parameters, there is a check box to enable or disable the service. If you make any changes, click on the “save” button in the bottom panel near the check box to save the new configuration.

1.8 Viewing Data



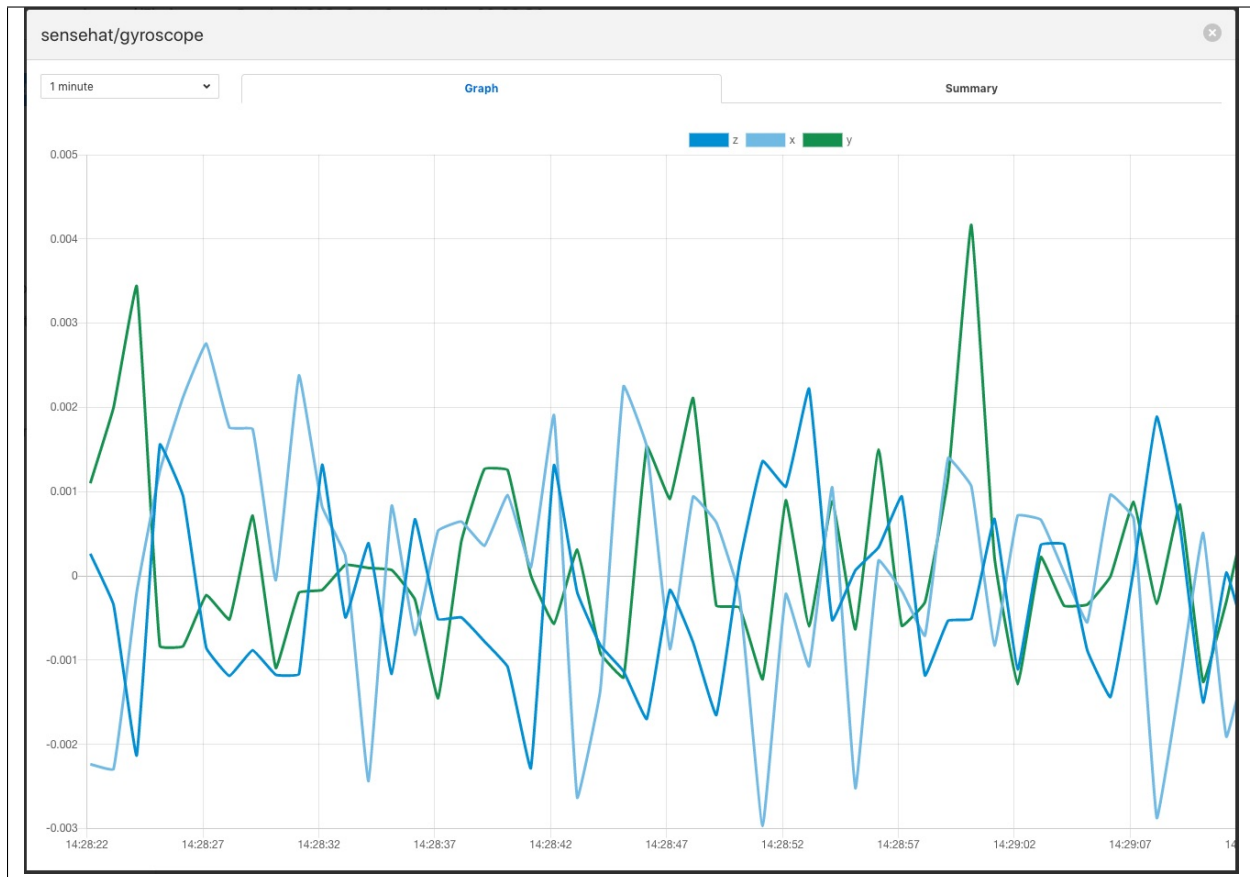
You can inspect all the data buffered by the Flir system on the Assets page. To access this page, click on “Assets & Readings” from the left-side menu bar.

This screen will display a list of every data asset in the system. Alongside each asset are two icons; one to display a graph of the asset and another to download the data stored for that asset as a CSV file.

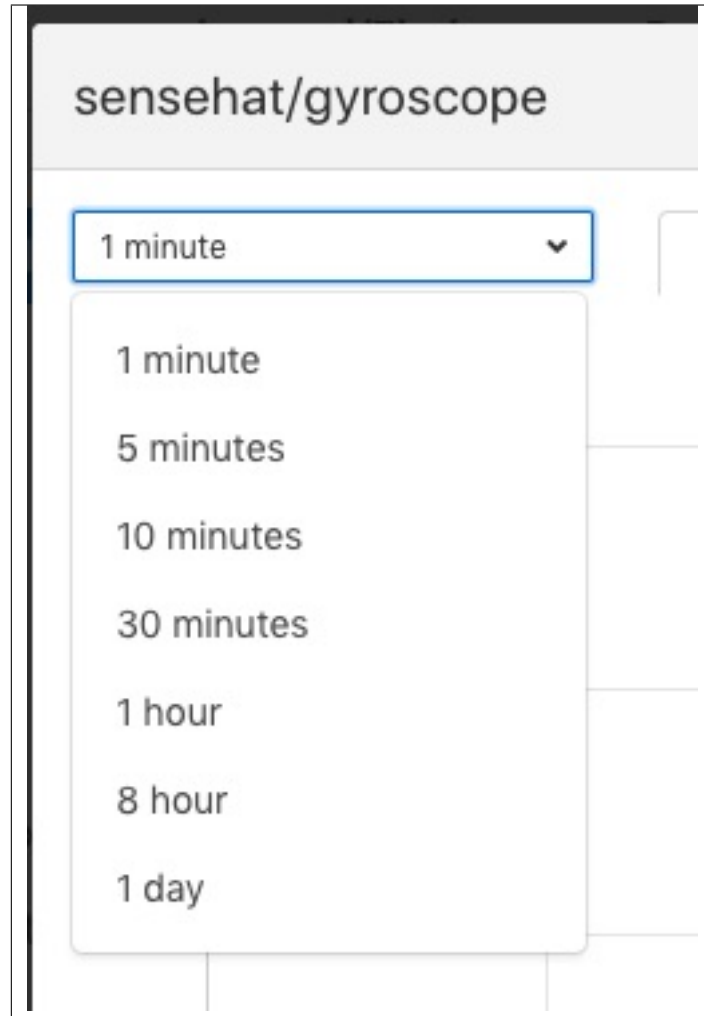
1.8.1 Display Graph



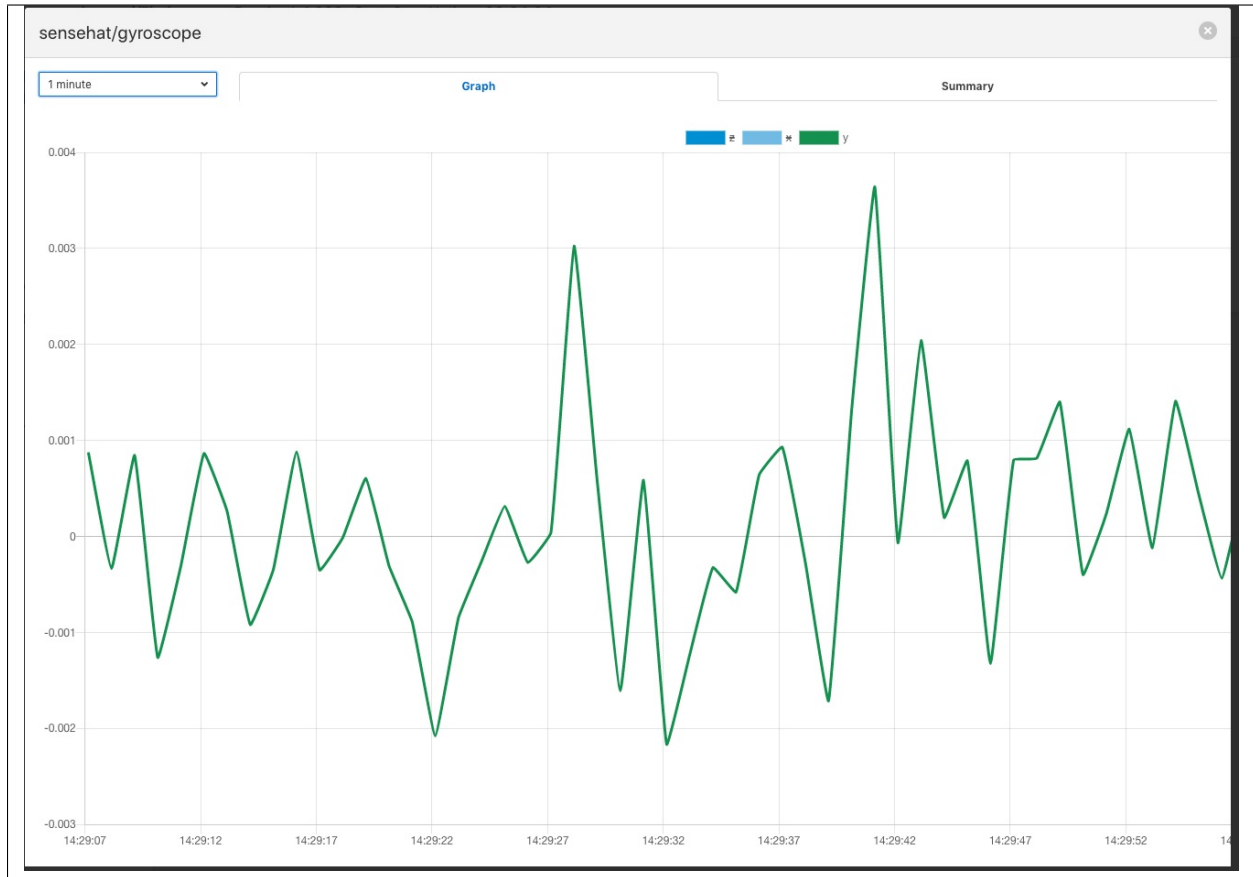
By clicking on the graph button next to each asset name, you can view a graph of individual data readings. A graph will be displayed with a plot for each data point within the asset.



It is possible to change the time period to which the graph refers by use of the plugin list in the top left of the graph.



Where an asset contains multiple data points each of these is displayed in a different colour. Graphs for particular data points can be toggled on and off by clicking on the key at the top of the graph. Those data points not should will be indicated by striking through the name of the data point.



A summary tab is also available, this will show the minimum, maximum and average values for each of the data points. Click on *Summary* to show the summary tab.



1.8.2 Download Data



By clicking on the download icon adjacent to each asset you can download the stored data for the asset. The format of the file is download is a CSV file that is designed to be loaded into a spreadsheet such as Excel, Numbers or OpenOffice Calc.

The file contains a header row with the names of the data points within the asset, the first column is always the timestamp when the reading was taken, the header for this being *timestamp*. The data is sorted in chronological order with the newest data first.

sensehat_gyroscope-readings			
timestamp	z	x	y
2020-05-04 14:30:49.145006	0.000792725	0.0010765493	0.0022465843
2020-05-04 14:30:48.145022	0.0010982286	-0.0004502609	0.000719551
2020-05-04 14:30:47.145006	0.0007928684	0.0032151192	-0.0011130939
2020-05-04 14:30:46.145008	-0.0013448559	0.0047423765	0.0001088944
2020-05-04 14:30:45.145000	-0.0004286431	0.0007723272	-0.0020291833
2020-05-04 14:30:44.144999	-0.0001233947	0.0013834909	0.0007194807
2020-05-04 14:30:43.145001	-0.000734292	-0.0001437888	0.0004143068

1.9 Sending Data to Other Systems

FogLAMP

● raspberry/FogLAMP

Received: 10,974 Sent: 0 Uptime: 00:44:14

Dashboard
Assets & Readings
South
North
Configuration
Schedules
Certificate Store
Backup & Restore
Logs

Audit
System
Tasks

Support
Settings

North Plugins

Create North Instance +

Process	Status	Sent
Plant Librarian	enabled	0

© 2018 DIANOMIC SYSTEMS. All Rights Reserved.

Data destinations are managed from the North Services screen. To access this screen, click on “North” from the menu bar on the left side of any screen.

The North Services screen displays the status of all data sending processes in the Flir system. Each data destination will display its status and the number of readings that have been collected.

1.9.1 Adding Data Destinations

To add a data destination, click on “Create North Instance+” in the upper right of the North Services screen. Flir will display a series of 3 screens to add the data destination:

1. The first screen will ask you to select the plugin for the data destination from the list of installed plugins. If you do not see the plugin you need, refer to the Installing Flir section of this manual. In addition, this screen allows you to specify a display name for the data destination. In addition, you can specify how frequently data will be forwarded to the destination in days, hours, minutes and seconds. Enter the number of days in the interval in the left box and the number of hours, minutes and seconds in format HH:MM:SS in the right box.
2. The second screen allows you to configure the plugin and the data assets it will send. See the section below for specifics of configuring a PI, EDS or OCS destination.
3. The final screen loads the plugin. You can specify whether it will be enabled immediately for data sending or to await enabling in the future.

1.9.2 Configuring Data Destinations

To modify the configuration of a data destination, click on its name in the North Services screen. This will display a list of all parameters available for that data source. If you make any changes, click on the “save” button in the top panel to save the new configuration. Click on the “x” button in the upper right corner to return to the North Services screen.

1.9.3 Enabling and Disabling Data Destinations

To enable or disable a data source, click on its name in the North Services screen. Under the list of data source parameters, there is a check box to enable or disable the service. If you make any changes, click on the “save” button in the bottom panel near the check box to save the new configuration.

1.9.4 Using the OMF plugin

OSISoft data historians are one of the most common destinations for Flir data. Flir supports the full range of OSISoft historians; the PI System, Edge Data Store (EDS) and OSISoft Cloud Services (OCS). To send data to a PI server you may use either the older PI Connector Relay or the newer PI Web API OMF endpoint. It is recommended that new users use the PI Web API OMF endpoint rather than the Connector Relay which is no longer supported by OSISoft.

1.10 PI Web API OMF Endpoint

To use the PI Web API OMF endpoint first ensure the OMF option was included in your PI Server when it was installed.

Now go to the Flir user interface, create a new North instance and select the “OMF” plugin on the first screen. The second screen will request the following information:

Endpoint	PI Web API
Server hostname	localhost
Server port, 0=use the default	0
Producer Token	omf_north_0001
Data Source	readings
Static Data	Location: Palo Alto, Company: Dianomic
Sleep Time Retry	1
Maximum Retry	3
HTTP Timeout	10
Integer Format	int64
Number Format	float64
Compression	<input type="checkbox"/>
Asset Framework hierarchies tree	/fledge/data_piwebapi/default
Asset Framework hierarchies rules	<div><div>1</div><div>{ }</div></div>
PI Web API Authentication Method	anonymous
PI Web API User Id	user_id
PI Web API Password
PI Web API Kerberos keytab file	piwebapi_kerberos_https.keytab

Select PI Web API from the Endpoint options.

- **Basic Information**

- **Endpoint:** Select what you wish to connect to, in this case PI Web API.
- **Send full structure:** Used to control if AF structure messages are sent to the PI server. If this is turned off then the data will not be placed in the asset framework.
- **Naming scheme:** Defines the naming scheme to be used when creating the PI points within the PI Server. See [Naming Scheme](#).
- **Server hostname:** The hostname or address of the PI Server.
- **Server port:** The port the PI Web API OMF endpoint is listening on. Leave as 0 if you are using the default port.
- **Data Source:** Defines which data is sent to the PI Server. The readings or Flir's internal statistics.
- **Static Data:** Data to include in every reading sent to PI. For example, you can use this to specify the location of the devices being monitored by the Flir server.

- **Asset Framework**

- **Asset Framework Hierarchies Tree:** The location in the Asset Framework into which the data will be inserted. All data will be inserted at this point in the Asset Framework unless a later rule overrides this.
- **Asset Framework Hierarchies Rules:** A set of rules that allow specific readings to be placed elsewhere in the Asset Framework. These rules can be based on the name of the asset itself or some metadata associated with the asset. See [Asset Framework Hierarchy Rules](#)

- **PI Web API authentication**

- **PI Web API Authentication Method:** The authentication method to be used, anonymous equates to no authentication, basic authentication requires a user name and password and Kerberos allows integration with your single sign on environment.
- **PI Web API User Id:** The user name to authenticate with the PI Web API.
- **PI Web API Password:** The password of the user we are using to authenticate.
- **PI Web API Kerberos keytab file:** The Kerberos keytab file used to authenticate.

- **Connection management (These should only be changed with guidance from support)**

- **Sleep Time Retry:** Number of seconds to wait before retrying the HTTP connection (Flir doubles this time after each failed attempt).
- **Maximum Retry:** Maximum number of times to retry connecting to the PI server.
- **HTTP Timeout:** Number of seconds to wait before Flir will time out an HTTP connection attempt.

- **Other (Rarely changed)**

- **Integer Format:** Used to match Flir data types to the data type configured in PI. This defaults to int64 but may be set to any OMF data type compatible with integer data, e.g. int32.
- **Number Format:** Used to match Flir data types to the data type configured in PI. The defaults is float64 but may be set to any OMF datatype that supports floating point values.
- **Compression:** Compress the readings data before sending it to the PI System.

1.11 EDS OMF Endpoint

To use the OSISoft Edge Data Store first install Edge Data Store on the same machine as your Flir instance. It is a limitation of Edge Data Store that it must reside on the same host as any system that connects to it with OMF.

Now go to the Flir user interface, create a new North instance and select the “OMF” plugin on the first screen. The second screen will request the following information:

Endpoint	Edge Data Store
Server hostname	localhost
Server port, 0=use the default	0
Producer Token	omf_north_0001
Data Source	readings
Static Data	Location: Palo Alto, Company: Dianomic
Sleep Time Retry	1
Maximum Retry	3
HTTP Timeout	10
Integer Format	int64
Number Format	float64
Compression	<input type="checkbox"/>
Asset Framework hierarchies tree	/fledge/data_piwebapi/default
Asset Framework hierarchies rules	<div>1</div> <div>{ }</div>
PI Web API Authentication Method	anonymous
PI Web API User Id	user_id
PI Web API Password
PI Web API Kerberos keytab file	piwebapi_kerberos_https.keytab
1.11. EDS OMF Endpoint	name_space
OCS Namespace	ocs_tenant_id
OCS Tenant ID	

Select Edge Data Store from the Endpoint options.

- **Basic Information**

- **Endpoint:** Select what you wish to connect to, in this case Edge Data Store.
- **Naming scheme:** Defines the naming scheme to be used when creating the PI points within the PI Server. See [Naming Scheme](#).
- **Server hostname:** The hostname or address of the PI Server. This must be the localhost for EDS.
- **Server port:** The port the Edge Datastore is listening on. Leave as 0 if you are using the default port.
- **Data Source:** Defines which data is sent to the PI Server. The readings or Flir's internal statistics.
- **Static Data:** Data to include in every reading sent to PI. For example, you can use this to specify the location of the devices being monitored by the Flir server.

- **Connection management (These should only be changed with guidance from support)**

- **Sleep Time Retry:** Number of seconds to wait before retrying the HTTP connection (Flir doubles this time after each failed attempt).
- **Maximum Retry:** Maximum number of times to retry connecting to the PI server.
- **HTTP Timeout:** Number of seconds to wait before Flir will time out an HTTP connection attempt.

- **Other (Rarely changed)**

- **Integer Format:** Used to match Flir data types to the data type configured in PI. This defaults to int64 but may be set to any OMF data type compatible with integer data, e.g. int32.
- **Number Format:** Used to match Flir data types to the data type configured in PI. The defaults is float64 but may be set to any OMF datatype that supports floating point values.
- **Compression:** Compress the readings data before sending it to the PI System.

1.12 OCS OMF Endpoint

Go to the Flir user interface, create a new North instance and select the “OMF” plugin on the first screen. The second screen will request the following information:

omf

Endpoint

OSIsoft Cloud Services

Server hostname

pi-server

Server port, 0=use the default

0

Producer Token

uid=5ced49c3-3a55-40e7-983f-c6cdcd5c5fd1&crt=20180620084'

Data Source

readings

Static Data

Location: Palo Alto, Company: Dianomic

Sleep Time Retry

1

Maximum Retry

3

HTTP Timeout

10

Integer Format

int64

Number Format

float64

Compression

Asset Framework hierarchies tree

/fledge/data_piwebapi/default

Asset Framework hierarchies rules

1

{}

PI Web API Authentication Method

anonymous

PI Web API User Id

user_id

PI Web API Password

....

PI Web API Kerberos

piwebapi_kerberos_https.keytab

Select OSIsoft Cloud Services from the Endpoint options.

- **Basic Information**

- **Endpoint:** Select what you wish to connect to, in this case OSIsoft Cloud Services.
- **Naming scheme:** Defines the naming scheme to be used when creating the PI points within the PI Server. See [Naming Scheme](#).
- **Data Source:** Defines which data is sent to the PI Server. The readings or Flir's internal statistics.
- **Static Data:** Data to include in every reading sent to PI. For example, you can use this to specify the location of the devices being monitored by the Flir server.

- **Authentication**

- **OCS Namespace:** Your namespace within the OSIsoft Cloud Services.
- **OCS Tenant ID:** Your OSIsoft Cloud Services tenant ID for your account.
- **OCS Client ID:** Your OSIsoft Cloud Services client ID for your account.
- **OCS Client Secret:** Your OCS client secret.

- **Connection management (These should only be changed with guidance from support)**

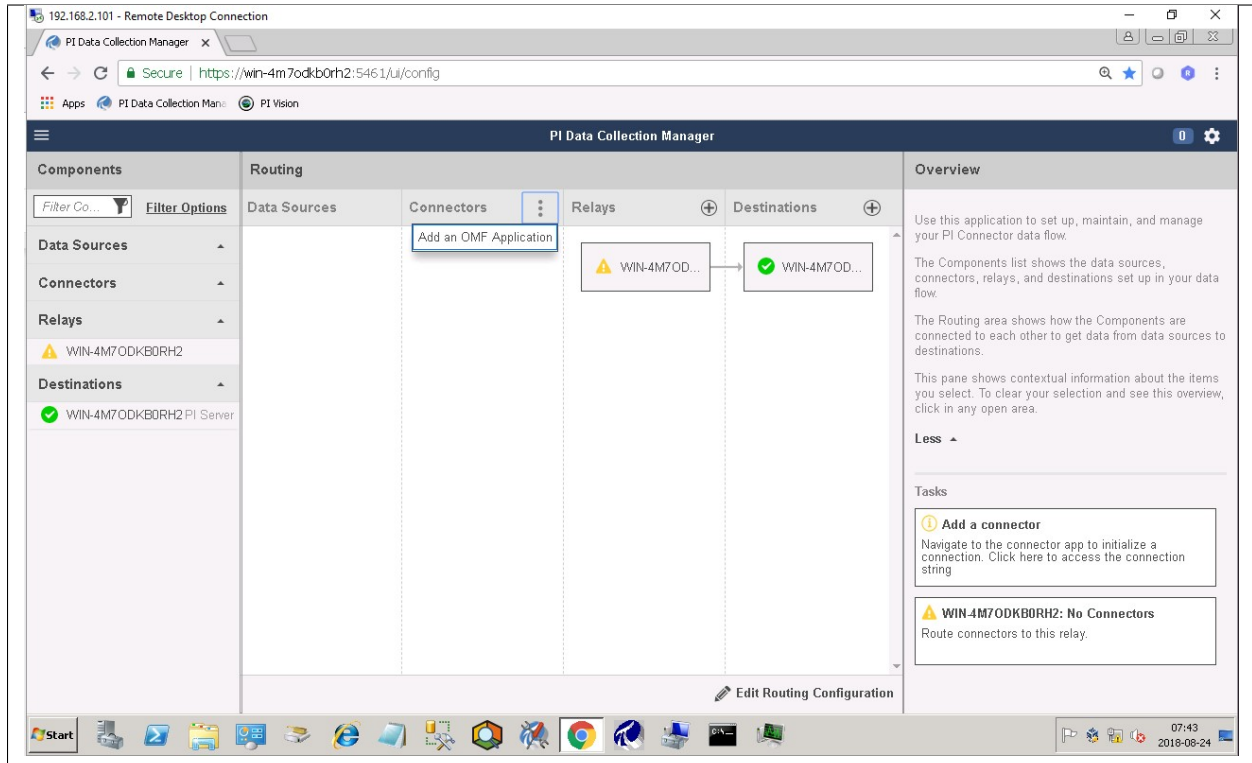
- **Sleep Time Retry:** Number of seconds to wait before retrying the HTTP connection (Flir doubles this time after each failed attempt).
- **Maximum Retry:** Maximum number of times to retry connecting to the PI server.
- **HTTP Timeout:** Number of seconds to wait before Flir will time out an HTTP connection attempt.

- **Other (Rarely changed)**

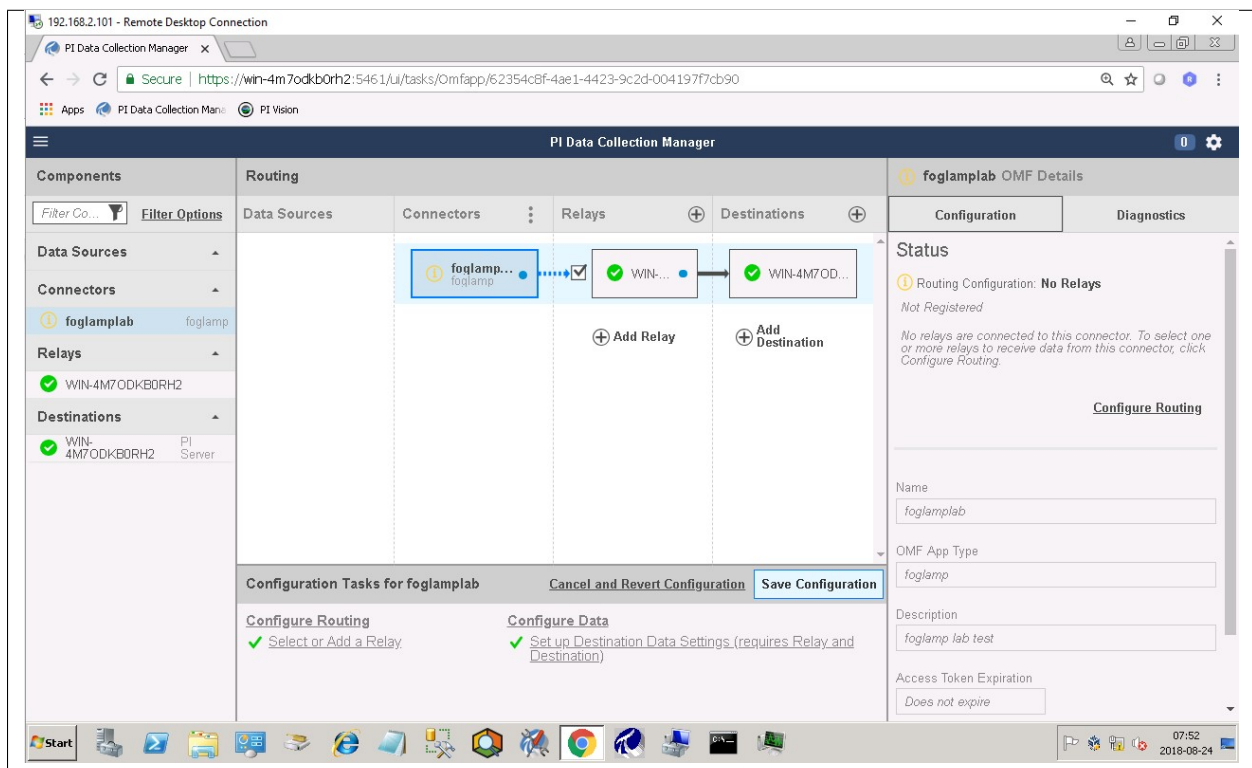
- **Integer Format:** Used to match Flir data types to the data type configured in PI. This defaults to int64 but may be set to any OMF data type compatible with integer data, e.g. int32.
- **Number Format:** Used to match Flir data types to the data type configured in PI. The default is float64 but may be set to any OMF datatype that supports floating point values.
- **Compression:** Compress the readings data before sending it to the PI System.

1.13 PI Connector Relay

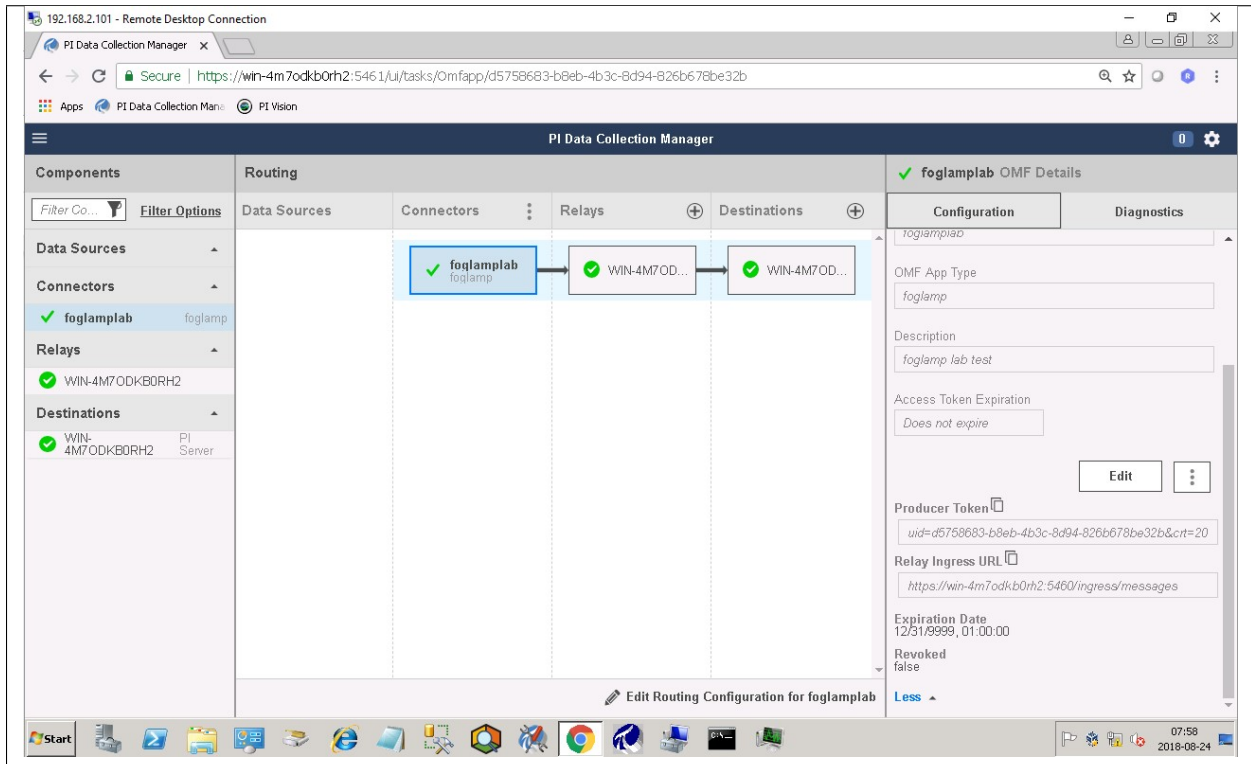
The PI Connector Relay was the original mechanism by which OMF data could be ingested into a PI Server, this has since been replaced by the PI Web API OMF endpoint. It is recommended that all new deployments should use the PI Web API endpoint as the Connector Relay has now been discontinued by OSIsoft. To use the Connector Relay, open and sign into the PI Relay Data Connection Manager.



To add a new connector for the Flir system, click on the drop down menu to the right of “Connectors” and select “Add an OMF application”. Add and save the requested configuration information.



Connect the new application to the OMF Connector Relay by selecting the new Flir application, clicking the check box for the OMF Connector Relay and then clicking “Save Configuration”.



Finally, select the new Flir application. Click “More” at the bottom of the Configuration panel. Make note of the Producer Token and Relay Ingress URL.

Now go to the Flir user interface, create a new North instance and select the “OMF” plugin on the first screen. The second screen will request the following information:

Endpoint	Connector Relay		
Server hostname	localhost		
Server port, 0=use the default	0		
Producer Token	omf_north_0001		
Data Source	readings		
Static Data	Location: Palo Alto, Company: Dianomic		
Sleep Time Retry	1		
Maximum Retry	3		
HTTP Timeout	10		
Integer Format	int64		
Number Format	float64		
Compression	<input type="checkbox"/>		
Asset Framework hierarchies tree	/fledge/data_piwebapi/default		
Asset Framework hierarchies rules	<table border="1"> <tr> <td>1</td> <td>{ }</td> </tr> </table>	1	{ }
1	{ }		
PI Web API Authentication Method	anonymous		
PI Web API User Id	user_id		
PI Web API Password		
PI Web API Kerberos keytab file	piwebapi_kerberos_https.keytab		
1.13. PI Connector Relay OCS Namespace	name_space		
OCS Tenant ID	ocs_tenant_id		

- **Basic Information**

- **Endpoint:** Select what you wish to connect to, in this case the Connector Relay.
- **Server hostname:** The hostname or address of the Connector Relay.
- **Server port:** The port the Connector Relay is listening on. Leave as 0 if you are using the default port.
- **Producer Token:** The Producer Token provided by PI
- **Data Source:** Defines which data is sent to the PI Server. The readings or Flir's internal statistics.
- **Static Data:** Data to include in every reading sent to PI. For example, you can use this to specify the location of the devices being monitored by the Flir server.

- **Connection management (These should only be changed with guidance from support)**

- **Sleep Time Retry:** Number of seconds to wait before retrying the HTTP connection (Flir doubles this time after each failed attempt).
- **Maximum Retry:** Maximum number of times to retry connecting to the PI server.
- **HTTP Timeout:** Number of seconds to wait before Flir will time out an HTTP connection attempt.

- **Other (Rarely changed)**

- **Integer Format:** Used to match Flir data types to the data type configured in PI. This defaults to int64 but may be set to any OMF data type compatible with integer data, e.g. int32.
- **Number Format:** Used to match Flir data types to the data type configured in PI. The defaults is float64 but may be set to any OMF datatype that supports floating point values.
- **Compression:** Compress the readings data before sending it to the PI System.

1.13.1 Naming Scheme

The naming of objects in the asset framework and of the attributes of those objects has a number of constraints that need to be understood when storing data into a PI Server using OMF. An important factor in this is the stability of your data structures. If, in your environment you have objects are liable to change, i.e. the types of attributes change or the number of attributes change between readings, then you may wish to take a different naming approach to if they do not.

This occurs because of a limitation of the OMF interface to the PI server. Data is sent to OMF in a number of stages, one of these is the definition of the types for the AF Objects. OMF let's a type be defined, but once defined it can not be changed. A new type must be created rather than changing the existing type. This means a new asset framework object is created each time a type changes.

The OMF plugin names objects in the asset framework based upon the asset name in the reading within Flir. Asset names are typically added to the readings in the south plugins, however they may be altered by filters between the south ingest and the north egress points in the data pipeline. Asset names can be overridden using the *OMF Hints* mechanism described below.

The attribute names used within the objects in the PI System are based on the names of the data points within each reading within Flir. Again *OMF Hints* can be used to override this mechanism.

The naming used within the objects in the Asset Framework is controlled by the *Naming Scheme* option

Concise No suffix or prefix is added to the asset name and property name when creating the objects in the AF framework and Attributes in the PI server. However if the structure of an asset changes a new AF Object will be created which will have the suffix -type*x* appended to it.

Use Type Suffix The AF Object names will be created from the asset names by appending the suffix -type*x* to the asset name. If and when the structure of an asset changes a new object name will be created with an updated suffix.

Use Attribute Hash Attribute names will be created using a numerical hash as a prefix.

Backward Compatibility The naming reverts to the rules that were used by version 1.9.1 and earlier of Flir, both type suffices and attribute hashes will be applied to the naming.

1.13.2 Asset Framework Hierarchy Rules

The asset framework rules allow the location of specific assets within the PI Asset Framework to be controlled. There are two basic type of hint;

- Asset name placement, the name of the asset determines where in the Asset Framework the asset is placed
- Meta data placement, metadata within the reading determines where the asset is placed in the Asset Framework

The rules are encoded within a JSON document, this document contains two properties in the root of the document; one for name based rules and the other for metadata based rules

```
{
  "names" :
  {
    "asset1" : "/Building1/EastWing/GroundFloor/Room4",
    "asset2" : "Room14"
  },
  "metadata" :
  {
    "exist" :
    {
      "temperature" : "temperatures",
      "power" : "/Electrical/Power"
    },
    "nonexist" :
    {
      "unit" : "Uncalibrated"
    },
    "equal" :
    {
      "room" :
      {
        "4" : "ElecticalLab",
        "6" : "FluidLab"
      }
    },
    "notequal" :
    {
      "building" :
      {
        "plant" : "/Office/Environment"
      }
    }
  }
}
```

The name type rules are simply a set of asset name and AF location pairs. The asset names must be complete names, there is no pattern matching within the names.

The metadata rules are more complex, four different tests can be applied;

- **exists:** This test looks for the existence of the named datapoint within the asset.
- **nonexist:** This test looks for the lack of a named datapoint within the asset.
- **equal:** This test looks for a named data point having a given value.
- **notequal:** This test looks for a name data point having a value different from that specified.

The *exist* and *nonexist* tests take a set of name/value pairs that are tested. The name is the datapoint name to examine and the value is the asset framework location to use. For example

```
"exist" :
{
  "temperature" : "temperatures",
  "power"       : "/Electrical/Power"
}
```

If an asset has a data point called *temperature* it will be stored in the AF hierarchy *temperatures*, if the asset had a data point called *power* the asset will be placed in the AF hierarchy */Electrical/Power*.

The *equal* and *notequal* tests take an object as a child, the name of the object is data point to examine, the child nodes are sets of values and locations. For example

```
"equal" :
{
  "room" :
  {
    "4" : "ElectricalLab",
    "6" : "FluidLab"
  }
}
```

In this case if the asset has a data point called *room* with a value of *4* then the asset will be placed in the AF location *ElectricalLab*, if it has a value of *6* then it is placed in the AF location *FluidLab*.

If an asset matches multiple rules in the ruleset it will appear in multiple locations in the hierarchy, the data is shared between each of the locations.

If an OMF Hint exists within a particular reading this will take precedence over generic rules.

The AF location may be a simple string or it may also include substitutions from other data points within the reading. For example if the reading has a data point called *room* that contains the room in which the readings were taken, an AF location of */BuildingA/\${room}* would put the reading in the asset framework using the value of the room data point. The reading

```
"reading" : {
  "temperature" : 23.4,
  "room"        : "B114"
}
```

would be put in the AF at */BuildingA/B114* whereas a reading of the form

```
"reading" : {
  "temperature" : 24.6,
  "room"        : "2016"
}
```

would be put at the location */BuildingA/2016*.

It is also possible to define defaults if the referenced data point is missing. Therefore in our example above if we used the location `/BuildingA/${room:unknown}` a reading without a `room` data point would be place in `/BuildingA/unknown`. If no default is given and the data point is missing then the level in the hierarchy is ignore. E.g. if we use our original location `/BuildingA/${room}` and we have the reading

```
"reading" : {
  "temperature" : 22.8,
}
```

this reading would be stored in `/BuildingA`.

1.13.3 OMF Hints

The OMF plugin also supports the concept of hints in the actual data that determine how the data should be treated by the plugin. Hints are encoded in a specially name data point within the asset, *OMFHint*. The hints themselves are encoded as JSON within a string.

1.14 Number Format Hints

A number format hint tells the plugin what number format to insert data into the PI Server as. The following will cause all numeric data within the asset to be written using the format *float32*.

```
"OMFHint" : { "number" : "float32" }
```

The value of the *number* hint may be any numeric format that is supported by the PI Server.

1.15 Integer Format Hints

an integer format hint tells the plugin what integer format to insert data into the PI Server as. The following will cause all integer data within the asset to be written using the format *integer32*.

```
"OMFHint" : { "number" : "integer32" }
```

The value of the *number* hint may be any numeric format that is supported by the PI Server.

1.16 Type Name Hints

A type name hint specifies that a particular name should be used when defining the name of the type that will be created to store the object in the Asset Framework. This will override the *Naming Scheme* currently configured.

```
"OMFHint" : { "typeName" : "substation" }
```

1.17 Type Hint

A type hint is similar to a type name hint, but instead of defining the name of a type to create it defines the name of an existing type to use. The structure of the asset *must* match the structure of the existing type with the PI Server, it is the responsibility of the person that adds this hint to ensure this is the case.

```
"OMFHint" : { "type" : "pump" }
```

1.18 Tag Name Hint

Specifies that a specific tag name should be used when storing data in the PI server.

```
"OMFHint" : { "tagName" : "AC1246" }
```

1.19 Datapoint Specific Hint

Hints may also be targeted to specific data points within an asset by using the datapoint hint. A *datapoint* hint takes a JSON object as it's value, this object defines the name of the datapoint and the hint to apply.

```
"OMFHint" : { "datapoint" : { "name" : "voltage:", "number" : "float32" } }
```

The above hint applies to the datapoint *voltage* in the asset and applies a *number format* hint to that datapoint.

1.20 Asset Framework Location Hint

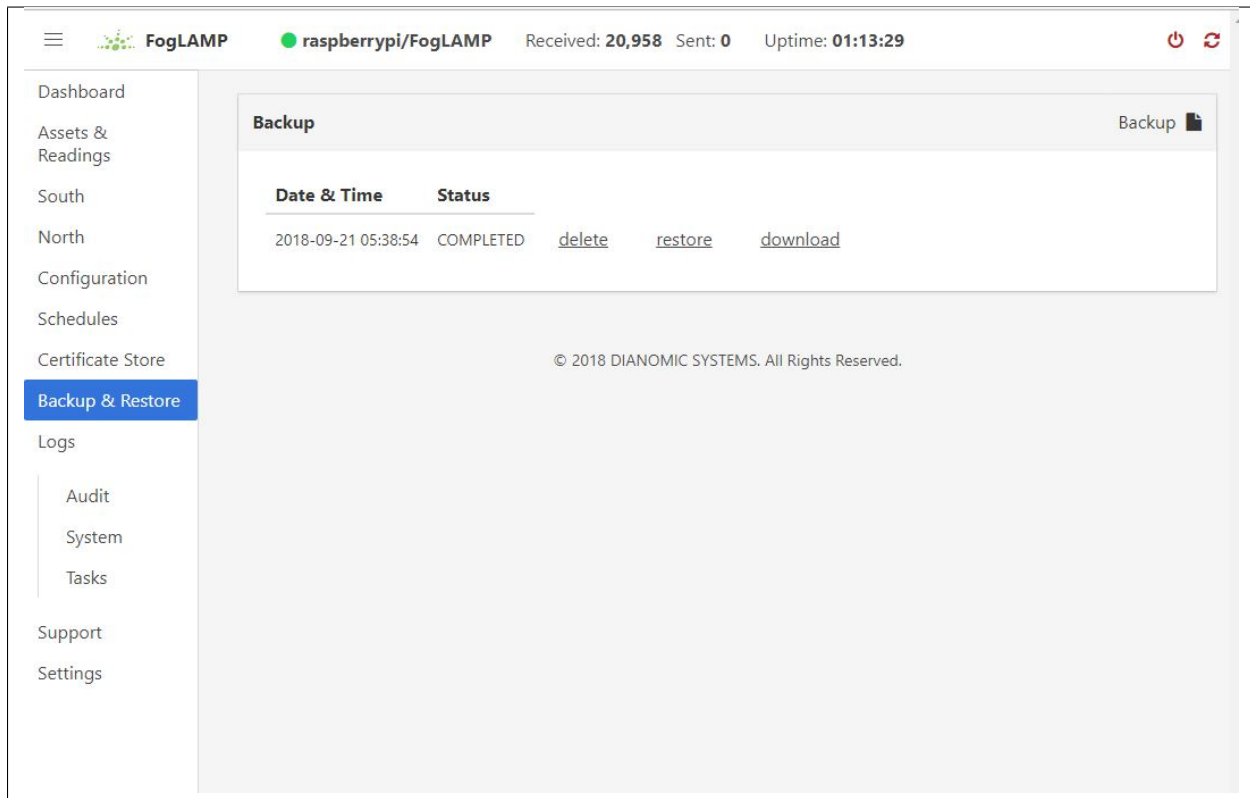
An asset framework location hint can be added to a reading to control the placement of that asset within the Asset Framework. An asset framework hint would be as follow

```
"OMFHint" : { "AFLocation" : "/UK/London/TowerHill/Floor4" }
```

1.21 Adding OMF Hints

An OMF Hint is implemented as a string data point on a reading with the data point name of *OMFHint*. It can be added at any point in the processing of the data, however a specific plugin is available for adding the hints, the .

1.22 Backing up and Restoring Flir



You can make a complete backup of all Flir data and configuration. To do this, click on “Backup & Restore” in the left menu bar. This screen will show a list of all backups on the system and the time they were created. To make a new backup, click the “Backup” button in the upper right corner of the screen. You will briefly see a “Running” indicator in the lower left of the screen. After a period of time, the new backup will appear in the list. You may need to click the refresh button in the upper left of the screen to refresh the list. You can restore, delete or download any backup simply by clicking the appropriate button next to the backup in the list.

1.23 Updating the software

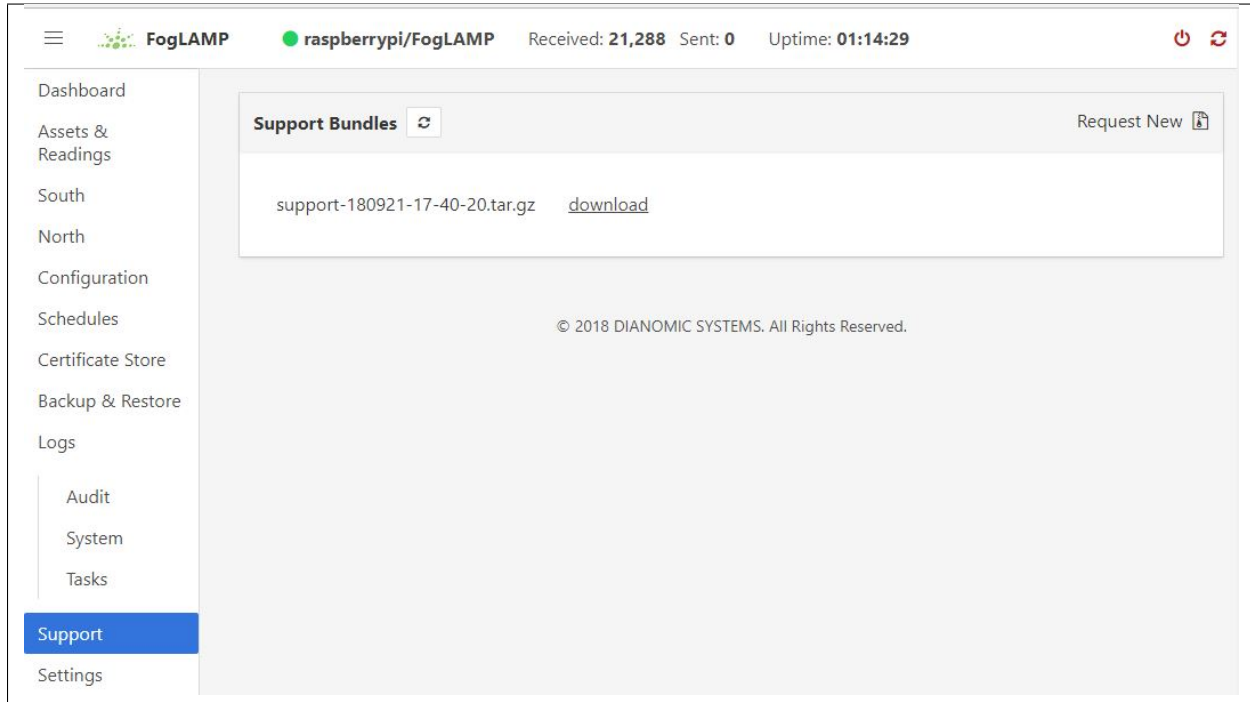
The FLIR Bridge software is installed using the Linux package manager *apt*. To update the software to the latest version you should login to the Bridge device using *ssh* or by connecting a keyboard and monitor. You may then update using the *apt* commands

```
apt update
apt upgrade
```

This will cause the latest software to be downloaded from the central software repository that holds the FLIR bridge software. It will also update the Linux software on your bridge device. During the process you will be asked to confirm the required actions.

Please note the bridge device should be connected to the network and have access to the internet using the standard http ports.

1.24 Troubleshooting and Support Information



FLIR Bridge keeps detailed logs of system events for both auditing and troubleshooting use. To access them, click “Logs” in the left menu bar. There are five logs in the system:

- **Audit:** Tracks all configuration changes and data uploads performed on the FLIR Bridge system.
- **Notifications:** If you are using the FLIR Bridge notification service this log will give details of notifications that have been triggered
- **Packages:** This log will give you information about the installation and upgrade of FLIR Bridge packages for services and plugins.
- **System:** All events and scheduled tasks and their status.
- **Tasks:** The most recent scheduled tasks that have run and their status

Do not hesitate to contact our Customer Support Center if you experience problems or have any questions.

For customer support help go to .

We have already seen that Flir can collect data from a variety of sources, buffer it locally and send it on to one or more destination systems. It is also possible to process the data within Flir to edit, augment or remove data as it traverses the Flir system. In the same way Flir makes extensive use of plugin components to add new sources of data and new destinations for that data, Flir also uses plugins to add processing filters to the Flir system.

2.1 Why Use Filters?

The concept behind filters is to create a set of small, useful pieces of functionality that can be inserted into the data flow from the south data ingress side to the north data egress side. By making these elements small and dedicated to a single task it increases the re-usability of the filters and greatly improves the chances when a new requirement is encountered that it can be satisfied by creating a filter pipeline from existing components or by augmenting existing components with the addition of any incremental processing required. The ultimate aim being to be able to create new applications within Flir by merely configuring filters from the existing pool of available filters into a suitable pipeline without the need to write any new code.

2.2 What Can Be Done?

Data processing is done via plugins that are known as *filters* in Flir, therefore it is not possible to give a definitive list of all the different processing that can occur, the design intent is that it is expandable by the user. The general types of things that can be done are;

- **Modify a value in a reading.** This could be as simple as applying a scale factor to convert from one measurement scale to another or more complex mathematical operation.
- **Modify asset or datapoint names.** Perform a simple textual substitution in order to change the name of an asset or a data point within that asset.
- **Add a new calculated value.** A new value can be calculated from a set of values, either based over a time period or based on a combination of different values, e.g. calculate power from voltage and current.

- **Add metadata to an asset.** This allows data such as units of measurement or information about the data source to be added to the data.
- **Compress data.** Only send data forward when the data itself shows significant change from previous values. This can be a useful technique to save bandwidth in low bandwidth or high cost network connections.
- **Conditionally forward data.** Only send data when a condition is satisfied or send low rate data unless some *interesting* condition is met.
- **Data conditioning.** Remove data from the data stream if the values are suspect or outside of reasonable conditions.

2.3 Where Can it Be Done?

Filters can be applied in two locations in the Flir system;

- In the south service as data arrives in Flir and before it is added to the storage subsystem for buffering.
- In the north tasks as the data is sent out to the upstream systems that receive data from the Flir system.

More than one filter can be added to a single south or north within a Flir instance. Filters are placed in an ordered pipeline of filters that are applied to the data in the order of the pipeline. The output of the first filter becomes the input to the second. Filters can thus be combined to perform complex sets of operations on a particular data stream into Flir or out of Flir.

The same filter plugin can appear in multiple places within a filter pipeline, a different instance is created for each and each one has its own configuration.

2.3.1 Adding a South Filter

In the following example we will add a filter to a south service. The filter we will use is the *expression* filter and we will convert the incoming value to a logarithmic scale. The south plugin used in this simple example is the *sinusoid* plugin that creates a simulated sine wave.

The process starts by selecting the *South* services in the Flir GUI from the left-hand menu bar. Then click on the south service of interest. This will display a dialog that allows the south service to be edited.

Sine South Service

Asset name

sinusoid

Enabled

☒

Show Advanced Config

Applications +

Cancel

Save

Service Info

Export Readings

Delete Service

Towards the bottom of this dialog is a section labeled *Applications* with a + icon to the right, select the + icon to add a filter to the south service. A filter wizard is now shown that allows you to select the filter you wish to add and give that filter a name.

Sine South Service

1 Plugin Name 2 Review Configuration

Plugin

- asset
- change
- delta
- expression**

Apply an expression to the data stream

[Install from available plugins](#)

Name

Back Next

Select the *expression* filter and enter a name in the dialog. Now click on the *Next* button. A new page in the wizard appears that allows the configuration of the filter.

Sine South Service

1 Plugin Name 2 Review Configuration

Datapoint Name LogSine

Expression to apply log(sinusoid)

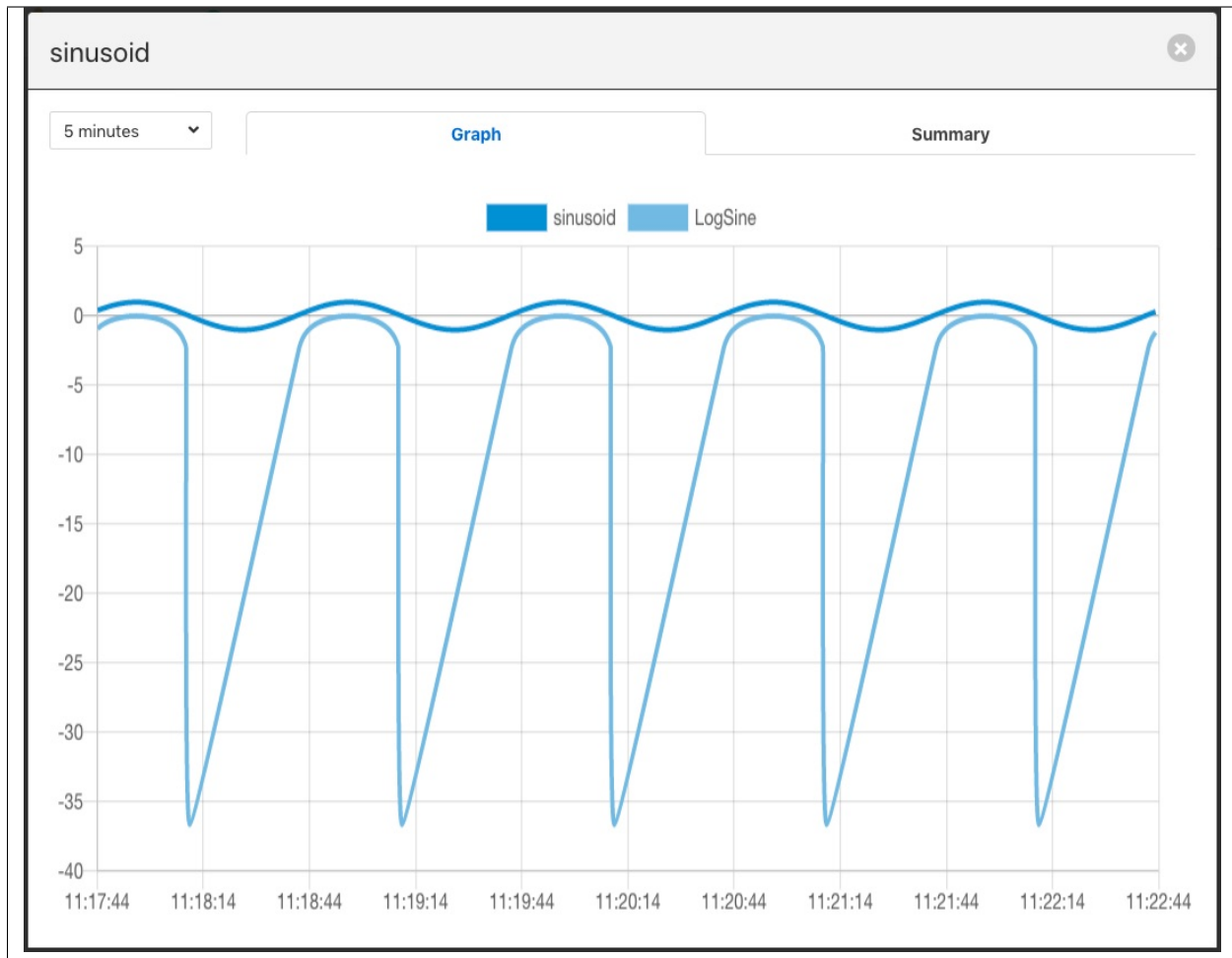
Enabled ☒

Previous Done

In the case of our expression filter we should add the expression we wish to execute *log(sinusoid)* and the name of the datapoint we wish to put the result in, *LogSine*. We can also choose to enable or disable the execution of this filter. We will enable it and click on *Done* to complete adding the filter.

Click on *Save* in the south edit dialog and our filter is now installed and running.

If we select the *Assets & Readings* option from the menu bar we can examine the sinusoid asset and view a graph of that asset. We will now see a second datapoint has been added, *LogSine* which is the result of executing our expression in the filter.



A second filter can be added in the same way, for example a *metadata* filter to create a pipeline. Now when we go back and view the south service we see two applications in the dialog.

Sine South Service

Asset name

sinusoid

Enabled

☒

[Show Advanced Config](#)

Applications +

≡ MyExpression

▼

≡ Location

▼

Cancel

Save

Service Info

http://localhost:37799

Export Readings

Delete Service

Reordering Filters

The order in which the filters are applied can be changed in the south service dialog by clicking and dragging one filter above another in the *Applications* section of dialog.

Sine South Service

Asset name sinusoid

Enabled ☒ [Show Advanced Config](#)

Applications +

- Location
- MyExpression

Cancel Save

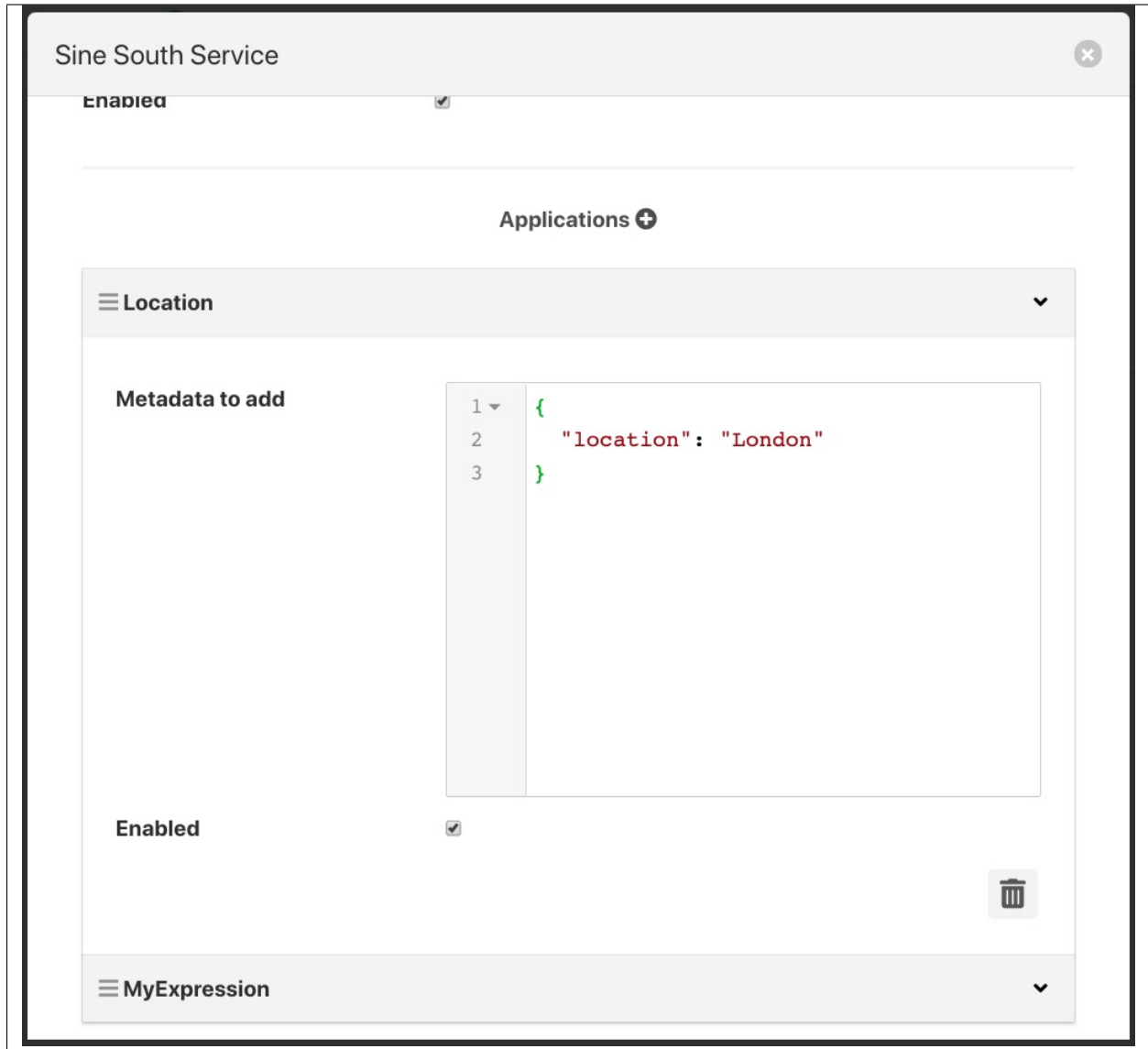
Service Info http://localhost:37799

Export Readings Delete Service

Filters are executed in a top to bottom order always. It may not matter in some cases what order a filter is executed in, in others it can have significant effect on the result.

Editing Filter Configuration

A filters configuration can be altered from the south service dialog by selecting the down arrow to the right of the filter name. This will open the edit area for that filter and show the configuration that can be altered.



You can also remove a filter from the pipeline of filters by select the trash can icon at the bottom right of the edit area for the filter.

2.3.2 Adding Filters To The North

Filters can also be added to the north in the same way as the south. The same set of filters can be applied, however some may be less useful in the north than in the south as they apply to all assets that are sent north.

In this example we will use the metadata filter to label all the data that goes north as coming via a particular Flir instance. As with the *South* service we start by selecting our north task from the *North* menu item in the left-hand menu bar.

PI Server

PI Web API Password

....

PI Web API Kerberos keytab file

piwebapi_kerberos_https.keytab

OCS Namespace

name_space

OCS Tenant ID

ocs_tenant_id

OCS Client ID

ocs_client_id

OCS Client Secret

....

Enabled

☒

Exclusive

☒

Interval

0

00:00:30

Show Advanced Config

Applications +

Cancel

Save

Delete Instance

At the bottom of the dialog there is a *Applications* area, you may have to scroll the dialog to find it, click on the + icon. A selection dialog appears that allows you to select the filter to use. Select the *metadata* filter.

The screenshot shows a web interface titled "PI Server" with a close button in the top right corner. A progress bar at the top indicates two steps: "1 Plugin Name" (highlighted with a green circle) and "2 Review Configuration". The main content area contains a "Plugin" dropdown menu with options: "fft", "FlirValidity" (with a tooltip "Metadata filter plugin"), "metadata" (highlighted), and "python27". Below the dropdown is a link "Install from available plugins". A "Name" input field contains the text "Floor". At the bottom, there are "Back" and "Next" buttons.

After clicking *Next* you will be shown the configuration page for the particular filter you have chosen. We will edit the JSON that defines the metadata tags to add and set a name of *floor* and a value of *1*.

PI Server

1 Plugin Name 2 Review Configuration

Metadata to add

```
1 {  
2   "floor": "1"  
3 }
```

Enabled ☒

Previous Done

After enabling and clicking on *Done* we save the north changes. All assets sent to this PI Server connection will now be tagged with the tag “floor” and value “1”.

Although this is a simple example of labeling data other things can be done here, such as limiting the rate we send data to the PI Server until an *interesting* condition becomes true, perhaps to save costs on an expensive link or prevent a network becoming loaded until normal operating conditions. Another option might be to block particular assets from being sent on this link, this could be useful if you have two destinations and you wish to send a subset of assets to each.

This example used a PI Server as the destination, however the same mechanism and filters may be used for any north destination.

2.4 Some Useful Filters

A number of simple filters are worthy of mention here, a complete list of the currently available filters in Flir can be found in the section .

2.4.1 Scale

The filter *flir-filter-scale* applies a scale factor and offset to the numeric values within an asset. This is useful for operations such as changing the unit of measurement of a value. An example might be to convert a temperature reading from Centigrade to Fahrenheit.

2.4.2 Metadata

The filter *flir-filter-metadata* will add metadata to an asset. This could be used to add information such as unit of measurement, machine data (make, model, serial no) or the location of the asset to the data.

2.4.3 Delta

The filter *flir-filter-delta* allows duplicate data to be removed, only forwarding data that changes by more than a configurable percentage. This can be useful if a value does not change often and there is a desire not to forward all the *similar* values in order to save network bandwidth or reduce storage requirements.

2.4.4 Rate

The filter *flir-filter-rate* is similar to the delta filter above, however it forwards data at a fixed rate that is lower the rate of the oncoming data but can send full rate data should an *interesting* condition be detected. The filter is configured with a rate to send data, the values sent at that rate are an average of the values seen since the last value was sent.

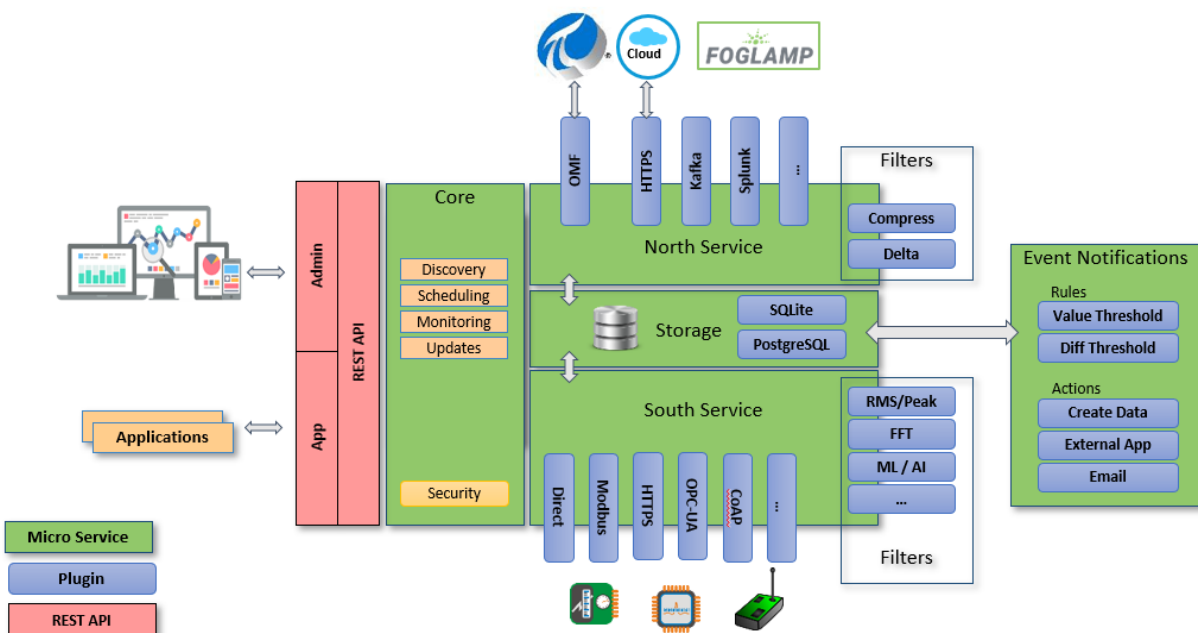
A rate of one reading per minute for example would average all the values for 1 minute and then send that average as the reading at the end of that minute. A condition can be added, when that condition is triggered all data is forwarded at full rate of the incoming data until a further condition is triggered that causes the reduced rate to be resumed.

CHAPTER 3

Flir Architecture

The following diagram shows the architecture of Flir:

- Components in blue are **plugins**. Plugins are light-weight modules that enable Flir to be extended. There are a variety of types of plugins: south-facing, north-facing, storage engine, filters, event rules and event delivery mechanisms. Plugins can be written in python (for fast development) or C++ (for high performance).
- Components in green are **microservices**. They can co-exist in the same operating environment or they can be distributed across multiple environments.



3.1 Flir Core

The Core microservice coordinates all of the Flir operations. Only one Core service can be active at any time.

Core functionality includes:

Scheduler: Flexible scheduler to bring up processes.

Configuration Management: maintain configuration of all Flir components. Enable software updates across all Flir components.

Monitoring: monitor all Flir components, and if a problem is discovered (such as an unresponsive microservice), attempt to self-heal.

REST API: expose external management and data APIs for functionality across all components.

Backup: Flir system backup and restore functionality.

Audit Logging: maintain logs of system changes for auditing purposes.

Certificate Storage: maintain security certificates for different components, including south services, north services, and API security.

User Management: maintain authentication and permission info on Flir administrators.

Asset Browsing: enable querying of stored asset data.

3.2 Storage Layer

The Storage microservice provides two principal functions: a) maintenance of Flir configuration and run-time state, and b) storage/buffering of asset data. The type of storage engine is pluggable, so in installations with a small footprint, a plugin for SQLite may be chosen, or in installations with a high number of concurrent requests and larger footprint Postgresql may be suitable. In micro installations, for example on Edge devices, in-memory temporary storage may be the best option.

3.3 Southbound Microservices

Southbound microservices offer bi-directional communication of data and metadata between Edge devices, such as sensors, actuators or PLCs and Flir. Smaller systems may have this service installed onboard Edge devices. Southbound components are typically deployed as always-running services, which continuously wait for new data. Alternatively, they can be deployed as single-shot tasks, which periodically spin up, collect data and spin down.

3.4 Northbound Microservices

Northbound microservices offer bi-directional communication of data and metadata between the Flir platform and larger systems located locally or in the cloud. Larger systems may be private and public Cloud data services, proprietary solutions or Flir instances with larger footprints. Northbound components are typically deployed as one-shot tasks, which periodically spin up and send data which has been batched, then spin down. However, they can also be deployed as continually-running services.

3.5 Filters

Filters are plugins which modify streams of data that flow through Flir. They can be deployed at ingress (in a South service), or at egress (in a North service). Typically, ingress filters are used to transform or enrich data, and egress filters are used to reduce flow to northbound pipes and infrastructure, i.e. by compressing or reducing data that flows out. Multiple filters can be applied in “pipelines”, and once configured, pipelines can be applied to multiple south or north services.

A sample of existing Filters:

Expression: apply an arbitrary mathematical equation across one or more assets.

Python35: run user-specified python code across one or more assets.

Metadata: apply tags to data, to note the device/location it came from, or to attribute data to a manufactured part.

RMS/Peak: summarize vibration data by generating a Root Mean Squared (RMS) across n samples.

FFT: generate a Fast Fourier Transform (FFT) of vibration data to discover component waveforms.

Delta: Only send data that has changed by a specified amount.

Rate: buffer data but don't send it, then if an error condition occurs, send the previous data.

3.6 Event Engine

The event engine maintains zero or more rule/action pairs. Each rule subscribes to desired asset data, and evaluates it. If the rule triggers, its associated action is executed.

Data Subscriptions: Rules can evaluate every data point for a specified asset, or they can evaluate the minimum, maximum or average of a specified window of data points.

Rules: the most basic rule evaluates if values are over/under a specified threshold. The Expression plugin will evaluate an arbitrary math equation across one or more assets. The Python35 plugin will execute user-specified python code to one or more assets.

Actions: A variety of delivery mechanisms exist to execute a python application, or create arbitrary data, or email/slack/hangout/communicate a message.

3.7 REST API

The Flir API provides methods to administer Flir, and to interact with the data inside it.

3.8 Graphical User Interface

A GUI enables administration of Flir. All GUI capability is through the REST API, so Flir can also be administered through scripts or other management tools. The GUI contains pages to:

Health: See if services are responsive. See data that's flowed in and out of Flir

Assets & Readings: analytics of data in Flir

South: manage south services

North: manage north services

Notifications: manage event engine rules and delivery mechanisms

Configuration Management: manage configuration of all components

Schedules: flexible scheduler management for processes and tasks

Certificate Store: manage certificates

Backup & Restore: backup/restore Flir

Logs: see system, notification, audit, packages and tasks logging information

Support: support bundle contents with system diagnostic reports

Settings: set/reset connection and GUI related settings

FLIR Bridge Plugins

The following set of plugins are available for FLIR Bridge. These plugins extend the functionality by adding new sources of data, new destinations, processing filters that can enhance or modify the data, rules for notification delivery and notification delivery mechanisms.

4.1 South Plugins

South plugins add new ways to get data into FLIR Bridge, a number of south plugins are available ready built or users may add new south plugins of their own by writing them in Python or C/C++.

Table 1: FLIR Bridge South Plugins

Name	Description
abb	A south plugin to pull data from the ABB cloud
am2315	FLIR Bridge south plugin for an AM2315 temperature and humidity sensor
b100-modbus-python	A south plugin to read data from a Dynamic Ratings B100 device over Modbus
beckhoff	A Beckhoff ADS data ingress plugin for FLIR Bridge, this monitors Beckhoff PLCs and returns the state of internal variables within the PLC
benchmark	A FLIR Bridge benchmark plugin to measure the ingestion rates on particular hardware
cc2650	A FLIR Bridge south plugin for the Texas Instruments SensorTag CC2650
coap	A south plugin for FLIR Bridge that pulls data from a COAP sensor
coral-enviro	A south plugin for the Google Coral Environmental Sensor Board
csv	A FLIR Bridge south plugin in C++ for reading CSV files
csv-async	A FLIR Bridge asynchronous plugin for reading CSV data
csvplayback	Plays a CSV at some configurable speed and each column of the file will become a datapoint of an asset using pandas library.
dht	A FLIR Bridge south plugin in C++ that interfaces to a DHT-11 temperature and humidity sensor
dht11	A FLIR Bridge south plugin that interfaces a DHT-11 temperature sensor
digiducer	South plugin for the Digiducer 333D01 vibration sensor

Continued on next page

Table 1 – continued from previous page

Name	Description
dnp3	A south plugin for FLIR Bridge that implements the DNP3 protocol
dt9837	A south plugin for the Data Translation DT9837 Series DAQ
edgeml	ML south plugin which forwards the video frames to a model running inside micro k8's; parses the response, generates readings and shows the detection results on browser.
envirophat	A FLIR Bridge south service for the Raspberry Pi EnviroPhat sensors
expression	A FLIR Bridge south plugin that uses a user define expression to generate data
flir-camera	Interface to all generations of Flir cameras supporting both generations of the REST API.
FlirAX8	A FLIR Bridge hybrid south plugin that uses flir-south-modbus-c to get temperature data from a Flir Thermal camera
game	The south plugin used for the FLIR Bridge lab session game involving remote controlled cars
gw65	FLIR Bridge plugin for getting vibration data from a set of FLIR GW65 vibration sensors
http	A Python south plugin for FLIR Bridge used to connect one FLIR Bridge instance to another
iec104	A south plugin to gather data using the IEC 104 protocol.
iec61850	A south plugin for collecting data via the IEC 61850 protocol
ina219	A FLIR Bridge south plugin for the INA219 voltage and current sensor
J1708	A plugin that uses the SAE J1708 protocol to load data from the ECU of heavy duty vehicles.
J1939	A CANBUS J1839 plugin to collect data into FLIR Bridge.
lathesim	A simulation plugin used as a demonstration to show how data can be collected within FLIR Bridge. This plugin simulates various properties of a lathe.
modbus-c	A FLIR Bridge south plugin that implements modbus-tcp and modbus-rtu
modbustcp	A FLIR Bridge south plugin that implements modbus-tcp in Python
mqtt	FLIR Bridge South MQTT Subscriber Plugin
mqtt-sparkplug	A FLIR Bridge south plugin that implements the Sparkplug API over MQTT
mqtt-scripted	An MQTT south plugin that allows a Python script to be added to decode the MQTT payload
opcua	A FLIR Bridge south service that pulls data from an OPC-UA server
openweathermap	A FLIR Bridge south plugin to pull weather data from OpenWeatherMap
person-detection	FLIR Bridge south service plugin that detects person in the live video stream
phidget	FLIR Bridge south code for different phidgets
piwebapi	A South plugin to ingest data from a PI Server using the PI Web API.
playback	A FLIR Bridge south plugin to replay data stored in a CSV file
pt100	A FLIR Bridge south plugin for the PT100 temperature sensor
random	A south plugin for FLIR Bridge that generates random numbers
randomwalk	A FLIR Bridge south plugin that returns data that with randomly generated steps
roxtec	A FLIR Bridge south plugin for the Roxtec cable gland project
s2opcua	An OPCUA south plugin based on the Safe & Secure OPCUA library. This plugin offers similar functionality to the flir-south-opcua plugin but also offers encryption and authentication.
s7	A south plugin that uses the S7 Communications protocol to read data from a Siemens S7 series PLC.
sarcos	A south plugin to process the Sarcos XO data files
sensehat	A FLIR Bridge south plugin for the Raspberry Pi Sensehat sensors
sensorphone	A FLIR Bridge south plugin the task to the iPhone SensorPhone app

Continued on next page

Table 1 – continued from previous page

Name	Description
simple-rest	A generic REST south plugin with support for a variety of common rest payloads and Python scripting to manipulate call results.
sinusoid	A FLIR Bridge south plugin that produces a simulated sine wave
suez	A south plugin to extract data from the Suez Water Insight API
systeminfo	A FLIR Bridge south plugin that gathers information about the system it is running on.
usb4704	A FLIR Bridge south plugin the Advantech USB-4704 data acquisition module
webcam-media	A FLIR Bridge south plugin that forwards image data, either directly from a webcam or from a directory of images
wind-turbine	A FLIR Bridge south plugin for a number of sensor connected to a wind turbine demo

4.2 North Plugins

North plugins add new destinations to which data may be sent by FLIR Bridge. A number of north plugins are available ready built or users may add new north plugins of their own by writing them in Python or C/C++.

Table 2: FLIR Bridge North Plugins

Name	Description
azure	A north plugin that sends data to Microsoft Azure IoT Core.
gcp	A north plugin to send data to Google Cloud Platform IoT Core
graphite	A north plugin for FLIR Bridge that sends data to the Graphite Carbon storage system.
harperdb	A north plugin that sends data to the HarperDB SQL/NoSQL data management platform
http	A Python implementation of a north plugin to send data between FLIR Bridge instances using HTTP
http-c	A FLIR Bridge north plugin that sends data between FLIR Bridge instances using HTTP/HTTPS
iec104	A FLIR Bridge north plugin for sending data using the IEC-104 protocol.
influxdb	A north plugin for sending data to InfluxDB
influxdbcloud	A north plugin to send data from FLIR Bridge to the InfluxDBCloud
kafka	A FLIR Bridge plugin for sending data north to Apache Kafka
kafka-python	A Python implementation of a north plugin that can send data to Apache Kafka
opcua	A north plugin for FLIR Bridge that makes it act as an OPC-UA server for the data it reads from sensors
splunk	A north plugin for sending data to Splunk
thingspeak	A FLIR Bridge north plugin to send data to Matlab's ThingSpeak cloud
timestream	A timestream north plugin

4.3 Filter Plugins

Filter plugins add new ways in which data may be modified, enhanced or cleaned as part of the ingress via a south service or egress to a destination system. A number of north plugins are available ready built or users may add new north plugins of their own by writing them in Python or C/C++.

It is also possible, using particular filters, to supply expressions or script snippets that can operate on the data as well.

This provides a simple way to process the data in FLIR Bridge as it is read from devices or written to destination systems.

Table 3: FLIR Bridge Filter Plugins

Name	Description
ADM-LD-prediction	Filter to detect whether a large discharge is required for an ADM centrifuge
asset	A FLIR Bridge processing filter that is used to block or allow certain assets to pass onwards in the data stream
asset-split	A filter to split an asset with multiple data points into several assets, each with a single data point.
blocktest	A filter designed to aid testing. It combines incoming readings into bigger blocks before sending onwards
change	A FLIR Bridge processing filter plugin that only forwards data that changes by more than a configurable amount
csv-writer	FLIR Bridge filter which writes selected readings passing through it out as a rotating sequence of .csv files.
dataframe	Turn streams of data assets into file data holding individual frames (probably in tmpfs).
delta	A FLIR Bridge processing filter plugin that removes duplicates from the stream of data and only forwards new values that differ from previous values by more than a given tolerance
downsample	A data downsampling filter which may be used to reduce the data rate using sampling or averaging techniques.
edgectl	Filter which takes image data, calls out to ML process, and forwards the inference from ML as asset contents.
ema	Generate exponential moving average datapoint: include a rate of current value and a rate of history values
eventrate	A filter designed for use in the north to trigger sending rates based on event notification assets
expression	A FLIR Bridge processing filter plugin that applies a user define formula to the data as it passes through the filter
fft	A FLIR Bridge processing filter plugin that calculates a Fast Fourier Transform across sensor data
fft2	Filter for FFT signal processing, finding peak frequencies, etc.
Flir-Validity	A FLIR Bridge processing filter used for processing temperature data from a Flir thermal camera
log	A FLIR Bridge filter that converts the readings data to a logarithmic scale. This is the example filter used in the plugin developers guide.
metadata	A FLIR Bridge processing filter plugin that adds metadata to the readings in the data stream
omfhint	A filter plugin that allows data to be added to assets that will provide extra information to the OMF north plugin.
python27	A FLIR Bridge processing filter that allows Python 2 code to be run on each sensor value.
python35	A FLIR Bridge processing filter that allows Python 3 code to be run on each sensor value.
rate	A FLIR Bridge processing filter plugin that sends reduced rate data until an expression triggers sending full rate data
rename	A FLIR Bridge processing filter that is used to modify the name of an asset, datapoint or with both
replace	Filter to replace characters in the names of assets and data points in readings object.

Continued on next page

Table 3 – continued from previous page

Name	Description
rms	A FLIR Bridge processing filter plugin that calculates RMS value for sensor data
rms-trigger	An RMS filter that uses a trigger asset rather than a fixed set of readings for each calculation
scale	A FLIR Bridge processing filter plugin that applies an offset and scale factor to the data
scale-set	A FLIR Bridge processing filter plugin that applies a set of scale factors to the data
sigfns	Signal processing functions
sigmacleanse	A data cleansing plugin that removes data that differs from the mean value by more than x sigma
simple-python	The simple Python filter plugin is analogous to the expression filter but accept Python code rather than the expression syntax
specgram	FLIR Bridge filter to generate spectrogram images for vibration data
statistics	Generic statistics filter for FLIR Bridge data that supports the generation of mean, mode, median, minimum, maximum, standard deviation and variance.
threshold	A FLIR Bridge processing filter that only forwards data when a threshold is crossed
velocity	Filter to process acceleration data to generate velocity and acceleration envelope
vibration_features	A filter plugin that takes a stream of vibration data and generates a set of features that characterise that data

4.4 Notification Rule Plugins

Notification rule plugins provide the logic that is used by the notification service to determine if a condition has been met that should trigger or clear that condition and hence send a notification. A number of notification plugins are available as standard, however as with any plugin the user is able to write new plugins in Python or C/C++ to extend the set of notification rules.

Table 4: FLIR Bridge Notification Rule Plugins

Name	Description
average	A FLIR Bridge notification rule plugin that evaluates an expression based sensor data notification rule plugin that triggers when sensors values depart from the moving average by more than a configured limit.
ML-bad-bearing	Notification rule plugin to detect bad bearing
outofbound	A FLIR Bridge notification rule plugin that triggers when sensors values exceed limits set in the configuration of the plugin.
periodic	A rule that periodically fires based on a timer when data is observed.
simple-expression	A FLIR Bridge notification rule plugin that evaluates an expression based sensor data
simple-sigma	A FLIR Bridge notification rule that will send a notification if the values being monitored differ from the mean for the value by more than a multiple of the current standard deviation.
watchdog	A watchdog rule plugin

4.5 Notification Delivery Plugins

Notification delivery plugins provide the mechanisms to deliver the notification messages to the systems that will receive them. A number of notification delivery plugins are available as standard, however as with any plugin the user

is able to write new plugins in Python or C/C++ to extend the set of notification rules.

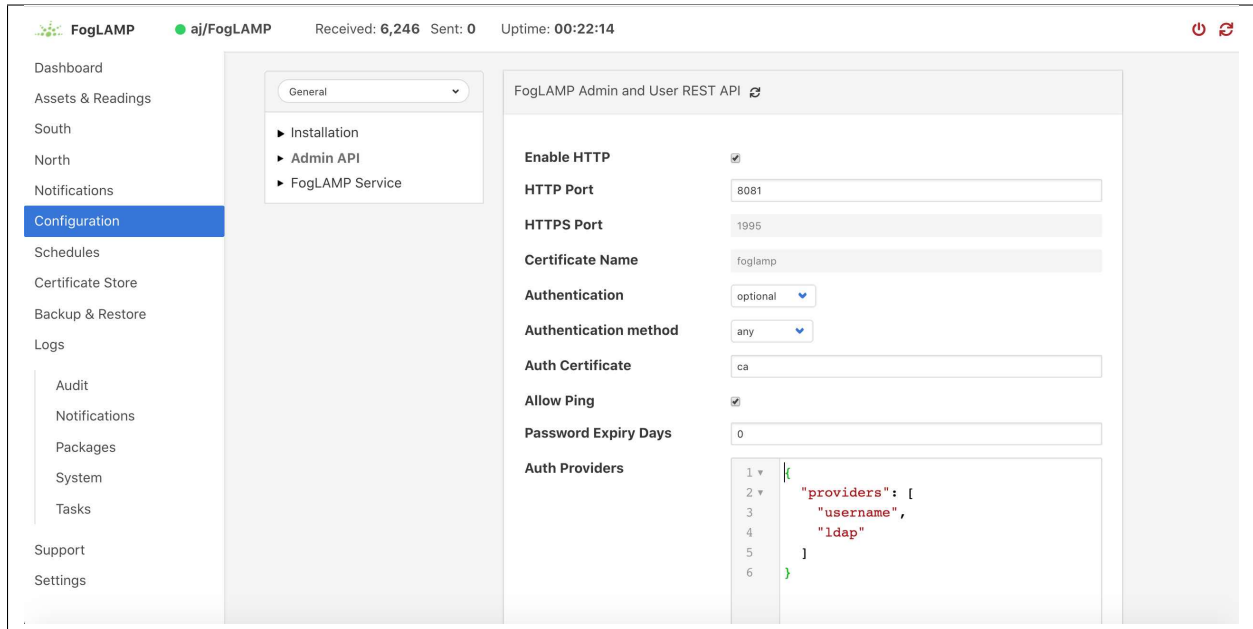
Table 5: FLIR Bridge Notification Delivery Plugins

Name	Description
alexa-notifyme	A FLIR Bridge notification delivery plugin that sends notifications to the Amazon Alexa platform
asset	A FLIR Bridge notification delivery plugin that creates an asset in FLIR Bridge when a notification occurs
blynk	A FLIR Bridge notification delivery plugin that sends notifications to the Blynk service
config	A notification delivery plugin that allows a configuration item within the local FLIR Bridge instance to be changed when the notification triggers or is cleared.
email	A FLIR Bridge notification delivery plugin that sends notifications via email
google-hangouts	A FLIR Bridge notification delivery plugin that sends alerts on the Google hang-out platform
ifttt	A FLIR Bridge notification delivery plugin that triggers an action of IFTTT
jira	A notification plugin that creates tickets in Jira
jsonconfig	A delivery mechanism that updates one element within a JSON configuration type configuration category item.
management	A notification delivery plugin the triggers the FLIR Bridge management service to check for updates to the configuration of FLIR Bridge
mqtt	A notification delivery plugin that sends messages via MQTT when a notification is triggered or cleared. This is the example used in the notification delivery plugin writers guide.
north	Deliver notification data via a FLIR Bridge north task
operation	A notification delivery plugin that will cause an operation to be trigger via the set point control operation API of a south service.
python35	A FLIR Bridge notification delivery plugin that runs an arbitrary Python 3 script
setpoint	A flir notification plugin that invokes a set point operation on a south service.
slack	A FLIR Bridge notification delivery plugin that sends notifications via the slack instant messaging platform
telegram	A FLIR Bridge notification delivery plugin that sends notifications via the telegram service
zendesk	A notification delivery plugin that will create tickets within Zendesk ticketing application

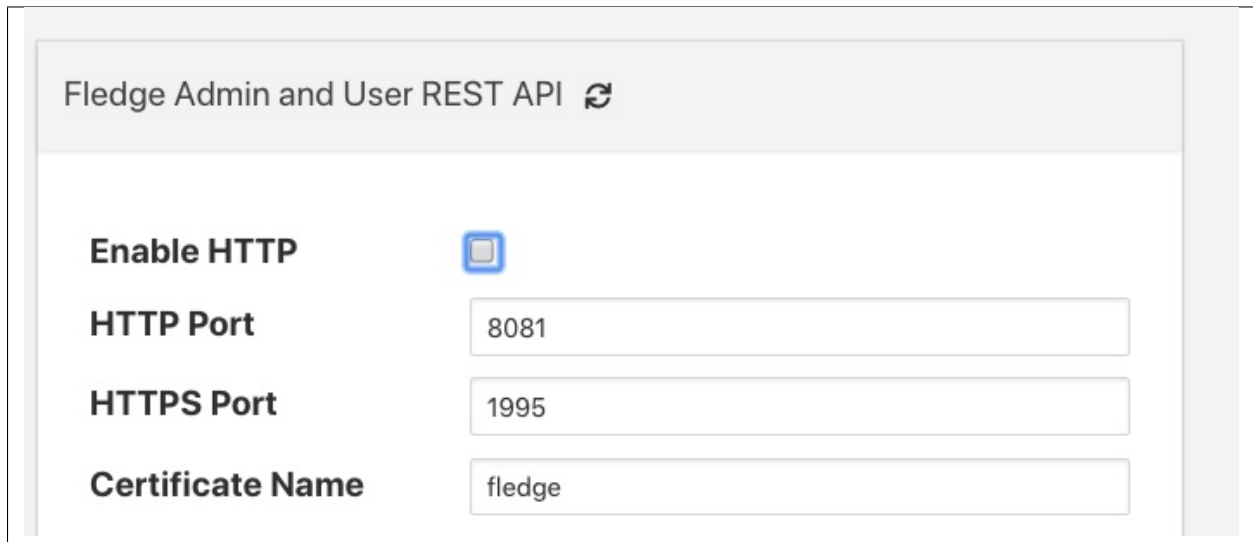
The default installation of a Flir service comes with security features turned off, there are several things that can be done to add security to Flir. The REST API by default support unencrypted HTTP requests, it can be switched to require HTTPS to be used. The REST API and the GUI can be protected by requiring authentication to prevent users being able to change the configuration of the Flir system. Authentication can be via username and password or by means of an authentication certificate.

5.1 Enabling HTTPS Encryption

Flir can support both HTTP and HTTPS as the transport for the REST API used for management, to switch between there two transport protocols select the *Configuration* option from the left-hand menu and the select *Admin API* from the configuration tree that appears,



The first option you will see is a tick box labeled *Enable HTTP*, to select HTTPS as the protocol to use this tick box should be deselected.



When this is unticked two options become active on the page, *HTTPS Port* and *Certificate Name*. The HTTPS Port is the port that Flir will listen on for HTTPS requests, the default for this is port 1995.

The *Certificate Name* is the name of the certificate that will be used for encryption. The default is to use a self signed certificate called *flir* that is created as part of the installation process. This certificate is unique per flir installation but is not signed by a certificate authority. If you require the extra security of using a signed certificate you may use the Flir [Certificate Store](#) functionality to upload a certificate that has been created and signed by a certificate authority.

After enabling HTTPS and selecting save you must restart Flir in order for the change to take effect. You must also update the connection setting in the GUI to use the HTTPS transport and the correct port.

Note: if using the default self-signed certificate you might need to authorise the browser to connect to IP:PORT. Just open a new browser tab and type the URL `https://YOUR_FLIR_IP:1995`

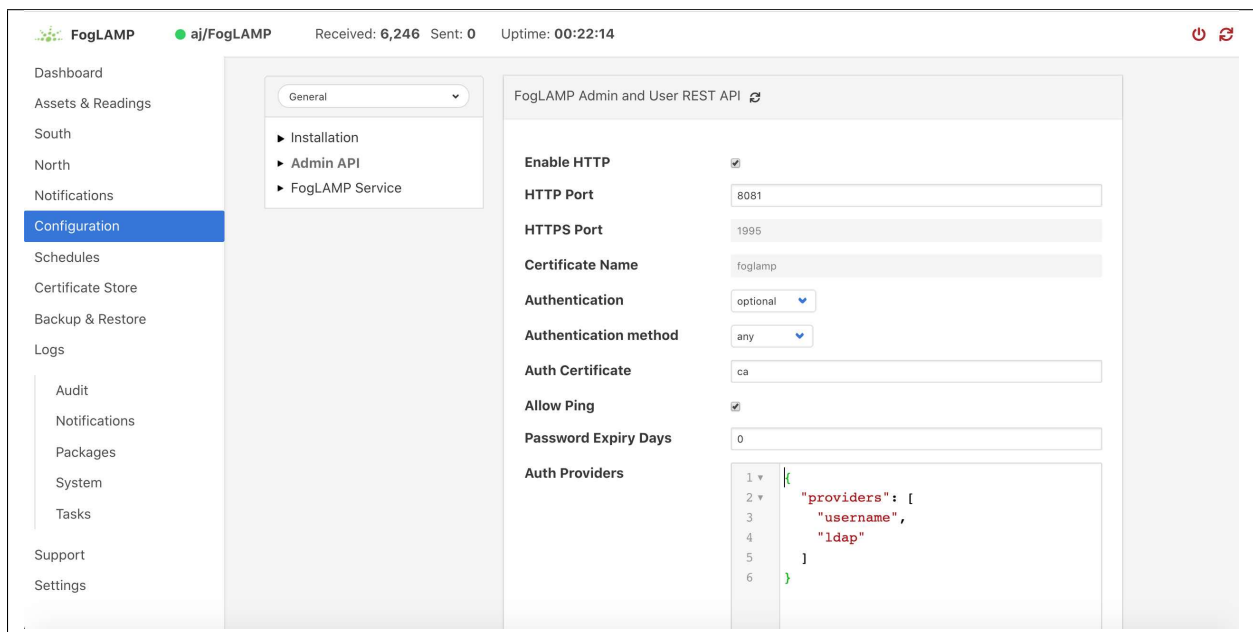
Then follow browser instruction in order to allow the connection and close the tab. In the Flir GUI you should see the green icon (Flir is running).



The image shows a 'Connection Setup' dialog box. It has two main sections: 'Host' and 'Port'. Under 'Host', there is a dropdown menu currently set to 'https' and a text input field containing 'foglamp-18.local'. Under 'Port', there is a text input field containing '1995'. At the bottom left of the dialog is a blue button labeled 'Set the URL & Restart'.

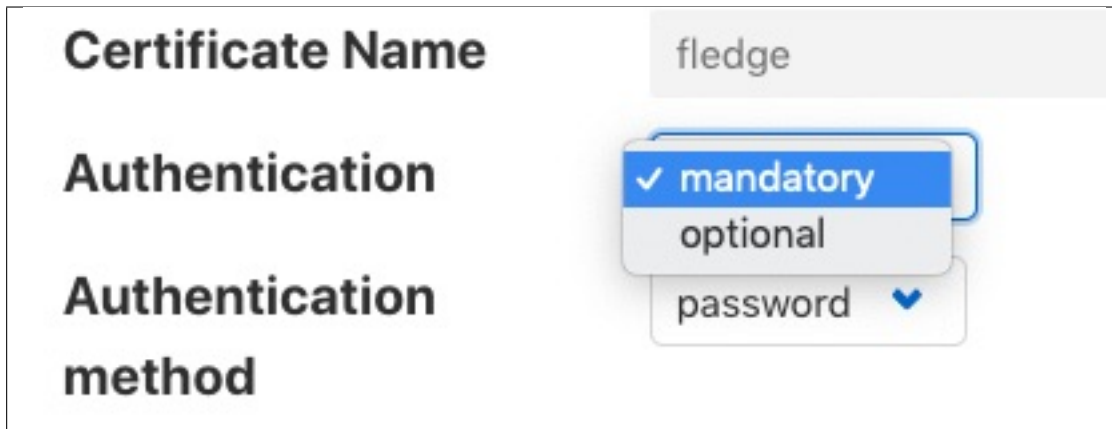
5.2 Requiring User Login

In order to set the REST API and GUI to force users to login before accessing Flir select the *Configuration* option from the left-hand menu and then select *Admin API* from the configuration tree that appears.



The image shows the FogLAMP configuration interface. On the left is a sidebar menu with options: Dashboard, Assets & Readings, South, North, Notifications, Configuration (highlighted), Schedules, Certificate Store, Backup & Restore, Logs, Audit, Notifications, Packages, System, Tasks, Support, and Settings. The main area is titled 'FogLAMP Admin and User REST API'. It contains several configuration fields: 'Enable HTTP' (checked), 'HTTP Port' (8081), 'HTTPS Port' (1995), 'Certificate Name' (foglamp), 'Authentication' (optional), 'Authentication method' (any), 'Auth Certificate' (ca), 'Allow Ping' (checked), 'Password Expiry Days' (0), and 'Auth Providers' (a JSON array containing 'username' and 'ldap').

Two particular items are of interest in this configuration category that is then displayed; *Authentication* and *Authentication method*

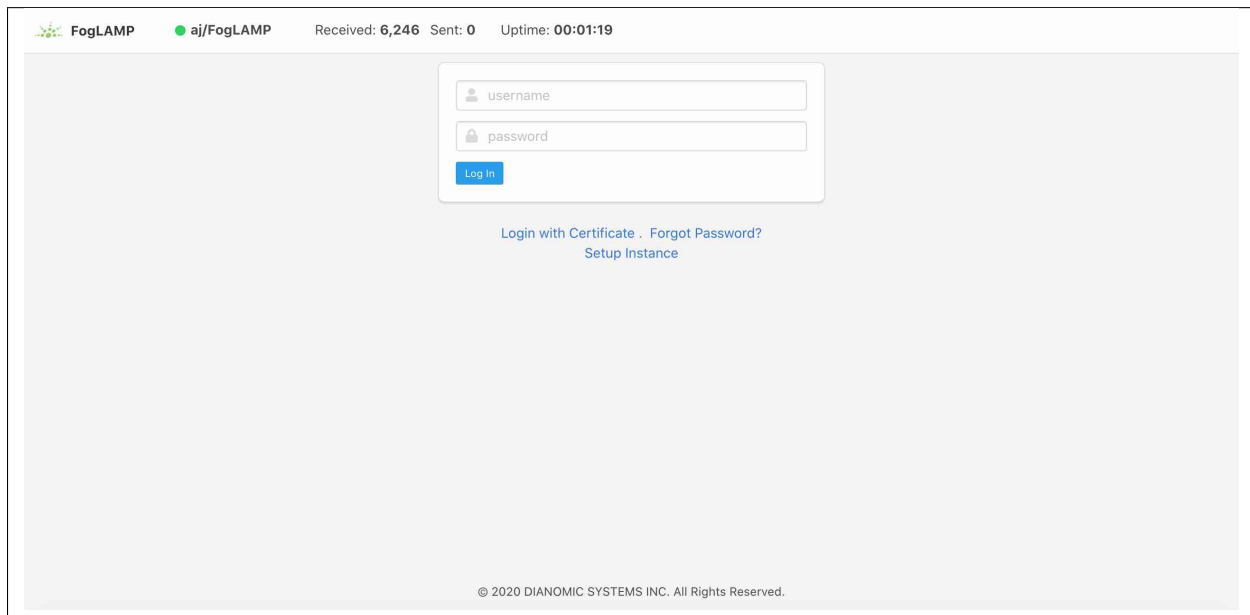


The screenshot shows a configuration dialog with three main sections: **Certificate Name**, **Authentication**, and **Authentication method**. The **Certificate Name** field is set to "fledge". The **Authentication** field has a dropdown menu open, showing three options: "mandatory" (selected with a checkmark), "optional", and "password". The **Authentication method** field has a dropdown menu open, showing three options: "mandatory", "optional", and "password" (selected with a checkmark).

Select the *Authentication* field to be mandatory and the *Authentication method* to be password. Click on *Save* at the bottom of the dialog.

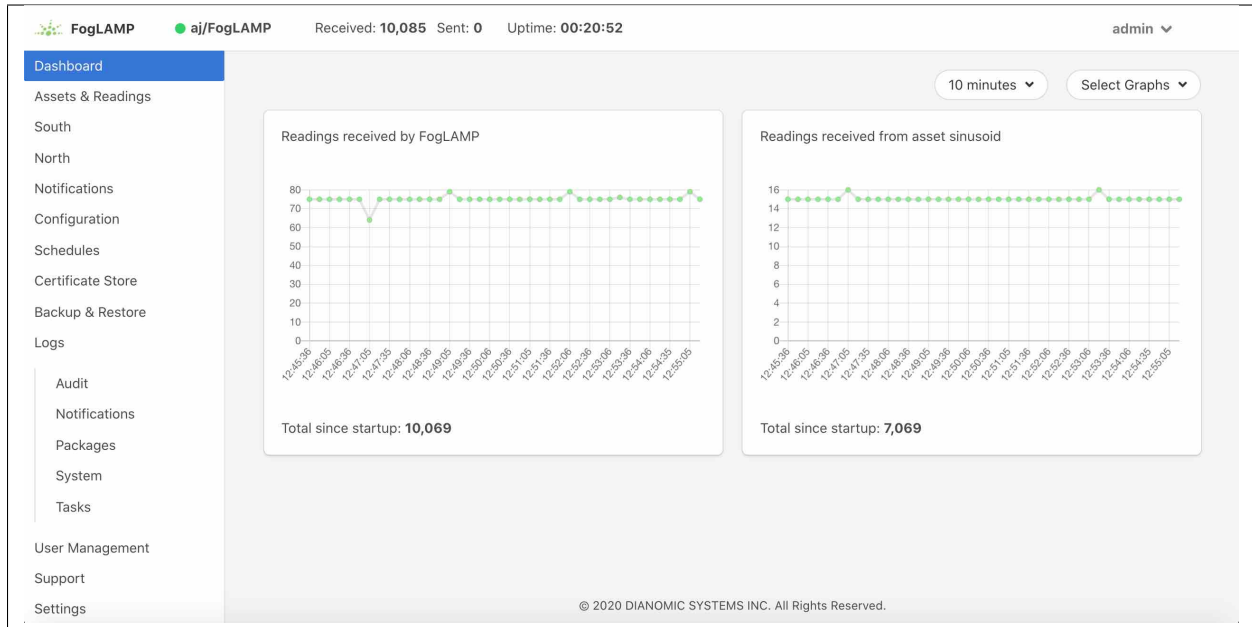
In order for the changes to take effect Flir must be restarted, this can be done in the GUI by selecting the restart item in the top status bar of Flir. Confirm the restart of Flir and wait for it to be restarted.

Once restarted refresh your browser page. You should be presented with a login request.



The screenshot shows the Flir login page. At the top, there is a status bar with the following information: "FogLAMP", "aj/FogLAMP", "Received: 6,246 Sent: 0", and "Uptime: 00:01:19". Below the status bar is a login form with two input fields: "username" and "password". Below the input fields is a "Log In" button. Below the "Log In" button are three links: "Login with Certificate", "Forgot Password?", and "Setup Instance". At the bottom of the page, there is a copyright notice: "© 2020 DIANOMIC SYSTEMS INC. All Rights Reserved."

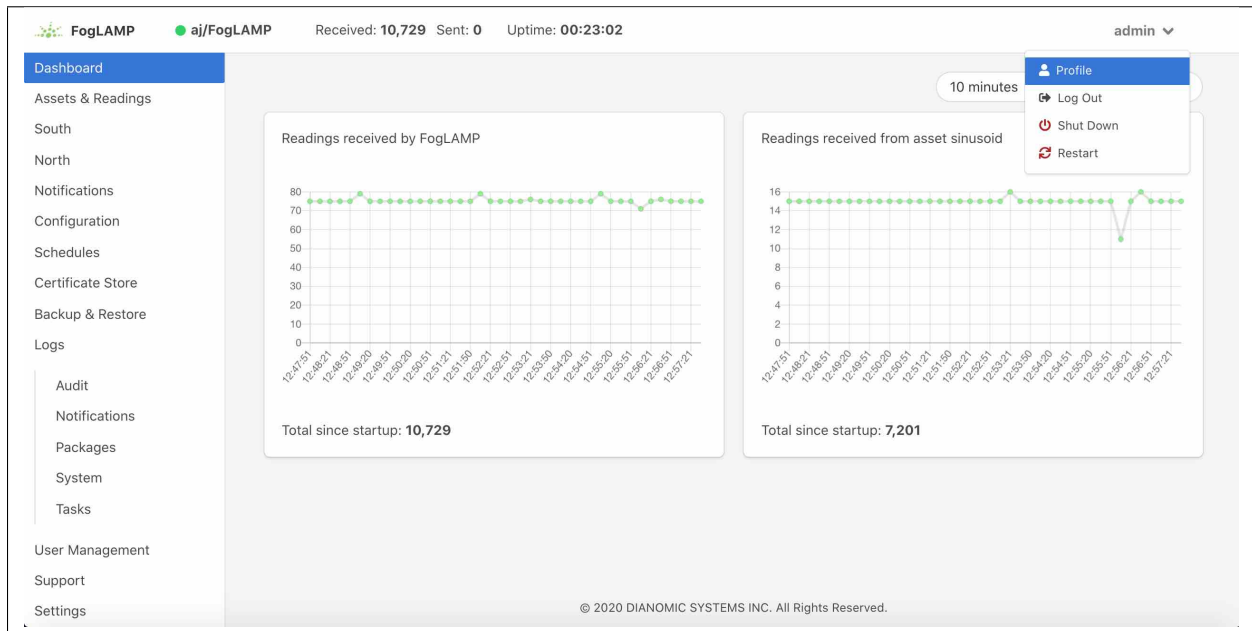
The default username is "admin" with a password of "flir". Use these to login to Flir, you should be presented with a slightly changed dashboard view.



The status bar now contains the name of the user that is currently logged in and a new option has appeared in the left-hand menu, *User Management*.

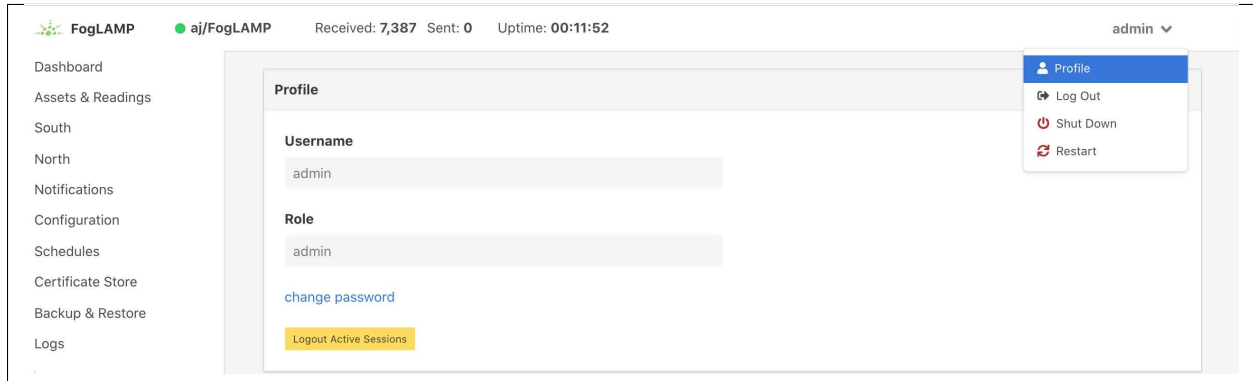
5.2.1 Changing Your Password

The top status bar of the Flir GUI now contains the user name on the right-hand side and a pull down arrow, selecting this arrow gives a number of options including one labeled *Profile*.



Note: This pulldown menu is also where the *Shutdown* and *Restart* options have moved.

Selecting the *Profile* option will display the profile for the user.



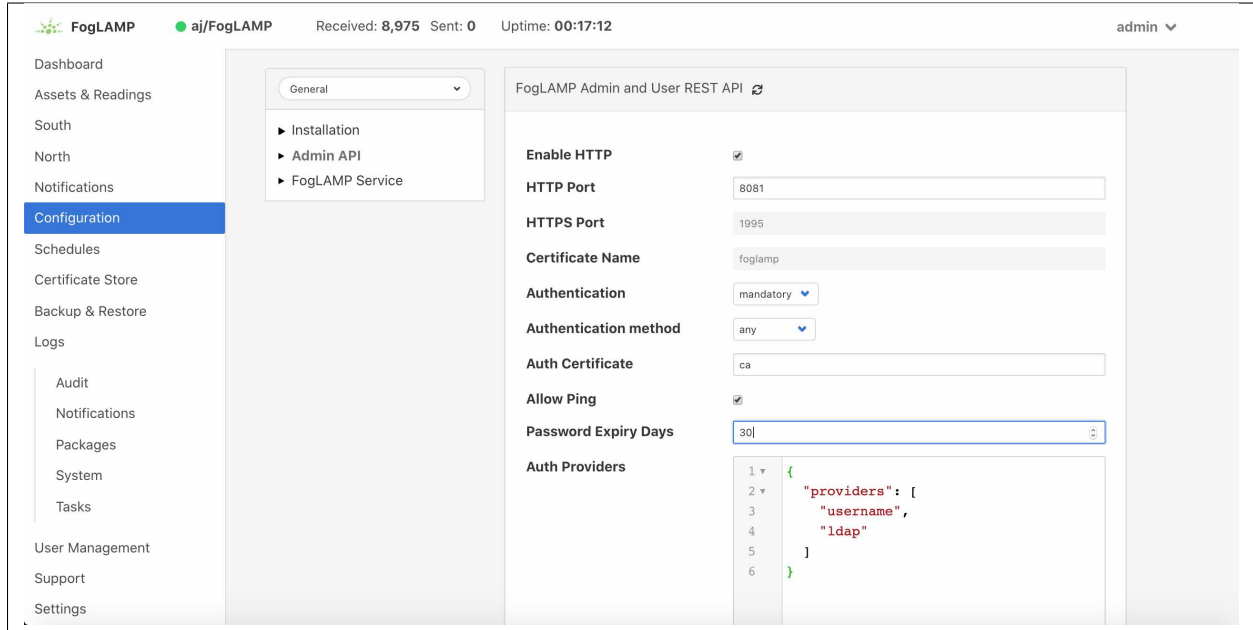
Towards the bottom of this profile display the *change password* option appears. Click on this text and a new password dialog will appear.



This popup can be used to change your password. On successfully changing your password you will be logged out of the user interface and will be required to log back in using this new password.

5.2.2 Password Rotation Mechanism

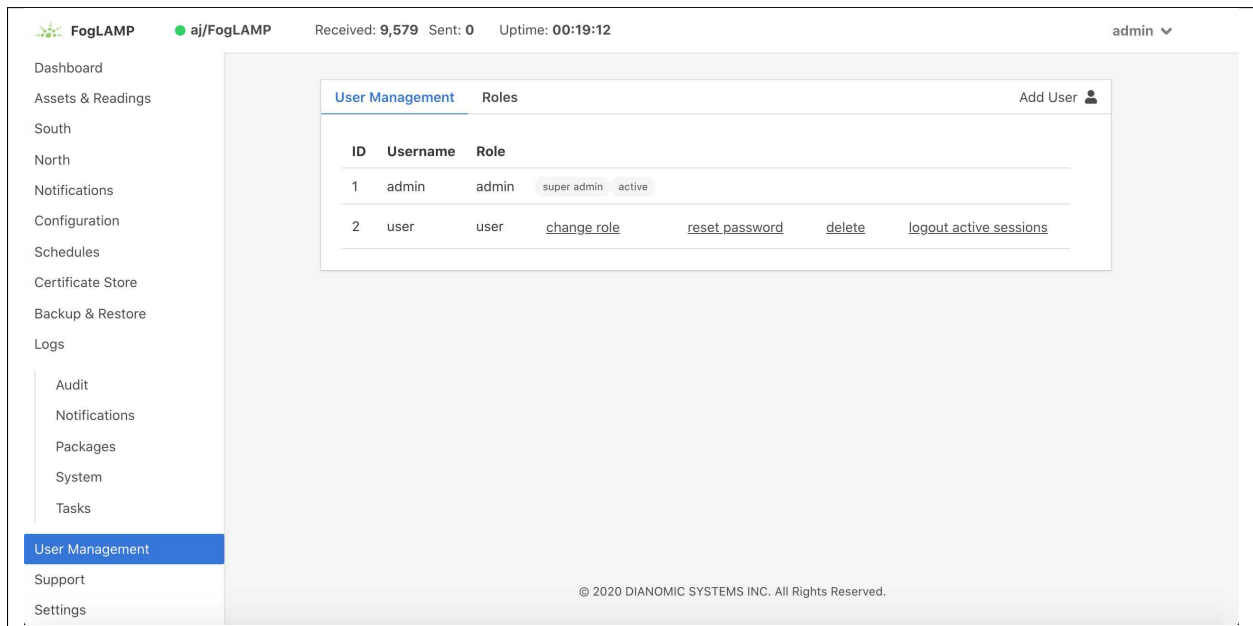
Flir provides a mechanism to limit the age of passwords in use within the system. A value for the maximum allowed age of a password is defined in the configuration page of the user interface.



Whenever a user logs into Flir the age of their password is checked against the maximum allowed password age. If their password has reached that age then the user is not logged in, but is instead forced to enter a new password. They must then login with that new password. In addition the system maintains a history of the last three passwords the user has used and prevents them being reused.

5.3 User Management

Once mandatory authentication has been enabled and the currently logged in user has the role *admin*, a new option appears in the GUI, *User Management*.



The user management pages allows

- Adding new users.
- Deleting users.
- Resetting user passwords.
- Changing the role of a user.

Flir currently supports two roles for users,

- **admin:** a user with admin role is able to fully configure Flir and also manage Flir users
- **user:** a user with this role is able to configure Flir but can not manage users

5.3.1 Adding Users

To add a new user from the *User Management* page select the *Add User* icon in the top right of the *User Management* pane. a new dialog will appear that will allow you to enter details of that user.

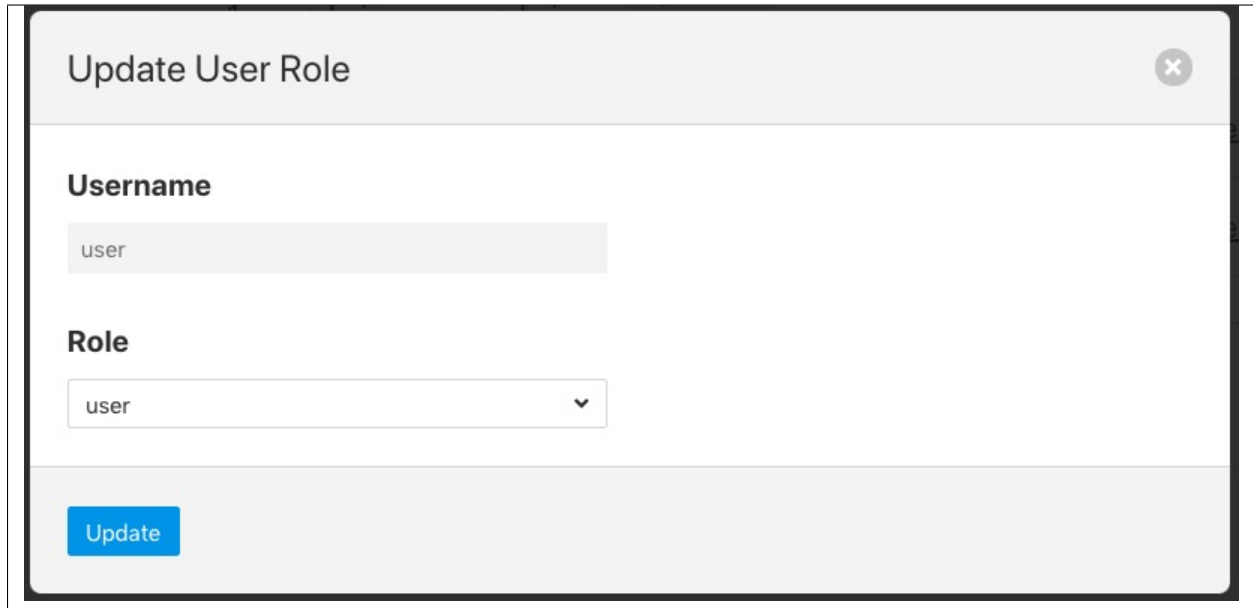


The image shows a 'Create User' dialog box with a title bar containing the text 'Create User' and a close button (X). The dialog contains four input fields: a dropdown menu for 'Role' with 'user' selected, a text field for 'mark', a password field with masked characters '.....', and another password field with masked characters '.....'. A blue 'Save' button is located at the bottom left of the dialog.

You can select a role for the new user, a user name and an initial password for the user. Only users with the role *admin* can add new users.

5.3.2 Changing User Roles

The role that a particular user has when the login can be changed from the *User Management* page. Simply select on the *change role* link next to the user you wish to change the role of.

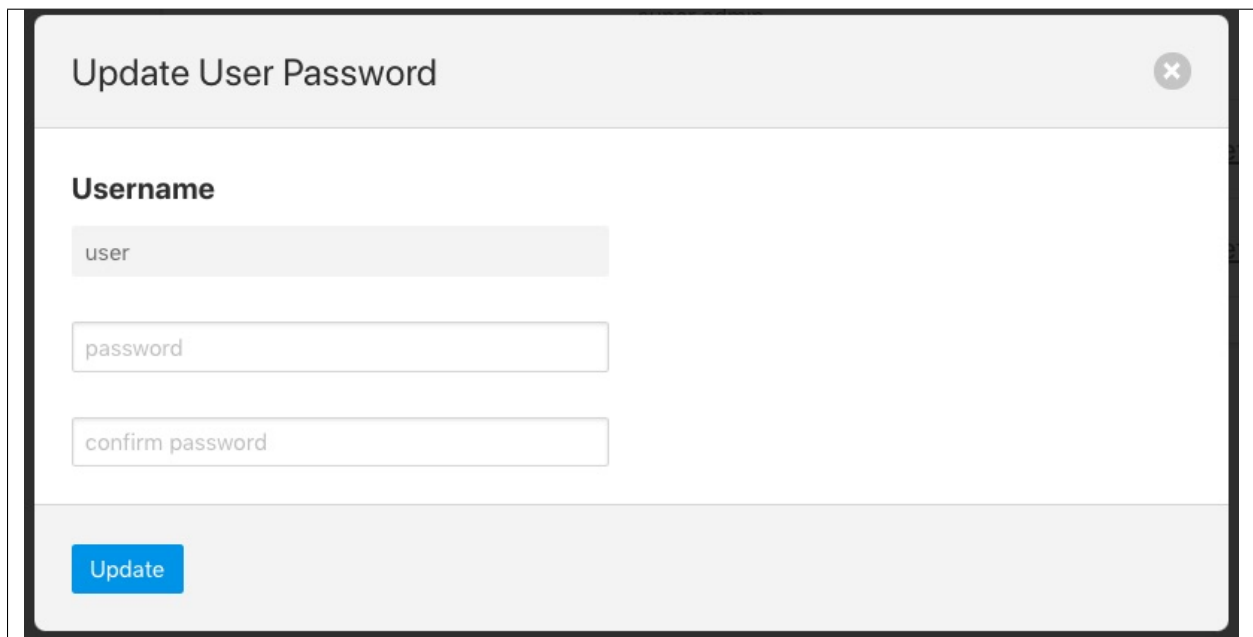


The dialog box titled "Update User Role" has a close button (X) in the top right corner. It contains two input fields: "Username" with the value "user" and "Role" with a dropdown menu showing "user". At the bottom left is a blue "Update" button.

Select the new role for the user from the drop down list and click on update. The new role will take effect the next time the user logs in.

5.3.3 Reset User Password

Users with the *admin* role may reset the password of other users. In the *User Management* page select the *reset password* link to the right of the user name of the user you wish to reset the password of. A new dialog will appear prompting for a new password to be created for the user.

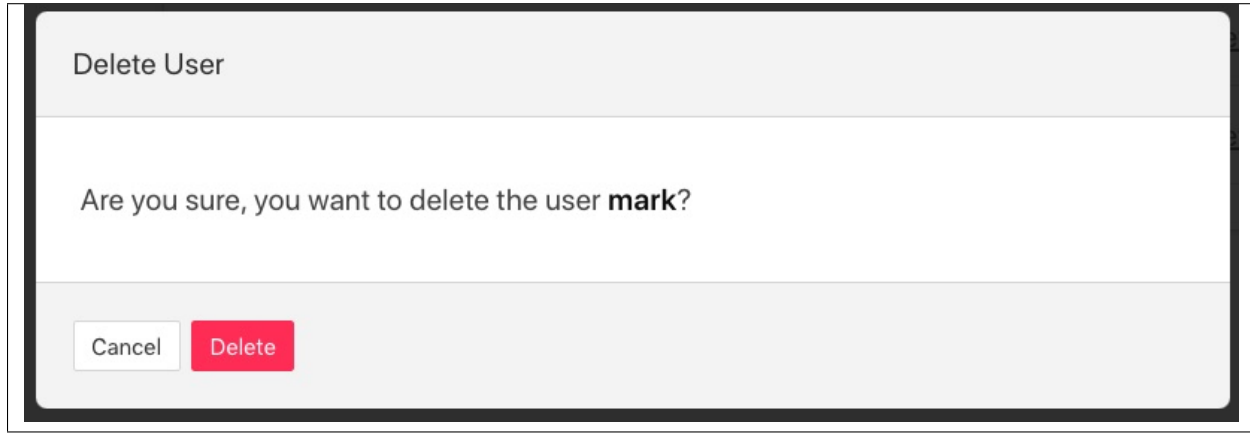


The dialog box titled "Update User Password" has a close button (X) in the top right corner. It contains three input fields: "Username" with the value "user", "password", and "confirm password". At the bottom left is a blue "Update" button.

Enter the new password and confirm that password by entering it a second time and click on *Update*.

5.3.4 Delete A User

Users may be deleted from the *User Management* page. Select the *delete* link to the right of the user you wish to delete. A confirmation dialog will appear. Select *Delete* and the user will be deleted.



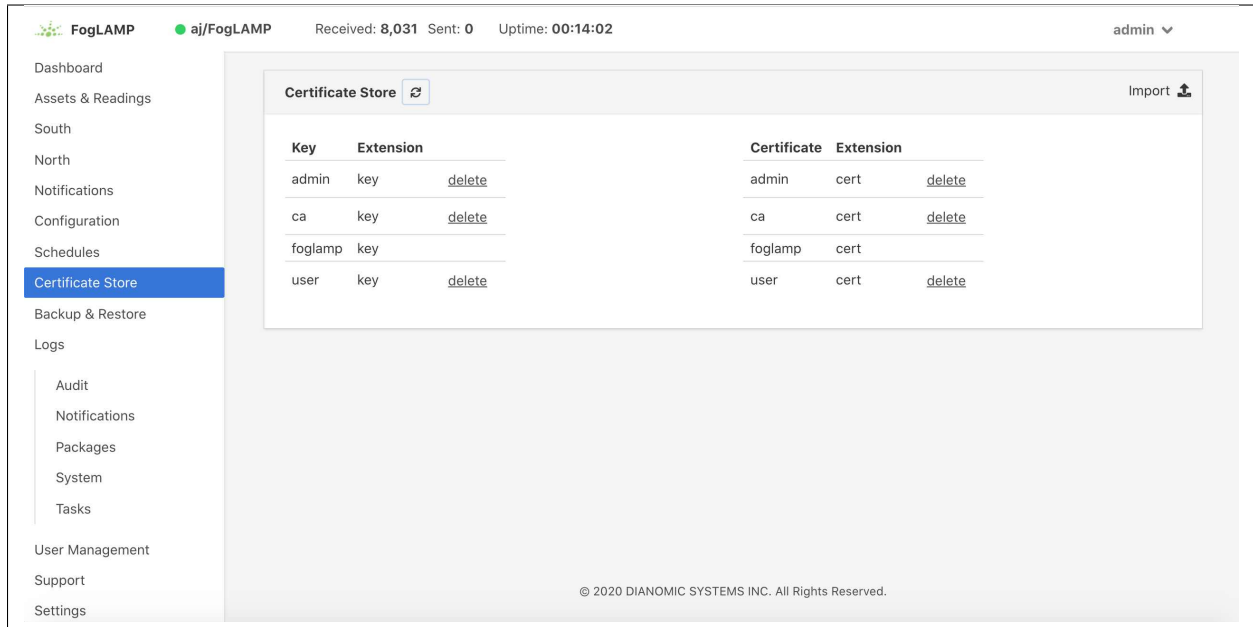
You can not delete the last user with role *admin* as this will prevent you from being able to manage Flir.

5.4 Certificate Store

The Flir *Certificate Store* allows certificates to be stored that may be referenced by various components within the system, in particular these certificates are used for the encryption of the REST API traffic and authentication. They may also be used by particular plugins that require a certificate of one type or another. A number of different certificate types re supported by the certificate store;

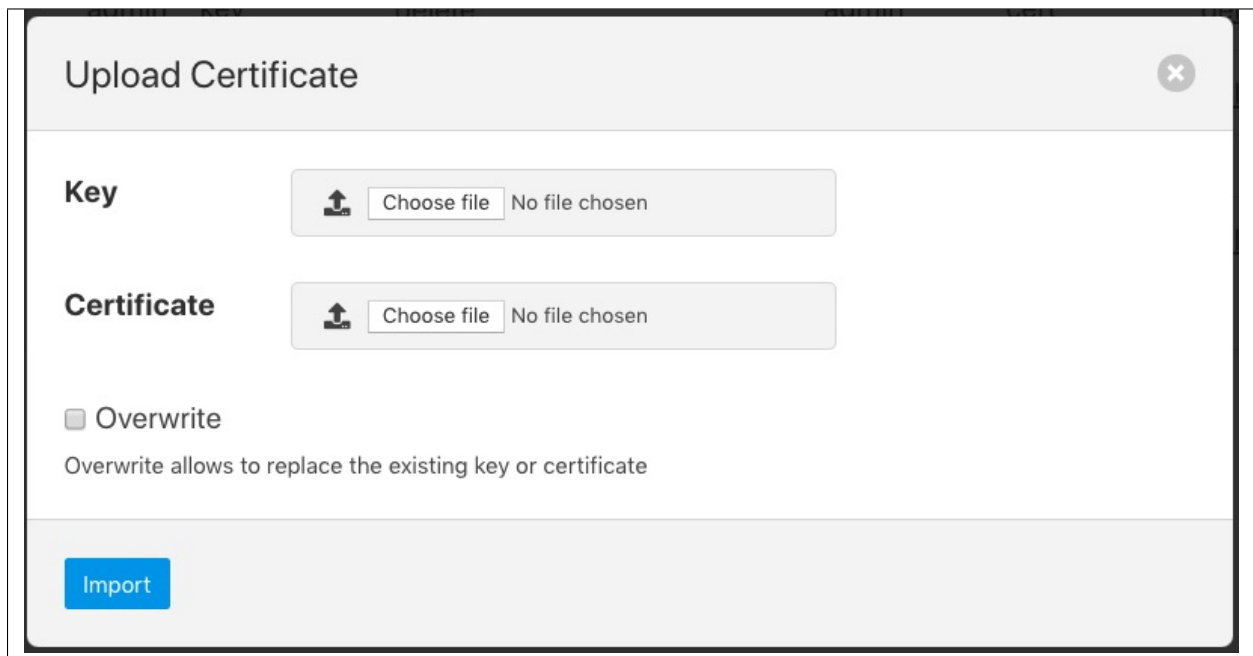
- PEM files as created by most certificate authorities
- CRT files as used by GlobalSign, VeriSign and Thawte
- Binary CER X.509 certificates
- JSON certificates as used by Google Cloud Platform

The *Certificate Store* functionality is available in the left-hand menu by selecting *Certificate Store*. When selected it will show the current content of the store.



Certificates may be removed by selecting the delete option next to the certificate name, note that the keys and certificates can be deleted independently. The self signed certificate that is created at installation time can not be deleted.

To add a new certificate select the *Import* icon in the top right of the certificate store display.



A dialog will appear that allows a key file and/or a certificate file to be selected and uploaded to the *Certificate Store*. An option allows to allow overwrite of an existing certificate. By default certificates may not be overwritten.

Buffering & Storage

One of the micro-services that makes up the core of a Flir implementation is the storage micro-service. This is responsible for

- storing the configuration of Flir
- buffering the data read from the south
- maintaining the Flir audit log
- persisting the state of the system

The storage service is configurable, like other services within Flir and uses plugins to extend the functionality of the storage system. These storage plugins provide the underlying mechanism by which data is stored within Flir. Flir can make use of either one or two of these plugins at any one time. If a single plugin is used then this plugin provides the storage for all data. If two plugins are used, one will be for the buffering of readings and the other for the storage of the configuration.

As standard Flir comes with 3 storage plugins

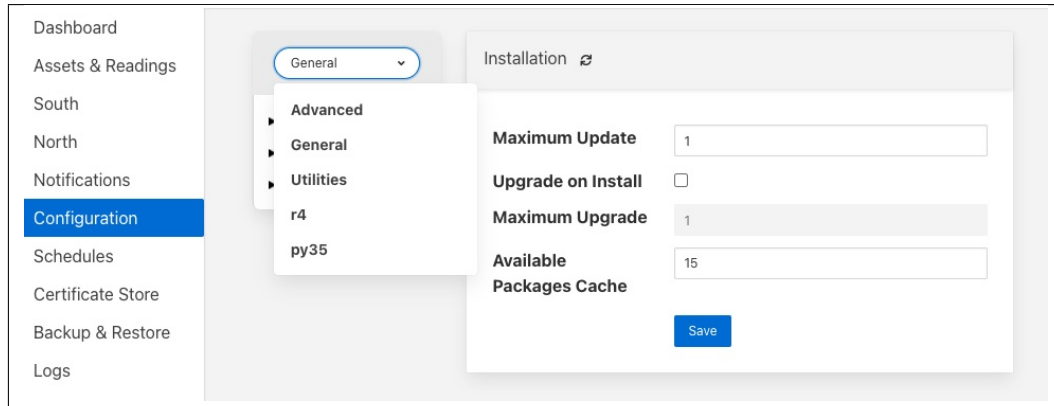
- **SQLite**: A plugin that can store both configuration data and the readings data using SQLite files as the backing store. The plugin uses multiple SQLite database to store different assets, allowing for high bandwidth data at the expense of limiting the number of assets that a single instance can ingest.,
- **SQLiteLB**: A plugin that can store both configuration data and the readings data using SQLite files as the backing store. This version of the SQLite plugin uses a single readings database and is better suited for environments that do not have very high bandwidth data. It does not limit the number of distinct assets that can be ingested.
- **PostgreSQL**: A plugin that can store both configuration and readings data which uses the PostgreSQL SQL server as a storage medium.
- **SQLiteMemory**: A plugin that can only be used to store reading data. It uses SQLite's in memory storage engine to store the reading data. This provides a high performance reading store however capacity is limited by available memory and if Flir is stopped or there is a power failure the buffered data will be lost.

The default configuration uses the SQLite disk based storage engine for both configuration and reading data

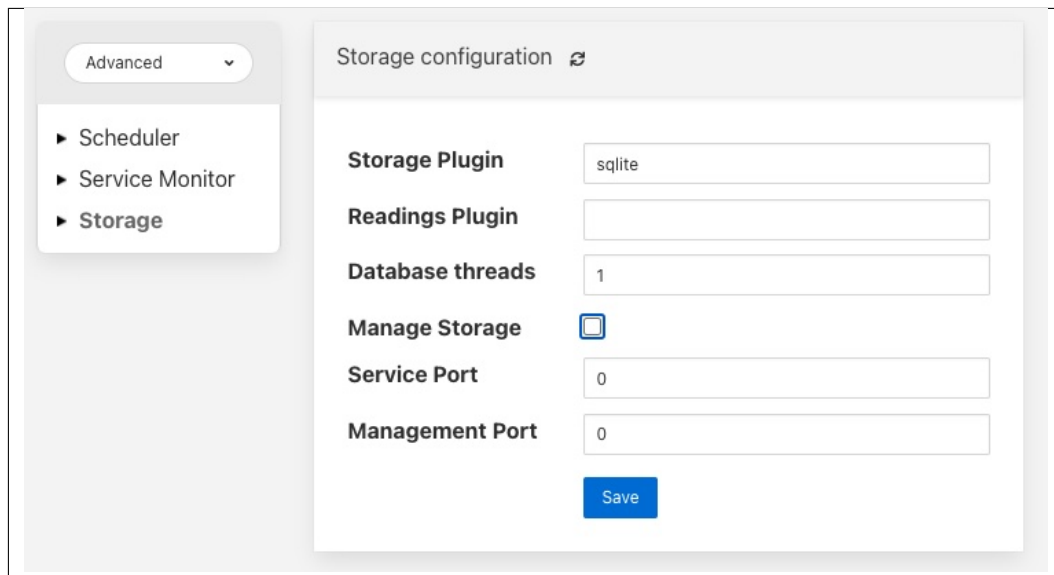
6.1 Configuring The Storage Plugin

Once installed the storage plugin can be reconfigured in much the same way as any Flir configuration, either using the API or the graphical user interface to set the storage engine and its options.

- Using the user interface to configuration the storage, select the *Configuration* item in the left hand menu bar.



- In the category pull down menu select *Advanced*.



- To change the storage plugin to use for both configuration and readings enter the name of the new plugin in the *Storage Plugin* entry field. If *Readings Plugin* is left empty then the storage plugin will also be used to store reading data. The default set of plugins installed with Flir that can be used as *Storage Plugin* values are:
 - *sqlite* - the SQLite file based storage engine.
 - *postgres* - the PostgreSQL server. Note the Postgres server is not installed by default when Flir is installed and must be installed before it can be used.
- The *Readings Plugin* may be set to any of the above and may also be set to use the SQLite In Memory plugin by entering the value *sqlitememory* into the configuration field.

- The *Database threads* field allows for the number of threads used for database housekeeping to be controlled. In normal circumstances 1 is sufficient. If performance issues are seen this can be increased however it is rarely required to be greater than 1 and can have counter productive effects on heavily loaded systems.
- The *Manage Storage* option is only used when the database storage uses an external database server, such as PostgreSQL. Toggling this option on causes Flir to start as stop the database server when Flir is started and stopped. If it is left off then Flir will assume the database server is running when it starts.
- The *Management Port* and *Service Port* options allow fixed ports to be assigned to the storage service. These settings are for debugging purposes only and the values should be set to 0 in normal operation.

Note: Additional storage engines may be installed to extend the set that is delivered with the standard Flir installation. These will be documented in the packages that provide the storage plugin.

Storage plugin configurations are not dynamic and Flir *must* be restarted after changing these values. Changing the plugin used to store readings will *not* cause the data in the previous storage system to be migrated to the new storage system and this data may be lost if it has not been sent onward from Flir.

6.1.1 SQLite Plugin Configuration

The SQLite plugin has a more complex set of configuration options that can be used to configure how and when it creates more database to accommodate more distinct assets. This plugin is designed to allow greater ingest rates for readings by separating the readings for each asset into a database table for that asset. It does however result in limiting the number of distinct assets that can be handled due to the requirement to handle large number of database files.

- **Purge Exclusions:** This option allows the user to specify that the purge process should not be applied to particular assets. The user can give a comma separated list of asset names that should be excluded from the purge process. Note, it is recommended that this option is only used for extremely low bandwidth, lookup data that would otherwise be completely purged from the system when the purge process runs.
- **Pool Size:** The number of connections to create in the database connection pool.
- **No. Readings per database:** This option control how many assets can be stored in a single database. Each asset will be stored in a distinct table within the database. Once all tables within a database are allocated the plugin will use more databases to store further assets.
- **No. databases allocate in advance:** This option defines how many databases are create initially by the SQLite plugin.

- **Database allocation threshold:** The number of unused databases that must exist within the system. Once the number of available databases falls below this value the system will begin the process of creating extra databases.
- **Database allocation size:** The number of databases to create when the above threshold is crossed. Database creation is a slow process and hence the tuning of these parameters can impact performance when an instance receives a large number of new asset names for which it has previously not allocated readings tables.

6.2 Installing A PostgreSQL server

The precise commands needed to install a PostgreSQL server vary for system to system, in general a packaged version of PostgreSQL is best used and these are often available within the standard package repositories for your system.

6.2.1 Ubuntu Install

On Ubuntu or other apt based distributions the command to install postgres:

```
sudo apt install -y postgresql postgresql-client
```

Now, make sure that PostgreSQL is installed and running correctly:

```
sudo systemctl status postgresql
```

Before you proceed, you must create a PostgreSQL user that matches your Linux user. Supposing that user is `<flir_user>`, type:

```
sudo -u postgres createuser -d <flir_user>
```

The `-d` argument is important because the user will need to create the Flir database.

A more generic command is:

```
sudo -u postgres createuser -d $(whoami)
```

6.2.2 CentOS/Red Hat Install

On CentOS and Red Hat systems, and other RPM based distributions the command is

```
sudo yum install -y https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_
↪64/pgdg-redhat-repo-latest.noarch.rpm
sudo yum install -y postgresql96-server
sudo yum install -y postgresql96-devel
sudo yum install -y rh-postgresql96
sudo yum install -y rh-postgresql96-postgresql-devel
sudo /usr/pgsql-9.6/bin/postgresql96-setup initdb
sudo systemctl enable postgresql-9.6
sudo systemctl start postgresql-9.6
```

At this point, Postgres has been configured to start at boot and it should be up and running. You can always check the status of the database server with `systemctl status postgresql-9.6`:

```
sudo systemctl status postgresql-9.6
```

Next, you must create a PostgreSQL user that matches your Linux user.

```
sudo -u postgres createuser -d $(whoami)
```

Finally, add `/usr/pgsql-9.6/bin` to your `PATH` environment variable in `$HOME/.bash_profile`. the new `PATH` setting in the file should look something like this:

```
PATH=$PATH:$HOME/.local/bin:$HOME/bin:/usr/pgsql-9.6/bin
```

6.3 SQLite Plugin Configuration

The SQLite storage engine has further options that may be used to configure its behavior. To access these configuration parameters click on the *sqlite* option under the *Storage* category in the configuration page.

The screenshot shows the 'Storage Plugin' configuration page. On the left, a sidebar has a dropdown menu with 'Advanced' selected, and a list of options: 'Scheduler', 'Service Monitor', 'Storage' (expanded), and 'sqlite'. The main content area is titled 'Storage Plugin' and contains the following settings:

- Pool Size:** 5
- No. Readings per database:** 15
- No. databases to allocate in advance:** 3
- Database allocation threshold:** 1
- Database allocation size:** 2

A blue 'Save' button is located at the bottom right of the configuration area.

Many of these configuration options control the performance of SQLite and it is important to have some background on how readings are stored within SQLite. The storage plugin stores readings for each distinct asset in a table for that asset. These tables are stored within a database. In order to improve concurrency multiple databases are used within the storage plugin. A set of parameters are used to define how these tables and databases are used.

- **Pool Size:** The number of connections to maintain to the database server.
- **No. Readings per database:** This controls the number of different assets that will be stored in each database file within SQLite.
- **No. databases to allocate in advance:** The number of SQLite databases that will be created at startup.
- **Database allocation threshold:** The point at which new databases are created. If the number of empty databases falls below this value then another set of databases will be created.
- **Database allocation size:** The number of database to allocate each time a new set of databases is required.

The setting of these parameters also imposes an upper limit on the number of assets that can be stored within a Flir instance as SQLite has a maximum limit of 61 databases that can be in use at any time. Therefore the maximum number of readings is 60 times the number of readings per database. One database is reserved for the configuration data.

Many factors will impact the performance of a Flir system

- The CPU, memory and storage performance of the underlying hardware
- The communication channel performance to the sensors
- The communications to the north systems
- The choice of storage system
- The external demands via the public REST API

Many of these are outside of the control of Flir itself, however it is possible to tune the way Flir will use certain resources to achieve better performance within the constraints of a deployment environment.

7.1 South Service Advanced Configuration

The south services within Flir each have a set of advanced configuration options defined for them. These are accessed by editing the configuration of the south service itself. There is a link titled *Show Advanced Config* to the right of the screen between the main configuration parameters and the *Enabled* option. Clicking on this link will show the following panel of advanced configuration options.

[Hide Advanced Config](#)

Maximum Reading Latency (mS)	<input type="text" value="5000"/>
Maximum buffered Readings	<input type="text" value="100"/>
Reading Rate	<input type="text" value="1"/>
Throttle	<input type="checkbox"/>
Reading Rate Per	<input type="text" value="second"/>
Minimum Log Level	<input type="text" value="warning"/>
Enabled	<input checked="" type="checkbox"/>

- *Maximum Reading Latency (mS)* - This is the maximum period of time for which a south service will buffer a reading before sending it onward to the storage layer. The value is expressed in milliseconds and it effectively defines the maximum time you can expect to wait before being able to view the data ingested by this south service.
- *Maximum buffered Readings* - This is the maximum number of readings the south service will buffer before attempting to send those readings onward to the storage service. This and the setting above work together to define the buffering strategy of the south service.
- *Reading Rate* - The rate at which polling occurs for this south service. This parameter only has effect if your south plugin is polled, asynchronous south services do not use this parameter. The units are defined by the setting of the *Reading Rate Per* item.
- *Throttle* - If enabled this allows the reading rate to be throttled by the south service. The service will attempt to poll at the rate defined by *Reading Rate*, however if this is not possible, because the readings are being forwarded out of the south service at a lower rate, the reading rate will be reduced to prevent the buffering in the south service from becoming overrun.
- *Reading Rate Per* - This defines the units to be used in the *Reading Rate* value. It allows the selection of per *second*, *minute* or *hour*.
- *Minimum Log Level* - This configuration option can be used to set the logs that will be seen for this service. It defines the level of logging that is sent to the syslog and may be set to *error*, *warning*, *info* or *debug*. Logs of the level selected and higher will be sent to the syslog.

7.1.1 Tuning Buffer Usage

The tuning of the south service allows the way the buffering is used within the south service to be controlled. Setting the latency value low results in frequent calls to send data to the storage service and therefore means data is more quickly available. However sending small quantities of data in each call the the storage system does not result in the most optimal use of the communications or of the storage engine itself. Setting a higher latency value results in more data being sent per transaction with the storage system and a more efficient system. The cost of this is the requirement for more in-memory storage within the south service.

Setting the *Maximum buffers Readings* value allows the user to place a cap on the amount of memory used to buffer within the south service, since when this value is reach, regardless of the age of the data and the setting of the latency parameter, the data will be sent to the storage service. Setting this to a smaller value allows tighter control on the memory footprint at the cost of less efficient use of the communication and storage service.

Tuning between performance, latency and memory usage is always a balancing act, there are situations where the performance requirements mean that a high latency will need to be incurred in order to make the most efficient use of the communications between the micro services and the transnational performance of the storage engine. Likewise the memory resources available for buffering may restrict the performance obtainable.

Notifications Service

Flir supports an optional service, known as the notification service that adds an event engine to the Flir installation. The notification services observed data as it flows into the Flir storage service buffer and processes that data against a set of rules that are configurable by the user to determine if an event has occurred. Events may be either when a condition that was previously not met being is, or a condition that was previously met becoming no longer true. The notification service can then send a notification when an event occurs or, in the case of a condition that is met, it can send notifications as long as that condition is met.

The notification services operates on data that is in the storage layer, and is independent of the individual south services. This means that the notification rules can use data from several south services to evaluate if a condition has occurred. Also the data that is observed by the notification is after any filtering rules have been applied in the south services but before any filtering that occurs in the north tasks. The mechanism used to allow the notification service to observe data is that the notifications register with the storage service to be given the values for particular assets as they arrive at the storage service. A notification may register for several assets and is free to buffer that data internally within the notification service. This registration does not impact how the data that is requested is treated in the rest of the system; it will still for example follow the normal processing rules to be sent onward to the north systems.

8.1 Notifications

The notification services manages *Notifications*, these are a set of parameters that it uses to determine if an event has occurred and a notification delivery should be made on the basis of that event.

A notification within the notification service consists of;

- A notification rule plugin that contains the logic to evaluate if a rule has been triggered, thus creating an event.
- A set of assets that are required to execute a notification rule.
- Information that defines how the data for each asset should be delivered to the notification rule.
- Configuration for the rule plugin that customizes that logic to this notification instance.
- A delivery plugin that provides the mechanism to delivery an event to destination for the notification.
- Configuration that may be required for the delivery plugin to operate.

8.1.1 Notification Rules

Notification rules are the logic that is used by the notification to determine if an event has occurred or not. An event is basically based on the values of a number of attributes, either at a single point in time or over a period of time. The notification services is delivered with one built in rule, this is a very simple rule called the *threshold rule* it simply looks at a single asset to determine if the value of a datapoint within the asset goes above or below a set value.

A notification rule has associated with it a set of configuration options, these define how the plugin behaves but also what data the plugin requires to execute the evaluation logic within the plugin. These configuration parameters can be divided into two sets; those items that define the data the rule requires from the notification service itself and those that relate directly to the logic of the rule.

A rule may work across one or more assets, the assets it requires are configured in the rule configuration and passed the the notification service to enable the service to subscribe to those assets and be sent that data by the storage service. A rule plugin may ask for every value of the asset as it changes or it may ask for a window of data. A window is defined as the values of an asset within a given time frame. An example might be the last 10 minutes of values. In the case of the window the rule may be passed the average value, minimum, maximum or all values in that window. The requirements about how data is delivered to a rule may be hard coded within the logic of a rule or may be part of the configuration a user of the rule should provide.

The second type of configuration parameter a rule might include are those that control the logic itself, in the example of the *threshold rule* this would be the threshold value itself and the control if the event is considered to have triggered if the value is above or below the threshold.

The section contains a full list of currently available rule plugins for Flir. As with other plugin types they are designed to be easily written by end users and developers, a guide is available for anyone wishing to write a notification rule plugin of their own.

8.1.2 Notification Types

Notifications can be delivered under a number of different conditions based on the state returned from a notification rule and how it related to the previous state returned by the notification rule, this is known as the notification type. A notification may be one of three types, these types are used to define when and how often notification are delivered.

One shot

A one shot notification is sent once when the notification triggers but will not be resent again if the notification triggers on successive evaluations. Once the evaluation does not trigger, the notification is cleared and will be sent again the next time the notification rule triggers.

One shot notifications may be further tailored with a maximum repeat frequency, e.g. no more than once in any 15 minute period.

Toggle

A toggle notification is sent when the notification rule triggers and will not be resent again until the rule fails to trigger, in exactly the same way as a one shot trigger. However in this case when the notification rule first stops triggering a cleared notification is sent.

Again this may be modified by the addition of a maximum repeat frequency.

Retriggered

A retriggered notification will continue to be sent when a notification rule triggers. The rate at which the notification is sent can be controlled by a maximum repeat frequency, e.g. send a notification every 5 minutes until the condition fails to trigger.

8.1.3 Notification Delivery

The notification service does not natively support any form of notification delivery, it relies upon a notification delivery plugin in order to delivery a notification of an event to a user or external system that should be alerted to the event that has occurred. Typical notification deliveries might be to alert a user via some form of paging or messaging system, push an event to an external application by sending some machine level message, execute an external program or code segment to make an action occur, switching on an indication light or in extreme cases maybe shutting down a machine for which a critical fault has been detected. The section contains a full list of currently available notification delivery plugins, however like other plugins these are easily extended and a guide is available for writing notification plugins to extend the available set of plugins.

8.2 Installing the Notification Service

The notification service is not part of the base Flir installation and is not a plugin, it is a separate microservice dedicated to the detection of events and the sending of notifications. The service is stored in a separate source repository, *flir-service-notification* and is packaged as a separate binary package for installation.

8.2.1 Building Notification Service

As with *Flir* itself there is always the option to build the notification service from the source code repository. This is only recommended if you also built your *Flir* from source code, if you did not then you should first do this before building the notification, otherwise you should install a binary package of the notification service.

The steps involved in building the notification service, assuming you have already built Flir itself and the environment variable *FLIR_ROOT* points to where you built your *Flir*, are;

```
$ git clone https://github.com/flir/flir-service-notification.git
...
$ cd flir-service-notification
$ ./requirements.sh
...
$ mkdir build
$ cd build
$ cmake ..
...
$ make
...
```

This will result in the creation of a notification service binary, you now need to copy that binary into the *Flir* installation. There are two options here, one if you used *make install* to create your installation and the other if you are running directly from the build environment.

If you used *make install* to create your *Flir* installation then simply run *make install* to install your notification service. This should be run from the *build* directory under the *flir-service-notification* directory.

```
$ make install
```

Note: You may need to run *make install* under a *sudo* command if your user does not have permissions to write to the installation directory. If you use a *DESTDIR=...* option to the *make install* of *Flir* then you should use the same *DESTDIR=...* option here also.

If you are running your *Flir* directly from the build environment, then execute the command

```
$ cp ./C/services/notification/flir.services.notification $FLIR_ROOT/services
```

8.2.2 Installing Notification Service Package

If you are using the packaged binaries for your system then you can use the package manager to install the *flir-service-notification* package. The exact command depends on your package manager and how you obtained your packages.

If you downloaded your packages then you should navigate to the directory that contains your package files and run the package manager. If you have deb package files run the command

```
$ sudo apt -y install ./flir-service-notification-1.7.0-armhf.deb
```

Note: The version number, 1.7.0 may be different on your system, this will depend which version you have downloaded. Also the armhf may be different for your machine architecture. Verify the precise name of your package before running the above command.

If you are using a RedHat or CentOS distribution and have rpm package files then run the command

```
$ sudo yum -y localinstall ./flir-service-notification-1.7.0-x86_64.deb
```

Note: The version number, 1.7.0 may be different on your system, this will depend which version you have downloaded. Verify the precise name of your package before running the above command.

If you have configured your system to search a package repository that contains the Flir packages then you can simply run the command

```
$ sudo apt-get -y install flir-service-notification
```

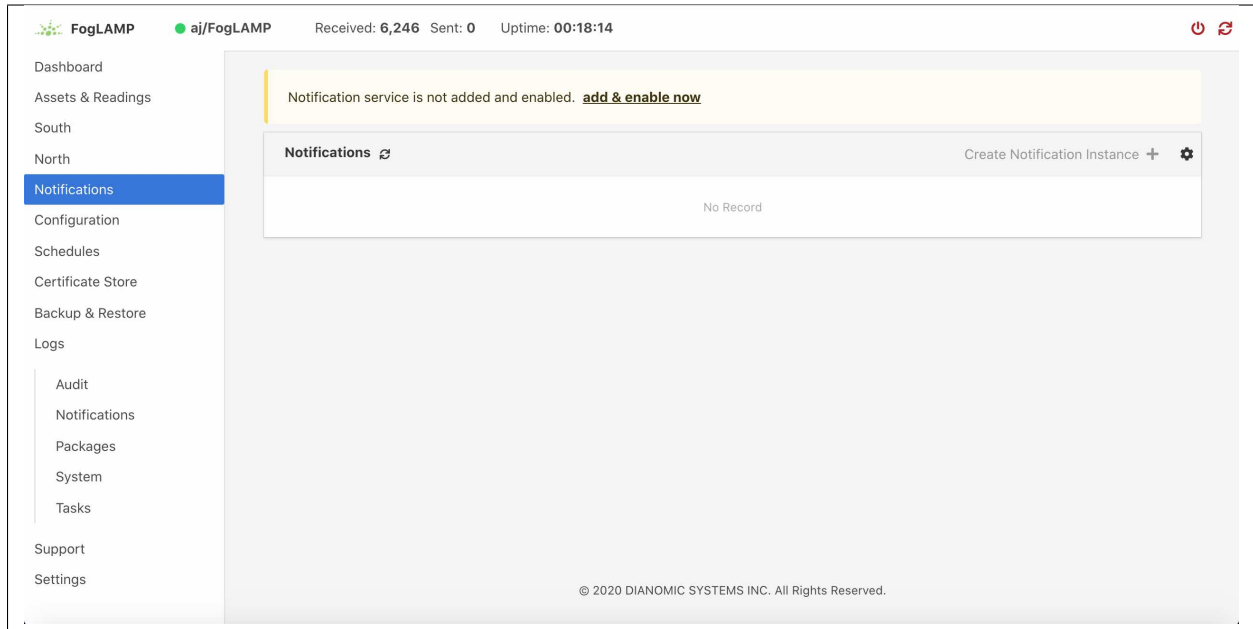
On a Debian/Ubuntu system, or

```
$ sudo yum -y install flir-service-notification
```

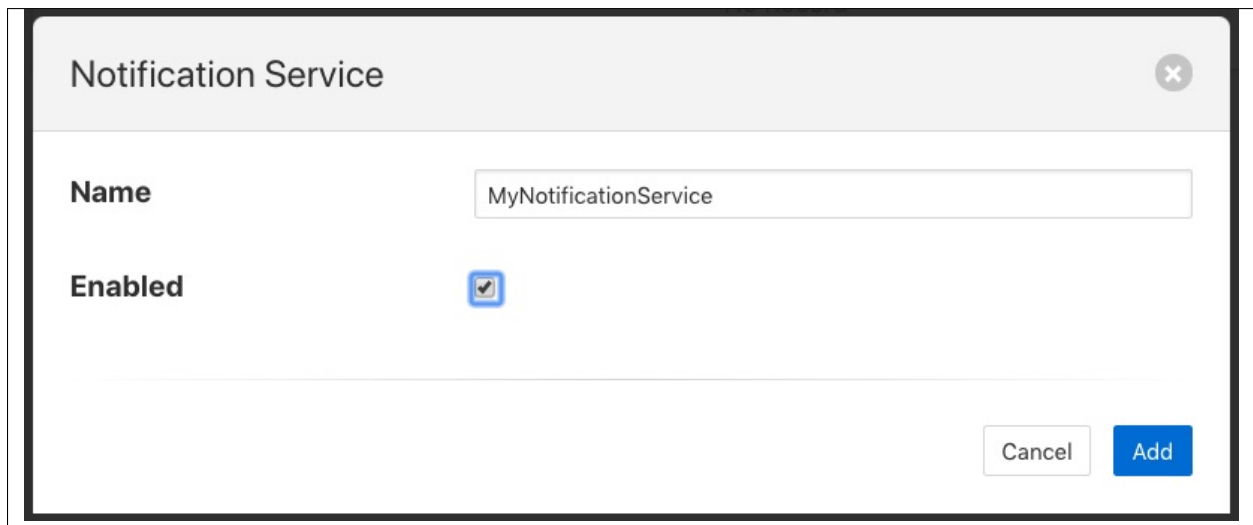
On a RedHat/CentOS system. This will install the latest version of the notification service on your machine.

8.3 Starting The Notification Service

Once installed you must configure Flir to start the notification service. This is simply done from the GUI by selecting the *Notifications* option from the left-hand menu. In the page that is then shown you will see a panel at the top that allows you to *add & enable now* the notification service. This only appears if one has not already been added.

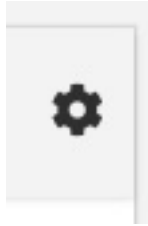


Select this link to *add & enable now* the notification service, a new dialog will appear that allows you to name and enable your service.



8.4 Configuring The Notification Service

Once the notification service has been added and enabled a new icon will appear in the *Notifications* page that allows you to configure the notification service. The icon appears in the top right and is in the shape of a gear wheel.



Clicking on this icon will display the notification service configuration dialog.

MyNotificationService Notification Service

Minimum Log Level

warning

Maximun number of delivery threads

2

Enabled

☒

Cancel

Save

Delete Service

You can use this dialog to control the level of logging that is done from the service by setting the *Minimum Log Level* to the least severity log level you wish to see. All log entries at the select level and of greater severity will be logged.

It is also possible to set the number of threads that will be used for delivering notifications. This defines how many notifications can be delivered in parallel. This only needs to be increased if the delivery process of any of the in use delivery plugins are long running.

The final setting allows you to disable the notification service.

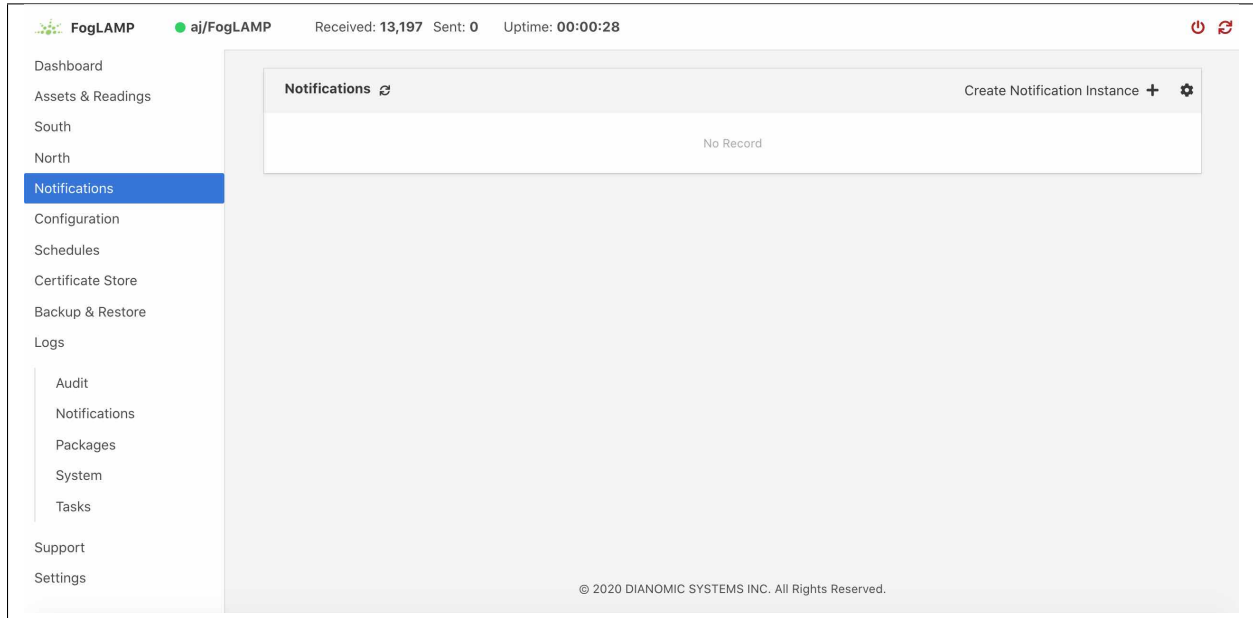
Once you have updated the configuration of the service click on *Save*.

It is also possible to delete the notification service using the *Delete Service* button at the bottom of this dialog.

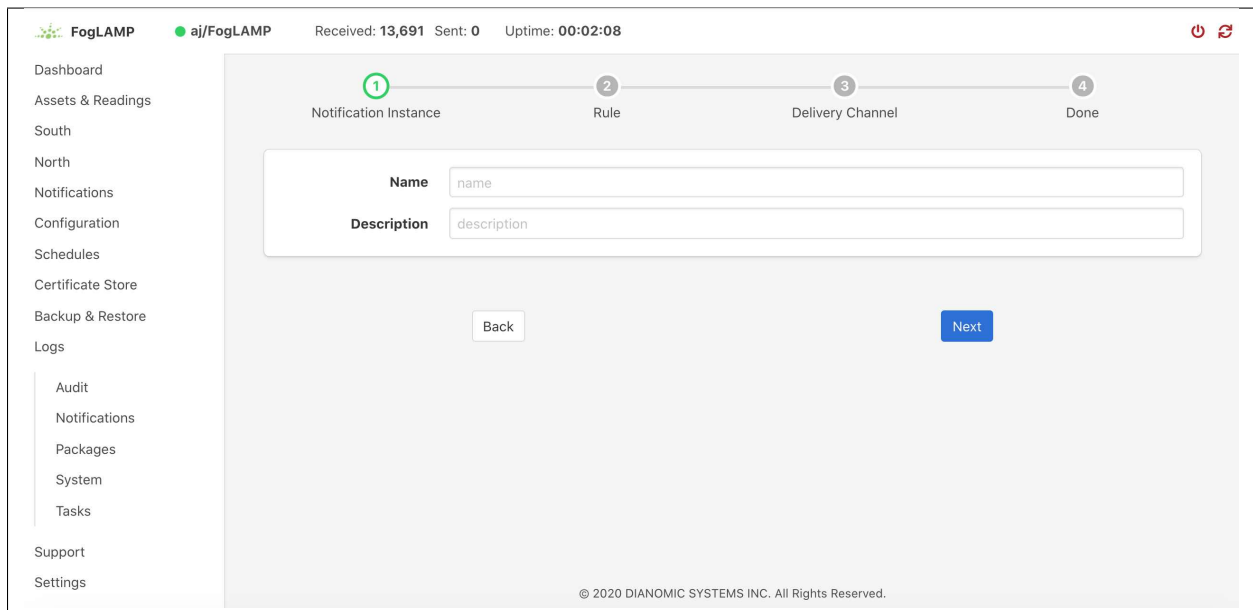
8.5 Using The Notification Service

8.5.1 Add A Notification

In order to add a notification, select the Notifications page in the left-hand menu, an empty set of notifications will appear.



Click on the + icon to add a new notification.



You will be presented with a dialog to enter a name and description for your notification.

The screenshot shows the first step of a four-step wizard. At the top, a progress bar has four numbered circles: 1 (green, active), 2 (grey), 3 (grey), and 4 (grey). Below the progress bar are the labels: 'Notification Instance', 'Rule', 'Delivery Channel', and 'Done'. The main content area has a 'Name' field with the text 'Above0.5' and a 'Description' field with the text 'Above0.5 notification instance'. At the bottom, there are 'Back' and 'Next' buttons.

Enter text for the name you require, a suggested description will be automatically added, however you can modify this to any string you desire. When complete click on the *Next* button to move forwards in the definition process. You can always click on *Previous* to go back a screen and modify what has been entered.

The screenshot shows the second step of the wizard. The progress bar now has circle 2 highlighted in green. The labels are 'Notification Instance', 'Rule', 'Delivery Channel', and 'Done'. The main content area features a 'Rule Plugin' dropdown menu with options: 'Average', 'OutOfBound', 'SimpleExpression', and 'Threshold' (which is highlighted in blue). To the right of the dropdown, the text 'Generate a notification when datapoint value crosses a boundary.' is displayed. Below the dropdown is a link labeled 'available plugins'. At the bottom, there are 'Previous' and 'Next' buttons.

You are presented with the set of installed rules on the system. If the rule you wish to use is not installed and you wish to install it then use the link *available plugins* to be presented with the list of plugins that are available to be installed.

Note: The *available plugins* link will only work if you have added the Flir package repository to the package manager of your system.

When you select a rule plugin a short description of what the rules does will be displayed to the right of the list. In this example we will use the threshold rule that is built into the notification service. Click on *Next* once you have selected

the rule you wish to use.

The screenshot displays a four-step configuration process for a notification. The steps are: 1. Notification Instance, 2. Rule (currently active), 3. Delivery Channel, and 4. Done. The 'Rule' step contains the following configuration fields:

Field	Value
Asset name	FastSine
Datapoint name	sinusoid
Condition	>
Trigger value	0.5
Evaluation data	Single Item
Window evaluation	Average
Time window	30

At the bottom of the configuration area, there are two buttons: 'Previous' and 'Next'.

You will be presented with the configuration parameters applicable to the rule you have chosen. Enter the name of the asset and the datapoint within that asset that you wish the rule to operate on. In the case of the *threshold* rule you can also define if you want the rule to trigger if the value is greater than, greater than or equal, less than or less than or equal to a *Trigger value*.

You can also choose to look at *Single Item* or *Window* data. If you choose the later you can then choose to define if the minimum, maximum or average within the window that must cross the threshold value.

The screenshot shows the 'Rule' configuration step (step 2 of 4) in a notification setup wizard. The progress bar at the top indicates the current step. The form contains the following fields:

- Asset name:** FastSine
- Datapoint name:** sinusoid
- Condition:** > (with a dropdown arrow)
- Trigger value:** 0.5
- Evaluation data:** A dropdown menu is open, showing options: Maximum, Minimum, and Average (selected with a checkmark).
- Window evaluation:** (This label is present but has no associated input field visible)
- Time window:** 30

At the bottom, there are 'Previous' and 'Next' buttons.

Once you have set the parameters for the rule click on the *Next* button to select the delivery plugin to use to delivery the notification data.

The screenshot shows the 'Delivery Channel' configuration step (step 3 of 4) in the notification setup wizard. The progress bar at the top indicates the current step. The form contains the following elements:

- Delivery Plugin:** A list of available plugins: alexa, asset, Blynk, and email.
- available plugins:** A link to view more options.

At the bottom, there are 'Previous' and 'Next' buttons.

A list of available delivery plugins will be presented, along with a similar link that allows you to install new delivery

plugins if desired. As you select a plugin a short text description will be displayed to the right of the plugin list. In this example we will select the *Slack* messaging platform for the delivery of the notification.

Once you have selected the plugin you wish to use click on the *Next* button.

The screenshot shows a four-step progress bar at the top: 1 Notification Instance, 2 Rule, 3 Delivery Channel (active), and 4 Done. Below the progress bar is a configuration form for the Slack Webhook plugin. The form includes three fields: 'Slack Webhook URL' with the value 'https://hooks.slack.com/services/T2GBZ52AF/BGFNTP7NG/YJxQwiJda5ZHMirFZqUjUci', 'Message Text' with the value 'The value of the sinusoid is greater than 0.5', and an 'Enabled' checkbox which is checked. At the bottom of the form are two buttons: 'Previous' and 'Next'.

You will then be presented with the configuration parameters the delivery plugin requires to deliver the notification. In the case of the *Slack* plugin this consists of the webhook that you should obtain from the *Slack* application and a message text that will be sent when the event triggers.

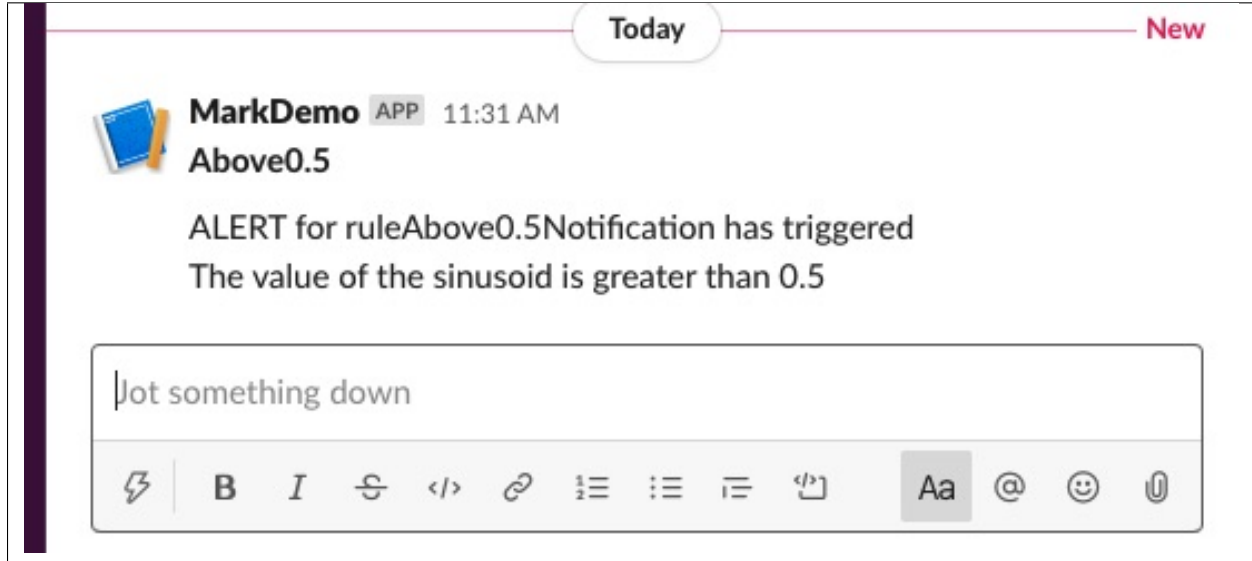
Note: You may disable the delivery of a notification separately to enabling or disabling the notification. This allows you to test the logic of a notification without delivering the notification. Entries will still be made in the notification log when delivery is disabled.

Once you have completed the configuration of the delivery plugin click on *Next* to move to the final stage in setting up your notification.

The screenshot shows the same four-step progress bar, but now step 4 'Done' is active. The configuration form below it has a 'Trigger' dropdown menu open, showing three options: 'one shot', 'retriggered', and 'toggled'. The 'one shot' option is selected. There is also an 'Enabled' checkbox which is checked. At the bottom right of the form is a 'Done' button.

The final stage of setting up your configuration is to set the notification type and the retrigger time for the notification. Enable the notification and click on *Done* to complete setting up your notification.

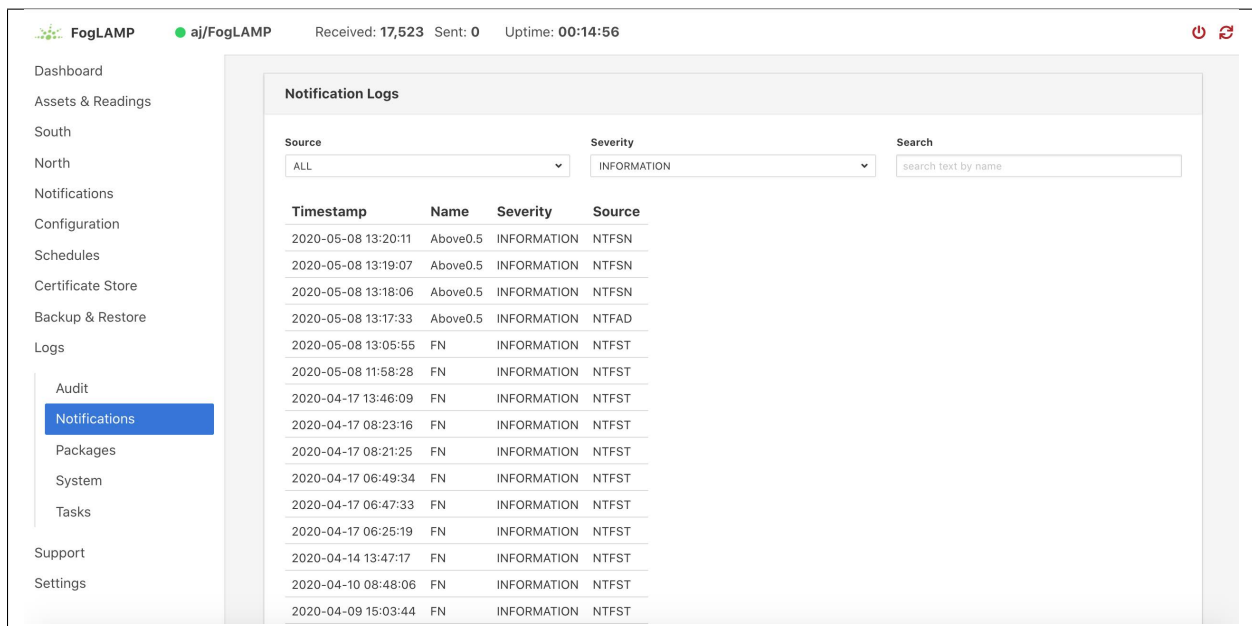
After a period of time, when a *sinusoid* value greater than 0.5 is received, a message will appear in your *Slack* window.



This will repeat at a maximum rate defined by the *Retrigger Time* whenever a value of greater than 0,5 is received.

Notification Log

You can see activity related to the notification service by selecting the *Notifications* option under *Logs* in the left-hand menu.



You may filter this output using the drop down menus along the top of the page. The list to the left defines the type of event that you filter, clicking on this list will show you the meaning of the different audit types.

Notification Logs

Count: 13

ALL

ALL

NTFDL - Notification Deleted

NTFAD - Notification Added

NTFSN - Notification Sent

NTFCL - Notification Cleared

NTFST - Notification Server Startup

NTFSD - Notification Server Shutdown

INFORMATION

Name

		Severity	Source
		INFORMATION	NTFSN
		INFORMATION	NTFSN
		INFORMATION	NTFSN
		INFORMATION	NTFSN
		INFORMATION	NTFST
		INFORMATION	NTFST
2020-04-21 09:20:15	MyNotificationService	INFORMATION	NTFST
2020-04-21 09:15:32	Above0.5	INFORMATION	NTFAD
2020-04-20 14:47:38	MyNotificationService	INFORMATION	NTFST
2020-04-20 14:28:17	MyNotificationService	INFORMATION	NTFST
2020-04-20 13:56:17	MyNotificationService	INFORMATION	NTFST
2020-04-20 13:54:27	MyNotificationService	INFORMATION	NTFST
2020-04-20 11:22:38	MyNotificationService	INFORMATION	NTFST

8.5.2 Editing Notifications

It is possible to update existing notifications or remove them using the *Notifications* option from the left-hand menu. Clicking on *Notifications* will bring up a list of the currently defined notifications within the system.

The screenshot displays the FogLAMP web interface. At the top, the header shows 'FogLAMP' with a green status indicator, 'aj/FogLAMP', and system statistics: 'Received: 16,844 Sent: 0 Uptime: 00:12:40'. On the left, a sidebar menu lists various sections: Dashboard, Assets & Readings, South, North, Notifications (highlighted in blue), Configuration, Schedules, Certificate Store, Backup & Restore, Logs, Audit, Notifications, Packages, System, Tasks, Support, and Settings. The main content area is titled 'Notifications' and includes a 'Create Notification Instance + ⚙️' button. Below this is a table with the following data:

Name	Channel	Rule	Type	Status
Above0.5	Telegram	Threshold	one shot	enabled

At the bottom of the interface, a copyright notice reads: '© 2020 DIANOMIC SYSTEMS INC. All Rights Reserved.'

Click on the name of the notification of interest to display the details of that notification and allow it to be edited.

Above0.5

Asset name

sinusoid

Datapoint name

sinusoid

Condition

>

Trigger value

0.5

Evaluation data

Single Item

Window evaluation

Maximum

Time window

30

Delivery Channel - slack

Slack Webhook URL

https://hooks.slack.com/services/T2GBZ52AF/BLH4E9VPX/SpTueiK9t73KSaNSe3

Message Text

The value of the sinusoid is greater than 0.5

Enabled

☒

Cancel

Save

A single page dialog appears that allows you to change any of the parameters of you notification.

Note: You can not change the rule plugin or delivery plugin you are using. If you wish to change either of these then you must delete this notification and create a new one with the desired plugins.

Once you have updated your notification click *Save* to action the changes.

If you wish to delete your notification this may be done by clicking the *Delete* button at the base of the dialog.

Set Point Control

Flir supports facilities that allows control of devices via the south service and plugins. This control is known as *set point control* as it is not intended for real time critical control of devices but rather to modify the behavior of a device based on one of many different information flows. The latency involved in these control operations is highly dependent on the control path itself and also the scheduling limitations of the underlying operating system. Hence the caveat that the control functions are not real time or guaranteed to be actioned within a specified time window.

9.1 Control Functions

There are two types of control function supported

- Modify the value in a device via the south service and plugin.
- Request the device to perform an action.

9.1.1 Set Point

Setting the value within the device is known as a set point action in Flir. This can be as simple as setting a speed variable within a controller for a fan or it may be more complete. Typically a south plugin would provide a set of values that can be manipulated, giving each a symbolic name that would be available for a set point command. The exact nature of these is defined by the south plugin.

9.1.2 Operation

Operations, as the name implies provides a means for the south service to request a device to perform an operation, such as reset or re-calibrate. The names of these operations and any arguments that can be given are defined within the south plugin and are specific to that south plugin.

9.2 Control Paths

Set point control may be invoked via a number of paths with Flir

- As the result of a notification within Flir itself.
- As a result of a request via the Flir public REST API.
- As a result of a control message flowing from a north side system into a north plugin and being routed onward to the south service.

Currently only the notification method is fully implemented within Flir.

The use of a notification in the Flir instance itself provides the fastest response for an edge notification. All the processing for this is done on the edge by Flir itself.

9.2.1 Edge Based Control

Edge based control is the name we use for a class of control applications that take place solely within the Flir instance at the edge. The data that is required for the control decision to be made is gathered in the Flir instance, the logic to trigger the control action runs in the Flir instance and the control action is taken within the Flir instance. Typically this will involve one or more south plugins to gather the data required to make the control decision, possibly some filters to process that data, the notification engine to make the decision and one or more south services to deliver the control messages.

As an example of how edge based control might work lets consider the following case.

We have a machine tool that is being monitored by Flir using the OPC/UA south plugin to read data from the machine tools controlling PLC. As part of that data we receive an asset which contains the temperature of the motor which is running the tool. We can assume this asset is called *MotorTemperature* and it contains a single data point called *temperature*.

We also have a fan unit that is able to cool that motor which is controlled via a Modbus interface. The modbus contains one a coil that toggles the fan on and off and a register that controls the speed of the fan. We configure the *flir-south-modbus* as a service called *MotorFan* with a control map that will map the coil and register to a pair of set points.

```
{
  "values" : [
    {
      "name" : "run",
      "coil" : 1
    },
    {
      "name" : "speed",
      "register" : 1
    }
  ]
}
```

Control

Use Control Map ▾

Control Map

```
1 {  
2   "values": [  
3     {  
4       "name": "run",  
5       "coil": 1  
6     },  
7     {  
8       "name": "speed",  
9       "register": 1  
10    }  
11  ]  
12 }
```

If the measured temperature of the motor going above 35 degrees centigrade we want to turn the fan on at 1200 RPM. We create a new notification to do this. The notification uses the *threshold* rule and triggers if the asset *MotorTemperature*, data point *temperature* is greater than 35.

1

2

3

4

Notification Instance

Rule

Delivery Channel

Done

Asset name

MotorTemperature

Datapoint name

temperature

Condition

> ▾

Trigger value

0.0

Evaluation data

Single Item ▾

Window evaluation

Average ▾

Time window

30

Previous

Next

We select the *setpoint* delivery plugin from the list and configure it.

1 Notification Instance 2 Rule 3 Delivery Channel 4 Done

Service MotorFan

Trigger Value

```

1 {
2   "values": {
3     "run": "1",
4     "speed": 1200
5   }
6 }

```

Cleared Value

```

1 {
2   "values": {
3     "run": "0"
4   }
5 }

```

Enabled ☒

Previous Next

- In *Service* we set the name of the service we are going to use to control the fan, in this case *MotorFan*
- In *Trigger Value* we set the control message we are going to send to the service. This will turn the fan on and set the speed to 1200RPM
- In *Cleared Value* we set the control message we are going to send to turn off the fan when the value falls below 35 degrees.

The plugin is enabled and we go on to set the notification type to toggled, since we want to turn off the fan if the motor cools down, and set a retrigger time to prevent the fan switching on and off too quickly. The notification type and the retrigger time are important parameters for tuning the behavior of the control system and are discussed in more detail below.

If we required the fan to speed up at a higher temperature then this could be achieved with a second notification. In this case it would have a higher threshold value and would set the speed to a higher value in the trigger condition and set it back to 1200 in the cleared condition. Since the notification type is *toggled* the notification service will ensure that these are called in the correct order.

Data Substitution

There is another option that can be considered in our example above that would allow the fan speed to be dependent on the temperature, the use of data substitution in the *setpoint* notification delivery.

Data substitution allows the values of a data point in the asset that caused the notification rule to trigger to be substituted into the values passed in the set point operation. The data that is available in the substitution is the same data that is given to the notification rule that caused the alert to be triggered. This may be a single asset with all of its data points for simple rules or may be multiple assets for more complex rules. If the notification rule is given averaged data then it is these averages that will be available rather than the individual values.

Parameters are substituted using a simple macro mechanism, the name of an asset and data point within the asset is inserted into the value surrounded by the \$ character. For example to substitute the value of the *temperature* data point of the *MotorTemperature* asset into the *speed* set point parameter we would define the following in the *Trigger Value*

```
{
  "values" : {
    "speed" : "$MotorTemperature.temperature$"
  }
}
```

Note that we separate the asset name from the data point name using a period character.

This would have the effect of setting the fan speed to the temperature of the motor. Whilst allowing us to vary the speed based on temperature it would probably not be what we want as the fan speed is too low. We need a way to map a temperature to a higher speed.

A simple option is to use the macro mechanism to append a couple of 0s to the temperature, a temperature of 21 degrees would result in a fan speed of 2100 RPM.

```
{
  "values" : {
    "speed" : "$MotorTemperature.temperature$00"
  }
}
```

This works, but is a little primitive and limiting. Another option is to add data to the asset that triggers the notification. In this case we could add an expression filter to create a new data point with a desired fan speed. If we were to add an expression filter and give it the expression *desiredSpeed = temperature > 20 ? temperature * 50 + 1200 : 0* then we would create a new data point in the asset called *desiredSpeed*. The value of *desiredSpeed* would be 0 if the temperature was 20 degrees or below, however for temperatures above it would be 1200 plus 50 times the temperature.

This new desired speed can then be used to set the temperature in the *setpoint* notification plugin.

```
{
  "values" : {
    "speed" : "$MotorTemperature.desiredSpeed$"
  }
}
```

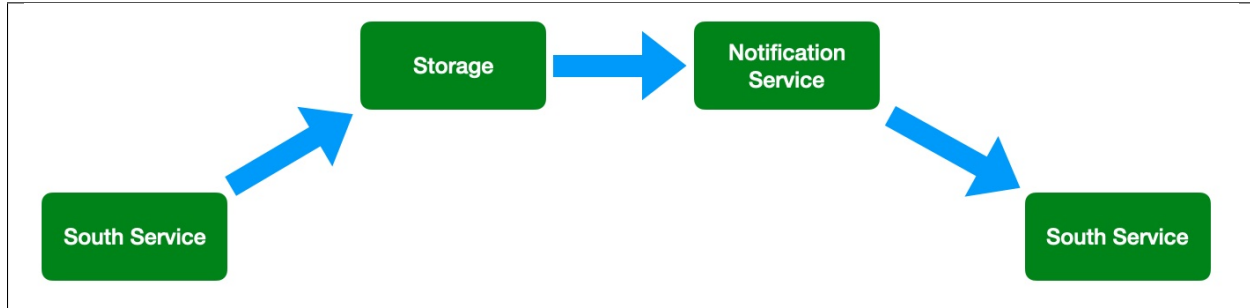
The user then has the choice of adding the desired speed item to the data stored in the north, or adding an asset filter in the north to remove this data point from the data that is sent onward to the north.

Tuning edge control systems

The set point control features of Flir are not intended to replace real time control applications such as would be seen in PLCs that are typically implemented in ladder logic, however Flir does allow for high performance control to be implemented within the edge device. The precise latency in control decisions is dependent on a large number of factors and there are various tuning parameters that can be used to reduce the latency in the control path.

In order to understand the latency inherent in the control path we should first start by examining that path to discover where latency can occur. To do this we will choose a simple case of a single south plugin that is gathering data required by a control decision within Flir. The control decision will be taken in a notification rule and delivered via the *flir-notify-setpoint* plugin to another south service.

A total of four services within Flir will be involved in the control path



- the south service that is gathering the data required for the decision
- the storage service that will dispatch the data to the notification service
- the notification service that will run the decision rule and trigger the delivery of the control message
- the south service that will send the control input to the device that is being controlled

Each of these services can add to that latency in the control path, however the way in which these are configured can significantly reduce that latency.

The south service that is gathering the data will typically be either polling a device or obtaining data asynchronously from the device. This will be sent to the ingest thread of the south service where it will be buffered before sending the data to the storage service.

The advanced settings for the south service can be used to trigger how often that data is sent to the storage service. Since it is the storage service that is responsible for routing the data onward to the notification service this impacts the latency of the delivery of the control messages.

[Hide Advanced Config](#)

Maximum Reading Latency (mS)	<input type="text" value="5000"/>
Maximum buffered Readings	<input type="text" value="100"/>
Reading Rate	<input type="text" value="1"/>
Throttle	<input type="checkbox"/>

The above shows the default configuration of a south service. In this case data will not be sent to the south service until there are either 100 readings buffered in the south service, or the oldest reading in the south service buffer has been in the buffer for 5000 milliseconds. In this example we are reading 1 new readings every second, therefore will send data to the storage service every 5 seconds, when the oldest reading in the buffer has been there for 5000mS. When it sends data it will send all the data it has buffered, in this case 5 readings as one block. If the oldest reading is the one that triggers the notification we have therefore introduced a 5 second latency into the control path.

The control path latency can be reduced by reducing the *Maximum Reading Latency* of this south plugin. This will of course put greater load on the system as a whole and should be done with caution as it increases the message traffic between the south service and the storage service.

The storage service has little impact on the latency, it is designed such that it will forward data it receives for buffering to the notification service in parallel to buffering it. The storage service will only forward data the notification service has subscribed to receive and will forward that data in the blocks it arrives at the storage service in. If a block of 5 readings arrives at the the storage service then all 5 will be sent to the notification service as a single block.

The next service in the edge control path is the notification service, this is perhaps the most complex step in the journey. The behavior of the notification service is very dependent upon how each individual notification instance has been configured, factors that are important are the notification type, the retrigger interval and the evaluation data options.

The notification type is used to determine when notifications are delivered to the delivery channel, in the case of edge control this might be the *setpoint* plugin or the *operation* plugin. Flir implements three options for the notification type

- **One shot:** A one shot notification is sent once when the notification triggers but will not be resent again if the notification triggers on successive evaluations. Once the evaluation does not trigger, the notification is cleared and will be sent again the next time the notification rule triggers. One shot notifications may be further tailored with a maximum repeat frequency, e.g. no more than once in any 15 minute period.
- **Toggle:** A toggle notification is sent when the notification rule triggers and will not be resent again until the rule fails to trigger, in exactly the same way as a one shot trigger. However in this case when the notification rule first stops triggering a cleared notification is sent. Again this may be modified by the addition of a maximum repeat frequency.
- **Retriggered:** A retriggered notification will continue to be sent when a notification rule triggers. The rate at which the notification is sent can be controlled by a maximum repeat frequency, e.g. send a notification every 5 minutes until the condition fails to trigger.

It is very important to choose the right type of notification in order to ensure the data delivered in your set point control path is what you require. The other factor that comes into play is the *Retrigger Time*, this defines a dead period during which notifications will not be sent regardless of the notification type.

Setting a retrigger time that is too high will mean that data that you expect to be sent will not be sent. For example if you a new value you wish to be updated once every 5 seconds then you should use a retrigger type notification and set the retrigger time to less than 5 seconds.

It is very important to understand however that the retrigger time defines when notifications can be delivered, it does not related to the interval between readings. As an example, assume we have a retrigger time of 1 second and a reading that arrives every 2 seconds that causes a notification to be sent.

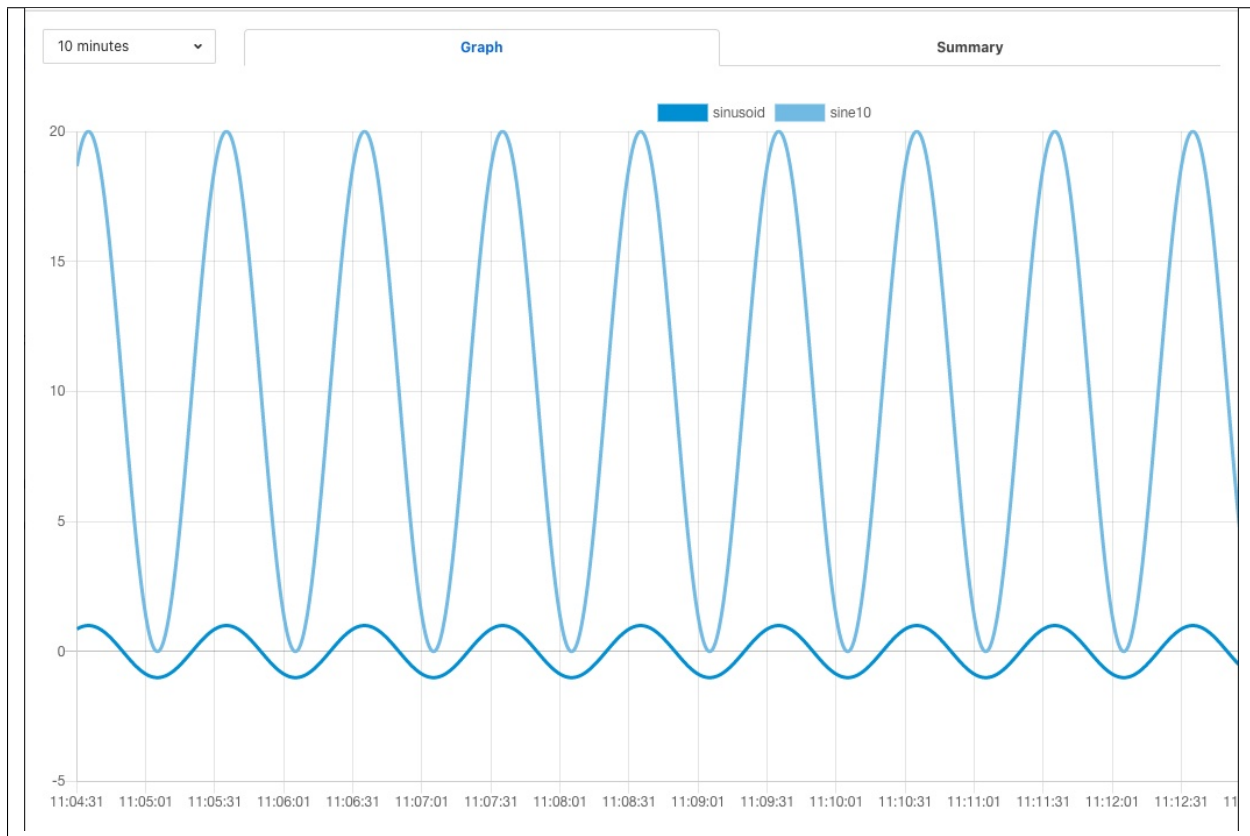
- If the south service is left with the default buffering configuration it will send the readings in a block to the storage service every 5 seconds, each block containing 2 readings.
- These are sent to the notification service in a single block of two readings.
- The notification will evaluate the rule against the first reading in the block.
- If the rule triggers the notification service will send the notification via the set point plugin.
- The notification service will now evaluate the rule against the second readings.
- If the rule triggers the notification service will note that it has been less than 1 second since it sent the last notification and it will not deliver another notification.

Therefore, in this case you appear to see only half of the data points you expect being delivered to you set point notification. In order to rectify this you must alter the tuning parameters of the south service to send data more frequently to the storage service.

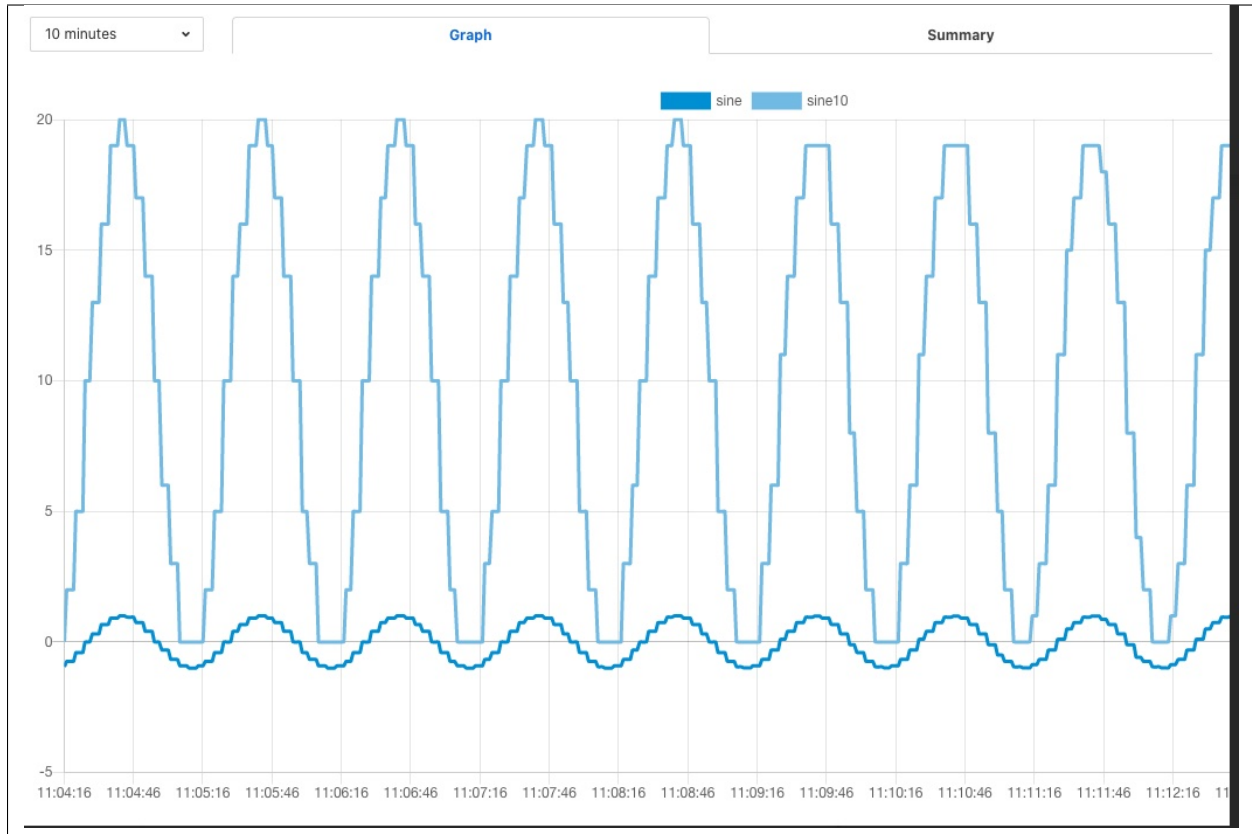
The final hop in the edge control path is the call from the notification service to the south service and the delivery via the plugin in the south service. This is done using the south service interface and is run on a separate thread in the south service. The result would normally be expected to be very low latency, however it should be noted that plugins commonly protect against simultaneous ingress and egress, therefore if the south service being used to deliver the data to the end device is also reading data from that device, there may be a requirement for the current read to complete before the write operation an commence.

To illustrate how the buffering in the south service might impact the data sent to the set point control service we will use a simple example of sine wave data being created by a south plugin and have every reading sent to a modbus device

and then read back from the modbus device. The input data as read at the south service gathering the data is a smooth sine wave,

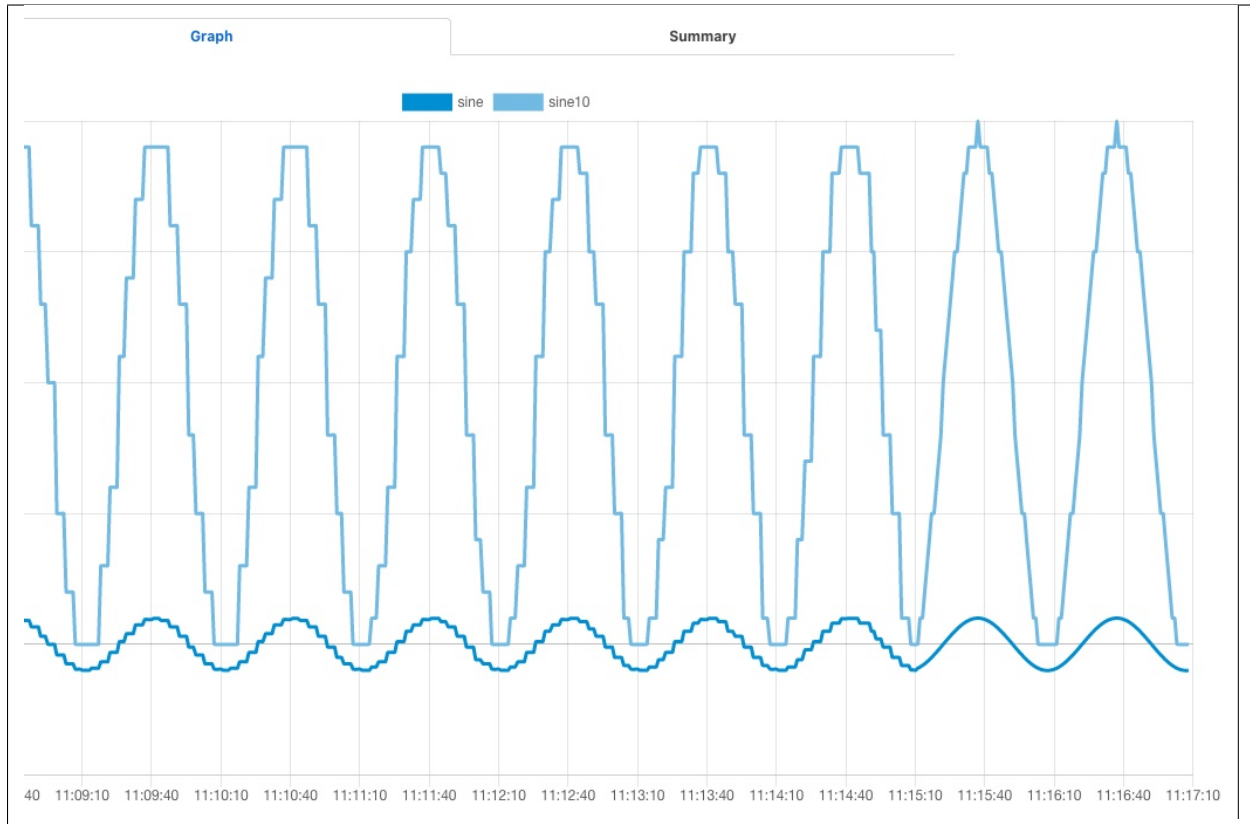


The data observed that is written to the modbus device is not however a clean sine wave as readings have been missed due to the retrigger time eliminating data that arrived in the same buffer.



Some jitter caused by occasional differences in the readings that arrive in a single block can be seen in the data as well.

Changing the buffering on the south service to only buffer a single reading results in a much smooth sine wave as can be seen below as the data is seen to transition from one buffering policy to the next.



At the left end of the graph the south service is buffering 5 readings before sending data onward, on the right end it is only buffering one reading.

Flir makes extensive use of plugin components to extend the base functionality of the platform. In particular, plugins are used to;

- Extend the set of sensors and actuators that Flir supports.
- Extend the set of services to which Flir will push accumulated data gathered from those sensors.
- The mechanism by which Flir buffers data internally.
- Filter plugins may be used to augment, edit or remove data as it flows through Flir.
- Rule plugins extend the rules that may trigger the delivery of notifications at the edge.
- Notification delivery plugins allow for new delivery mechanisms to be integrated into Flir.

This chapter presents the plugins that are bundled with Flir, how to write and use new plugins to support different sensors, protocols, historians and storage devices. It will guide you through the process and entry points that are required for the various different types of plugin.

There are also numerous plugins that are available as separate packages or in separate repositories that may be used with Flir.

10.1 Plugins

In this version of Flir you have six types of plugins:

- **South Plugins** - They are responsible for communication between Flir and the sensors and actuators they support. Each instance of a Flir South microservice will use a plugin for the actual communication to the sensors or actuators that that instance of the South microservice supports.
- **North Plugins** - They are responsible for taking reading data passed to them from the South bound service and doing any necessary conversion to the data and providing the protocol to send that converted data to a north-side task.
- **Storage Plugins** - They sit between the Storage microservice and the physical data storage mechanism that stores the Flir configuration and readings data. Storage plugins differ from other plugins in that they are written

exclusively in C/C++, however they share the same common attributes and entry points that the other filter must support.

- **Filter Plugins** - Filter plugins are used to modify data as it flows through Flir. Filter plugins may be combined into a set of ordered filters that are applied as a pipeline to either the south ingress service or the north egress task that sends data to external systems.
- **Notification Rule Plugins** - These are used by the optional notification service in order to evaluate data that flows into the notification service to determine if a notification should be sent.
- **Notification Delivery Plugins** - These plugins are used by the optional notification service to deliver a notification to a system when a notification rule has triggered. These plugins allow the mechanisms to deliver notifications to be extended.

10.1.1 Plugins in this version of Flir

This version of Flir provides the following plugins in the main repository:

Type	Name	Initial Status	Description	Availability	Notes
Storage	SQLite	Enabled	SQLite storage for data and metadata	Ubuntu: x86_64 Ubuntu Core: x86, ARM Raspbian	
Storage	Postgres	Disabled	PostgreSQL storage for data and metadata	Ubuntu: x86_64 Ubuntu Core: x86, ARM Raspbian	
North	OMF	Disabled	OSISoft Message Format sender to PI Connector Relay OMF	Ubuntu: x86_64 Ubuntu Core: x86, ARM Raspbian	It works with PI Connector Relay OMF 1.2.X and 2.2. The plugin also works against EDS and OCS.

In addition to the plugins in the main repository, there are many other plugins available in separate repositories, a list of the is maintained within this document.

10.1.2 Installing New Plugins

As a general rule and unless the documentation states otherwise, plugins should be installed in two ways:

- When the plugin is available as **package**, it should be installed when **Flir is running**. This is the required method because the package executed pre and post-installation tasks that require Flir to run.
- When the plugin is available as **source code**, it should be installed when **Flir is either running or not**. You will want to manually move the plugin code into the right location where Flir is installed, add pre-requisites and execute the REST commands necessary to start the plugin **after** you have started Flir if it is not running when you start this process.

For example, this is the command to use to install the *OpenWeather* South plugin:

```
$ sudo systemctl status flir.service
flir.service - LSB: Flir
   Loaded: loaded (/etc/init.d/flir; bad; vendor preset: enabled)
   Active: active (running) since Wed 2018-05-16 01:32:25 BST; 4min 1s ago
     Docs: man:systemd-sysv-generator(8)
```

(continues on next page)

(continued from previous page)

```

CGroup: /system.slice/flir.service
└─13741 python3 -m flir.services.core
└─13746 /usr/local/flir/services/storage --address=0.0.0.0 --port=40138

May 16 01:36:09 ubuntu python3[13741]: Flir[13741] INFO: scheduler: flir.services.
→core.scheduler.scheduler: Process started: Schedule 'stats collection' process
→'stats coll
                                ['tasks/statistics', '--port=40138', '--
→address=127.0.0.1', '--name=stats collector']
...
Flir v1.3.1 running.
Flir Uptime: 266 seconds.
Flir records: 0 read, 0 sent, 0 purged.
Flir does not require authentication.
=== Flir services:
flir.services.core
=== Flir tasks:
$
$ sudo cp flir-south-openweathermap-1.2-x86_64.deb /var/cache/apt/archives/.
$ sudo apt install /var/cache/apt/archives/flir-south-openweathermap-1.2-x86_64.deb
Reading package lists... Done
Building dependency tree
Reading state information... Done
Note, selecting 'flir-south-openweathermap' instead of '/var/cache/apt/archives/flir-
→south-openweathermap-1.2-x86_64.deb'
The following packages were automatically installed and are no longer required:
  linux-headers-4.4.0-109 linux-headers-4.4.0-109-generic linux-headers-4.4.0-119_
→linux-headers-4.4.0-119-generic linux-headers-4.4.0-121 linux-headers-4.4.0-121-
→generic
  linux-image-4.4.0-109-generic linux-image-4.4.0-119-generic linux-image-4.4.0-121-
→generic linux-image-extra-4.4.0-109-generic linux-image-extra-4.4.0-119-generic
  linux-image-extra-4.4.0-121-generic
Use 'sudo apt autoremove' to remove them.
The following NEW packages will be installed
  flir-south-openweathermap
0 to upgrade, 1 to newly install, 0 to remove and 0 not to upgrade.
Need to get 0 B/3,404 B of archives.
After this operation, 0 B of additional disk space will be used.
Selecting previously unselected package flir-south-openweathermap.
(Reading database ... 211747 files and directories currently installed.)
Preparing to unpack .../flir-south-openweathermap-1.2-x86_64.deb ...
Unpacking flir-south-openweathermap (1.2) ...
Setting up flir-south-openweathermap (1.2) ...
openweathermap plugin installed.
$
$ flir status
Flir v1.3.1 running.
Flir Uptime: 271 seconds.
Flir records: 36 read, 0 sent, 0 purged.
Flir does not require authentication.
=== Flir services:
flir.services.core
flir.services.south --port=42066 --address=127.0.0.1 --name=openweathermap
=== Flir tasks:
$

```

You may also install new plugins directly from within the Flir GUI, however you will need to have setup your Linux

machine to include the Flir package repository in the list of repositories the Linux package manager searches for new packages.

10.2 Writing and Using Plugins

A plugin has a small set of external entry points that must exist in order for Flir to load and execute that plugin. Currently plugins may be written in either Python or C/C++, the set of entry points is the same for both languages. The entry points detailed here will be presented for both languages, a more indepth discussion of writing plugins in C/C++ will then follow.

10.2.1 Common Flir Plugin API

Every plugin provides at least one common API entry point, the *plugin_info* entry point. It is used to obtain information about a plugin before it is initialised and used. It allows Flir to determine what type of plugin it is, e.g. a South bound plugin or a North bound plugin, obtain default configuration information for the plugin and determine version information.

Plugin Information

The information entry point is implemented as a call, *plugin_info*, that takes no arguments. Data is returned from this API call as a JSON document with certain well known properties.

A typical Python implementation of this would simply return a fixed dictionary object that encodes the required properties.

```
def plugin_info():
    """ Returns information about the plugin.

    Args:
    Returns:
        dict: plugin information
    Raises:
    """

    return {
        'name': 'DHT11 GPIO',
        'version': '1.0',
        'mode': 'poll',
        'type': 'south',
        'interface': '1.0',
        'config': _DEFAULT_CONFIG
    }
```

These are the properties returned by the JSON document:

- **Name** - A textual name that will be used for reporting purposes for this plugin.
- **Version** - This property allows the version of the plugin to be communicated to the plugin loader. This is used for reporting purposes only and has no effect on the way Flir interacts with the plugin.
- **Type** - The type of the plugin, used by the plugin loader to determine if the plugin is being used correctly. The type is a simple string and may be South, North, Storage, Filter, Rule or Delivery.

Note: If you browse the Flir code you may find old plugins with type *device*: this was the type used to indicate a South plugin and it is now deprecated.

- **Interface** - This property reports the version of the plugin API to which this plugin was written. It allows Flir to support upgrades of the API whilst being able to recognise the version that a particular plugin is compliant with. Currently all interfaces are version 1.0.
- **Configuration** - This allows the plugin to return a JSON document which contains the default configuration of the plugin. This is in line with the extensible plugin mechanism of Flir, each plugin will return a set of configuration items that it wishes to use, this will then be used to extend the set of Flir configuration items. This structure, a JSON document, includes default values but no actual values for each configuration option. The first time Flir's configuration manager sees a category it will register the category and create values for each item using the default value in the configuration document. On subsequent calls the value already in the configuration manager will be used. This mechanism allows the plugin to extend the set of configuration variables whilst giving the user the opportunity to modify the value of these configuration items. It also allow new versions of plugins to add new configuration items whilst retaining the values of previous items. And new items will automatically be assigned the default value for that item. As an example, a plugin that wishes to maintain two configuration variables, say a GPIO pin to use and a polling interval, would return a configuration document that looks as follows:

```
{
  'pollInterval': {
    'description': 'The interval between poll calls to the device poll routine_
↪expressed in milliseconds.',
    'type': 'integer',
    'default': '1000'
  },
  'gpiopin': {
    'description': 'The GPIO pin into which the DHT11 data pin is connected',
    'type': 'integer',
    'default': '4'
  }
}
```

A C/C++ plugin returns the same information as a structure, this structure includes the JSON configuration document as a simple C string.

```
#include <plugin_api.h>

extern "C" {

/**
 * The plugin information structure
 */
static PLUGIN_INFORMATION info = {
    "MyPlugin",           // Name
    "1.0.1",              // Version
    0,                    // Flags
    PLUGIN_TYPE_SOUTH,    // Type
    "1.0.0",              // Interface version
    default_config         // Default configuration
};

/**
 * Return the information about this plugin
```

(continues on next page)

(continued from previous page)

```

*/
PLUGIN_INFORMATION *plugin_info()
{
    return &info;
}

```

In the above example the constant *default_config* is a string that contains the JSON configuration document. In order to make the JSON easier to manage a special macro is defined in the *plugin_api.h* header file. This macro is called *QUOTE* and is designed to ease the quoting requirements to create this JSON document.

```

const char *default_config = QUOTE({
    "plugin" : {
        "description" : "My example plugin in C++",
        "type" : "string",
        "default" : "MyPlugin",
        "readonly" : "true"
    },
    "asset" : {
        "description" : "The name of the asset the plugin will produce",
        "type" : "string",
        "default" : "MyAsset"
    }
});

```

Plugin Initialization

The plugin initialization is called after the service that has loaded the plugin has collected the plugin information and resolved the configuration of the plugin but before any other calls will be made to the plugin. The initialization routine is called with the resolved configuration of the plugin, this includes values as opposed to the defaults that were returned in the *plugin_info* call.

This call is used by the plugin to do any initialization or state creation it needs to do. The call returns a handle which will be passed into each subsequent call of the plugin. The handle allows the plugin to have state information that is maintained and passed to it whilst allowing for multiple instances of the same plugin to be loaded by a service if desired. It is equivalent to a this or self pointer for the plugin, although the plugin is not defined as a class.

In Python a simple example of a sensor that reads a GPIO pin for data, we might choose to use that configured GPIO pin as the handle we pass to other calls.

```

def plugin_init(config):
    """ Initialise the plugin.

    Args:
        config: JSON configuration document for the device configuration category
    Returns:
        handle: JSON object to be used in future calls to the plugin
    Raises:
        """

    handle = config['gpioin']['value']
    return handle

```

A C/C++ plugin should return a value in a *void* pointer that can then be dereferenced in subsequent calls. A typical C++ implementation might create an instance of a class and use that instance as the handle for the plugin.


```

/**
 * Initialise the plugin, called to get the plugin handle
 */
PLUGIN_HANDLE plugin_init(ConfigCategory *config)
{
    MyPluginClass *plugin = new MyPluginClass();

    plugin->configure(config);

    return (PLUGIN_HANDLE)plugin;
}

```

It should also be observed in the above C/C++ example the *plugin_init* call is passed a pointer to a *ConfigCategory* class that encapsulates the JSON configuration category for the plugin. Details of the *ConfigCategory* class are available in the section .

Plugin Shutdown

The plugin shutdown method is called as part of the shutdown sequence of the service that loaded the plugin. It gives the plugin the opportunity to do any cleanup operations before terminating. As with all calls it is passed the handle of our plugin instance. Plugins can not prevent the shutdown and do not have to implement any actions. In our simple sensor example there is nothing to do in order to shutdown the plugin.

A C/C++ plugin might use this *plugin_shutdown* call to delete the plugin class instance it created in the corresponding *plugin_init* call.

```

/**
 * Shutdown the plugin
 */
void plugin_shutdown(PLUGIN_HANDLE *handle)
{
    MyPluginClass *plugin = (MyPluginClass *)handle;

    delete plugin;
}

```

Plugin Reconfigure

The plugin reconfigure method is called whenever the configuration of the plugin is changed. It allows for the dynamic reconfiguration of the plugin whilst it is running. The method is called with the handle of the plugin and the updated configuration document. The plugin should take whatever action it needs to and return a new or updated copy of the handle that will be passed to future calls.

The plugin reconfigure method is shared between most but not all plugin types. In particular it does not exist for the shorted lived plugins that are created to perform a single operation and then terminated. These are the north plugins and the notification delivery plugins.

Using a simple Python example of our sensor reading a GPIO pin, we extract the new pin number from the new configuration data and return that as the new handle for the plugin instance.

```

def plugin_reconfigure(handle, new_config):
    """ Reconfigures the plugin, it should be called when the configuration of the_
    ↪ plugin is changed during the
        operation of the device service.
        The new configuration category should be passed.

```

(continues on next page)

(continued from previous page)

```

    Args:
        handle: handle returned by the plugin initialisation call
        new_config: JSON object representing the new configuration category for the_
→category
    Returns:
        new_handle: new handle to be used in the future calls
    Raises:
        """

    new_handle = new_config['gpioin']['value']
    return new_handle

```

In C/C++ the *plugin_reconfigure* class is very similar, note however that the *plugin_reconfigure* call is passed the JSON configuration category as a string and not a *ConfigCategory*, it is easy to parse and create the C++ class however, a name for the category must be given however.

```

/**
 * Reconfigure the plugin
 */
void plugin_reconfigure(PLUGIN_HANDLE *handle, string& newConfig)
{
    ConfigCategory      config("newConfiguration", newConfig);
    MyPluginClass      *plugin = (MyPluginClass *)*handle;

    plugin->configure(&config);
}

```

It should be noted that the *plugin_reconfigure* call may be delivered in a separate thread for a C/C++ plugin and that the plugin should implement any mutual exclusion mechanisms that are required based on the actions of the *plugin_reconfigure* method.

10.3 South Plugins

South plugins are used to communicate with sensors and actuators, there are two modes of plugin operation; *asyncio* and *polled*.

10.3.1 Polled Mode

Polled mode is the simplest form of South plugin that can be written, a poll routine is called at an interval defined in the plugin configuration. The South service determines the type of the plugin by examining at the mode property in the information the plugin returns from the *plugin_info* call.

Plugin Poll

The plugin *poll* method is called periodically to collect the readings from a poll mode sensor. As with all other calls the argument passed to the method is the handle returned by the initialization call, the return of the method should be the JSON payload of the readings to return.

The JSON payload returned, as a Python dictionary, should contain the properties; asset, timestamp, key and readings.

Property	Description
asset	The asset key of the sensor device that is being read
timestamp	A timestamp for the reading data
key	A UUID which is the unique key of this reading
readings	The reading data itself as a JSON object

It is important that the *poll* method does not block as this will prevent the proper operation of the South microservice. Using the example of our simple DHT11 device attached to a GPIO pin, the *poll* routine could be:

```
def plugin_poll(handle):
    """ Extracts data from the sensor and returns it in a JSON document as a Python_
    ↪dict.

    Available for poll mode only.

    Args:
        handle: handle returned by the plugin initialisation call
    Returns:
        returns a sensor reading in a JSON document, as a Python dict, if it is_
    ↪available
        None - If no reading is available
    Raises:
        DataRetrievalError
    """

    try:
        humidity, temperature = Adafruit_DHT.read_retry(Adafruit_DHT.DHT11, handle)
        if humidity is not None and temperature is not None:
            time_stamp = str(datetime.now(tz=timezone.utc))
            readings = { 'temperature': temperature , 'humidity' : humidity }
            wrapper = {
                'asset': 'dht11',
                'timestamp': time_stamp,
                'key': str(uuid.uuid4()),
                'readings': readings
            }
            return wrapper
        else:
            return None

    except Exception as ex:
        raise exceptions.DataRetrievalError(ex)

    return None
```

10.3.2 Async IO Mode

In asyncio mode the plugin inserts itself into the event processing loop of the South Service itself. This is a more complex mechanism and is intended for plugins that need to block or listen for incoming data via a network.

Plugin Start

The *plugin_start* method, as with other plugin calls, is called with the plugin handle data that was returned from the *plugin_init* call. The *plugin_start* call will only be called once for a plugin, it is the responsibility of *plugin_start* to

install the plugin code into the python event handling system for asyncIO. Assuming an example whereby the interface to a sensor is via HTTP and the sensor will make HTTP POST calls to our plugin in order to send data into Flir, a *plugin_start* for this scenario would create a web application endpoint for reception of the POST command.

```
loop = asyncio.get_event_loop()
app = web.Application(middlewares=[middleware.error_middleware])
app.router.add_route('POST', '/', SensorPhoneIngest.render_post)
handler = app.make_handler()
coro = loop.create_server(handler, host, port)
server = asyncio.ensure_future(coro)
```

This code first gets the event loop for this Python execution, it then creates the web application and adds a route for the POST request. In this case it is calling the *render_post* method of the object *SensorPhone*. It then goes on to create the handler and install the web server instance into the event system.

Async Handler

The async handler is defined for incoming message has the responsibility of taking the sensor data and ingesting that into Flir. Unlike the poll mechanism, this is done from within the handler rather than by passing the data back to the South service itself. A convenient method exists for ingesting readings, *Ingest.add_readings*. This call is passed an asset, timestamp, key and readings document for the asset and will do everything else required to make sure the readings are stored in the Flir buffer. In the case of our HTTP based example above, the code would create the items needed to generate the arguments to the *Ingest.add_readings* call, by creating data items and retrieving them from the payload sent by the sensor.

```
try:
    if not Ingest.is_available():
        increment_discarded_counter = True
        message = {'busy': True}
    else:
        payload = await request.json()

        asset = 'SensorPhone'
        timestamp = str(datetime.now(tz=timezone.utc))
        messages = payload.get('messages')

        if not isinstance(messages, list):
            raise ValueError('messages must be a list')

        for readings in messages:
            key = str(uuid.uuid4())
            await Ingest.add_readings(asset=asset, timestamp=timestamp, key=key,
↳ readings=readings)
except ...
```

It would then respond to the HTTP request and return. Since the handler is embedded in the event loop this will happen in the context of a coroutine and would happen each time a new POST request is received.

```
message['status'] = code
return web.json_response(message)
```

10.3.3 A South Plugin Example In Python: the DHT11 Sensor

Let's try to put all the information together and write a plugin. We can continue to use the example of an inexpensive sensor, the DHT11, used to measure temperature and humidity, directly wired to a Raspberry PI. This plugin is available on github, .

First, here is a set of links where you can find more information regarding this sensor:

-
-
-

The Hardware

The DHT sensor is directly connected to a Raspberry PI 2 or 3. You may decide to buy a sensor and a resistor and solder them yourself, or you can buy a ready-made circuit that provides the correct output to wire to the Raspberry PI. shows a DHT11 with resistor that you can buy online.

The sensor can be directly connected to the Raspberry PI GPIO (General Purpose Input/Output). An introduction to the GPIO and the pinset is available . In our case, you must connect the sensor on these pins:

- **VCC** is connected to PIN #2 (5v Power)
- **GND** is connected to PIN #6 (Ground)
- **DATA** is connected to PIN #7 (BCM 4 - GPCLK0)

shows the sensor wired to the Raspberry PI and is a zoom into the wires used.

The Software

For this plugin we use the ADAAfruit Python Library (links to the GitHub repository are above). First, you must install the library (in future versions the library will be provided in a ready-made package):

```
$ git clone https://github.com/adafruit/Adafruit_Python_DHT.git
Cloning into 'Adafruit_Python_DHT'...
remote: Counting objects: 249, done.
remote: Total 249 (delta 0), reused 0 (delta 0), pack-reused 249
Receiving objects: 100% (249/249), 77.00 KiB | 0 bytes/s, done.
Resolving deltas: 100% (142/142), done.
$ cd Adafruit_Python_DHT
$ sudo apt-get install build-essential python-dev
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
build-essential python-dev
...
$ sudo python3 setup.py install
running install
running bdist_egg
running egg_info
creating Adafruit_DHT.egg-info
...
$
```

The Plugin

This is the code for the plugin:

```
# -*- coding: utf-8 -*-

# FLIR_BEGIN
# See: http://flir.readthedocs.io/
# FLIR_END

""" Plugin for a DHT11 temperature and humidity sensor attached directly
    to the GPIO pins of a Raspberry Pi

    This plugin uses the Adafruit DHT library, to install this perform
    the following steps:

        git clone https://github.com/adafruit/Adafruit_Python_DHT.git
        cd Adafruit_Python_DHT
        sudo apt-get install build-essential python-dev
        sudo python setup.py install

    To access the GPIO pins flir must be able to access /dev/gpiomem,
    the default access for this is owner and group read/write. Either
    Flir must be added to the group or the permissions altered to
    allow Flir access to the device.
    """

from datetime import datetime, timezone
import uuid

from flir.common import logger
from flir.services.south import exceptions

__author__ = "Mark Riddoch"
__copyright__ = "Copyright (c) 2017 OSIsoft, LLC"
__license__ = "Apache 2.0"
__version__ = "${VERSION}"

_DEFAULT_CONFIG = {
    'plugin': {
        'description': 'Python module name of the plugin to load',
        'type': 'string',
        'default': 'dht11'
    },
    'pollInterval': {
        'description': 'The interval between poll calls to the device poll routine,
        ↪expressed in milliseconds.',
        'type': 'integer',
        'default': '1000'
    },
    'gpioPin': {
        'description': 'The GPIO pin into which the DHT11 data pin is connected',
        'type': 'integer',
        'default': '4'
    }
}
```

(continues on next page)

(continued from previous page)

```

_LOGGER = logger.setup(__name__)
""" Setup the access to the logging system of Flir """

def plugin_info():
    """ Returns information about the plugin.

    Args:
    Returns:
        dict: plugin information
    Raises:
    """

    return {
        'name': 'DHT11 GPIO',
        'version': '1.0',
        'mode': 'poll',
        'type': 'south',
        'interface': '1.0',
        'config': _DEFAULT_CONFIG
    }

def plugin_init(config):
    """ Initialise the plugin.

    Args:
        config: JSON configuration document for the device configuration category
    Returns:
        handle: JSON object to be used in future calls to the plugin
    Raises:
    """

    handle = config['gpiopin']['value']
    return handle

def plugin_poll(handle):
    """ Extracts data from the sensor and returns it in a JSON document as a Python_
↪dict.

    Available for poll mode only.

    Args:
        handle: handle returned by the plugin initialisation call
    Returns:
        returns a sensor reading in a JSON document, as a Python dict, if it is_
↪available
        None - If no reading is available
    Raises:
        DataRetrievalError
    """

    try:
        humidity, temperature = Adafruit_DHT.read_retry(Adafruit_DHT.DHT11, handle)
        if humidity is not None and temperature is not None:

```

(continues on next page)

(continued from previous page)

```

        time_stamp = str(datetime.now(tz=timezone.utc))
        readings = {'temperature': temperature, 'humidity': humidity}
        wrapper = {
            'asset':      'dht11',
            'timestamp': time_stamp,
            'key':        str(uuid.uuid4()),
            'readings':   readings
        }
        return wrapper
    else:
        return None

except Exception as ex:
    raise exceptions.DataRetrievalError(ex)

return None

def plugin_reconfigure(handle, new_config):
    """ Reconfigures the plugin, it should be called when the configuration of the
    ↪ plugin is changed during the
        operation of the device service.
        The new configuration category should be passed.

    Args:
        handle: handle returned by the plugin initialisation call
        new_config: JSON object representing the new configuration category for the
    ↪ category
    Returns:
        new_handle: new handle to be used in the future calls
    Raises:
        """

    new_handle = new_config['gpiopin']['value']
    return new_handle

def plugin_shutdown(handle):
    """ Shutdowns the plugin doing required cleanup, to be called prior to the device
    ↪ service being shut down.

    Args:
        handle: handle returned by the plugin initialisation call
    Returns:
    Raises:
        """
    pass

```

Building Flir and Adding the Plugin

If you have not built Flir yet, follow the steps described . After the build, you can optionally install Flir following steps.

- If you have started Flir from the build directory, copy the structure of the *flir-south-dht11/python/* directory into the *python* directory:


```
$ cd ~/Flir
$ cp -R ~/flir-south-dht11/python/flir/plugins/south/dht11 python/flir/plugins/south/
$
```

- If you have installed Flir by executing `sudo make install`, copy the structure of the `flir-south-dht11/python/` directory into the installed `python` directory:

```
$ sudo cp -R ~/flir-south-dht11/python/flir/plugins/south/dht11 /usr/local/flir/
↪python/flir/plugins/south/
$
```

Note: If you have installed Flir using an alternative *DESTDIR*, remember to add the path to the destination directory to the `cp` command.

- Add service

```
$ curl -sX POST http://localhost:8081/flir/service -d '{"name": "dht11", "type":
↪"south", "plugin": "dht11", "enabled": true}'
```

Note: Each plugin repo has its own debian packaging script and documentation, And that is the recommended way to go! As above method(s) may need explicit action for linux and/or python dependencies installation.

Using the Plugin

Once south plugin is added as an enabled service, You are ready to use the DHT11 plugin.

```
$ curl -X GET http://localhost:8081/flir/service | jq
```

Let's see what we have collected so far:

```
$ curl -s http://localhost:8081/flir/asset | jq
[
  {
    "count": 158,
    "asset_code": "dht11"
  }
]
$
```

Finally, let's extract some values:

```
$ curl -s http://localhost:8081/flir/asset/dht11?limit=5 | jq
[
  {
    "timestamp": "2017-12-30 14:41:39.672",
    "reading": {
      "temperature": 19,
      "humidity": 62
    }
  },
  {
    "timestamp": "2017-12-30 14:41:35.615",
```

(continues on next page)

(continued from previous page)

```

    "reading": {
      "temperature": 19,
      "humidity": 63
    }
  },
  {
    "timestamp": "2017-12-30 14:41:34.087",
    "reading": {
      "temperature": 19,
      "humidity": 62
    }
  },
  {
    "timestamp": "2017-12-30 14:41:32.557",
    "reading": {
      "temperature": 19,
      "humidity": 63
    }
  },
  {
    "timestamp": "2017-12-30 14:41:31.028",
    "reading": {
      "temperature": 19,
      "humidity": 63
    }
  }
]
$

```

Clearly we will not see many changes in temperature or humidity, unless we place our thumb on the sensor or we blow warm breathe on it :-)

```

$ curl -s http://localhost:8081/flir/asset/dht11?limit=5 | jq
[
  {
    "timestamp": "2017-12-30 14:43:16.787",
    "reading": {
      "temperature": 25,
      "humidity": 95
    }
  },
  {
    "timestamp": "2017-12-30 14:43:15.258",
    "reading": {
      "temperature": 25,
      "humidity": 95
    }
  },
  {
    "timestamp": "2017-12-30 14:43:13.729",
    "reading": {
      "temperature": 24,
      "humidity": 95
    }
  },
  {

```

(continues on next page)

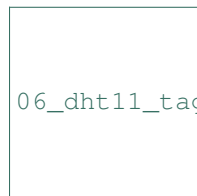
(continued from previous page)

```

    "timestamp": "2017-12-30 14:43:12.201",
    "reading": {
      "temperature": 24,
      "humidity": 95
    }
  },
  {
    "timestamp": "2017-12-30 14:43:05.616",
    "reading": {
      "temperature": 22,
      "humidity": 95
    }
  }
]
$

```

Needless to say, the North plugin will send the buffered data to the PI system using the OMF plugin or any other north system using the appropriate north plugin.



10.4 South Plugins in C

South plugins written in C/C++ are no different in use to those written in Python, it is merely a case that they are implemented in a different language. The same options of polled or asynchronous methods still exist and the enduser of Flir is not aware in which language the plugin has been written.

10.4.1 Polled Mode

Polled mode is the simplest form of South plugin that can be written, a poll routine is called at an interval defined in the plugin advanced configuration. The South service determines the type of the plugin by examining the mode property in the information the plugin returns from the *plugin_info* call.

Plugin Poll

The plugin *poll* method is called periodically to collect the readings from a poll mode sensor. As with all other calls the argument passed to the method is the handle returned by the *plugin_init* call, the return of the method should be a *Reading* instance that contains the data read.

The *Reading* class consists of

Property	Description
assetName	The asset key of the sensor device that is being read
userTimestamp	A timestamp for the reading data
datapoints	The reading data itself as a set of datapoint instances

More detail regarding the *Reading* class can be found in the section .

It is important that the *poll* method does not block as this will prevent the proper operation of the South microservice. Using the example of our simple DHT11 device attached to a GPIO pin, the *poll* routine could be:

```
/**
 * Poll for a plugin reading
 */
Reading plugin_poll (PLUGIN_HANDLE *handle)
{
    DHT11 *dht11 = (DHT11*)handle;
    return dht11->takeReading();
}
```

Where our *DHT11* class has a method *takeReading* as follows

```
/**
 * Take reading from sensor
 *
 * @param firstReading This flag indicates whether this is the first reading to be
 * taken from sensor,
 * if so get it reliably even if takes multiple retries.
 * Subsequently (firstReading=false),
 * if reading from sensor fails, last good reading is returned.
 */
Reading DHT11::takeReading (bool firstReading)
{
    static uint8_t sensorData[4] = {0,0,0,0};

    bool valid = false;
    unsigned int count=0;
    do {
        valid = readSensorData(sensorData);
        count++;
    } while (!valid && firstReading && count < MAX_SENSOR_READ_RETRIES);

    if (firstReading && count >= MAX_SENSOR_READ_RETRIES)
        Logger::getLogger()->error("Unable to get initial valid reading from
        DHT11 sensor connected to pin %d even after %d tries", m_pin, MAX_SENSOR_READ_
        RETRIES);

    vector<Datapoint *> vec;

    ostringstream tmp;
    tmp << ((unsigned int)sensorData[0]) << "." << ((unsigned int)sensorData[1]);
    DatapointValue dpv1(stod(tmp.str()));
    vec.push_back(new Datapoint("Humidity", dpv1));

    ostringstream tmp2;
    tmp2 << ((unsigned int)sensorData[2]) << "." << ((unsigned_
    int)sensorData[3]);
    DatapointValue dpv2(stod(tmp2.str()));
    vec.push_back(new Datapoint ("Temperature", dpv2));

    return Reading(m_assetName, vec);
}
```

We are creating two *DatapointValues* for the Humidity and Temperature values returned by reading the DHT11 sensor.

Plugin Poll Returning Multiple Values

It is possible in a C/C++ plugin to have a plugin that returns multiple readings in a single call to a poll routine. This is done by setting the interface version of 2.0.0 rather than 1.0.0. In this interface version the *plugin_poll* call returns a vector of *Reading* rather than a single *Reading*.

```
/**
 * Poll for a plugin reading
 */
std::vector<Reading *> *plugin_poll(PLUGIN_HANDLE *handle)
{
    Modbus *modbus = (Modbus *)handle;

    if (!handle)
        throw runtime_error("Bad plugin handle");
    return modbus->takeReading();
}
```

10.4.2 Async IO Mode

In asyncio mode the plugin runs either a separate thread or uses some incoming event from a device or callback mechanism to trigger sending data to Flir. The asynchronous mode uses two additional entry points to the plugin, one to register a callback on which the plugin sends data, *plugin_register_ingest* and another to start the asynchronous behavior *plugin_start*.

Plugin Register Ingest

The *plugin_register_ingest* call is used to allow the south service to pass a callback function to the plugin that the plugin uses to send data to the service every time the plugin has some new data.

```
/**
 * Register ingest callback
 */
void plugin_register_ingest(PLUGIN_HANDLE *handle, INGEST_CB cb, void *data)
{
    MyPluginClass *plugin = (MyPluginClass *)handle;

    if (!handle)
        throw new exception();
    plugin->registerIngest(data, cb);
}
```

The plugin should store the callback function pointer and the data associated with the callback such that it can use that information to pass a reading to the south service. The following code snippets show how a plugin class might store the callback and data and then use it to send readings into Flir at a later stage.

```
/**
 * Record the ingest callback function and data in member variables
 *
 * @param data The Ingest function data
 * @param cb The callback function to call
 */
void MyPluginClass::registerIngest(void *data, INGEST_CB cb)
{

```

(continues on next page)

(continued from previous page)

```

        m_ingest = cb;
        m_data = data;
    }

    /**
     * Called when a data is available to send to the south service
     *
     * @param points      The points in the reading we must create
     */
    void MyPluginClass::ingest(Reading& reading)
    {
        (*m_ingest)(m_data, reading);
    }

```

Plugin Start

The *plugin_start* method, as with other plugin calls, is called with the plugin handle data that was returned from the *plugin_init* call. The *plugin_start* call will only be called once for a plugin, it is the responsibility of *plugin_start* to take whatever action is required in the plugin in order to start the asynchronous actions of the plugin. This might be to start a thread, register an endpoint for a remote connection or call an entry point in a third party library to start asynchronous processing.

```

    /**
     * Start the Async handling for the plugin
     */
    void plugin_start(PLUGIN_HANDLE *handle)
    {
        MyPluginClass *plugin = (MyPluginClass *)handle;

        if (!handle)
            return;
        plugin->start();
    }

    /**
     * Start the asynchronous processing thread
     */
    void MyPluginClass::start()
    {
        m_running = true;
        m_thread = new thread(threadWrapper, this);
    }

```

10.4.3 Set Point Control

South plugins can also be used to exert control on the underlying device to which they are connected. This is not intended for use as a substitute for real time control systems, but rather as a mechanism to make non-time critical changes to a device or to trigger an operation on the device.

To make a south plugin support control features there are two steps that need to be taken

- Tag the plugin as supporting control

- Add the entry points for control

Enable Control

A plugin enables control features by means of the flags in the plugin information data structure which is returned by the *plugin_info* entry point of the plugin. The flag value *SP_CONTROL* should be added to the flags of the plugin.

```
/**
 * The plugin information structure
 */
static PLUGIN_INFORMATION info = {
    PLUGIN_NAME,           // Name
    VERSION,               // Version
    SP_CONTROL,            // Flags - add control
    PLUGIN_TYPE_SOUTH,     // Type
    "1.0.0",               // Interface version
    CONFIG                 // Default configuration
};
```

Adding this flag will cause the south service to do a number of things when it loads the plugin;

- The south service will attempt to resolve the two control entry points.
- A toggle will be added to the advanced configuration category of the service that will permit the disabling of control services.
- A security category will be added to the south service that contains the access control lists and permissions associated with the service.

Control Entry Points

Two entry points are supported for control operations in the south plugin

- **plugin_write**: which is used to set the value of a parameter within the plugin or device
- **plugin_operation**: which is used to perform an operation on the plugin or device

The south plugin can support one or both of these entry points as appropriate for the plugin.

Write Entry Point

The write entry point is used to set data in the plugin or write data into the device.

The plugin write entry point is defined as follows

```
bool plugin_write(PLUGIN_HANDLE *handle, string name, string value)
```

Where the parameters are;

- **handle** the handle of the plugin instance
- **name** the name of the item to be changed
- **value** a string presentation of the new value to assign to the item

The return value defines if the write was successful or not. True is returned for a successful write.

```
bool plugin_write(PLUGIN_HANDLE *handle, string& name, string& value)
{
    Random *random = (Random *)handle;

    return random->write(operation, name, value);
}
```

In this case the main logic of the write operation is implemented in a class that contains all the plugin logic. Note that the assumption here, and a design pattern often used by plugin writers, is that the *PLUGIN_HANDLE* is actually a pointer to a C++ class instance.

In this case the implementation in the plugin class is as follows:

```
bool Random::write(string& name, string& value)
{
    if (name.compare("mode") == 0)
    {
        if (value.compare("relative") == 0)
        {
            m_mode = RELATIVE_MODE;
        }
        else if (value.compare("absolute") == 0)
        {
            m_mode = ABSOLUTE_MODE;
        }
        Logger::getLogger()->error("Unknown mode requested '%s' ignored.",
↪value.c_str());
        return false;
    }
    else
    {
        Logger::getLogger()->error("Unknown control item '%s' ignored.", name.c_
↪str());
        return false;
    }
    return true;
}
```

In this case the code is relatively simple as we assume there is a single control parameter that can be written, the mode of operation. We look for the known name and if a different name is passed an error is logged and false is returned. If the correct name is passed in we then check the value and take the appropriate action. If the value is not a recognized value then an error is logged and we again return false.

In this case we are merely setting a value within the plugin, this could equally well be done via configuration and would in that case be persisted between restarts. Normally control would not be used for this, but rather for making a change with the connected device itself, such as changing a PLC register value. This is simply an example to demonstrate the mechanism.

Operation Entry Point

The plugin will support an operation entry point. This will execute the given operation synchronously, it is expected that this operation entry point will be called using a separate thread, therefore the plugin should implement operations in a thread safe environment.

The plugin write operation entry point is defined as follows


```
bool plugin_operation(PLUGIN_HANDLE *handle, string& operation, int count, PLUGIN_
↳PARAMETER **params)
```

Where the parameters are;

- **handle** the handle of the plugin instance
- **operation** the name of the operation to be executed
- **count** the number of parameters
- **params** a set of name/value pairs that are passed to the operation

The *operation* parameter should be used by the plugin to determine which operation is to be performed, that operation may also be passed a number of parameters. The count of these parameters are passed to the plugin in the *count* argument and the actual parameters are passed in an array of key/value pairs as strings.

The return from the call is a boolean result of the operation, a failure of the operation or a call to an unrecognized operation should be indicated by returning a false value. If the operation succeeds a value of true should be returned.

The following example shows the implementation of the plugin operation entry point.

```
bool plugin_operation(PLUGIN_HANDLE *handle, string& operation, int count, PLUGIN_
↳PARAMETER **params)
{
    Random *random = (Random *)handle;

    return random->operation(operation, count, params);
}
```

In this case the main logic of the operation is implemented in a class that contains all the plugin logic. Note that the assumption here, and a design pattern often used by plugin writers, is that the *PLUGIN_HANDLE* is actually a pointer to a C++ class instance.

In this case the implementation in the plugin class is as follows:

```
/**
 * SetPoint operation. We support reseeding the random number generator
 */
bool Random::operation(const std::string& operation, int count, PLUGIN_PARAMETER_
↳**params)
{
    if (operation.compare("seed") == 0)
    {
        if (count)
        {
            if (params[0]->name.compare("seed")
↳
            {
                long seed = strtol(params[0]->value.c_str(), NULL,
↳10);

                srand(seed);
            }
            else
            {
                return false;
            }
        }
        else
        {
            srand(time(0));
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
    Logger::getLogger()->info("Reseeded random number generator");
    return true;
}
Logger::getLogger()->error("Unrecognised operation %s", operation.c_str());
return false;
}

```

In this example, the operation method checks the name of the operation to perform, only a single operation is supported by this plugin. If this operation name differs the method will log an error and return false. If the operation is recognized it will check for any arguments passed in, retrieve and use it. In this case an optional *seed* argument may be passed.

There is no actual machine connected here, therefore the operation occurs within the plugin. In the case of a real machine the operation would most likely cause an action on a machine, for example a request to the machine to re-calibrate itself.

10.4.4 A South Plugin Example In C/C++: the DHT11 Sensor

Using the same example as before, the DHT11 temperature and humidity sensor, let's look at how to create the plugin in C/C++.

The Software

For this plugin we use the wiringpi C library to connect to the hardware of the Raspberry Pi

```

$ sudo apt-get install wiringpi
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
wiringpi
...
$

```

The Plugin

This is the code for the plugin.cpp file that provides the plugin API:

```

/*
 * Flir south plugin.
 *
 * Copyright (c) 2018 OSisoft, LLC
 *
 * Released under the Apache 2.0 Licence
 *
 * Author: Amandeep Singh Arora
 */
#include <dht11.h>
#include <plugin_api.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

```

(continues on next page)

(continued from previous page)

```

#include <string>
#include <logger.h>
#include <plugin_exception.h>
#include <config_category.h>
#include <rapidjson/document.h>
#include <version.h>

using namespace std;
#define PLUGIN_NAME "dht11_V2"

/**
 * Default configuration
 */
const static char *default_config = QUOTE({
    "plugin" : {
        "description" : "DHT11 C south plugin",
        "type" : "string",
        "default" : PLUGIN_NAME,
        "readonly": "true"
    },
    "asset" : {
        "description" : "Asset name",
        "type" : "string",
        "default" : "dht11",
        "order": "1",
        "displayName": "Asset Name",
        "mandatory" : "true"
    },
    "pin" : {
        "description" : "Rpi pin to which DHT11 is attached",
        "type" : "integer",
        "default" : "7",
        "displayName": "Rpi Pin"
    }
});

/**
 * The DHT11 plugin interface
 */
extern "C" {

/**
 * The plugin information structure
 */
static PLUGIN_INFORMATION info = {
    PLUGIN_NAME,           // Name
    VERSION,               // Version
    0,                     // Flags
    PLUGIN_TYPE_SOUTH,     // Type
    "1.0.0",               // Interface version
    default_config          // Default configuration
};

/**
 * Return the information about this plugin
 */

```

(continues on next page)

(continued from previous page)

```

PLUGIN_INFORMATION *plugin_info()
{
    return &info;
}

/**
 * Initialise the plugin, called to get the plugin handle
 */
PLUGIN_HANDLE plugin_init(ConfigCategory *config)
{
    unsigned int pin;

    if (config->itemExists("pin"))
    {
        pin = stoul(config->getValue("pin"), nullptr, 0);
    }

    DHT11 *dht11= new DHT11(pin);

    if (config->itemExists("asset"))
        dht11->setAssetName(config->getValue("asset"));
    else
        dht11->setAssetName("dht11");

    Logger::getLogger()->info("m_assetName set to %s", dht11->getAssetName());

    return (PLUGIN_HANDLE) dht11;
}

/**
 * Poll for a plugin reading
 */
Reading plugin_poll(PLUGIN_HANDLE *handle)
{
    DHT11 *dht11 = (DHT11*)handle;
    return dht11->takeReading();
}

/**
 * Reconfigure the plugin
 */
void plugin_reconfigure(PLUGIN_HANDLE *handle, string& newConfig)
{
    ConfigCategory conf("dht", newConfig);
    DHT11 *dht11 = (DHT11*)*handle;

    if (conf.itemExists("asset"))
        dht11->setAssetName(conf.getValue("asset"));
    if (conf.itemExists("pin"))
    {
        unsigned int pin = stoul(conf.getValue("pin"), nullptr, 0);
        dht11->setPin(pin);
    }
}

/**
 * Shutdown the plugin

```

(continues on next page)

(continued from previous page)

```

*/
void plugin_shutdown(PLUGIN_HANDLE *handle)
{
    DHT11 *dht11 = (DHT11*)handle;
    delete dht11;
}
};

```

The full source code, including the *DHT11* class can be found in [GitHub](#)

Building Flir and Adding the Plugin

If you have not built Flir yet, follow the steps described . After the build, you can optionally install Flir following steps.

- Clone the *flir-south-dht* repository

```

$ git clone https://github.com/flir/flir-south-dht.git
...
$

```

- Set the environment variable `FLIR_ROOT` to the directory in which you built Flir

```

$ export FLIR_ROOT=~/.flir
$

```

- Go to the location in which you cloned the *flir-south-dht* repository and create a build directory and run `cmake` in that directory

```

$ cd ~/.flir-south-dht
$ mkdir build
$ cd build
$ cmake ..
...
$

```

- Now make the plugin

```

$ make
$

```

- If you have started Flir from the build directory, copy the plugin into the destination directory

```

$ mkdir -p $FLIR_ROOT/plugins/south/dht
$ cp libdht.so $FLIR_ROOT/plugins/south/dht
$

```

- If you have installed Flir by executing `sudo make install`, copy the plugin into the destination directory

```

$ sudo mkdir -p /usr/local/flir/plugins/south/dht
$ sudo cp libdht.so /usr/local/flir/plugins/south/dht
$

```

Note: If you have installed Flir using an alternative *DESTDIR*, remember to add the path to the destination directory to the `cp` command.

- Add service

```
$ curl -sX POST http://localhost:8081/flir/service -d '{"name": "dht", "type": "south", "plugin": "dht", "enabled": true}'
```

You may now use the C/C++ plugin in exactly the same way as you used a Python plugin earlier.

10.5 C++ Support Classes

A number of support classes exist within the common library that forms part of every Flir plugin.

10.5.1 Reading

The *Reading* class and the associated *Datapoint* and *DatapointValue* classes provide the mechanism within C++ classes to manipulated the reading asset data. The public part of the *Reading* class is currently defined as follows;

```
class Reading {
public:
    Reading(const std::string& asset, Datapoint *value);
    Reading(const std::string& asset, std::vector<Datapoint *> values);
    Reading(const std::string& asset, std::vector<Datapoint *> values,
↳const std::string& ts);
    Reading(const Reading& orig);

    ~Reading();
    void addDatapoint(Datapoint *value);
    Datapoint *removeDatapoint(const std::string&
↳name);

    std::string toJSON(bool minimal = false) const;
    std::string getDatapointsJSON() const;
    // Return AssetName
    const std::string& getAssetName() const { return m_asset;
↳ };

    // Set AssetName
    void setAssetName(std::string assetName) {
↳m_asset = assetName; };
    unsigned int getDatapointCount() { return m_values.
↳size(); };

    void removeAllDatapoints();
    // Return Reading datapoints
    const std::vector<Datapoint *> getReadingData() const { return m_
↳values; };

    // Return reference to Reading datapoints
    std::vector<Datapoint *>& getReadingData() { return m_values; };
    unsigned long getId() const { return m_id; };
    unsigned long getTimestamp() const { return
↳(unsigned long)m_timestamp.tv_sec; };
    unsigned long getUserTimestamp() const { return
↳(unsigned long)m_userTimestamp.tv_sec; };
    void setId(unsigned long id) { m_id = id; }
↳;
}
```

(continues on next page)

(continued from previous page)

```

        void                                setTimestamp(unsigned long ts) { m_
↪timestamp.tv_sec = (time_t)ts; };
        void                                setTimestamp(struct timeval tm) { m_
↪timestamp = tm; };
        void                                setTimestamp(const std::string&_
↪timestamp);
        void                                getTimestamp(struct timeval *tm) {_
↪*tm = m_timestamp; };
        void                                setUserTimestamp(unsigned long uTs) {_
↪m_userTimestamp.tv_sec = (time_t)uTs; };
        void                                setUserTimestamp(struct timeval tm) {_
↪m_userTimestamp = tm; };
        void                                setUserTimestamp(const std::string&_
↪timestamp);
        void                                getUserTimestamp(struct timeval *tm)
↪{ *tm = m_userTimestamp; };

        typedef enum dateTimeFormat { FMT_DEFAULT, FMT_STANDARD, FMT_ISO8601 }
↪ readingTimeFormat;

        // Return Reading asset time - ts time
        const std::string getAssetDateTime(readingTimeFormat datetimeFmt =_
↪FMT_DEFAULT, bool addMs = true) const;
        // Return Reading asset time - user_ts time
        const std::string getAssetDateUserTime(readingTimeFormat datetimeFmt_
↪= FMT_DEFAULT, bool addMs = true) const;
    }

```

The *Reading* class contains a number of items that are mapped to the JSON representation of data that is sent to the Flir storage service and are used by the various services and plugins within Flir.

- **Asset Name:** The name of the asset. The asset name is set in the constructor of the reading and retrieved via the *getAssetName()* method.
- **Timestamp:** The timestamp when the reading was first seen within Flir.
- **User Timestamp:** The timestamp for the actual data in the reading. This may differ from the value of Timestamp if the device itself is able to supply a timestamp value.
- **Datapoints:** The actual data of a reading stored in a Datapoint class.

The *Datapoint* class provides a name for each data point within a *Reading* and the tagged type data for the reading value. The public definition of the *Datapoint* class is as follows;

```

class Datapoint {
    public:
        /**
         * Construct with a data point value
         */
        Datapoint(const std::string& name, DatapointValue& value) : m_
↪name(name), m_value(value);
        ~Datapoint();
        /**
         * Return asset reading data point as a JSON
         * property that can be included within a JSON
         * document.
         */
        std::string toJSONProperty();

```

(continues on next page)

(continued from previous page)

```

    const std::string getName() const;
    void setName(std::string name);
    const DatapointValue getData() const;
    DatapointValue& getData();
}

```

Closely associated with the *Datapoint* is the *DatapointValue* which uses a tagged union to store the values. The public definition of the *DatapointValue* is as follows;

```

class DatapointValue {
public:
    /**
     * Constructors for the various types
     */
    DatapointValue(const std::string& value;
    DatapointValue(const long value);
    DatapointValue(const double value);
    DatapointValue(const std::vector<double>& values);
    DatapointValue(std::vector<Datapoint*>& values, bool isDict)
    DatapointValue(const DatapointValue& obj)

    DatapointValue& operator=(const DatapointValue& rhs)
    ~DatapointValue();

    void deleteNestedDPV();

    /**
     * Set the value for the various types
     */
    void setValue(long value);
    void setValue(double value);

    /**
     * Return the value as the various types
     */
    std::string toString() const;
    long toInt() const;
    double toDouble() const;

    typedef enum DatapointTag
    {
        T_STRING,
        T_INTEGER,
        T_FLOAT,
        T_FLOAT_ARRAY,
        T_DP_DICT,
        T_DP_LIST
    } dataTagType;
    dataTagType getType() const;
    std::string getTypeStr() const;
    std::vector<Datapoint*>& getDpVec();
}

```


10.5.2 Configuration Category

The *ConfigCategory* class is a support class for managing configuration information within a plugin and is passed to the plugin entry points. The public definition of the class is as follows;

```
class ConfigCategory {
    public:
        enum ItemType {
            UnknownType,
            StringItem,
            EnumerationItem,
            JsonItem,
            BoolItem,
            NumberItem,
            DoubleItem,
            ScriptItem,
            CategoryType,
            CodeItem
        };

        ConfigCategory(const std::string& name, const std::string& json);
        ConfigCategory() {};
        ConfigCategory(const ConfigCategory& orig);
        ~ConfigCategory();

        void addItem(const std::string& name,
            ↪const std::string description,
            ↪const std::string def,
            ↪const std::string description,
            ↪std::string& value,
            ↪options);
        void removeItems();
        void removeItemsType(ItemType type);
        void keepItemsType(ItemType type);
        bool extractSubcategory(ConfigCategory &
            ↪subCategories);
        void setDescription(const std::string&
            ↪description);
        std::string getName() const;
        std::string getDescription() const;
        unsigned int getCount() const;
        bool itemExists(const std::string& name)
            ↪const;
        bool setItemDisplayName(const std::string&
            ↪name, const std::string& displayName);
        std::string getValue(const std::string& name)
            ↪const;
        std::string getType(const std::string& name)
            ↪const;
        std::string getDescription(const std::string&
            ↪name) const;
        std::string getDefault(const std::string& name)
            ↪const;
        bool setDefault(const std::string& name,
            ↪const std::string& value);
};
```

(continues on next page)

(continued from previous page)

```

std::string
↪name) const;
std::vector<std::string>
↪const;
std::string
↪const;
std::string
↪const;
std::string
↪const;
bool
↪const;
bool
↪name) const;
bool
bool
bool
↪const;
bool
↪const;
bool
↪const;
std::string
std::string
↪const;
ConfigCategory&
ConfigCategory&
void
void
std::string
↪itemName) const;
enum ItemAttribute
↪MANDATORY_ATTR, FILE_ATTR;
std::string
↪itemName,
ItemAttribute
↪itemAttribute) const;
}

getDisplayNames(const std::string& name)
getOptions(const std::string& name)
getLength(const std::string& name)
getMinimum(const std::string& name)
getMaximum(const std::string& name)
isString(const std::string& name)
isEnumeration(const std::string& name)
isJSON(const std::string& name) const;
isBool(const std::string& name) const;
isNumber(const std::string& name)
isDouble(const std::string& name)
isDeprecated(const std::string& name)
toJson(const bool full=false) const;
itemsToJson(const bool full=false)
operator=(ConfigCategory const& rhs);
operator+=(ConfigCategory const& rhs);
setItemValueFromDefault();
checkDefaultValuesOnly() const;
itemToJson(const std::string& name,
ItemAttribute attr) const;
{ ORDER_ATTR, READONLY_ATTR,
getItemAttribute(const std::string& name,
ItemAttribute attr) const;

```

Although *ConfigCategory* is a complex class, only a few of the methods are commonly used within a plugin

- **itemExists:** - used to test if an expected configuration item exists within the configuration category.
- **getValue:** - return the value of a configuration item from within the configuration category
- **isBool:** - tests if a configuration item is of boolean type
- **isNumber:** - tests if a configuration item is a number
- **isDouble:** - tests if a configuration item is valid to be represented as a double
- **isString:** - tests if a configuration item is a string

10.5.3 Logger

The *Logger* class is used to write entries to the syslog system within Flir. A singleton *Logger* exists which can be obtained using the following code snippet;

```
Logger *logger = Logger::getLogger();
logger->error("An error has occurred within the plugin processing");
```

It is then possible to log messages at one of five different log levels; *debug*, *info*, *warn*, *error* or *fatal*. Messages may be logged using standard printf formatting strings. The public definition of the *Logger* class is as follows;

```
class Logger {
    public:
        Logger(const std::string& application);
        ~Logger();
        static Logger *getLogger();
        void debug(const std::string& msg, ...);
        void printLongString(const std::string&);
        void info(const std::string& msg, ...);
        void warn(const std::string& msg, ...);
        void error(const std::string& msg, ...);
        void fatal(const std::string& msg, ...);
        void setMinLevel(const std::string& level);
};
```

The various log levels should be used as follows;

- **debug**: should be used to output messages that are relevant only to a programmer that is debugging the plugin.
- **info**: should be used for information that is meaningful to the end users, but should not normally be logged.
- **warn**: should be used for warning messages that will normally be logged but reflect a condition that does not prevent the plugin from operating.
- **error**: should be used for conditions that cause a temporary failure in processing within the plugin.
- **fatal**: should be used for conditions that cause the plugin to fail processing permanently, possibly requiring a restart of the microservice in order to resolve.

10.6 Hybrid Plugins

In addition to plugins written in Python and C/C++ it is possible to have a hybrid plugin that is a combination of an existing plugin and configuration for that plugin. This is useful in a situation whereby there are multiple sensors or devices that you connect to Flir that have common configuration. It allows devices to be added without repeating the common configuration.

Using our example of a *DHT11* sensor connected to a GPIO pin, if we wanted to create a new plugin for a *DHT11* that was always connected to pin 4 then we could do this by creating a JSON file as below that supplies a fixed default value for the GPIO pin.

```
{
    "description" : "A DHT11 sensor connected to GPIO pin 4",
    "name" : "DHT11-4",
    "connection" : "DHT11",
    "defaults" : {
        "pin" : {
            "default" : "4"
        }
    }
}
```

This creates a new hybrid plugin called DHT11-4 that is installed by copying this file into the `plugins/south/DHT11-4` directory of your installation. Once installed it can be treated as any other south plugin within Flir. The effect of this hybrid plugin is to load the *DHT11* plugin and always set the configuration parameter called “pin” to the value “4”. The item “pin” will be hidden from the user in the Flir GUI when they create the instance of the plugin. This allows for a simpler and more streamlined user experience when adding plugins with common configuration.

The items in the JSON file are;

Name	Description
description	A description of the hybrid plugin. This will appear the right of the selection list in the Flir user interface when the plugin is selected.
name	The name of the plugin itself. This must match the filename of the JSON file and also the name of the directory the file is placed in.
connection	The name of the underlying plugin that will be used as the basis for this hybrid plugin. This must be a C/C++ or Python plugin, it can not be another hybrid plugin.
defaults	The set of values to default in this hybrid plugin. These are configuration parameters of the underlying plugin that will be fixed in the hybrid plugin. Each hybrid plugin can have one or many values here.

It may not be difficult to enter the GPIO pin in each case in this example, where it becomes more useful is for plugins such as *Modbus* where a complex map is required to be entered in a JSON document. By using a hybrid plugin we can define the map we need once and then add new sensors of the same type without having to repeat the map. An example of this would be the Flir AX8 camera that requires a total of 176 Modbus registers to be mapped into 88 different values in an asset. A hybrid plugin *flir-south-FlirAX8* defines that mapping once and as a result adding a new Flir AX8 camera is as simple as selecting the FlirAX8 hybrid plugin and entering the IP address of the camera.

10.7 North Plugins

North plugins are used in North tasks and microservices to extract data buffered in Flir and send it Northbound, i.e. to a server or a service in the Cloud or in an Enterprise data center. We currently have two North plugins, one to send data to an OSIsoft PI Server and one to the OSIsoft Cloud Service.

10.7.1 The OMF Plugin

The OMF Plugin is used by a North task to send data to an OSIsoft PI server via a PI Connector Relay or PI Web API, it can also send to Edge Data Store or OSIsoft Cloud Services. All these destinations share a single protocol for communication, OMF. OMF stands for OSIsoft Message Format, it is the JSON format defined by OSIsoft to send IoT data to a PI server via a Connector Relay server.

The plugin is designed to send two streams of data:

- The data collected by South microservices and buffered into Flir
- The statistics generated by Flir

The streams are managed by two different North tasks using the same plugin, but with a different configuration. The two tasks are registered in the list of scheduled jobs and they can be identified using the `schedule` API call:

```
$ curl -sX GET http://localhost:8081/flir/schedule
{
  "schedules": [
    {
      "id": "ef8bd42b-da9f-47c4-ade8-751ce9a504be",
```

(continues on next page)

(continued from previous page)

```

    "name": "OMF to PI north",
    "processName": "north_c",
    "type": "INTERVAL",
    "repeat": 30.0,
    "time": 0,
    "day": null,
    "exclusive": true,
    "enabled": false
  },
  {
    "id": "27501b35-e0cd-4340-afc2-a4465fe877d6",
    "name": "Stats OMF to PI north",
    "processName": "north_c",
    "type": "INTERVAL",
    "repeat": 30.0,
    "time": 0,
    "day": null,
    "exclusive": true,
    "enabled": true
  },
  ...
]
}

```

The output of API call above shows three interesting tasks: the two tasks associated to the OMF plugin, the one to send data (*OMF to PI north*) and the one to send statistics (*Stats OMF to PI north*).

The two scheduled tasks are associated to two configuration items that can be retrieved using the category API call. The items are named OMF to PI north and Stats OMF to PI north.

```

$ curl -sX GET http://localhost:8081/flir/category/OMF%20to%20PI%20north
{
  "enable": {
    "description": "A switch that can be used to enable or disable execution of the_
↪sending process.",
    "type": "boolean",
    "readonly": "true",
    "default": "true",
    "value": "true"
  },
  "streamId": {
    "description": "Identifies the specific stream to handle and the related_
↪information, among them the ID of the last object streamed.",
    "type": "integer",
    "readonly": "true",
    "default": "0",
    "value": "4",
    "order": "16"
  },
  "plugin": {
    "description": "PI Server North C Plugin",
    "type": "string",
    "default": "OMF",
    "readonly": "true",
    "value": "OMF"
  },
  "source": {

```

(continues on next page)

(continued from previous page)

```

        "description": "Defines the source of the data to be sent on the stream, this
↪may be one of either readings, statistics or audit.",
        "type": "enumeration",
        "options": [
            "readings",
            "statistics"
        ],
        "default": "readings",
        "order": "5",
        "displayName": "Data Source",
        "value": "readings"
    },
    ...}
$ curl -sX GET http://localhost:8081/flir/category/Stats%20OMF%20to%20PI%20north
{
  "enable": {
    "description": "A switch that can be used to enable or disable execution of the
↪sending process.",
    "type": "boolean",
    "readonly": "true",
    "default": "true",
    "value": "true"
  },
  "streamId": {
    "description": "Identifies the specific stream to handle and the related
↪information, among them the ID of the last object streamed.",
    "type": "integer",
    "readonly": "true",
    "default": "0",
    "value": "5",
    "order": "16"
  },
  "plugin": {
    "description": "PI Server North C Plugin",
    "type": "string",
    "default": "OMF",
    "readonly": "true",
    "value": "OMF"
  },
  "source": {
    "description": "Defines the source of the data to be sent on the stream, this may
↪be one of either readings, statistics or audit.",
    "type": "enumeration",
    "options": [
        "readings",
        "statistics"
    ],
    "default": "readings",
    "order": "5",
    "displayName": "Data Source",
    "value": "statistics"
  },
  ...}
$
```

In order to activate the tasks, you must change their status. First you must collect their id (from the GET method of the schedule API call), then you must use the IDs with the PUT method of the same call:

```
$ curl -sX PUT http://localhost:8081/flir/schedule/ef8bd42b-da9f-47c4-ade8-751ce9a504be -d '{ "enabled" : true}'
{
  "schedule": {
    "id": "ef8bd42b-da9f-47c4-ade8-751ce9a504be",
    "name": "OMF to PI north",
    "processName": "north_c",
    "type": "INTERVAL",
    "repeat": 30,
    "time": 0,
    "day": null,
    "exclusive": true,
    "enabled": true
  }
}
$ curl -sX PUT http://localhost:8081/flir/schedule/27501b35-e0cd-4340-afc2-a4465fe877d6 -d '{ "enabled" : true}'
{
  "schedule": {
    "id": "27501b35-e0cd-4340-afc2-a4465fe877d6",
    "name": "Stats OMF to PI north",
    "processName": "north_c",
    "type": "INTERVAL",
    "repeat": 30,
    "time": 0,
    "day": null,
    "exclusive": true,
    "enabled": true
  }
}
$
```

At this point, the configuration has been enriched with default values of the tasks:

```
$ curl -sX GET http://localhost:8081/flir/category/OMF%20to%20PI%20north
{
  "enable": {
    "description": "A switch that can be used to enable or disable execution of the_
    ↪ sending process.",
    "type": "boolean",
    "readonly": "true",
    "default": "true",
    "value": "true"
  },
  "streamId": {
    "description": "Identifies the specific stream to handle and the related_
    ↪ information, among them the ID of the last object streamed.",
    "type": "integer",
    "readonly": "true",
    "default": "0",
    "value": "4",
    "order": "16"
  },
  "plugin": {
    "description": "PI Server North C Plugin",
    "type": "string",
    "default": "OMF",

```

(continues on next page)

(continued from previous page)

```

    "readonly": "true",
    "value": "OMF"
  },
  "source": {
    "description": "Defines the source of the data to be sent on the stream, this_
↪may be one of either readings, statistics or audit.",
    "type": "enumeration",
    "options": [
      "readings",
      "statistics"
    ],
    "default": "readings",
    "order": "5",
    "displayName": "Data Source",
    "value": "readings"
  },
  ...}
$ curl -sX GET http://localhost:8081/flir/category/Stats%20OMF%20to%20PI%20north
{
  "enable": {
    "description": "A switch that can be used to enable or disable execution of the_
↪sending process.",
    "type": "boolean",
    "readonly": "true",
    "default": "true",
    "value": "true"
  },
  "streamId": {
    "description": "Identifies the specific stream to handle and the related_
↪information, among them the ID of the last object streamed.",
    "type": "integer",
    "readonly": "true",
    "default": "0",
    "value": "5",
    "order": "16"
  },
  "plugin": {
    "description": "PI Server North C Plugin",
    "type": "string",
    "default": "OMF",
    "readonly": "true",
    "value": "OMF"
  },
  "source": {
    "description": "Defines the source of the data to be sent on the stream, this may_
↪be one of either readings, statistics or audit.",
    "type": "enumeration",
    "options": [
      "readings",
      "statistics"
    ],
    "default": "readings",
    "order": "5",
    "displayName": "Data Source",
    "value": "statistics"
  },
  ...}

```

(continues on next page)

(continued from previous page)

\$

OMF Plugin Configuration

The following table presents the list of configuration options available for the task that sends data to OMF (category *OMF to PI north*):

Item	Type	Default	Description
AFMap	JSON	{ }	Defines a set of rules to address where assets should be placed in the AF hierarchy.
compression	boolean	true	Compress readings data before sending to PI server
DefaultAFLocation	integer	/flir/data_piwebapi/default	Defines the hierarchies tree in Asset Framework in which the assets will be created, each level is separated by /, PI Web API only.
enable	boolean	True	A switch that can be used to enable or disable execution of the sending process.
formatInteger	string	int64	OMF format property to apply to the type Integer.
formatNumber	string	float64	OMF format property to apply to the type Number
notBlockingErrors	JSON	“{ “errors400” : [“Redefinition of the type with the same ID is not allowed”, “Invalid value type for the property”, “Property does not exist in the type definition”, “Container is not defined”, “Unable to find the property of the container of type”] }”	These errors are considered not blocking in the communication with the PI Server, the sending operation will proceed with the next block of data if one of these is encountered.
OCSCClientSecret	boolean	ocs_client_secret	Client secret associated to the specific OCS account, it is used to authenticate the source for using the OCS API.
OCSCClientId	string	ocs_client_id	Client id associated to the specific OCS account, it is used to authenticate the source for using the OCS API.
OCSTenantId	string	ocs_tenant_id	Tenant id associated to the specific OCS account
OCSNamespace	string	name_space	Specifies the OCS namespace where the information are stored and it is used for the interaction with the OCS API.
OMFHttpTimeout	integer	10	Timeout in seconds for the HTTP operations with the OMF PI Connector Relay
OMFMaxRetry	integer	1	Seconds between each retry for the communication with the OMF PI Connector Relay, NOTE : the time is doubled at each attempt.
PIWebAPIKerberosKeytabFileName	string	piwebapi_kerberos_https.keytab	Keytab file name used for Kerberos authentication in PI Web API.
PIWebAPIAuthenticationMethod	enum	anonymous	Defines the authentication method to be used with the PI Web API.
PIWebAPIPassword	password	password	Password of the user of PI Web API to be used with the basic access authentication.
PIWebAPIUserId	string	user_id	User id of PI Web API to be used with the basic access authentication.
PIServerEndpoint	enumeration	Connector Relay	Select the endpoint among PI Web API, connector Relay, OSIsoft Cloud Services or Edge Data Store
plugin	string	OMF	PI Server North C Plugin

The following table presents the list of configuration options available for the task that sends statistics to OMF (category *Stats OMF to PI north*):

Item	Type	Default	Description
AFMap	JSON	{ }	Defines a set of rules to address where assets should be placed in the AF hierarchy.
compression	boolean	true	Compress readings data before sending to PI server
DefaultAFLocation	integer	/flir/data_piwebapi/default	Defines the hierarchies tree in Asset Framework in which the assets will be created, each level is separated by /, PI Web API only.
enable	boolean	True	A switch that can be used to enable or disable execution of the sending process.
formatInteger	string	int64	OMF format property to apply to the type Integer.
formatNumber	string	float64	OMF format property to apply to the type Number
notBlockingErrors	JSON	“{ “errors400” : [“Redefinition of the type with the same ID is not allowed”, “Invalid value type for the property”, “Property does not exist in the type definition”, “Container is not defined”, “Unable to find the property of the container of type”] }”	These errors are considered not blocking in the communication with the PI Server, the sending operation will proceed with the next block of data if one of these is encountered.
OCSCClientSecret	boolean	ocs_client_secret	Client secret associated to the specific OCS account, it is used to authenticate the source for using the OCS API.
OCSCClientId	string	ocs_client_id	Client id associated to the specific OCS account, it is used to authenticate the source for using the OCS API.
OCSTenantId	string	ocs_tenant_id	Tenant id associated to the specific OCS account
OCSNamespace	string	name_space	Specifies the OCS namespace where the information are stored and it is used for the interaction with the OCS API.
OMFHttpTimeout	integer	10	Timeout in seconds for the HTTP operations with the OMF PI Connector Relay
OMFMaxRetry	integer	1	Seconds between each retry for the communication with the OMF PI Connector Relay, NOTE : the time is doubled at each attempt.
PIWebAPIKerberosKeytabFileName	string	piwebapi_kerberos_https.keytab	Keytab file name used for Kerberos authentication in PI Web API.
PIWebAPIAuthenticationMethod	enum	anonymous	Defines the authentication method to be used with the PI Web API.
PIWebAPIPassword	password	password	Password of the user of PI Web API to be used with the basic access authentication.
PIWebAPIUserId	string	user_id	User id of PI Web API to be used with the basic access authentication.
PIServerEndpoint	enumeration	Connector Relay	Select the endpoint among PI Web API, connector Relay, OSIsoft Cloud Services or Edge Data Store
plugin	string	OMF	PI Server North C Plugin

The last parameter to review is the *OMF Type*. The call is the GET method `flir/category/OMF_TYPES`, which returns an integer value that identifies the measurement type:

```
$ curl -sX GET http://localhost:8081/flir/category/OMF_TYPES
{
  "type-id": {
    "description": "Identify sensor and measurement types",
    "type": "integer",
    "default": "0001",
    "value": "0001"
  }
}
```

If you change the value, you can easily identify the set of data sent to and then stored into PI.

Changing the OMF Plugin Configuration

Before you send data to the PI server, it is likely that you need to apply more changes to the configuration. The most important items to change are:

- **URL** : the URL to the PI Connector Relay OMF. It is usually composed by the name or address of the Windows server where the Connector Relay service is running, the port associated to the service and the ingress/messages API call. The communication is via HTTPS protocol.
- **producerToken** : the token provided by the Data Collection Manager when the PI administrator sets the use of Flir.
- **type-id** : the measurement type for the stream of data.
- **source** : this parameter should be set to *readings* (default) when the plugin is used to send data collected by South microservices, and to *statistics* when the plugin is used to send Flir statistics to the PI system.

An example of the changes to apply to the plugins to send data to the PI system is available here [here](#).

Data in the PI System

Once the North plugins have been set properly, you should expect to see data automatically sent and stored in the PI Server. More specifically, the process of the plugin is the following:

- **Assets** buffered in Flir are stored as *elements* in the PI System. - *PI Asset Framework* is automatically update with the new assets. - JSON objects captured as part of the *reading* in Flir become *attributes* in the PI Data Archive
- The **Producer Token** is used to authenticate and create the hierarchy of elements in the *PI Asset Framework*
- The configuration object named as **Static Data** is added as a set of *attributes* in the PI Data Archive

System	Object	Value
Flir	Producer Token	readings_001
	OMF Type	001
	Static Data	{ "Company": "Dianomic", "Location": "Palo Alto" }
	Asset	fogbench/accelerometer
	Reading	[{"reading":{"y":1,"z":1,"x":-1}, "timestamp":"2018-05-14 19:27:06.788"]
PI	Element Template	[OMF.readings_001 Connector.0001_fogbench/accelerometer_typename_sensor]
	Attribute Template	[OMF.readings_001 Connector.0001_fogbench/accelerometer_typename_sensor]
		Company Configuration Item, Excluded, String
		Location Configuration Item, Excluded, String
		x Excluded, Int64
		y Excluded, Int64
		z Excluded, Int64
	Element	flir > readings_001 > fogbench/accelerometer
	Attributes	Company Dianomic 1970-01-01 00:00:00
		Location Palo Alto 1970-01-01 00:00:00
		x -1 2018-05-14 19:27:06.788
		y -1 2018-05-14 19:27:06.788
		z -1 2018-05-14 19:27:06.788

10.8 Storage Plugins

Storage plugins are used to interact with the Storage Microservice and provide the persistent storage of information for Flir.

The current version of Flir comes with three storage plugins:

- The **SQLite plugin**: this is the default plugin and it is used for general purpose storage on constrained devices.
- The **SQLite In Memory plugin**: this plugin can be used in conjunction with one of the other storage plugins and will provide an in memory storage system for reading data only. Configuration data is stored using the *SQLite* or *PostgreSQL* plugins.
- The **PostgreSQL plugin**: this plugin can be set on request (or it can be built as a default plugin from source) and it is used for a more significant demand of storage on relatively larger systems.

10.8.1 Data and Metadata

Persistency is split in two blocks:

- **Metadata persistency**: it refers to the storage of metadata for Flir, such as the configuration of the plugins, the scheduling of jobs and tasks and the the storage of statistical information.
- **Data persistency**: it refers to the storage of data collected from sensors and devices by the South microservices. The *SQLite In Memory* plugin is an example of a storage plugin designed to store only the data.

In the current implementation of Flir, metadata and data use the same Storage plugin by default. Administrators can select different plugins for these two categories of data, with the most common configuration of this type to use the *SQLite In Memory* storage service for data and *SQLite* for the metadata. This is set by editing the storage configuration file. Currently there is no interface within Flir to change the storage configuration.

The storage configuration file is stored in the Flir data directory as etc/storage.json, the default storage configuration file is

```
{
  "plugin": {
    "value": "sqlite",
    "description": "The main storage plugin to load"
  },
  "readingPlugin": {
    "value": "",
    "description": "The storage plugin to load for readings data. If blank the main_
↪storage plugin is used."
  },
  "threads": {
    "value": "1",
    "description": "The number of threads to run"
  },
  "managedStatus": {
    "value": "false",
    "description": "Control if Flir should manage the storage provider"
  },
  "port": {
    "value": "0",
    "description": "The port to listen on"
  },
  "managementPort": {
    "value": "0",
    "description": "The management port to listen on."
  }
}
```

This sets the storage plugin to use as the *SQLite* plugin and leaves the *readingPlugin* blank. If the *readingPlugin* is blank then readings will be stored via the main plugin, if it is populated then a separate plugin will be used to store the readings. As an example, to store the readings in the *SQLite In Memory* plugin the storage.json file would be

```
{
  "plugin": {
    "value": "sqlite",
    "description": "The main storage plugin to load"
  },
  "readingPlugin": {
    "value": "sqlitememory",
    "description": "The storage plugin to load for readings data. If blank the main_
↪storage plugin is used."
  },
  "threads": {
    "value": "1",
    "description": "The number of threads to run"
  },
  "managedStatus": {
    "value": "false",
    "description": "Control if Flir should manage the storage provider"
  },
  "port": {
    "value": "0",
    "description": "The port to listen on"
  },
  "managementPort": {
```

(continues on next page)

(continued from previous page)

```
"value": "0",  
"description": "The management port to listen on."  
}  
}
```

Flir must be restarted for changes to the storage.json file to take effect.

In addition to the definition of the plugins to use, the storage.json file also has a number of other configuration options for the storage service.

- **threads:** The number of threads to use to accept incoming REST requests. This is normally set to 1, increasing the number of threads has minimal impact on performance in normal circumstances.
- **managedStatus:** This configuration option allows Flir to manage the underlying storage system. If, for example you used a database server and you wished Flir to start and stop that server as part of the Flir start up and shut down procedure you would set this option to “true”.
- **port:** This option can be used to make the storage service listen on a fixed port. This is normally not required, but can be used for diagnostic purposes.
- **managementPort:** As with *port* above this can be used for diagnostic purposes to fix the management API port for the storage service.

10.8.2 Common Elements for Storage Plugins

In designing the Storage API and plugins, we have first of all considered that there may be a large number of use cases for data and metadata persistence, therefore we have designed a flexible architecture that poses very few limitations. In practice, this means that developers can build their own Storage plugin and they can rely on anything they want to use as persistent storage. They can use a memory structure, or even a pass-through library, a file, a message queue system, a time series database, a relational database, NoSQL or something else.

After having praised the flexibility of the Storage plugins, let’s provide guidelines about the basic functionality they should provide, bearing in mind that such functionality may not be relevant for some use cases.

- **Metadata persistency:** As mentioned before, one of the main reasons to use a Storage plugin is to safely store the configuration of the Flir components. Since the configuration must survive to a system crash or reboot, it is fair to say that such information should be stored in one or more files or in a database system.
- **Data buffering:** The second most important feature of a Storage plugin is the ability to buffer (or store) data coming from the outside world, typically from the South microservices. In some cases this feature may not be necessary, since administrators may want to send data to other systems as soon as possible, using a North task of microservice. Even in situations where data can be sent up North instantaneously, you should consider these scenarios:
 - Flir may be installed in areas where the network is unreliable. The North plugins will provide the logic of retrying to gain connectivity and resending data when the connection has been lost in the middle of the transfer operations.
 - North services may rely on the use of networks that provide time windows to operate.
 - Historians and other systems may work better when data is transferred in blocks instead of a constant streaming.
- **Data purging:** Data may persist for the time needed by any specific use case, but it is pretty common that after a while (it can be seconds or minutes, but also day or months) data is no longer needed in Flir. For this reason, the Storage plugin is able to purge data. Purging may be by time or by space usage, in conjunction with the fact that data may have been already transferred to other systems.

- **Data backup/restore:** Data, but especially metadata (i.e. configuration), can be backed up and stored safely on other systems. In case of crash and recovery, the same data may be restored into Flir. Flir provides a set of generic API to execute backup and restore operations.

10.9 Filter Plugins

Filter plugins provide a mechanism to alter the data stream as it flows through a flir instance, filters may be applied in south or north micro-services and may form a pipeline of multiple processing elements through which the data flows. Filters applied in a south service will only process data that is received by the south service, whilst filters placed in the north will process all data that flows out of that north interface.

Filters may;

- augment data by adding static metadata or calculated values to the data
- remove data from the stream
- add data to the stream
- modify data in the stream

It should be noted that there are some alternatives to creating a filter if you wish to make simple changes to the data stream. There are a number of existing filters that provide a degree of programmability. These include the which allows an arbitrary mathematical formula to be applied to the data or the which allows a small include Python script to be applied to the data.

Filter plugins may be written in C++ or Python and have a very simple interface. The plugin mechanism and a subset of the API is common between all types of plugins including filters.

10.9.1 Configuration

Filters use the same configuration mechanism as the rest of Flir, using a JSON document to describe the configuration parameters. As with any other plugin the structure is defined by the plugin and retrieve by the `plugin_info` entry point. This is then matched with the database content to pass the configured values to the `plugin_init` entry point.

10.9.2 C++ Filter Plugin API

The filter API consists of a small number of C function entry points, these are called in a strict order and based on the same set of common API entry points for all Flir plugins.

Plugin Information

The *plugin_info* entry point is the first entry point that is called in a filter plugin and returns the plugin information structure. This is the exact same call that every Flir plugin must support and is used to determine the type of the plugin and the configuration category defaults for the plugin.

A typical implementation of *plugin_info* would merely return a pointer to a static `PLUGIN_INFORMATION` structure.

```
PLUGIN_INFORMATION *plugin_info()
{
    return &info;
}
```

Plugin Initialise

The *plugin_init* entry point is called after *plugin_info* has been called and before any data is passed to the filter. It is called at the phase where the service is setting up the filter pipeline and provides the filter with its configuration category that now contains the user supplied values and the destination to which the filter will send the output of the filter.

```
PLUGIN_HANDLE plugin_init(ConfigCategory* config,
                          OUTPUT_HANDLE *outHandle,
                          OUTPUT_STREAM output)
{
}
```

The *config* parameter is the configuration category with the user supplied values inserted, the *outHandle* is a handle for the next filter in the chain and the *output* is a function pointer to call to send the data to the next filter in the chain. The *outHandle* and *output* arguments should be stored for future use in the *plugin_ingest* when data is to be forwarded within the pipeline.

The *plugin_init* function returns a handle that will be passed to all subsequent plugin calls. This handle can be used to store state that needs to be passed between calls. Typically the *plugin_init* call will create a C++ class that implements the filter and return a point to the instance as the handle. The instance can then be used to store the state of the filter, including the output handle and callback that needs to be used.

Filter classes can also be used to buffer data between calls to the *plugin_ingest* entry point, allowing a filter to defer the processing of the data until it has a sufficient quantity of buffered data available to it.

Plugin Ingest

The *plugin_ingest* entry point is the workhorse of the filter, it is called with sets of readings to process and then passes on the new set of readings to the next filter in the pipeline. The process of passing on the data to the next filter is via the *OUTPUT_STREAM* function pointer. A filter does not have to output data each time it ingests data, it is free to output no data or to output more or less data than it was called with.

```
void plugin_ingest(PLUGIN_HANDLE *handle,
                  READINGSET *readingSet)
{
}
```

The number of readings that a filter is called with will depend on the environment it is run in and what any filters earlier in the filter pipeline have produced. A filter that requires a particular sample size in order to process a result should therefore be prepared to buffer data across multiple calls to *plugin_ingest*. Several examples of filters that do this are available for reference.

The *plugin_ingest* call may send data onwards in the filter pipeline by using the stored *output* and *outHandle* parameters passed to *plugin_init*.

```
(*output)(outHandle, readings);
```

Plugin Reconfigure

As with other plugin types the filter may be reconfigured during its lifetime. When a reconfiguration operation occurs the *plugin_reconfigure* method will be called with the new configuration for the filter.

```
void plugin_reconfigure(PLUGIN_HANDLE *handle, const std::string& newConfig)
{
}

```

Plugin Shutdown

As with other plugins a shutdown call exists which may be used by the plugin to perform any cleanup that is required when the filter is shut down.

```
void plugin_shutdown(PLUGIN_HANDLE *handle)
{
}

```

C++ Helper Class

It is expected that filters will be written as C++ classes, with the plugin handle being used as a mechanism to store and pass the pointer to the instance of the filter class. In order to make it easier to write filters a base *FlirFilter* class has been provided, it is recommended that you derive your specific filter class from this base class in order to simplify the implementation

```
class FlirFilter {
public:
    FlirFilter(const std::string& filterName,
              ConfigCategory& filterConfig,
              OUTPUT_HANDLE *outHandle,
              OUTPUT_STREAM output);
    ~FlirFilter() {};
    const std::string&
        getName() const { return m_name; };
    bool
        isEnabled() const { return m_enabled; };
    ConfigCategory&
        getConfig() { return m_config; };
    void
        disableFilter() { m_enabled = false; };
    void
        setConfig(const std::string& newConfig);
public:
    OUTPUT_HANDLE*
        m_data;
    OUTPUT_STREAM
        m_func;
protected:
    std::string
        m_name;
    ConfigCategory
        m_config;
    bool
        m_enabled;
};

```

10.9.3 C++ Filter Example

The following example is a simple data processing example. It applies the `log()` function to numeric data in the data stream

Plugin Interface

Most plugins written in C++ have a source file that encapsulates the C API to the plugin, this is traditionally called `plugin.cpp`. The example plugin follows this model with the content of `plugin.cpp` shown below.

The first section includes the filter class that is the actual implementation of the filter logic and defines the JSON configuration category. This uses the *QUOTE* macro in order to make the JSON definition more readable.

```
/*
 * Flir "log" filter plugin.
 *
 * Copyright (c) 2020 Dianomic Systems
 *
 * Released under the Apache 2.0 Licence
 *
 * Author: Mark Riddoch
 */

#include <logFilter.h>
#include <version.h>

#define FILTER_NAME "log"
const static char *default_config = QUOTE({
    "plugin" : {
        "description" : "Log filter plugin",
        "type" : "string",
        "default" : FILTER_NAME,
        "readonly": "true"
    },
    "enable": {
        "description": "A switch that can be used to enable or
↳disable execution of the log filter.",
        "type": "boolean",
        "displayName": "Enabled",
        "default": "false"
    },
    "match" : {
        "description" : "An optional regular expression to match in
↳the asset name.",
        "type": "string",
        "default": "",
        "order": "1",
        "displayName": "Asset filter"}
});

using namespace std;
```

We then define the plugin information contents that will be returned by the *plugin_info* call.

```
/**
 * The Filter plugin interface
 */
extern "C" {

/**
 * The plugin information structure
 */
static PLUGIN_INFORMATION info = {
    FILTER_NAME,           // Name
    VERSION,               // Version
    0,                     // Flags
    PLUGIN_TYPE_FILTER,    // Type
    "1.0.0",               // Interface version
};
```

(continues on next page)

(continued from previous page)

```

    default_config          // Default plugin configuration
};

```

The final section of this file consists of the entry points themselves and the implementation. The majority of this consist of calls to the LogFilter class that in this case implements the logic of the filter.

```

/**
 * Return the information about this plugin
 */
PLUGIN_INFORMATION *plugin_info()
{
    return &info;
}

/**
 * Initialise the plugin, called to get the plugin handle.
 * We merely create an instance of our LogFilter class
 *
 * @param config      The configuration category for the filter
 * @param outHandle   A handle that will be passed to the output stream
 * @param output      The output stream (function pointer) to which data is passed
 * @return            An opaque handle that is used in all subsequent calls to the
 * ↪ plugin
 */
PLUGIN_HANDLE plugin_init(ConfigCategory* config,
                          OUTPUT_HANDLE *outHandle,
                          OUTPUT_STREAM output)
{
    LogFilter *log = new LogFilter(FILTER_NAME,
                                   *config,
                                   outHandle,
                                   output);

    return (PLUGIN_HANDLE) log;
}

/**
 * Ingest a set of readings into the plugin for processing
 *
 * @param handle      The plugin handle returned from plugin_init
 * @param readingSet  The readings to process
 */
void plugin_ingest(PLUGIN_HANDLE *handle,
                  READINGSET *readingSet)
{
    LogFilter *log = (LogFilter *) handle;
    log->ingest(readingSet);
}

/**
 * Plugin reconfiguration method
 *
 * @param handle      The plugin handle
 * @param newConfig   The updated configuration
 */
void plugin_reconfigure(PLUGIN_HANDLE *handle, const std::string& newConfig)

```

(continues on next page)

(continued from previous page)

```
{
    LogFilter *log = (LogFilter *)handle;
    log->reconfigure(newConfig);
}

/**
 * Call the shutdown method in the plugin
 */
void plugin_shutdown(PLUGIN_HANDLE *handle)
{
    LogFilter *log = (LogFilter *) handle;
    delete log;
}

// End of extern "C"
};
```

Filter Class

Although it is not mandatory it is good practice to encapsulate the filter logic in a class, these classes are derived from the FlirFilter class

```
#ifndef _LOG_FILTER_H
#define _LOG_FILTER_H
/*
 * Flir "Log" filter plugin.
 *
 * Copyright (c) 2020 Dianomic Systems
 *
 * Released under the Apache 2.0 Licence
 *
 * Author: Mark Riddoch
 */
#include <filter.h>
#include <reading_set.h>
#include <config_category.h>
#include <string>
#include <logger.h>
#include <mutex>
#include <regex>
#include <math.h>

/**
 * Convert the incoming data to use a logarithmic scale
 */
class LogFilter : public FlirFilter {
public:
    LogFilter(const std::string& filterName,
              ConfigCategory& filterConfig,
              OUTPUT_HANDLE *outHandle,
              OUTPUT_STREAM output);
    ~LogFilter();
    void ingest(READINGSET *readingSet);
    void reconfigure(const std::string& newConfig);
```

(continues on next page)

(continued from previous page)

```

private:
    void                handleConfig(ConfigCategory& config);
    std::string          m_match;
    std::regex           *m_regex;
    std::mutex           m_configMutex;
};

#endif

```

Filter Class Implementation

The following is the code that implements the filter logic

```

/*
 * Flir "Log" filter plugin.
 *
 * Copyright (c) 2020 Dianomic Systems
 *
 * Released under the Apache 2.0 Licence
 *
 * Author: Mark Riddoch
 */
#include <logFilter.h>

using namespace std;

/**
 * Constructor for the LogFilter.
 *
 * We call the constructor of the base class and handle the initial
 * configuration of the filter.
 *
 * @param   filterName      The name of the filter
 * @param   filterConfig    The configuration category for this filter
 * @param   outHandle       The handle of the next filter in the chain
 * @param   output          A function pointer to call to output data to the next_
 * filter
 */
LogFilter::LogFilter(const std::string& filterName,
                    ConfigCategory& filterConfig,
                    OUTPUT_HANDLE *outHandle,
                    OUTPUT_STREAM output) : m_regex(NULL),
                    FlirFilter(filterName, filterConfig, outHandle, _
 * output)
{
    handleConfig(filterConfig);
}

/**
 * Destructor for this filter class
 */
LogFilter::~LogFilter()
{
    if (m_regex)

```

(continues on next page)

(continued from previous page)

```

        delete m_regex;
    }

    /**
     * The actual filtering code
     *
     * @param readingSet The reading data to filter
     */
    void
    LogFilter::ingest(READINGSET *readingSet)
    {
        lock_guard<mutex> guard(m_configMutex);

        if (isEnabled()) // Filter enable, process the readings
        {
            const vector<Reading *>& readings = ((ReadingSet *)readingSet)->
↳getAllReadings();
            for (vector<Reading *>::const_iterator elem = readings.begin();
                elem != readings.end(); ++elem)
            {
                // If we set a matching regex then compare to the name of
↳this asset

                if (!m_match.empty())
                {
                    string asset = (*elem)->getAssetName();
                    if (!regex_match(asset, *m_regex))
                    {
                        continue;
                    }
                }

                // We are modifying this asset so put an entry in the asset
↳tracker

                AssetTracker::getAssetTracker()->
↳addAssetTrackingTuple(getName(), (*elem)->getAssetName(), string("Filter"));

                // Get a reading DataPoints
                const vector<Datapoint *>& dataPoints = (*elem)->
↳getReadingData();

                // Iterate over the datapoints
                for (vector<Datapoint *>::const_iterator it = dataPoints.
↳begin(); it != dataPoints.end(); ++it)
                {
                    // Get the reference to a DataPointValue
                    DatapointValue& value = (*it)->getData();

                    /*
                     * Deal with the T_INTEGER and T_FLOAT types.
                     * Try to preserve the type if possible but
                     * if a floating point log function is applied
                     * then T_INTEGER values will turn into T_FLOAT.
                     * If the value is zero we do not apply the log
↳function

                     */
                    if (value.getType() == DatapointValue::T_INTEGER)
                    {

```

(continues on next page)

(continued from previous page)

```

        long ival = value.toInt();
        if (ival != 0)
        {
            double newValue = log((double)ival);
            value.setValue(newValue);
        }
    }
    else if (value.getType() == DatapointValue::T_FLOAT)
    {
        double dval = value.toDouble();
        if (dval != 0.0)
        {
            value.setValue(log(dval));
        }
    }
    else
    {
        // do nothing for other types
    }
}

// Pass on all readings in this case
(*m_func)(m_data, readingSet);
}

/**
 * Reconfiguration entry point to the filter.
 *
 * This method runs holding the configMutex to prevent
 * ingest using the regex class that may be destroyed by this
 * call.
 *
 * Pass the configuration to the base FilterPlugin class and
 * then call the private method to handle the filter specific
 * configuration.
 *
 * @param newConfig The JSON of the new configuration
 */
void
LogFilter::reconfigure(const std::string& newConfig)
{
    lock_guard<mutex> guard(m_configMutex);
    setConfig(newConfig); // Pass the configuration to the base_
    ↪class
    handleConfig(m_config);
}

/**
 * Handle the filter specific configuration. In this case
 * it is just the single item "match" that is a regex
 * expression
 *
 * @param config The configuration category
 */
void
```

(continues on next page)

(continued from previous page)

```

LogFilter::handleConfig(ConfigCategory& config)
{
    if (config.itemExists("match"))
    {
        m_match = config.getValue("match");
        if (m_regex)
            delete m_regex;
        m_regex = new regex(m_match);
    }
}

```

10.9.4 Python Filter API

Filters may also be written in Python, the API is very similar to that of a C++ filter and consists of the same set of entry points.

Plugin Information

As with C++ filters this is the first entry point called, it returns a Python dictionary that describes the filter.

```

def plugin_info():
    """ Returns information about the plugin
    Args:
    Returns:
        dict: plugin information
    Raises:
    """

```

Plugin Initialisation

The *plugin_init* call is used to pass the resolved configuration to the plugin and also pass in the handle of the next filter in the pipeline and a callback that should be called with the output data of the filter.

```

def plugin_init(config, ingest_ref, callback):
    """ Initialise the plugin
    Args:
        config: JSON configuration document for the Filter plugin configuration_
↪category
        ingest_ref:
        callback:
    Returns:
        data: JSON object to be used in future calls to the plugin
    Raises:
    """

```

Plugin Ingestion

The *plugin_ingest* method is used to pass data into the plugin, the plugin will then process that data and call the callback that was passed into the *plugin_init* entry point with the *ingest_ref* handle and the data to send along the filter pipeline.

```
def plugin_ingest(handle, data):
    """ Modify readings data and pass it onward

    Args:
        handle: handle returned by the plugin initialisation call
        data: readings data
    """
```

The *data* is arranged as an array of Python dictionaries, each of which is a *Reading*. Typically the data can be processed by traversing the array

```
for elem in data:
    process(elem)
```

Plugin Reconfigure

The *plugin_reconfigure* entry point is called whenever a configuration change occurs for the filters configuration category.

```
def plugin_reconfigure(handle, new_config):
    """ Reconfigures the plugin

    Args:
        handle: handle returned by the plugin initialisation call
        new_config: JSON object representing the new configuration category for the
        ↪category
    Returns:
        new_handle: new handle to be used in the future calls
    """
```

Plugin Shutdown

Called when the plugin is to be shutdown to allow it to perform any cleanup operations.

```
def plugin_shutdown(handle):
    """ Shutdowns the plugin doing required cleanup.

    Args:
        handle: handle returned by the plugin initialisation call
    Returns:
        plugin shutdown
    """
```

10.9.5 Python Filter Example

The following is an example of a Python filter that calculates an exponential moving average.

```
# -*- coding: utf-8 -*-

# Flir_BEGIN
# See: http://flir.readthedocs.io/
# Flir_END
```

(continues on next page)

(continued from previous page)

```

""" Module for EMA filter plugin

Generate Exponential Moving Average
The rate value (x) allows to include x% of current value
and (100-x)% of history
A datapoint called 'ema' is added to each reading being filtered
"""

import time
import copy
import logging

from flir.common import logger
import filter_ingest

__author__ = "Massimiliano Pinto"
__copyright__ = "Copyright (c) 2020 Dianomic Systems"
__license__ = "Apache 2.0"
__version__ = "${VERSION}"

_LOGGER = logger.setup(__name__, level = logging.WARN)

# Filter specific objects
the_callback = None
the_ingest_ref = None

# latest ema value
latest = None
# rate value
rate = None
# datapoint name
datapoint = None
# plugin shutdown indicator
shutdown_in_progress = False

_DEFAULT_CONFIG = {
    'plugin': {
        'description': 'Exponential Moving Average filter plugin',
        'type': 'string',
        'default': 'ema',
        'readonly': 'true'
    },
    'enable': {
        'description': 'Enable ema plugin',
        'type': 'boolean',
        'default': 'false',
        'displayName': 'Enabled',
        'order': "3"
    },
    'rate': {
        'description': 'Rate value: include % of current value',
        'type': 'float',
        'default': '0.07',
        'displayName': 'Rate',
        'order': "2"
    },
}

```

(continues on next page)

(continued from previous page)

```

    'datapoint': {
        'description': 'Datapoint name for calculated ema value',
        'type': 'string',
        'default': 'ema',
        'displayName': 'EMA datapoint',
        'order': "1"
    }
}

def compute_ema(reading):
    """ Compute EMA

    Args:
        A reading data
    """
    global rate, latest, datapoint
    for attribute in list(reading):
        if not latest:
            latest = reading[attribute]
            latest = reading[attribute] * rate + latest * (1 - rate)
            reading[datapoint] = latest

def plugin_info():
    """ Returns information about the plugin

    Args:
    Returns:
        dict: plugin information
    Raises:
    """
    return {
        'name': 'ema',
        'version': '1.8.2',
        'mode': "none",
        'type': 'filter',
        'interface': '1.0',
        'config': _DEFAULT_CONFIG
    }

def plugin_init(config, ingest_ref, callback):
    """ Initialise the plugin

    Args:
        config: JSON configuration document for the Filter plugin configuration_
        ↪category ingest_ref:
        callback:

    Returns:
        data: JSON object to be used in future calls to the plugin
    Raises:
    """
    data = copy.deepcopy(config)

    global the_callback, the_ingest_ref, rate, datapoint

    the_callback = callback

```

(continues on next page)

(continued from previous page)

```

the_ingest_ref = ingest_ref
rate = float(config['rate']['value'])
datapoint = config['datapoint']['value']

_LOGGER.debug("plugin_init for filter EMA called")

return data

def plugin_reconfigure(handle, new_config):
    """ Reconfigures the plugin

    Args:
        handle: handle returned by the plugin initialisation call
        new_config: JSON object representing the new configuration category for the_
category
    Returns:
        new_handle: new handle to be used in the future calls
    """
    global rate, datapoint
    rate = float(new_config['rate']['value'])
    datapoint = new_config['datapoint']['value']
    _LOGGER.debug("Old config for ema plugin {} \n new config {}".format(handle, new_
category))
    new_handle = copy.deepcopy(new_config)

    return new_handle

def plugin_shutdown(handle):
    """ Shutdowns the plugin doing required cleanup.

    Args:
        handle: handle returned by the plugin initialisation call
    Returns:
        plugin shutdown
    """
    global shutdown_in_progress, the_callback, the_ingest_ref, rate, latest, datapoint
    shutdown_in_progress = True
    time.sleep(1)
    the_callback = None
    the_ingest_ref = None
    rate = None
    latest = None
    datapoint = None

    _LOGGER.info('filter ema plugin shutdown.')

def plugin_ingest(handle, data):
    """ Modify readings data and pass it onward

    Args:
        handle: handle returned by the plugin initialisation call
        data: readings data
    """
    global shutdown_in_progress, the_callback, the_ingest_ref

```

(continues on next page)

(continued from previous page)

```

if shutdown_in_progress:
    return

if handle['enable']['value'] == 'false':
    # Filter not enabled, just pass data onwards
    filter_ingest.filter_ingest_callback(the_callback, the_ingest_ref, data)
    return

# Filter is enabled: compute EMA for each reading
for elem in data:
    compute_ema(elem['readings'])

# Pass data onwards
filter_ingest.filter_ingest_callback(the_callback, the_ingest_ref, data)

_LOGGER.debug("ema filter_ingest done")

```

10.10 Notification Delivery Plugins

Notification delivery plugins are used by the notification system to send a notification to some other system or device. They are the transport that allows the event to be notified to that other system or device.

Notification delivery plugins may be written in C or C++ and have a very simple interface. The plugin mechanism and a subset of the API is common between all types of plugins including filters. This documentation is based on the . The sends MQTT messages to a configurable MQTT topic when a notification is triggered and cleared.

10.10.1 Configuration

Notification Delivery plugins use the same configuration mechanism as the rest of Flir, using a JSON document to describe the configuration parameters. As with any other plugin the structure is defined by the plugin and retrieved by the *plugin_info* entry point. This is then matched with the database content to pass the configured values to the *plugin_init* entry point.

10.10.2 Notification Delivery Plugin API

The notification delivery plugin API consists of a small number of C function entry points, these are called in a strict order and based on the same set of common API entry points for all Flir plugins.

Plugin Information

The *plugin_info* entry point is the first entry point that is called in a notification delivery plugin and returns the plugin information structure. This is the exact same call that every Flir plugin must support and is used to determine the type of the plugin and the configuration category defaults for the plugin.

A typical implementation of *plugin_info* would merely return a pointer to a static `PLUGIN_INFORMATION` structure.

```

PLUGIN_INFORMATION *plugin_info()
{
    return &info;
}

```

Plugin Initialise

The second call that is made to the plugin is the *plugin_init* call, that is used to retrieve a handle on the plugin instance and to configure the plugin.

```
PLUGIN_HANDLE plugin_init(ConfigCategory* config)
{
    MQTT *mqtt = new MQTT(config);
    return (PLUGIN_HANDLE)mqtt;
}
```

The *config* parameter is the configuration category with the user supplied values inserted, these values are used to configure the behavior of the plugin. In the case of our MQTT example we use this to call the constructor of our MQTT class.

```
/**
 * Construct a MQTT notification plugin
 *
 * @param category The configuration of the plugin
 */
MQTT::MQTT(ConfigCategory *category)
{
    if (category->itemExists("broker"))
        m_broker = category->getValue("broker");
    if (category->itemExists("topic"))
        m_topic = category->getValue("topic");
    if (category->itemExists("trigger_payload"))
        m_trigger = category->getValue("trigger_payload");
    if (category->itemExists("clear_payload"))
        m_clear = category->getValue("clear_payload");
}
```

This constructor merely stores values out of the configuration category as private member variables of the MQTT class.

We return the pointer to our MQTT class as the handle for the plugin. This allows subsequent calls to the plugin to reference the instance created by the *plugin_init* call.

Plugin Delivery

This is the API call made whenever the plugin needs to send a triggered or cleared notification state. It may be called multiple times within the lifetime of a plugin.

```
bool plugin_deliver(PLUGIN_HANDLE handle,
                    const std::string& deliveryName,
                    const std::string& notificationName,
                    const std::string& triggerReason,
                    const std::string& message)
{
    MQTT *mqtt = (MQTT *)handle;
    return mqtt->notify(notificationName, triggerReason, message);
}
```

The delivery call is passed the handle, which gives us the MQTT class instance on this case, the name of the notification, a trigger reason, which is a JSON document and a message. The trigger reason JSON document contains information about why the delivery call was made, including the triggered or cleared status, the timestamp of the

reading that caused the notification to trigger and the name of the asset or assets involved in the notification rule that triggered this delivery event.

```
{
  "reason": "triggered",
  "asset": ["sinusoid"],
  "timestamp": "2020-11-18 11:52:33.960530+00:00"
}
```

The return from the *plugin_deliver* entry point is a boolean that indicates if the delivery succeeded or not.

In the case of our MQTT example we call the notify method of the class, this then interacts with the MQTT broker.

```
/**
 * Send a notification via MQTT broker
 *
 * @param notificationName The name of this notification
 * @param triggerReason    Why the notification is being sent
 * @param message          The message to send
 */
bool MQTT::notify(const string& notificationName, const string& triggerReason, const_
↳ string& message)
{
    string          payload = m_trigger;
    MQTTClient      client;

    lock_guard<mutex> guard(m_mutex);

    // Parse the JSON that represents the reason data
    Document doc;
    doc.Parse(triggerReason.c_str());
    if (!doc.HasParseError() && doc.HasMember("reason"))
    {
        if (!strcmp(doc["reason"].GetString(), "cleared"))
            payload = m_clear;
    }

    // Connect to the MQTT broker
    MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
    MQTTClient_message pubmsg = MQTTClient_message_initializer;
    MQTTClient_deliveryToken token;
    int rc;

    if ((rc = MQTTClient_create(&client, m_broker.c_str(), CLIENTID,
MQTTCLIENT_PERSISTENCE_NONE, NULL)) != MQTTCLIENT_SUCCESS)
    {
        Logger::getLogger()->error("Failed to create client, return code %d\n
↳ ", rc);
        return false;
    }

    conn_opts.keepAliveInterval = 20;
    conn_opts.cleansession = 1;
    if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
    {
        Logger::getLogger()->error("Failed to connect, return code %d\n", rc);
        return false;
    }
}
```

(continues on next page)

(continued from previous page)

```

    // Construct the payload
    pubmsg.payload = (void *)payload.c_str();
    pubmsg.payloadlen = payload.length();
    pubmsg.qos = 1;
    pubmsg.retained = 0;

    // Publish the message
    if ((rc = MQTTClient_publishMessage(client, m_topic.c_str(), &pubmsg, &
    token)) != MQTTCLIENT_SUCCESS)
    {
        Logger::getLogger()->error("Failed to publish message, return code
    %d\n", rc);
        return false;
    }

    // Wait for completion and disconnect
    rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
    if ((rc = MQTTClient_disconnect(client, 10000)) != MQTTCLIENT_SUCCESS)
        Logger::getLogger()->error("Failed to disconnect, return code %d\n",
    rc);
    MQTTClient_destroy(&client);
    return true;
}

```

Plugin Reconfigure

As with other plugin types the notification delivery plugin may be reconfigured during its lifetime. When a reconfiguration operation occurs the *plugin_reconfigure* method will be called with the new configuration for the plugin.

```

void plugin_reconfigure(PPLUGIN_HANDLE *handle, const std::string& newConfig)
{
    MQTT *mqtt = (MQTT *)handle;
    mqtt->reconfigure(newConfig);
    return;
}

```

In the case of our MQTT example we call the reconfigure method of our MQTT class. In this method the new values are copied into the local member variables of the instance.

```

/**
 * Reconfigure the MQTT delivery plugin
 *
 * @param newConfig The new configuration
 */
void MQTT::reconfigure(const string& newConfig)
{
    ConfigCategory category("new", newConfig);
    lock_guard<mutex> guard(m_mutex);
    m_broker = category.getValue("broker");
    m_topic = category.getValue("topic");
    m_trigger = category.getValue("trigger_payload");
    m_clear = category.getValue("clear_payload");
}

```

The mutex is used here to prevent the plugin reconfiguration occurring when we are delivering a notification. The same mutex is held in the notify method of the MQTT class.

Plugin Shutdown

As with other plugins a shutdown call exists which may be used by the plugin to perform any cleanup that is required when the plugin is shut down.

```
void plugin_shutdown(PLUGIN_HANDLE *handle)
{
    MQTT *mqtt = (MQTT *)handle;
    delete mqtt;
}
```

In the case of our MQTT example we merely destroy the instance of the MQTT class and allow the destructor of that class to do any cleanup that is required. In the case of this example there is no cleanup required.

10.11 Testing Your Plugin

The first step in testing your new plugin is to put the plugin in the location in which your Flir system will be loading it from. The exact location depends on the way you installed your Flir system and the type of plugin.

If your Flir system was installed from a package and you used the default installation path, then your plugin must be stored under the directory `/usr/local/flir`. If you installed Flir in a nonstandard location or you have built it from the source code, then the plugin should be stored under the directory `$FLIR_ROOT`.

A C/C++ plugin or a hybrid plugin should be placed in the directory `plugins/<type>/<plugin name>` under the installed directory described above. Where `<type>` is one of `south`, `filter`, `north`, `notificationRule` or `notificationDelivery`. And `<plugin name>` is the name you gave your plugin.

A south plugin written in C/C++ and called DHT11, for a system installed from a package, would be installed in a directory called `/usr/local/flir/plugins/south/DHT11`. Within that directory Flir would expect to find a file called `libDHT11.so`.

A south hybrid plugin called MD1421, for a development system built from source would be installed in `$(FLIR_ROOT)/plugins/south/MD1421`. In this directory a JSON file called `MD1421.json` should exist, this is what the system will read to create the plugin.

A Python plugin should be installed in the directory `python/flir/plugins/<plugin type>/<plugin name>` under the installed directory described above. Where `<type>` is one of `south`, `filter`, `north`, `notificationRule` or `notificationDelivery`. And `<plugin name>` is the name you gave your plugin.

A Python filter plugin call normalise, on a system installed from a package in the default location should be copied into a directory `/usr/local/flir/python/flir/plugins/filter/normalise`. Within this directory should be a file called `normalise.py` and an empty file called `__init__.py`.

10.11.1 Initial Testing

After you have copied your plugin into the correct location you can test if Flir is able to see it by running the API call `/flir/plugins/installed`. This will list all the installed plugins and their versions.

```
$ curl http://localhost:8081/flir/plugins/installed | jq
{
  "plugins": [
```

(continues on next page)

(continued from previous page)

```

{
  "name": "pi_server",
  "type": "north",
  "description": "PI Server North Plugin",
  "version": "1.0.0",
  "installedDirectory": "north/pi_server",
  "packageName": ""
},
{
  "name": "ocs",
  "type": "north",
  "description": "OCS (OSIsoft Cloud Services) North Plugin",
  "version": "1.0.0",
  "installedDirectory": "north/ocs",
  "packageName": ""
},
{
  "name": "http_north",
  "type": "north",
  "description": "HTTP North Plugin",
  "version": "1.8.1",
  "installedDirectory": "north/http_north",
  "packageName": "flir-north-http-north"
},
{
  "name": "GCP",
  "type": "north",
  "description": "Google Cloud Platform IoT-Core",
  "version": "1.8.1",
  "installedDirectory": "north/GCP",
  "packageName": "flir-north-gcp"
},
...
}

```

Note, in the above example the *jq* program has been used to format the returned JSON and the output has been truncated for brevity.

If your plugin does not appear it may be because there was a problem loading it or because the *plugin_info* call returned a bad value. Examine the syslog file to see if there are any errors recorded during the above API call.

10.11.2 C/C++ Common Faults

Common faults for C/C++ plugins are that a symbol could not be resolved when the plugin was loaded or the JSON for the default configuration is malformed.

There is a utility called *get_plugin_info* that is used by Python code to call the C *plugin_info* call, this can be used to ascertain the cause of some problems. It should return the default configuration of your plugin and will verify that your plugin has no undefined symbols.

The location of *get_plugin_info* will depend on the type of installation you have. If you have built from source then it can be found in *./cmake_build/C/plugins/utls/get_plugin_info*. If you have installed a package, or run *make install*, you can find it in */usr/local/flir/extras/C/get_plugin_info*.

The utility is passed the library file of your plugin as its first argument and the function to call, usually *plugin_info*.

```
$ get_plugin_info plugins/north/GCP/libGCP.so plugin_info
{"name": "GCP", "version": "1.8.1", "type": "north", "interface": "1.0.0", "flag": 0,
  "config": { "plugin": { "description": "Google Cloud Platform IoT-Core", "type":
    "string", "default": "GCP", "readonly": "true" }, "project_id": { "description":
    "The GCP IoT Core Project ID", "type": "string", "default": "", "order": "1",
    "displayName": "Project ID" }, "region": { "description": "The GCP Region", "type":
    "enumeration", "options": [ "us-centrall", "europe-west1", "asia-east1" ],
    "default": "us-centrall", "order": "2", "displayName": "The GCP Region" },
    "registry_id": { "description": "The Registry ID of the GCP Project", "type":
    "string", "default": "", "order": "3", "displayName": "Registry ID" }, "device_id":
    { "description": "Device ID within GCP IoT Core", "type": "string", "default":
    "", "order": "4", "displayName": "Device ID" }, "key": { "description": "Name
    of the key file to use", "type": "string", "default": "", "order": "5",
    "displayName": "Key Name" }, "algorithm": { "description": "JWT algorithm", "type":
    "enumeration", "options": [ "ES256", "RS256" ], "default": "RS256", "order":
    "6", "displayName": "JWT Algorithm" }, "source": { "description": "The source of
    data to send", "type": "enumeration", "default": "readings", "order": "8",
    "displayName": "Data Source", "options": [ "readings", "statistics" ] } } }
```

If there is an undefined symbol you will get an error from this utility. You can also check the validity of your JSON configuration by piping the output to a program such as jq.

10.11.3 Running Under a Debugger

If you have a C/C++ plugin that crashes you may want to run the plugin under a debugger. To build with debug symbols use the CMake option `-DCMAKE_BUILD_TYPE=Debug` when you create the *Makefile*.

Running a Service Under the Debugger

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

The easiest approach to run under a debugger is

- Create the service that uses your plugin, say a south service and name that service as you normally would.
- Disable that service from being started by Flir
- Use the flir status script to find the arguments to pass the service

```
$ scripts/flir status
Flir v1.8.2 running.
Flir Uptime: 1451 seconds.
Flir records: 200889 read, 200740 sent, 120962 purged.
Flir does not require authentication.
=== Flir services:
flir.services.core
flir.services.storage --address=0.0.0.0 --port=39821
flir.services.south --port=39821 --address=127.0.0.1 --name=AX8
flir.services.south --port=39821 --address=127.0.0.1 --name=Sine
=== Flir tasks:
```

- Note the `--port=` and `--address=` arguments
- Set your `LD_LIBRARY_PATH`. This is normally done in the script that launches Flir but will need to be run as a manual step when running under the debugger.

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/flir/lib
```

If you built from source rather than installing a package you will need to include the libraries you built

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${FLIR_ROOT}/cmake_build/C/lib
```

- Load the service you wish to use to run your plugin, e.g a south service, under the debugger

```
$ gdb services/flir.services.south
```

- Run the service passing the `-port=` and `-address=` arguments you noted above and add `-d` and `-name=` with the name of your service.

```
(gdb) run --port=39821 --address=127.0.0.1 --name=ServiceName -d
```

Where *ServiceName* is the name you gave your service

- You can now use the debugger in the way you normally would to find any issues.

Running a Task Under the Debugger

Running a task under the debugger is much the same as running a service, you will first need to find the management port and address of the core management service. Create the task, e.g. a north sending process in the same way as you normally would and disable it. You will also need to set your `LD_LIBRARY_PATH` as with running a service under the debugger.

If you are using a plugin with a task, such as the north sending process task, then the command to use to start the debugger is

```
$ gdb tasks/sending_process
```

Running the Storage Service Under the Debugger

Running the storage service under the debugger is more difficult as you can not start the storage service after Flir has started, the startup of the storage service is coordinated by the core due to the nature of how configuration is stored. It is possible however to attach a debugger to a running storage service.

- Run a command to find the process ID of the storage service

```
$ ps aux | grep flir.services.storage
flir 23318 0.0 0.3 270848 12388 ?        Ssl  10:00   0:01 /usr/local/
↪flir/services/flir.services.storage --address=0.0.0.0 --port=33761
flir 31033 0.0 0.0 13136 1084 pts/1    S+   10:37   0:00 grep --
↪color=auto flir.services.storage
```

- Use the process ID of the flir service as an argument to gdb. Note you will need to run gdb as root on some systems

```
$ sudo gdb /usr/local/flir/services/flir.services.storage 23318
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
↪gpl.html>
This is free software: you are free to change and redistribute it.
```

(continues on next page)

(continued from previous page)

```

There is NO WARRANTY, to the extent permitted by law.  Type "show_
↳copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from services/flir.services.storage...done.
Attaching to program: /usr/local/flir/services/flir.services.storage, _
↳process 23318
[New LWP 23320]
[New LWP 23321]
[New LWP 23322]
[New LWP 23330]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.
↳so.1".
0x00007f47a3e05d2d in __GI___pthread_timedjoin_ex_
↳(threadid=139945627997952, thread_return=0x0, abstime=0x0,
   block=<optimized out>) at pthread_join_common.c:89
89  pthread_join_common.c: No such file or directory.
(gdb)

```

- You can now use gdb to set break points etc and debug the storage service and plugins.

If you are debugging a plugin that crashes the system when readings are processed you should disable the south services until you have connected the debugger to the storage system. If you have a system that is setup and crashes, use the `--safe-mode` flag to the startup of Flir in order to disable all processes and services. This will allow you to disable services or to run a particular service manually.

10.11.4 Using strace

You can also use a similar approach to that of running gdb to use the *strace* command to trace system calls and signals

- Create the service that uses your plugin, say a south service and name that service as you normally would.
- Disable that service from being started by Flir
- Use the flir status script to find the arguments to pass the service

```

$ scripts/flir status
Flir v1.8.2 running.
Flir Uptime: 1451 seconds.
Flir records: 200889 read, 200740 sent, 120962 purged.
Flir does not require authentication.
=== Flir services:
flir.services.core
flir.services.storage --address=0.0.0.0 --port=39821
flir.services.south --port=39821 --address=127.0.0.1 --name=AX8
flir.services.south --port=39821 --address=127.0.0.1 --name=Sine
=== Flir tasks:

```

- Note the `--port=` and `--address=` arguments
- Run `strace` with the service adding the same set of arguments you used in `gdb` when running the service

```
$ strace services/flir.services.south --port=39821 --address=127.0.0.1 --
↪name=ServiceName -d
```

Where *ServiceName* is the name you gave your service

10.11.5 Memory Leaks and Corruptions

The same approach can be used to make use of the `valgrind` command to find memory corruption and leak issues in your plugin

- Create the service that uses your plugin, say a south service and name that service as you normally would.
- Disable that service from being started by Flir
- Use the flir status script to find the arguments to pass the service

```
$ scripts/flir status
Flir v1.8.2 running.
Flir Uptime: 1451 seconds.
Flir records: 200889 read, 200740 sent, 120962 purged.
Flir does not require authentication.
=== Flir services:
flir.services.core
flir.services.storage --address=0.0.0.0 --port=39821
flir.services.south --port=39821 --address=127.0.0.1 --name=AX8
flir.services.south --port=39821 --address=127.0.0.1 --name=Sine
=== Flir tasks:
```

- Note the `--port=` and `--address=` arguments
- Run `strace` with the service adding the same set of arguments you used in `gdb` when running the service

```
$ valgrind --leak-check=full services/flir.services.south --port=39821 -
↪--address=127.0.0.1 --name=ServiceName -d
```

Where *ServiceName* is the name you gave your service

10.11.6 Python Plugin Info

It is also possible to test the loading and validity of the `plugin_info` call in a Python plugin.

- From the `/usr/include/flir` or `${FLIR_ROOT}` directory run the command

```
python3 -c 'from flir.plugins.south.<name>.<name> import plugin_info;
↪print(plugin_info())'
```

Where `<name>` is the name of your plugin.


```
python3 -c 'from flir.plugins.south.sinusoid.sinusoid import plugin_info;
↳ print(plugin_info())'
{'name': 'Sinusoid Poll plugin', 'version': '1.8.1', 'mode': 'poll', 'type':
↳ 'south', 'interface': '1.0', 'config': {'plugin': {'description': 'Sinusoid
↳ Poll Plugin which implements sine wave with data points', 'type': 'string',
↳ 'default': 'sinusoid', 'readonly': 'true'}}, 'assetName': {'description': 'Name
↳ of Asset', 'type': 'string', 'default': 'sinusoid', 'displayName': 'Asset name',
↳ 'mandatory': 'true'}}
```

This allows you to confirm the plugin can be loaded and the *plugin_info* entry point can be called.

You can also check your default configuration. Although in Python this is usually harder to get wrong.

```
$ python3 -c 'from flir.plugins.south.sinusoid.sinusoid import plugin_info;
↳ print(plugin_info()["config"])'
{'plugin': {'description': 'Sinusoid Poll Plugin which implements sine wave with data
↳ points', 'type': 'string', 'default': 'sinusoid', 'readonly': 'true'}, 'assetName':
↳ {'description': 'Name of Asset', 'type': 'string', 'default': 'sinusoid',
↳ 'displayName': 'Asset name', 'mandatory': 'true'}}
```


11.1 The Flir REST API

Users, administrators and applications interact with Flir via a REST API. This section presents a full reference of the API.

Note: The Flir REST API should not be confused with the internal REST API used by Flir tasks and microservices to communicate with each other.

11.1.1 Introducing the Flir REST API

The REST API is the route into the Flir appliance, it provides all user and program interaction to configure, monitor and manage the Flir system. A separate specification will define the contents of the API, in summary however it is designed to allow for:

- The complete configuration of the Flir appliance
- Access to monitoring statistics for the Flir appliance
- User and role management for access to the API
- Access to the data buffer contents

Port Usage

In general Flir components use dynamic port allocation to determine which port to use, the admin API is however an exception to this rule. The Admin API port has to be known to end-users and any user interface or management system that uses it, therefore the port on which the admin API listens must be consistent and fixed between invocations. This does not mean however that it can not be changed by the user. The user must have the option to define the port to use by the admin API to listen on. To achieve this the port will be stored in the configuration data for the admin API, using the configuration category *AdminAPI*, see Configuration. Administrators who have access to the appliance

can find information regarding the port and the protocol to used (i.e. HTTP or HTTPS) in the *pid* file stored in *\$FLIR_DATA/var/run/*:

```
$ cat data/var/run/flir.core.pid
{ "adminAPI" : { "protocol" : "HTTP",
                  "port"      : 8081,
                  "addresses" : [ "0.0.0.0" ] },
  "processID" : 3585 }
```

Flir is shipped with a default port for the admin API to use, however the user is free to change this after installation. This can be done by first connecting to the port defined as the default and then modifying the port using the admin API. Flir should then be restarted to make use of this new port.

Infrastructure

There are two REST API's that allow external access to Flir, the **Administration API** and the **User API**. The User API is intended to allow access to the data in the Flir storage layer which buffers sensor readings, and it is not part of this current version.

The Administration API is the first API is concerned with all aspects of managing and monitoring the Flir appliance. This API is used for all configuration operations that occur beyond basic installation.

11.2 Administration API Reference

This section presents the list of administrative API methods in alphabetical order.

11.2.1 Audit Trail

The audit trail API is used to interact with the audit trail log tables in the storage microservice. In Flir, log information is stored in the system log where the microservice is hosted. All the relevant information used for auditing are instead stored inside Flir and they are accessible through the Admin REST API. The API allows the reading but also the addition of extra audit logs, as if such logs are created within the system.

audit

The *audit* methods implement the audit trail, they are used to create and retrieve audit logs.

GET Audit Entries

GET */flir/audit* - return a list of audit trail entries sorted with most recent first.

Request Parameters

- **limit** - limit the number of audit entries returned to the number specified
- **skip** - skip the first n entries in the audit table, used with limit to implement paged interfaces
- **source** - filter the audit entries to be only those from the specified source
- **severity** - filter the audit entries to only those of the specified severity

Response Payload

The response payload is an array of JSON objects with the audit trail entries.

Name	Type	Description	Example
timestamp	timestamp	The timestamp when the audit trail item was written.	2018-04-16 14:33:18.215
source	string	The source of the audit trail entry.	CoAP
severity	string	The severity of the event that triggered the audit trail entry to be written. This will be one of SUCCESS, FAILURE, WARNING or INFORMATION.	FAILURE
details	object	A JSON object that describes the detail of the audit trail event.	{ "message": "Sensor readings discarded due to malformed payload" }

Example

```
$ curl -s http://localhost:8081/flir/audit?limit=2
{ "totalCount" : 24,
  "audit"      : [ { "timestamp" : "2018-02-25 18:58:07.748",
                    "source"     : "SRVRG",
                    "details"    : { "name" : "COAP" },
                    "severity"   : "INFORMATION" },
                  { "timestamp" : "2018-02-25 18:58:07.742",
                    "source"     : "SRVRG",
                    "details"    : { "name" : "HTTP_SOUTH" },
                    "severity"   : "INFORMATION" },
                  { "timestamp" : "2018-02-25 18:58:07.390",
                    "source"     : "START",
                    "details"    : {},
                    "severity"   : "INFORMATION" }
                ]
}
$ curl -s http://localhost:8081/flir/audit?source=SRVUN&limit=1
{ "totalCount" : 4,
  "audit"      : [ { "timestamp" : "2018-02-25 05:22:11.053",
                    "source"     : "SRVUN",
                    "details"    : { "name": "COAP" },
                    "severity"   : "INFORMATION" }
                ]
}
$
```

POST Audit Entries

POST /flir/audit - create a new audit trail entry.

The purpose of the create method on an audit trail entry is to allow a user interface or an application that is using the Flir API to utilize the Flir audit trail and notification mechanism to raise user defined audit trail entries.

Request Payload

The request payload is a JSON object with the audit trail entry minus the timestamp.

Name	Type	Description	Example
source	string	The source of the audit trail entry.	LOGGN
severity	string	The severity of the event that triggered the audit trail entry to be written. This will be one of SUCCESS, FAILURE, WARNING or INFORMATION.	FAILURE
details	object	A JSON object that describes the detail of the audit trail event.	{ "message" : "Internal System Error" }

Response Payload

The response payload is the newly created audit trail entry.

Name	Type	Description	Example
timestamp	timestamp	The timestamp when the audit trail item was written.	2018-04-16 14:33:18.215
source	string	The source of the audit trail entry.	LOGGN
severity	string	The severity of the event that triggered the audit trail entry to be written. This will be one of SUCCESS, FAILURE, WARNING or INFORMATION.	FAILURE
details	object	A JSON object that describes the detail of the audit trail event.	{ "message" : "Internal System Error" }

Example

```
$ curl -X POST http://localhost:8081/flir/audit \
-d '{ "severity": "FAILURE", "details": { "message": "Internal System Error" },
  ↪ "source": "LOGGN" }'
{ "source": "LOGGN",
  "timestamp": "2018-04-17 11:49:55.480",
  "severity": "FAILURE",
  "details": { "message": "Internal System Error" }
}
$
$ curl -X GET http://localhost:8081/flir/audit?severity=FAILURE
{ "totalCount": 1,
  "audit": [ { "timestamp": "2018-04-16 18:32:28.427",
               "source" : "LOGGN",
               "details" : { "message": "Internal System Error" },
               "severity" : "FAILURE" }
            ]
}
$
```

11.2.2 Configuration Management

Configuration management is an important aspect of the REST API, however due to the discoverable form of the configuration of Flir the API itself is fairly small.

The configuration REST API interacts with the configuration manager to create, retrieve, update and delete the configuration categories and values. Specifically all updates must go via the management layer as this is used to trigger the notifications to the components that have registered interest in configuration categories. This is the means by which the dynamic reconfiguration of Flir is achieved.

category

The *category* interface is part of the Configuration Management for Flir and it is used to create, retrieve, update and delete configuration categories and items.

GET categor(ies)

GET /flir/category - return the list of known categories in the configuration database

Response Payload

The response payload is a JSON object with an array of JSON objects, one per valid category.

Name	Type	Description	Example
key	string	The category key, each category has a unique textual key that defines it.	network
description	string	A description of the category that may be used for display purposes.	Network Settings
display-Name	string	Name of the category that may be used for display purposes.	Network Settings

Example

```
$ curl -X GET http://localhost:8081/flir/category
{
  "categories":
  [
    {
      "key": "SCHEDULER",
      "description": "Scheduler configuration",
      "displayName": "Scheduler"
    },
    {
      "key": "SMNTR",
      "description": "Service Monitor",
      "displayName": "Service Monitor"
    },
    {
      "key": "rest_api",
      "description": "Flir Admin and User REST API",
      "displayName": "Admin API"
    },
    {
      "key": "service",
      "description": "Flir Service",
      "displayName": "Flir Service"
    },
    {
      "key": "Installation",
      "description": "Installation",
      "displayName": "Installation"
    },
    {
      "key": "General",
      "description": "General",

```

(continues on next page)

(continued from previous page)

```

    "displayName": "General"
  },
  {
    "key": "Advanced",
    "description": "Advanced",
    "displayName": "Advanced"
  },
  {
    "key": "Utilities",
    "description": "Utilities",
    "displayName": "Utilities"
  }
]
}
$

```

GET category

GET /flir/category/{name} - return the configuration items in the given category.

Path Parameters

- **name** is the name of one of the categories returned from the GET /flir/category call.

Response Payload

The response payload is a set of configuration items within the category, each item is a JSON object with the following set of properties.

Name	Type	Description	Example
description	string	A description of the configuration item that may be used in a user interface.	The IPv4 network address of the Flir server
type	string	A type that may be used by a user interface to know how to display an item.	IPv4
default	string	An optional default value for the configuration item.	127.0.0.1
displayName	string	Name of the category that may be used for display purposes.	IPv4 address
order	integer	Order at which category name will be displayed.	1
value	string	The current configured value of the configuration item. This may be empty if no value has been set.	192.168.0.27

Example

```

$ curl -X GET http://localhost:8081/flir/category/rest_api
{
  "enableHttp": {
    "description": "Enable HTTP (disable to use HTTPS)",
    "type": "boolean",
    "default": "true",
    "displayName": "Enable HTTP",
    "order": "1",

```

(continues on next page)

(continued from previous page)

```

        "value": "true"
    },
    "httpPort": {
        "description": "Port to accept HTTP connections on",
        "type": "integer",
        "default": "8081",
        "displayName": "HTTP Port",
        "order": "2",
        "value": "8081"
    },
    "httpsPort": {
        "description": "Port to accept HTTPS connections on",
        "type": "integer",
        "default": "1995",
        "displayName": "HTTPS Port",
        "order": "3",
        "validity": "enableHttp==\"false\"",
        "value": "1995"
    },
    "certificateName": {
        "description": "Certificate file name",
        "type": "string",
        "default": "flir",
        "displayName": "Certificate Name",
        "order": "4",
        "validity": "enableHttp==\"false\"",
        "value": "flir"
    },
    "authentication": {
        "description": "API Call Authentication",
        "type": "enumeration",
        "options": [
            "mandatory",
            "optional"
        ],
        "default": "optional",
        "displayName": "Authentication",
        "order": "5",
        "value": "optional"
    },
    "authMethod": {
        "description": "Authentication method",
        "type": "enumeration",
        "options": [
            "any",
            "password",
            "certificate"
        ],
        "default": "any",
        "displayName": "Authentication method",
        "order": "6",
        "value": "any"
    },
    "authCertificateName": {
        "description": "Auth Certificate name",
        "type": "string",
        "default": "ca",

```

(continues on next page)

(continued from previous page)

```

    "displayName": "Auth Certificate",
    "order": "7",
    "value": "ca"
  },
  "allowPing": {
    "description": "Allow access to ping, regardless of the authentication required_
↪and authentication header",
    "type": "boolean",
    "default": "true",
    "displayName": "Allow Ping",
    "order": "8",
    "value": "true"
  },
  "passwordChange": {
    "description": "Number of days after which passwords must be changed",
    "type": "integer",
    "default": "0",
    "displayName": "Password Expiry Days",
    "order": "9",
    "value": "0"
  },
  "authProviders": {
    "description": "Authentication providers to use for the interface (JSON array_
↪object)",
    "type": "JSON",
    "default": "{\"providers\": [\"username\", \"ldap\"] }",
    "displayName": "Auth Providers",
    "order": "10",
    "value": "{\"providers\": [\"username\", \"ldap\"] }"
  }
}
$

```

GET category item

GET /flir/category/{name}/{item} - return the configuration item in the given category.

Path Parameters

- **name** - the name of one of the categories returned from the GET /flir/category call.
- **item** - the item within the category to return.

Response Payload

The response payload is a configuration item within the category, each item is a JSON object with the following set of properties.

Name	Type	Description	Example
description	string	A description of the configuration item that may be used in a user interface.	The IPv4 network address of the Flir server
type	string	A type that may be used by a user interface to know how to display an item.	IPv4
default	string	An optional default value for the configuration item.	127.0.0.1
displayName	string	Name of the category that may be used for display purposes.	IPv4 address
order	integer	Order at which category name will be displayed.	1
value	string	The current configured value of the configuration item. This may be empty if no value has been set.	192.168.0.27

Example

```
$ curl -X GET http://localhost:8081/flir/category/rest_api/httpsPort
{
  "description": "Port to accept HTTPS connections on",
  "type": "integer",
  "default": "1995",
  "displayName": "HTTPS Port",
  "order": "3",
  "validity": "enableHttp==\"false\"",
  "value": "1995"
}
$
```

PUT category item

PUT /flir/category/{name}/{item} - set the configuration item value in the given category.

Path Parameters

- **name** - the name of one of the categories returned from the GET /flir/category call.
- **item** - the item within the category to set.

Request Payload

A JSON object with the new value to assign to the configuration item.

Name	Type	Description	Example
value	string	The new value of the configuration item.	192.168.0.27

Response Payload

The response payload is the newly updated configuration item within the category, the item is a JSON object with the following set of properties.

Name	Type	Description	Example
description	string	A description of the configuration item that may be used in a user interface.	The IPv4 network address of the Flir server
type	string	A type that may be used by a user interface to know how to display an item.	IPv4
default	string	An optional default value for the configuration item.	127.0.0.1
displayName	string	Name of the category that may be used for display purposes.	IPv4 address
order	integer	Order at which category name will be displayed.	1
value	string	The current configured value of the configuration item. This may be empty if no value has been set.	192.168.0.27

Example

```
$ curl -X PUT http://localhost:8081/flir/category/rest_api/httpsPort \
-d '{ "value" : "1996" }'
{
  "description": "Port to accept HTTPS connections on",
  "type": "integer",
  "default": "1995",
  "displayName": "HTTPS Port",
  "order": "3",
  "validity": "enableHttp=="false"",
  "value": "1996"
}
$
```

DELETE category item

DELETE /flir/category/{name}/{item}/value - unset the value of the configuration item in the given category.

This will result in the value being returned to the default value if one is defined. If not the value will be blank, i.e. the value property of the JSON object will exist with an empty value.

Path Parameters

- **name** - the name of one of the categories returned from the GET /flir/category call.
- **item** - the the item within the category to return.

Response Payload

The response payload is the newly updated configuration item within the category, the item is a JSON object object with the following set of properties.

Name	Type	Description	Example
description	string	A description of the configuration item that may be used in a user interface.	The IPv4 network address of the Flir server
type	string	A type that may be used by a user interface to know how to display an item.	IPv4
default	string	An optional default value for the configuration item.	127.0.0.1
displayName	string	Name of the category that may be used for display purposes.	IPv4 address
order	integer	Order at which category name will be displayed.	1
value	string	The current configured value of the configuration item. This may be empty if no value has been set.	127.0.0.1

Example

```
$ curl -X DELETE http://localhost:8081/flir/category/rest_api/httpsPort/value
{
  "description": "Port to accept HTTPS connections on",
  "type": "integer",
  "default": "1995",
  "displayName": "HTTPS Port",
  "order": "3",
  "validity": "enableHttp==\"false\"",
  "value": "1995"
}
$
```

POST category

POST /flir/category - creates a new category

Request Payload

A JSON object that defines the category.

Name	Type	Description	Example
key	string	The key that identifies the category. If the key already exists as a category then the contents of this request is merged with the data stored.	backup
description	string	A description of the configuration category	Backup configuration
items	list	An optional list of items to create in this category	
name	string	The name of a configuration item	destination
description	string	A description of the configuration item	The destination to which the backup will be written
type	string	The type of the configuration item	string
default	string	An optional default value for the configuration item	/backup

NOTE: with list we mean a list of JSON objects in the form of { obj1,obj2,etc. }, to differ from the concept of *array*,

i.e. [obj1,obj2,etc.]

Example

```
$ curl -X POST http://localhost:8081/flir/category
-d '{ "key": "My Configuration", "description": "This is my new configuration",
      "value": { "item one": { "description": "The first item", "type": "string",
↪ "default": "one" },
                  "item two": { "description": "The second item", "type": "string",
↪ "default": "two" },
                  "item three": { "description": "The third item", "type": "string",
↪ "default": "three" } } }'
{ "description": "This is my new configuration", "key": "My Configuration", "value": {
  "item one": { "default": "one", "type": "string", "description": "The first item
↪ ", "value": "one" },
  "item two": { "default": "two", "type": "string", "description": "The second
↪ item", "value": "two" },
  "item three": { "default": "three", "type": "string", "description": "The third
↪ item", "value": "three" } }
}
```

11.2.3 Task Management

The task management API's allow an administrative user to monitor and control the tasks that are started by the task scheduler either from a schedule or as a result of an API request.

task

The *task* interface allows an administrative user to monitor and control Flir tasks.

GET task

GET /flir/task - return the list of all known task running or completed

Request Parameters

- **name** - an optional task name to filter on, only executions of the particular task will be reported.
- **state** - an optional query parameter that will return only those tasks in the given state.

Response Payload

The response payload is a JSON object with an array of task objects.

Name	Type	Description	Example
id	string	A unique identifier for the task. This takes the form of a uuid and not a Linux process id as the ID's must survive restarts and failovers	0a787bf3-4f48-4235-ae9a-2816f8ac76cc
name	string	The name of the task	purge
state	string	The current state of the task	Running
start-Time	times-tamp	The date and time the task started	2018-04-17 08:32:15.071
end-Time	times-tamp	The date and time the task ended This may not exist if the task is not completed.	2018-04-17 08:32:14.872
exit-Code	integer	Exit Code of the task.	0
reason	string	An optional reason string that describes why the task failed.	No destination available to write backup

Example

```
$ curl -X GET http://localhost:8081/flir/task
{
  "tasks": [
    {
      "id": "a9967d61-8bec-4d0b-8aa1-8b4dfb1d9855",
      "name": "stats collection",
      "processName": "stats collector",
      "state": "Complete",
      "startTime": "2020-05-28 09:21:58.650",
      "endTime": "2020-05-28 09:21:59.155",
      "exitCode": 0,
      "reason": ""
    },
    {
      "id": "7706b23c-71a4-410a-a03a-9b517dcd8c93",
      "name": "stats collection",
      "processName": "stats collector",
      "state": "Complete",
      "startTime": "2020-05-28 09:22:13.654",
      "endTime": "2020-05-28 09:22:14.160",
      "exitCode": 0,
      "reason": ""
    },
    ... ] }
$
$ curl -X GET http://localhost:8081/flir/task?name=purge
{
  "tasks": [
    {
      "id": "c24e006d-22f2-4c52-9f3a-391a9b17b6d6",
      "name": "purge",
      "processName": "purge",
      "state": "Complete",
      "startTime": "2020-05-28 09:44:00.175",
      "endTime": "2020-05-28 09:44:13.915",
      "exitCode": 0,
      "reason": ""
    },
    {
```

(continues on next page)

(continued from previous page)

```
"id": "609f35e6-4e89-4749-ac17-841ae3ee2b31",
"name": "purge",
"processName": "purge",
"state": "Complete",
"startTime": "2020-05-28 09:44:15.165",
"endTime": "2020-05-28 09:44:28.154",
"exitCode": 0,
"reason": ""
},
... ] }
$
$ curl -X GET http://localhost:8081/flir/task?state=complete
{
"tasks": [
{
  "id": "a9967d61-8bec-4d0b-8aa1-8b4dfb1d9855",
  "name": "stats collection",
  "processName": "stats collector",
  "state": "Complete",
  "startTime": "2020-05-28 09:21:58.650",
  "endTime": "2020-05-28 09:21:59.155",
  "exitCode": 0,
  "reason": ""
},
{
  "id": "7706b23c-71a4-410a-a03a-9b517dcd8c93",
  "name": "stats collection",
  "processName": "stats collector",
  "state": "Complete",
  "startTime": "2020-05-28 09:22:13.654",
  "endTime": "2020-05-28 09:22:14.160",
  "exitCode": 0,
  "reason": ""
},
... ] }
$
```

GET task latest

GET /flir/task/latest - return the list of most recent task execution for each name.

This call is designed to allow a monitoring interface to show when each task was last run and what the status of that task was.

Request Parameters

- **name** - an optional task name to filter on, only executions of the particular task will be reported.
- **state** - an optional query parameter that will return only those tasks in the given state.

Response Payload

The response payload is a JSON object with an array of task objects.

Name	Type	Description	Example
id	string	A unique identifier for the task. This takes the form of a uuid and not a Linux process id as the ID's must survive restarts and failovers	0a787bf3-4f48-4235-ae9a-2816f8ac76cc
name	string	The name of the task	purge
state	string	The current state of the task	Running
start-Time	times-tamp	The date and time the task started	2018-04-17 08:32:15.071
end-Time	times-tamp	The date and time the task ended This may not exist if the task is not completed.	2018-04-17 08:32:14.872
exit-Code	integer	Exit Code of the task.	0
reason	string	An optional reason string that describes why the task failed.	No destination available to write backup
pid	integer	Process ID of the task.	17481

Example

```
$ curl -X GET http://localhost:8081/flir/task/latest
{
  "tasks": [
    {
      "id": "ea334d3b-8a33-4a29-845c-8be50efd44a4",
      "name": "certificate checker",
      "processName": "certificate checker",
      "state": "Complete",
      "startTime": "2020-05-28 09:35:00.009",
      "endTime": "2020-05-28 09:35:00.057",
      "exitCode": 0,
      "reason": "",
      "pid": 17481
    },
    {
      "id": "794707da-dd32-471e-8537-5d20dc0f401a",
      "name": "stats collection",
      "processName": "stats collector",
      "state": "Complete",
      "startTime": "2020-05-28 09:37:28.650",
      "endTime": "2020-05-28 09:37:29.138",
      "exitCode": 0,
      "reason": "",
      "pid": 17926
    }
  ]
}

$ curl -X GET http://localhost:8081/flir/task/latest?name=purge
{
  "tasks": [
    {
      "id": "609f35e6-4e89-4749-ac17-841ae3ee2b31",
      "name": "purge",
      "processName": "purge",
      "state": "Complete",
      "startTime": "2020-05-28 09:44:15.165",
      "endTime": "2020-05-28 09:44:28.154",
```

(continues on next page)

(continued from previous page)

```
"exitCode": 0,  
"reason": "",  
"pid": 20914  
}  
]  
}  
$
```

GET task by ID

GET /flir/task/{id} - return the task information for the given task

Path Parameters

- **id** - the uuid of the task whose data should be returned.

Response Payload

The response payload is a JSON object containing the task details.

Name	Type	Description	Example
id	string	A unique identifier for the task. This takes the form of a uuid and not a Linux process id as the ID's must survive restarts and failovers	0a787bf3-4f48-4235-ae9a-2816f8ac76cc
name	string	The name of the task	purge
state	string	The current state of the task	Running
start-Time	times-tamp	The date and time the task started	2018-04-17 08:32:15.071
end-Time	times-tamp	The date and time the task ended This may not exist if the task is not completed.	2018-04-17 08:32:14.872
exit-Code	integer	Exit Code of the task.	0
reason	string	An optional reason string that describes why the task failed.	No destination available to write backup

Example

```
$ curl -X GET http://localhost:8081/flir/task/ea334d3b-8a33-4a29-845c-8be50efd44a4  
{  
  "id": "ea334d3b-8a33-4a29-845c-8be50efd44a4",  
  "name": "certificate checker",  
  "processName": "certificate checker",  
  "state": "Complete",  
  "startTime": "2020-05-28 09:35:00.009",  
  "endTime": "2020-05-28 09:35:00.057",  
  "exitCode": 0,  
  "reason": ""  
}  
$
```

Cancel task by ID

PUT /flir/task/{id}/cancel - cancel a task

Path Parameters

- **id** - the uuid of the task to cancel.

Response Payload

The response payload is a JSON object with the details of the cancelled task.

Name	Type	Description	Example
id	string	A unique identifier for the task. This takes the form of a uuid and not a Linux process id as the ID's must survive restarts and failovers	0a787bf3-4f48-4235-ae9a-2816f8ac76cc
name	string	The name of the task	purge
state	string	The current state of the task	Running
start-Time	times-tamp	The date and time the task started	2018-04-17 08:32:15.071
end-Time	times-tamp	The date and time the task ended This may not exist if the task is not completed.	2018-04-17 08:32:14.872
reason	string	An optional reason string that describes why the task failed.	No destination available to write backup

Example

```
$ curl -X PUT http://localhost:8081/flir/task/ea334d3b-8a33-4a29-845c-8be50efd44a4/
→cancel
{"id": "ea334d3b-8a33-4a29-845c-8be50efd44a4", "message": "Task cancelled successfully"}
→ }
$
```

11.2.4 Other Administrative API calls**ping**

The *ping* interface gives a basic confidence check that the Flir appliance is running and the API aspect of the appliance is functional. It is designed to be a simple test that can be applied by a user or by an HA monitoring system to test the liveness and responsiveness of the system.

GET ping

GET /flir/ping - return liveness of Flir

NOTE: the GET method can be executed without authentication even when authentication is required. This behaviour is configurable via a configuration option.

Response Payload

The response payload is some basic health information in a JSON object.

Name	Type	Description	Example
uptime	numeric	Time in seconds since Flir started	2113.076449394226
dataRead	numeric	A count of the number of sensor readings	1452
dataSent	numeric	A count of the number of readings sent to PI	347
dataPurged	numeric	A count of the number of readings purged	226
authenticationOptional	boolean	When true, the REST API does not require authentication. When false, users must successfully login in order to call the rest API. Default is <i>true</i>	true
serviceName	string	Name of service	Flir
hostName	string	Name of host machine	flir
ipAddresses	list	IPv4 and IPv6 address of host machine	["10.0.0.0","123:234:345:456:567:567:567:567"]
health	string	Health of Flir services	"green"
safeMode	boolean	True if Flir is started in safe mode (only core and storage services will be started)	2113.076449394226

Example

```
$ curl -s http://localhost:8081/flir/ping
{
  "uptime": 276818,
  "dataRead": 0,
  "dataSent": 0,
  "dataPurged": 0,
  "authenticationOptional": true,
  "serviceName": "Flir",
  "hostName": "flir",
  "ipAddresses": [
    "x.x.x.x",
    "x:x:x:x:x:x:x:x"
  ],
  "health": "green",
  "safeMode": false
}
```

statistics

The *statistics* interface allows the retrieval of live statistics and statistical history for the Flir device.

GET statistics

GET /flir/statistics - return a general set of statistics

Response Payload

The response payload is a JSON document with statistical information (all numerical), these statistics are absolute counts since Flir started.

Key	Description
BUFFERED	Readings currently in the Flir buffer
DISCARDED	Readings discarded by the South Service before being placed in the buffer. This may be due to an error in the readings themselves.
PURGED	Readings removed from the buffer by the purge process
READINGS	Readings received by Flir
UNSENT	Readings filtered out in the send process
UNSNPURGED	Readings that were purged from the buffer before being sent

Example

```
$ curl -s http://localhost:8081/flir/statistics
[ {
  "key": "BUFFERED",
  "description": "Readings currently in the Flir buffer",
  "value": 0
},
...
{
  "key": "UNSNPURGED",
  "description": "Readings that were purged from the buffer before being sent",
  "value": 0
},
... ]
$
```

GET statistics/history

GET /flir/statistics/history - return a historical set of statistics. This interface is normally used to check if a set of sensors or devices are sending data to Flir, by comparing the recent statistics and the number of readings received for an asset.

Request Parameters

- **limit** - limit the result set to the *N* most recent entries.

Response Payload

A JSON document containing an array of statistical information, these statistics are delta counts since the previous entry in the array. The time interval between values is a constant defined that runs the gathering process which populates the history statistics in the storage layer.

Key	Description
interval	The interval in seconds between successive statistics values
statistics[].BUFFERED	Readings currently in the Flir buffer
statistics[].DISCARDED	Readings discarded by the South Service before being placed in the buffer. This may be due to an error in the readings themselves.
statistics[].PURGED	Readings removed from the buffer by the purge process
statistics[].READINGS	Readings received by Flir
statistics[*NORTH_TASK_NAME]	The number of readings sent to the PI system via the OMF plugin with north instance name
statistics[].UNSENT	Readings filtered out in the send process
statistics[].UNSNPURGED	Readings that were purged from the buffer before being sent
statistics[*ASSET-CODE*]	The number of readings received by Flir since startup with name <i>asset-code</i>

Example

```
$ curl -s http://localhost:8081/flir/statistics/history?limit=2
{
  "interval": 15,
  "statistics": [
    {
      "history_ts": "2020-06-01 11:21:04.357",
      "READINGS": 0,
      "BUFFERED": 0,
      "UNSENT": 0,
      "PURGED": 0,
      "UNSNPURGED": 0,
      "DISCARDED": 0,
      "Readings Sent": 0
    },
    {
      "history_ts": "2020-06-01 11:20:48.740",
      "READINGS": 0,
      "BUFFERED": 0,
      "UNSENT": 0,
      "PURGED": 0,
      "UNSNPURGED": 0,
      "DISCARDED": 0,
      "Readings Sent": 0
    }
  ]
}
```

11.3 User API Reference

The user API provides a mechanism to access the data that is buffered within Flir. It is designed to allow users and applications to get a view of the data that is available in the buffer and do analysis and possibly trigger actions based on recently received sensor readings.

In order to use the entry points in the user API, with the exception of the `/flir/authenticate` entry point, there must be an authenticated client calling the API. The client must provide a header field in each request, `authtoken`, the value of which is the token that was retrieved via a call to `/flir/authenticate`. This token must be checked for validity and also that the authenticated entity has user or admin permissions.

11.3.1 Browsing Assets

asset

The `asset` method is used to browse all or some assets, based on search and filtering.

GET all assets

GET `/flir/asset` - Return an array of asset codes buffered in Flir and a count of assets by code.

Response Payload

An array of JSON objects, one per asset.

Name	Type	Description	Example
[].assetCode	string	The code of the asset	fogbench/accelerometer
[].count	number	The number of recorded readings for the asset code	22359

Example

```
$ curl -s http://localhost:8081/flir/asset
[ { "count": 18, "assetCode": "fogbench/accelerometer" },
  { "count": 18, "assetCode": "fogbench/gyroscope" },
  { "count": 18, "assetCode": "fogbench/humidity" },
  { "count": 18, "assetCode": "fogbench/luxometer" },
  { "count": 18, "assetCode": "fogbench/magnetometer" },
  { "count": 18, "assetCode": "fogbench/mouse" },
  { "count": 18, "assetCode": "fogbench/pressure" },
  { "count": 18, "assetCode": "fogbench/switch" },
  { "count": 18, "assetCode": "fogbench/temperature" },
  { "count": 18, "assetCode": "fogbench/wall clock" } ]
$
```

GET asset readings

GET /flir/asset/{code} - Return an array of readings for a given asset code.

Path Parameters

- **code** - the asset code to retrieve.

Request Parameters

- **limit** - set the limit of the number of readings to return. If not specified, the defaults is 20 readings.

Response Payload

An array of JSON objects with the readings data for a series of readings sorted in reverse chronological order.

Name	Type	Description	Example
[].timestamp	timestamp	The time at which the reading was received.	2018-04-16 14:33:18.215
[].reading	JSON object	The JSON reading object received from the sensor.	{ "reading": { "x": 0, "y": 0, "z": 1 } }

Example

```
$ curl -s http://localhost:8081/flir/asset/fogbench%2Faccelerometer
[ { "reading": { "x": 0, "y": -2, "z": 0 }, "timestamp": "2018-04-19 14:20:59.692" },
  { "reading": { "x": 0, "y": 0, "z": -1 }, "timestamp": "2018-04-19 14:20:54.643" },
  { "reading": { "x": -1, "y": 2, "z": 1 }, "timestamp": "2018-04-19 14:20:49.899" },
  { "reading": { "x": -1, "y": -1, "z": 1 }, "timestamp": "2018-04-19 14:20:47.026" },
  { "reading": { "x": -1, "y": -2, "z": -2 }, "timestamp": "2018-04-19 14:20:42.746" },
  { "reading": { "x": 0, "y": 2, "z": 0 }, "timestamp": "2018-04-19 14:20:37.418" },
  { "reading": { "x": -2, "y": -1, "z": 2 }, "timestamp": "2018-04-19 14:20:32.650" },
  { "reading": { "x": 0, "y": 0, "z": 1 }, "timestamp": "2018-04-19 14:06:05.870" },
  { "reading": { "x": 1, "y": 1, "z": 1 }, "timestamp": "2018-04-19 14:06:05.870" },
  { "reading": { "x": 0, "y": 0, "z": -1 }, "timestamp": "2018-04-19 14:06:05.869" },
  ... ]
```

(continues on next page)

(continued from previous page)

```

{ "reading": { "x": 2, "y": -1, "z": 0 }, "timestamp": "2018-04-19 14:06:05.868" },
{ "reading": { "x": -1, "y": -2, "z": 2 }, "timestamp": "2018-04-19 14:06:05.867" },
{ "reading": { "x": 2, "y": 1, "z": 1 }, "timestamp": "2018-04-19 14:06:05.867" },
{ "reading": { "x": 1, "y": -2, "z": 1 }, "timestamp": "2018-04-19 14:06:05.866" },
{ "reading": { "x": 2, "y": -1, "z": 1 }, "timestamp": "2018-04-19 14:06:05.865" },
{ "reading": { "x": 0, "y": -1, "z": 2 }, "timestamp": "2018-04-19 14:06:05.865" },
{ "reading": { "x": 0, "y": -2, "z": 1 }, "timestamp": "2018-04-19 14:06:05.864" },
{ "reading": { "x": -1, "y": -2, "z": 0 }, "timestamp": "2018-04-19 13:45:15.881" } ]
↪ ]
$
$ curl -s http://localhost:8081/flir/asset/fogbench%2Faccelerometer?limit=5
[ { "reading": { "x": 0, "y": -2, "z": 0 }, "timestamp": "2018-04-19 14:20:59.692" },
  { "reading": { "x": 0, "y": 0, "z": -1 }, "timestamp": "2018-04-19 14:20:54.643" },
  { "reading": { "x": -1, "y": 2, "z": 1 }, "timestamp": "2018-04-19 14:20:49.899" },
  { "reading": { "x": -1, "y": -1, "z": 1 }, "timestamp": "2018-04-19 14:20:47.026" },
  { "reading": { "x": -1, "y": -2, "z": -2 }, "timestamp": "2018-04-19 14:20:42.746" } ]
↪ ]
$

```

GET asset reading

GET /flir/asset/{code}/{reading} - Return an array of single readings for a given asset code.

Path Parameters

- **code** - the asset code to retrieve.
- **reading** - the sensor from the assets JSON formatted reading.

Request Parameters

- **limit** - set the limit of the number of readings to return. If not specified, the default is 20 single readings.

Response Payload

An array of JSON objects with a series of readings sorted in reverse chronological order.

Name	Type	Description	Example
timestamp	timestamp	The time at which the reading was received.	2018-04-16 14:33:18.215
{reading}	JSON object	The value of the specified reading.	"temperature": 20

Example

```

$ curl -s http://localhost:8081/flir/asset/fogbench%2Fhumidity/temperature
[ { "temperature": 20, "timestamp": "2018-04-19 14:20:59.692" },
  { "temperature": 33, "timestamp": "2018-04-19 14:20:54.643" },
  { "temperature": 35, "timestamp": "2018-04-19 14:20:49.899" },
  { "temperature": 0, "timestamp": "2018-04-19 14:20:47.026" },
  { "temperature": 37, "timestamp": "2018-04-19 14:20:42.746" },
  { "temperature": 47, "timestamp": "2018-04-19 14:20:37.418" },
  { "temperature": 26, "timestamp": "2018-04-19 14:20:32.650" },
  { "temperature": 12, "timestamp": "2018-04-19 14:06:05.870" },
  { "temperature": 38, "timestamp": "2018-04-19 14:06:05.869" },
  { "temperature": 7, "timestamp": "2018-04-19 14:06:05.869" },
  { "temperature": 21, "timestamp": "2018-04-19 14:06:05.868" },
  { "temperature": 5, "timestamp": "2018-04-19 14:06:05.867" },

```

(continues on next page)

(continued from previous page)

```
{ "temperature": 40, "timestamp": "2018-04-19 14:06:05.867" },
{ "temperature": 39, "timestamp": "2018-04-19 14:06:05.866" },
{ "temperature": 29, "timestamp": "2018-04-19 14:06:05.865" },
{ "temperature": 41, "timestamp": "2018-04-19 14:06:05.865" },
{ "temperature": 46, "timestamp": "2018-04-19 14:06:05.864" },
{ "temperature": 10, "timestamp": "2018-04-19 13:45:15.881" } ]
$
$ curl -s http://localhost:8081/flir/asset/fogbench%2Faccelerometer?limit=5
[ { "temperature": 20, "timestamp": "2018-04-19 14:20:59.692" },
{ "temperature": 33, "timestamp": "2018-04-19 14:20:54.643" },
{ "temperature": 35, "timestamp": "2018-04-19 14:20:49.899" },
{ "temperature": 0, "timestamp": "2018-04-19 14:20:47.026" },
{ "temperature": 37, "timestamp": "2018-04-19 14:20:42.746" } ]
$
```

GET asset reading summary

GET /flir/asset/{code}/{reading}/summary - Return minimum, maximum and average values of a reading by asset code.

Path Parameters

- **code** - the asset code to retrieve.
- **reading** - the sensor from the assets JSON formatted reading.

Response Payload

An array of JSON objects with a series of readings sorted in reverse chronological order.

Name	Type	Description	Example
{reading}.average	number	The average value of the set of sensor values selected in the query string	27
{reading}.min	number	The minimum value of the set of sensor values selected in the query string	0
{reading}.max	number	The maximum value of the set of sensor values selected in the query string	47

Example

```
$ curl -s http://localhost:8081/flir/asset/fogbench%2Fhumidity/temperature/summary
{ "temperature": { "max": 47, "min": 0, "average": 27 } }
$
```

GET timed average asset reading

GET /flir/asset/{code}/{reading}/series - Return minimum, maximum and average values of a reading by asset code in a time series. The default interval in the series is one second.

Path Parameters

- **code** - the asset code to retrieve.
- **reading** - the sensor from the assets JSON formatted reading.

Request Parameters

- **limit** - set the limit of the number of readings to return. If not specified, the defaults is 20 single readings.

Response Payload

An array of JSON objects with a series of readings sorted in reverse chronological order.

Name	Type	Description	Example
times-tamp	times-tamp	The time the reading represents.	2018-04-16 14:33:18
average	number	The average value of the set of sensor values selected in the query string	27
min	number	The minimum value of the set of sensor values selected in the query string	0
max	number	The maximum value of the set of sensor values selected in the query string	47

Example

```
$ curl -s http://localhost:8081/flir/asset/fogbench%2Fhumidity/temperature/series
[ { "timestamp": "2018-04-19 14:20:59", "max": 20, "min": 20, "average": 20 },
  { "timestamp": "2018-04-19 14:20:54", "max": 33, "min": 33, "average": 33 },
  { "timestamp": "2018-04-19 14:20:49", "max": 35, "min": 35, "average": 35 },
  { "timestamp": "2018-04-19 14:20:47", "max": 0, "min": 0, "average": 0 },
  { "timestamp": "2018-04-19 14:20:42", "max": 37, "min": 37, "average": 37 },
  { "timestamp": "2018-04-19 14:20:37", "max": 47, "min": 47, "average": 47 },
  { "timestamp": "2018-04-19 14:20:32", "max": 26, "min": 26, "average": 26 },
  { "timestamp": "2018-04-19 14:06:05", "max": 46, "min": 5, "average": 27.8 },
  { "timestamp": "2018-04-19 13:45:15", "max": 10, "min": 10, "average": 10 } ]

$
$ curl -s http://localhost:8081/flir/asset/fogbench%2Fhumidity/temperature/series
[ { "timestamp": "2018-04-19 14:20:59", "max": 20, "min": 20, "average": 20 },
  { "timestamp": "2018-04-19 14:20:54", "max": 33, "min": 33, "average": 33 },
  { "timestamp": "2018-04-19 14:20:49", "max": 35, "min": 35, "average": 35 },
  { "timestamp": "2018-04-19 14:20:47", "max": 0, "min": 0, "average": 0 },
  { "timestamp": "2018-04-19 14:20:42", "max": 37, "min": 37, "average": 37 } ]
```

12.1 Flir v1

12.1.1 v1.9.2

Release Date: 2021-09-29

- **Flir Core**

- New Features:

- * The ability for south plugins to persist data between executions of south services has been added for plugins written in C/C++. This follows the same model as already available for north plugins.
- * Notification delivery plugins now also receive the data that caused the rule to trigger. This can be used to deliver values in the notification delivery plugins.
- * A new option has been added to the sqlite storage plugin only that allows assets to be excluded from consideration in the purge process.
- * A new purge process has been added to control the growth of statistics history and audit trails. This new process is known as the “System Purge” process.
- * The support bundle has been updated to include details of the packages installed.
- * The package repository API endpoint has been updated to support Ubuntu 20.04 repository end point.
- * The handling of updates from RPM package repositories has been improved.
- * The certificate store has been updated to support more formats of certificates, including DER, P12 and PFX format certificates.
- * The documentation has been updated to include an improved & detailed introduction to filters.
- * The OMF north plugin documentation has been re-organised and updated to include the latest features that have been introduced to this plugin.

- * A new section has been added to the documentation that discusses the tuning of the edge based control path.

– Bug Fix:

- * A rare race condition during ingestion of readings would cause the south service to terminate and restart. This has now been resolved.
- * In some circumstances it was seen that north services could send the same data more than once. This has now been corrected.
- * An issue that caused an intermittent error in the tracking of data sent north has been resolved. This only impacted north services and not north tasks.
- * An optimisation has been added to prevent north plugins being sent empty data sets when the filter chain removes all the data in a reading set.
- * An issue that prevented a north service restarting correctly when certain combinations of filters were present has been resolved.
- * The API for retrieving the list of backups on the system has been improved to honour the limit and offset parameters.
- * An issue with the restore operation always restoring the latest backup rather than the chosen backup has been resolved.
- * The support package failed to include log data if binary data had been written to syslog. This has now been resolved.
- * The configuration category for the system purge was in the incorrect location with the configuration category tree, this has now been correctly placed underneath the “Utilities” item.
- * It was not possible to set a notification to always retrigger as there was a limitation that there must always be 1 second between notification triggers. This restriction has now been removed and it is possible to set a retrigger time of zero.
- * An error in the documentation for the plugin developers guide which incorrectly documented how to build debug binaries has been corrected.

• GUI

– New Features:

- * The user interface has been updated to improve the filtering of logs when a large number of services have been defined within the instance.
- * The user interface input validation for hostnames and port has been improved in the setup screen. A message is now displayed when an incorrect port or address is entered.
- * The user interface now prompts to accept a self signed certificate if one is configured.

– Bug Fix:

- * If a south or north plugin included a script type configuration item the GUI failed to allow the service or task using this plugin to be created correctly. This has now been resolved.
- * The ability to paste into password fields has been enabled in order to allow copy/paste of keys, tokens etc into configuration of the south and north services.
- * An issue that could result in filters not being correctly removed from a pipeline of 2 or more filters has been resolved.

• Plugins

– New Features:

- * A new south plugin has been added that can be used to support a number of REST based APIs. The plugin allows processing of JSON payloads or with the addition of Python scripting other payload formats may also be supported. This plugin also supports a choice of methods to control the set of readings data that will be returned.
 - * A new OPC/UA south plugin has been created based on the Safe and Secure OPC/UA library. This plugin supports authentication and encryption mechanisms.
 - * A new plugin has been added to fetch data from the Suez Water cloud API service.
 - * Control features have now been added to the modbus south plugin that allows the writing of registers and coils via the south service control channel.
 - * The modbus south control flow has been updated to use both 0x06 and 0x10 function codes. This allows items that are split across multiple modbus registers to be written in a single write operation.
 - * The MQTT Scripted south plugin has been updated to allow multiple assets to be ingested in a single plugin.
 - * The MQTT Scripted south plugin has been enhanced to support MQTTS as well as MQTT.
 - * The MQTT scripted plugin has been updated to support the return of a specific asset as well as values.
 - * The OMF plugin has been updated to support more complex scenarios for the placement of assets with the PI Asset Framework.
 - * The OMF north plugin hinting mechanism has been extended to support asset framework hierarchy hints.
 - * The OMF north plugin now defaults to using a concise naming scheme for tags in the PI server.
 - * The Kafka north plugin has been updated to allow timestamps of higher granularity than 1 second, previously timestamps would be truncated to the previous second.
 - * The Kafka north plugin has been enhanced to give the option of sending JSON objects as strings to Kafka, as previously the default, or sending them as JSON objects.
 - * The HTTP-C north plugin has been updated to allow the inclusion of customer HTTP headers.
 - * The Python35 Filter plugin did not correctly handle string type data points. This has now been resolved.
 - * The vibration velocity filter has been updated to support multiple channel data.
 - * The MQTT broker package now supports RPM platforms.
 - * The OMF Hint filter documentation has been updated to describe the use of regular expressions when defining the asset name to which the hint should be applied.
 - * The Beckhoff south plugin documentation has been updated to include details on how to create the AMS route in a number of different scenarios.
- Bug Fix:
- * An issue with string data that had quote characters embedded within the reading data has been resolved. This would cause data to be discarded with a bad formatting message in the log.
 - * An issue that could result in the configuration for the incorrect plugin being displayed has now been resolved.
 - * An issue with the modbus south plugin that could cause resource starvation in the threads used for set point write operations has been resolved.
 - * A race condition in the modbus south that could cause an issue if the plugin configuration is changed during a set point operation.

- * Importing the Pandas Python library into the script within the MQTT scripted plugin previously failed due to the way Pandas uses global variables. This has now been resolved such that Pandas can be imported, however it should be noted that a filter can not import Pandas if the south plugin already imports Pandas.
- * When using the South MQTT Scripted plugin, if the Python script returned an asset name as well as a reading the asset name would be corrupted on second and subsequent calls. This has now been resolved.
- * The MQTT scripted plugin would occasionally fail to shutdown cleanly. This issue has now been resolved.
- * The MQTT Scripted plugin could not previously deal with payloads that consisted of a simple negative number. This has now been corrected.
- * An issue with the MQTT notification plugin and the MQTT scripted plugin when installing with RPM packages has been resolved.
- * The CSV playback south plugin installation on CentOS 7 platforms has now been corrected.
- * The digiducer south plugin has been updated to support the latest release of the underlying libraries that support it.
- * The error handling of the OMF north plugin has been improved such that assets that contain data types that are not supported by the OMF endpoint of the PI Server are removed and other data continues to be sent to the PI Server.
- * The Kafka north plugin was not always able to reconnect if the Kafka service was not available when it was first started. This issue has now been resolved.
- * The Kafka north plugin would on occasion duplicate data if a connection failed and was later reconnected. This has been resolved.
- * A number of fixes have been made to the Kafka north plugin, these include; fixing issues caused by quoted data in the Kafka payload, sending timestamps accurate to the millisecond, fixing an issue that caused data duplication and switching the the user timestamp.
- * A problem with the quoting of string type data points on the North HTTP-C plugin has been fixed.
- * String type variables in the OPC/UA north plugin were incorrectly having extra quotes added to them. This has now been resolved.
- * The delta filter previously did not manage calculating delta values when a datapoint changed from being an integer to a floating point value or vice versa. This has now been resolved and delta values are correctly calculated when these changes occur.
- * The vibration features plugin has been updated to run on Ubuntu 20 platforms.
- * The signal processing filter plugin now installs correctly on CentOS platforms.
- * The data frames filter plugin is now supported on RPM based platforms.
- * An issue with the vibration features filter on Ubuntu 18 has been resolved.
- * The example path shown in the DHT11 plugin in the developers guide was incorrect, this has now been fixed.

12.1.2 v1.9.1

Release Date: 2021-05-27

- **Flir Core**

- New Features:

- * Support has been added for Ubuntu 20.04 LTS.
- * The core components have been ported to build and run on CentOS 8
- * A new option has been added to the command line tool that controls the system. This option, called purge, allows all readings related data to be purged from the system whilst retaining the configuration. This allows a system to be tested and then reset without losing the configuration.
- * A new service interface has been added to the south service that allows set point control and operations to be performed via the south interface. This is the first phase of the set point control feature in the product.
- * The documentation has been improved to include the new control functionality in the south plugin developers guide.
- * An improvement has been made to the documentation layout for default plugins to make the GUI able to find the plugin documentation.
- * Documentation describing the installation of PostgreSQL on CentOS has been updated.
- * The documentation has been updated to give more detail around the topic of self-signed certificates.

- Bug Fix:

- * A security flaw that allowed non-privileged users to update the certificate store has been resolved.
- * A bug that prevented users being created with certificate based authentication rather than password based authentication has been fixed.
- * Switching storage plugins from SQLite to PostgreSQL caused errors in some circumstances. This has now been resolved.
- * The HTTP code returned by the ping command has been updated to correctly report 401 errors if the option to allow ping without authentication is turned off.
- * The HTTP error code returned when the notification service is not available has been corrected.
- * Disabling and re-enabling the backup and restore task schedules sometimes caused a restart of the system. This has now been resolved.
- * The error message returned when schedules could not be enabled or disabled has been improved.
- * A problem related to readings with nested data not correctly getting copied has been resolved.
- * An issue that caused problems if a service was deleted and then a new service was recreated using the name of the previously deleted service has been resolved.

- **GUI**

- New Features:

- * Links to the online help have been added on a number of screens in the user interface.
 - * Improvements have been made to the user management screens of the GUI.

- **Plugins**

- New Features:

- * North services now support Python as well as C++ plugins.
 - * A new south plugin has been created to read data from the ABB cloud service.
 - * A new south plugin has been added for getting vibration data from a set of FLIR GW65 vibration sensors.

- * A new delivery notification plugin has been added that uses the set point control mechanism to invoke an action in the south plugin.
 - * A new notification delivery mechanism has been implemented that uses the set point control mechanism to assert control on a south service. The plugin allows you to set the values of one or more control items on the notification triggered and set a different set of values when the notification rule clears.
 - * Support has been added in the OPC/UA north plugin for array data. This allows FFT spectrum data to be represented in the OPC/UA server.
 - * The documentation for the OPC/UA north plugin has been updated to recommend running the plugin as a service.
 - * A new storage plugin has been added that uses SQLite. This is designed for situations with low bandwidth sensors and stores all the readings within a single SQLite file.
 - * The CSV Writer filter has been updated to support writing encrypted files.
 - * Support has been added to use RTSP video streams in the person detection plugin.
 - * The delta filter has been updated to allow an optional set of asset specific tolerances to be added in addition to the global tolerance used by the plugin when deciding to forward data.
 - * The Python script run by the MQTT scripted plugin now receives the topic as well as the message.
 - * The OMF plugin has been updated in line with recommendations from the OMF group regarding the use of SCRF Defense.
 - * The OMFHint plugin has been updated to support wildcarding of asset names in the rules for the plugin.
 - * New documentation has been added to help in troubleshooting PI connection issues.
 - * The `pi_server` and `ocs` north plugins are deprecated in favour of the newer and more feature rich OMF north plugin. These deprecated plugins cannot be used in north services and are only provided for backward compatibility when run as north tasks. These plugins will be removed in a future release.
- Bug Fix:
- * The OMF plugin has been updated to better deal with nested data.
 - * Some improvements to error handling have been added to the InfluxDB north plugin for version 1.x of InfluxDB.
 - * The Python 35 filter stated it used the Python version 3.5 always, in reality it uses whatever Python 3 version is installed on your system. The documentation has been updated to reflect this.
 - * The Asset Split filter plugin previously logged debug messages by default, this has now been resolved.
 - * Fixed a bug that treated arrays of bytes as if they were strings in the OPC/UA south plugin.
 - * The FFT2 filter used a single asset name for all output FFT's. If an incoming asset had multiple data points they would each have a separate FFT applied to them and then output with the same asset name. This caused confusion. Now if there are multiple data points each will have a unique asset name for the output FFT. This asset name is made up of the configured output asset name with the data point name appended. For example an inout asset having X, Y and Z data points with the output asset configured to be FFT will result in 3 assets, FFTX, FFTY and FFTZ.
 - * The HTTP North C plugin would not correctly shutdown, this effected reconfiguration when run as an always on service. This issue has now been resolved.
 - * The description of the statistics filter was incorrect, this has now been corrected.

- * An issue with the SQLite In Memory storage plugin that caused database locks under high load conditions has been resolved.

13.1 Introduction

The bundled OMF north plugin in Flir can use a number of different authentication schemes when communicating with the various OSIssoft products. The PI Web API method in the [OMF](#) plugin supports the use of a Kerberos scheme.

The Flir *requirements.sh* script installs the Kerberos client to allow the integration with what in the specific terminology is called KDC (the Kerberos server).

13.2 PI-Server as the North endpoint

The OSI *Connector Relay* allows token authentication while *PI Web API* supports Basic and Kerberos.

There could be more than one configuration to allow the Kerberos authentication, the easiest one is the Windows server on which the PI-Server is executed act as the Kerberos server also.

The Windows Active directory should be installed and properly configured for allowing the Windows server to authenticate Kerberos requests.

13.3 North plugin

The North plugin has a set of configurable options that should be changed, using either the Flir API or the Flir GUI, to select the Kerberos authentication.

The North plugin supports the configurable option *PIServerEndpoint* for allowing to select the target among:

- Connector Relay
- PI Web API
- Edge Data Store
- OSIssoft Cloud Services

The *PIWebAPIAuthenticationMethod* option permits to select the desired authentication among:

- anonymous
- basic
- kerberos

The Kerberos authentication requires a keytab file, the *PIWebAPIKerberosKeytabFileName* option specifies the name of the file expected under the directory:

```
${FLIR_ROOT}/data/etc/kerberos
```

NOTE:

- *A keytab is a file containing pairs of Kerberos principals and encrypted keys (which are derived from the Kerberos password). A keytab file allows to authenticate to various remote systems using Kerberos without entering a password.*

the *AFHierarchyLevel* option allows to specific the first level of the hierarchy that will be created into the Asset Framework and will contain the information for the specific North plugin.

13.4 Flir server configuration

The server on which Flir is going to be executed needs to be properly configured to allow the Kerberos authentication.

The following steps are needed:

- *IP Address resolution for the KDC*
- *Kerberos client configuration*
- *Kerberos keytab file setup*

13.4.1 IP Address resolution of the KDC

The Kerberos server name should be resolved to the corresponding IP Address, editing the */etc/hosts* is one of the possible and the easiest way, sample row to add:

```
192.168.1.51    pi-server.dianomic.com pi-server
```

try the resolution of the name using the usual *ping* command:

```
$ ping -c 1 pi-server.dianomic.com

PING pi-server.dianomic.com (192.168.1.51) 56(84) bytes of data.
64 bytes from pi-server.dianomic.com (192.168.1.51): icmp_seq=1 ttl=128 time=0.317 ms
64 bytes from pi-server.dianomic.com (192.168.1.51): icmp_seq=2 ttl=128 time=0.360 ms
64 bytes from pi-server.dianomic.com (192.168.1.51): icmp_seq=3 ttl=128 time=0.455 ms
```

NOTE:

- *the name of the KDC should be the first in the list of aliases*

13.4.2 Kerberos client configuration

The server on which Flir runs act like a Kerberos client and the related configuration file should be edited for allowing the proper Kerberos server identification. The information should be added into the `/etc/krb5.conf` file in the corresponding section, for example:

```
[libdefaults]
    default_realm = DIANOMIC.COM

[realms]
    DIANOMIC.COM = {
        kdc = pi-server.dianomic.com
        admin_server = pi-server.dianomic.com
    }
```

13.4.3 Kerberos keytab file

The keytab file should be generated on the Kerberos server and copied into the Flir server in the directory:

```
${FLIR_DATA}/etc/kerberos
```

NOTE:

- if **FLIR_DATA** is not set its value should be `$FLIR_ROOT/data`.

The name of the file should match the value of the North plugin option `PIWebAPIKerberosKeytabFileName`, by default `piwebapi_kerberos_https.keytab`

```
$ ls -l ${FLIR_DATA}/etc/kerberos
-rwxrwxrwx 1 flir flir 91 Jul 17 09:07 piwebapi_kerberos_https.keytab
-rw-rw-r-- 1 flir flir 199 Aug 13 15:30 README.rst
```

The way the keytab file is generated depends on the type of the Kerberos server, in the case of Windows Active Directory this is an sample command:

```
ktpass -princ HTTPS/pi-server@DIANOMIC.COM -mapuser Administrator@DIANOMIC.COM -pass_
↪Password -crypto AES256-SHA1 -ptype KRB5_NT_PRINCIPAL -out C:\Temp\piwebapi_
↪kerberos_https.keytab
```

13.4.4 Troubleshooting the Kerberos authentication

- 1) check the North plugin configuration, a sample command

```
curl -s -S -X GET http://localhost:8081/flir/category/North_Readings_to_PI | jq ".|
↪{URL, \"PIServerEndpoint\", PIWebAPIAuthenticationMethod, PIWebAPIKerberosKeytabFileName,
↪AFHierarchyLevel}\""
```

- 2) check the presence of the keytab file

```
$ ls -l ${FLIR_ROOT}/data/etc/kerberos
-rwxrwxrwx 1 flir flir 91 Jul 17 09:07 piwebapi_kerberos_https.keytab
-rw-rw-r-- 1 flir flir 199 Aug 13 15:30 README.rst
```

- 3) verify the reachability of the Kerberos server (usually the PI-Server) - Network reachability

```
$ ping pi-server.dianomic.com
PING pi-server.dianomic.com (192.168.1.51) 56(84) bytes of data.
64 bytes from pi-server.dianomic.com (192.168.1.51): icmp_seq=1 ttl=128 time=5.07 ms
64 bytes from pi-server.dianomic.com (192.168.1.51): icmp_seq=2 ttl=128 time=1.92 ms
```

Kerberos reachability and keys retrieval

```
$ kinit -p HTTPS/pi-server@DIANOMIC.COM
Password for HTTPS/pi-server@DIANOMIC.COM:
$ klist
Ticket cache: FILE:/tmp/krb5cc_1001
Default principal: HTTPS/pi-server@DIANOMIC.COM

Valid starting          Expires              Service principal
09/27/2019 11:51:47    09/27/2019 21:51:47  krbtgt/DIANOMIC.COM@DIANOMIC.COM
        renew until 09/28/2019 11:51:46
$
```

13.5 Kerberos authentication on RedHat/CentOS

RedHat and CentOS version 7 provide by default an old version of curl and the related libcurl and it does not support Kerberos, output of the curl provided by CentOS:

```
$ curl -V
curl 7.29.0 (x86_64-redhat-linux-gnu) libcurl/7.29.0 NSS/3.36 zlib/1.2.7 libidn/1.28_
↳libssh2/1.4.3
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps pop3 pop3s rtsp_
↳scp sftp smtp smtps telnet tftp
Features: AsynchDNS GSS-Negotiate IDN IPv6 Largefile NTLM NTLM_WB SSL libz unix-
↳sockets
```

The *requirements.sh* evaluates if the default version 7.29.0 is installed and in this case it will download the sources, build and install the version 7.65.3 to provide Kerberos authentication, output of the curl after the upgrade:

```
$ curl -V
curl 7.65.3 (x86_64-unknown-linux-gnu) libcurl/7.65.3 OpenSSL/1.0.2k-fips zlib/1.2.7
Release-Date: 2019-07-19
Protocols: dict file ftp ftps gopher http https imap imaps pop3 pop3s rtsp smb smbs_
↳smtp smtps telnet tftp
Features: AsynchDNS GSS-API HTTPS-proxy IPv6 Kerberos Largefile libz NTLM NTLM_WB_
↳SPNEGO SSL UnixSockets
```

The sources are downloaded from the curl repository [curl sources](#), the curl homepage is available at [curl homepage](#).

The following external plugins are currently available to extend the functionality of Flir.

14.1 FLIR Bridge South Plugins

14.1.1 ABB Ability Smart Cloud Service

The *flir-south-abb* plugin is designed to pull data from the ABB Ability™ Smart Sensor Cloud into FLIR Bridge. It pulls data for a list of ABB assets into the local FLIR Bridge system at a rate defined for the service.

To create a south service with the ABB plugin

- Click on *South* in the left hand menu bar
- Select *ABB* from the plugin list
- Name your service and click *Next*

1 Plugin & Service Name 2 Review Configuration 3 Done

ABB Assets

```

1 {
2   "assets": [
3     ""
4   ]
5 }

```

ABB Service

Username

Auth. Key

Asset Structure

- Configure the plugin
 - **ABB Assets:** A list of the assets in the ABB cloud service that should be read. This is a JSON document with an array called assets which contains the assets name as strings.
 - **ABB Service:** The hostname of the ABB service to which to connect. Usually this is the default api.smartsensor.abb.com.
 - **Username:** The ABB cloud user name.
 - **Auth. Key:** The authentication key that has been created in the ABB cloud for the given username.
 - **Asset Structure:** This defines how the FLIR Bridge assets that will be created should be organized.

Asset Structure

✓ Single Asset
Group Assets
Individual Asset

* **Single Asset:** A single asset in the ABB cloud will be stored as a single asset in FLIR Bridge with the same name as the ABB asset. Within each FLIR Bridge asset a data point

will be created for each data value within the asset using the ABB measurement type name.

- * **Group Assets:** An asset will be created for each group of sensors for each asset within the ABB cloud. The asset will be named `<ABB asset>_<group name>`. Within each FLIR Bridge asset a data point will be created for each data value within the group using the ABB measurement type name.
- * **Individual Assets:** An asset will be created for each data item for each ABB cloud asset. The asset will be named `<ABB asset>_<item name>`.

14.1.2 AM2315 Temperature & Humidity Sensor



The *flir-south-am2315* is a south plugin for a temperature and humidity sensor. The sensor connects via the I2C bus and can provide temperature data in the range -40°C to $+125^{\circ}\text{C}$ with an accuracy of 0.1°C .

The plugin will produce a single asset that has two data points; temperature and humidity.

Note: The AM2315 is only available on the Raspberry Pi as it requires an I2C bus connection

To create a south service with the AM2315 plugin

- Click on *South* in the left hand menu bar
- Select *am2315* from the plugin list
- Name your service and click *Next*

- Configure the plugin

- **Asset Name:** The name of the asset that will be created. To help when multiple AM2315 sensors are used a %M may be added to the asset name. This will be replaced with the I2C address of the sensor.
 - **I2C Address:** The I2C address of the sensor, this allows multiple sensors to be added to the same I2C bus.
- Click *Next*
 - Enable the service and click on *Done*

Wiring The Sensor

The following table details the four connections that must be made from the sensor to the Raspberry Pi GPIO connector.

Colour	Name	GPIO Pin	Description
Red	VDD	Pin 2 (5V)	Power (3.3V - 5V)
Yellow	SDA	Pin 3 (SDA)	Serial Data
Black	GND	Pin 6 (GND)	Ground
White	SCL	Pin 5 (SCL)	Serial Clock

14.1.3 Beckhoff TwinCAT

The *flir-south-beckhoff* plugin is a plugin that allows collection of data from Beckhoff PLC's using the TwinCAT 2 or TwinCAT 3 protocols. It utilises the ADS library to allow updates the the values held within the PLC to be captured in FLIR Bridge and sent onward as with any other data in FLIR Bridge.

The plugin uses a subscription model to register for changes to variables within the PLC and each of these becomes a data point in the asset that is created within FLIR Bridge.

To create a south service with the Beckhoff TwinCAT plugin

- Click on *South* in the left hand menu bar
- Select *Beckhoff* from the plugin list
- Name your service and click *Next*

1 Plugin & Service Name 2 Review Configuration 3 Done

Asset Name: beckhoff

EtherCAT Server: ads-server

Remote NetId: 192.168.0.231.1.1

Protocol: Automatic

Source NetId:

TwinCAT Map:

```

1 {
2   "items": [
3     {
4       "datapoint": "engine",
5       "name": "MAIN.engine"
6     }
7   ]
8 }

```

- Configure the plugin
 - **Asset Name:** The default asset name that is used for the data that is extracted from the PLC if the map does not define an explicit asset name.
 - **EtherCAT Server:** The hostname or IP address of the ADS master, this is the IP address of the Beckhoff PLC.
 - **Remote NetId:** The Beckhoff netId of the PLC. This is normally the IP address of the PLC with .1.1 appended to it.
 - **Protocol:** Define if the Automatic, TwinCAT 2 or TwinCAT 3 protocol is to be used. If *Automatic* is chosen the plugin will attempt to determine if the PLC supports TwinCAT 2 or TwinCAT 3.
 - **Source NetId:** The Beckhoff AMS NetId to assign to this plugin. This may be left blank, in which case an NetId will be generated from the IP address of the machine. However in some circumstances this is not acceptable or does not work correctly. A source NetId must always be provided when running within a container.
 - **TwinCAT Map:** A JSON document that is the data mapping for the PLC. This defines what variables are to be extracted from the PLC. See below for details of the map format.
- You must also authorise the FLIR Bridge plugin by adding an AMS route on your PLC

Adding AMS Route

Sample AMS route:

Name:	MyAdsClient
AMS Net Id:	192.168.0.1.1.1 # Derived from the IP address of your FLIR Bridge
Address:	192.168.0.1 # The IP address of your FLIR Bridge
Transport Type:	TCP/IP

Routes can be configured using one of several different methods;

TwinCAT Engineering: Go to the tree item SYSTEM/Routes and add a static route.

TwinCAT Systray: Open the context menu by right click the TwinCAT systray icon. (not available on Windows CE devices)

TC2: Go to Properties/AMS Router/Remote Computers. This requires a restart of TwinCAT on your PLC

TC3: Go to Router/Edit routes.

TcAmsRemoteMgr: Windows CE devices can be configured locally (TC2/TC3). Tool location: /Hard Disk/System/TcAmsRemoteMgr.exe. If uses TwinCAT 2 then a restart will be required after adding the AMS Route.

IPC Diagnose: Beckhoff IPC's provide a web interface for diagnose and configuration. Further information: [Beckhoff Device Manager](<http://infosys.beckhoff.de/content/1033/devicemanager/index.html?id=286>)

Map Format

The map is a JSON document that describes the variables to be extracted from the PLC. The variables may be defined either by name or by group and index id. Each variable will become a datapoint with the asset added to FLIR Bridge. The map itself is a single JSON array called "items", with each element in the array being an object that define the variable and what to do with it.

These objects have the following members within them

Key	Description
as-set	An optional element that defines an asset code that should be used to store the variable. If this is not given then the default asset code for the plugin is used.
datapoint	The name of the datapoint into which the variable is stored within the asset code. The datapoint name must be given for each object in the map.
name	The variable name within the PLC that is extracted. This may be obtained either by examining the PLC code that is running or by extracted from the .TPY file for the PLC. Either name or group and index must be given for each item in the map.
group	The numeric group within the PLC from which data is extracted. This allow data to be extracted without the use of variable names. It is not recommended for production use as it is very dependent on the layout of the PLC code, using variable names is more robust than group and index.
index	The numeric index within the group from which to extract data, see above.

Example

An example TwinCAT map is should below

```
{
  "items": [
    {
      "datapoint": "engine",
      "name": "MAIN.engine"
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```
]
}
```

This is based on the simulation that is available from Beckhoff and creates a single data point within the default asset called engine. It is populated with the value of the internal PLC variable *MAIN.engine*. A new asset will be created and added to the FLIR Bridge buffer every time this variable changes.

Multiple items may be read from the PLC by adding an element for each to the *items* array. For example to extract the two variables *MAIN.oilPressure* and *Main.engineSpeed* from the PLC a map as shown below could be used.

```
{
  "items": [
    {
      "datapoint": "oilPressure",
      "name": "MAIN.oilPressure"
    },
    {
      "datapoint": "rpm",
      "name": "MAIN.engineSpeed"
    }
  ]
}
```

Testing

The easiest way to test the Beckhoff plugin is to setup a simulation on a windows machine and run the Beckhoff PLC in simulator mode. The Beckhoff PLC can be freely downloaded from the Beckhoff site.

```
https://beckhoff.co.uk/english/download/tc3-downloads.htm?id=1905053019883865
```

This is designed to be run on a Windows 7 machine.

You can then create some sample variables to try to link to.

Downloading the code from Beckhoff includes a simple example that can be run that defines an engine variable, this is the example for which the default configuration is setup for.

Note: You will need to setup a static route in the Beckhoff PLC with the AMSNetId and IP address for the plugin and the type as TCP/IP.

14.1.4 CC2650 SensorTag



The *flir-south-cc2650* is a plugin that connects using Bluetooth to a Texas Instruments . The SensorTag offers 10 sensors within a small, low powered package which may be read by this plugin and ingested into FLIR Bridge. These sensors include;

- ambient light
- magnetometer
- humidity
- pressure
- accelerometer
- gyroscope
- object temperature
- digital microphone

Note: The sensor requires that you have a Bluetooth low energy adapter available that supports at least BLE 4.0.

To create a south service with the

- Click on *South* in the left hand menu bar
- Select *cc2650* from the plugin list
- Name your service and click *Next*

The screenshot shows a configuration wizard with three steps: 1. Plugin & Service Name, 2. Review Configuration, and 3. Done. The 'Review Configuration' step is active, displaying a list of configuration items on the left and their values in input fields on the right. The items include Bluetooth Address (B0:91:22:EA:79:04), Asset Name Prefix (CC2650/%M/), Shutdown Threshold (10), Connection Timeout (3), Temperature Sensor (checked), Temperature Sensor Name (temperature), Luminance Sensor (unchecked), Luminance Sensor Name (luminance), Humidity Sensor (unchecked), Humidity Sensor Name (humidity), Pressure Sensor (unchecked), Pressure Sensor Name (pressure), Movement Sensor (unchecked), Gyroscope Sensor Name (gyroscope), Accelerometer Sensor Name (accelerometer), Magnetometer Sensor Name (magnetometer), Battery Data (unchecked), and Battery Sensor Name (battery). At the bottom, there are 'Previous' and 'Next' buttons.

Configuration Item	Value
Bluetooth Address	B0:91:22:EA:79:04
Asset Name Prefix	CC2650/%M/
Shutdown Threshold	10
Connection Timeout	3
Temperature Sensor	<input checked="" type="checkbox"/>
Temperature Sensor Name	temperature
Luminance Sensor	<input type="checkbox"/>
Luminance Sensor Name	luminance
Humidity Sensor	<input type="checkbox"/>
Humidity Sensor Name	humidity
Pressure Sensor	<input type="checkbox"/>
Pressure Sensor Name	pressure
Movement Sensor	<input type="checkbox"/>
Gyroscope Sensor Name	gyroscope
Accelerometer Sensor Name	accelerometer
Magnetometer Sensor Name	magnetometer
Battery Data	<input type="checkbox"/>
Battery Sensor Name	battery

- Configure the plugin
 - **Bluetooth Address:** The Bluetooth MAC address of the device
 - **Asset Name Prefix:** A prefix to add to the asset name
 - **Shutdown Threshold:** The time in seconds allowed for a shutdown operation to complete
 - **Connection Timeout:** The Bluetooth connection timeout to use when attempting to connect to the device
 - **Temperature Sensor:** A toggle to include the temperature data in the data ingested
 - **Temperature Sensor Name:** The data point name to assign the temperature data
 - **Luminance Sensor:** Toggle to control the inclusion of the ambient light data
 - **Luminance Sensor Name:** The data point name to use for the luminance data
 - **Humidity Sensor:** A toggle to include the humidity data
 - **Humidity Sensor Name:** The data point name to use for the humidity data
 - **Pressure Sensor:** A toggle to control the inclusion of pressure data
 - **Pressure Sensor Name:** The name to be used for the data point that will contain the atmospheric pressure data

- **Movement Sensor:** A toggle that controls the inclusion of movement data gathered from the gyroscope, accelerometer and magnetometer
- **Gyroscope Sensor Name:** The data point name to use for the gyroscope data
- **Accelerometer Sensor Name:** The name of the data point that will record the accelerometer data
- **Magnetometer Sensor Name:** The name to use for the magnetometer data
- **Battery Data:** A toggle to control inclusion of the state of charge of the battery
- **Battery Sensor Name:** The data point name for the battery charge percentage
- Click *Next*
- Enable the service and click on *Done*

14.1.5 CoAP

The *flir-south-coap* plugin implements a passive listener that will accept data from sensors implementing the CoAP protocol. CoAP is an Internet application protocol for constrained devices to send data over the internet, it is similar to HTTP but may be run over UDP or TCP and is considerably simplified to allow implementation in small footprint devices. CoAP stands for Constrained Application Protocol.

The plugin listens for POST requests to the URI defined in the configuration. It expects the content of this PUT request to be a CBOR payload which it will expand and create assets for the items read from the CBOR payload.

To create a south service with the CoAP plugin

- Click on *South* in the left hand menu bar
- Select *coap* from the plugin list
- Name your service and click *Next*

The screenshot shows a configuration window for the CoAP plugin. At the top, a progress bar indicates three steps: 1. Plugin & Service Name, 2. Review Configuration (the current step, highlighted with a green circle), and 3. Done. Below the progress bar, there are two input fields: 'Port' with the value '5683' and 'URI' with the value 'sensor-values'. At the bottom, there are two buttons: 'Previous' and 'Next'.

- Configure the plugin
 - **Port:** The port on which the CoAP plugin will listen
 - **URI:** The URI the plugin expects to receive POST requests
- Click *Next*
- Enable the service and click on *Done*

14.1.6 Simple CSV Plugin

The *flir-south-csv* plugin is a simple plugin for reading comma separated variable files and injecting them as if there were sensor data. There are a number of variants of plugin that support this functionality with varying degrees of sophistication. These may also be considered as simple examples of how to write plugin code.

This particular CSV reader supports single or multi-column CSV files, without timestamps in the file. It assumes every value is a data value. If the multi-column option is not set then it will read data from the file up until a newline or a comma character and make that as single data point in an asset and return that.

If the multi-column option is selected then each column in the CSV file becomes a data point within a single asset. It is assumed that every row of the CSV file will have the same number of values.

Upon reaching the end of the file the plugin will restart sending data from the beginning of the file.

To create a south service with the csv plugin

- Click on *South* in the left hand menu bar
- Select *Csv* from the plugin list
- Name your service and click *Next*

The screenshot shows a configuration window for the CSV plugin. At the top, a progress bar indicates three steps: 1. Plugin & Service Name, 2. Review Configuration (highlighted with a green circle), and 3. Done. Below the progress bar, the configuration form contains the following fields:

- Asset Name:** A text box containing the value "Vibration".
- Datapoint:** A text box containing the value "ch".
- Multi-Column:** A checkbox that is currently unchecked.
- Path Of File:** An empty text box.

At the bottom of the form, there are two buttons: "Previous" (disabled) and "Next" (active/blue).

- Configure the plugin
 - **Asset Name:** The name of the asset that will be created
 - **Datapoint:** The name of the data point to insert. If multi-column is selected this becomes the prefix of the name, with the column number appended to create the full name
 - **Multi-Column:** If selected then each row of the CSV file is treated as a single asset with each column becoming a data point within that asset.
 - **Path Of File:** The file that should be read by the CSV plugin, this may be any location within the host operating system. The FLIR Bridge process should have permission to read this file.
- Click *Next*
- Enable the service and click on *Done*

14.1.7 CSV Playback

The plugin plays a csv file inside some given directory in file system (The default being FLIR_ROOT/data). It converts the columns of csv file into readings which are datapoints of an output asset. The plugin plays readings at some configured rate.

We can also convert the columns of csv file into some other data type. For example from float to integer. The converted data will be part of reading not the CSV file.

The plugin has the ability to play the readings in either burst or continuous mode. In burst mode all readings are ingested into database at once and there is no adjustment of timestamp of a single reading. Whereas in continuous mode readings are ingested one by one and the timestamp of each reading is adjusted according to sampling rate. (For example if sampling rate is 8000 then the user_ts of every reading differs by 125 micro seconds.)

We can also copy the timestamp if present in the CSV file. This time stamp becomes the user_ts of a reading.

The plugin can also play the file in a loop which means it can start again if end of the file has reached.

The plugin can also play a file that has variable columns in every line.

The screenshot shows a configuration window for the CSV Playback plugin, specifically the 'Review Configuration' step (indicated by a green circle with the number 2). The window has a progress bar at the top with three steps: 1. Plugin & Service Name, 2. Review Configuration, and 3. Done. The configuration fields are as follows:

- Asset name:** vibration
- CSV directory name:** FLEDGE_DATA
- CSV file pattern:** (empty field)
- Header processing method:** do_not_skip (dropdown menu)
- Data point for header rows:** metadata
- Number of rows to skip or pass in datapoint:** 1 (spinner)
- Dynamic columns:** ☐
- Column processing method:** pick_from_file (dropdown menu)
- Auto generate prefix:** column
- Column names and data types for explicit:** (empty field)

- **‘assetName’:** type: string default: **‘vibration’**: The output asset that contains the readings.
- **‘csvDirName’:** type: string default: **‘FLIR_DATA’**: The directory where CSV file exists. Default is FLIR_DATA or FLIR_ROOT/data
- **‘csvFileName’:** type: string default: **‘’**: CSV file name or pattern to search inside directory. Not necessarily an exact file name. If there are multiple files matching with the pattern, then the plugin will pick the first file in alphabetical order. If postProcessMethod is rename or delete then it will rename or delete the played file and pick the next one and so on.
- **‘headerMethod’:** type: enumeration default: **‘do_not_skip’**: The method for processing the header of csv file.
 1. skip_rows : If this is selected then the plugin will skip a given number of rows. The number of rows should be given in noOfRows config parameter given below.

2. `pass_in_datapoint` : If this is selected then the given number of rows will be combined into a string. This string will be present inside some given datapoint. Useful in cases where we want to ingest meta data along with readings from the csv file.
 3. `do_not_skip`: This option will not take any action on the header.
- **‘datapointForCombine’**: **type: string default: ‘metadata’**: If header method is `pass_in_datapoint` then it is the datapoint name where the given number of rows will get combined.
 - **‘noOfRows’**: **type: integer default: ‘1’**: No. of rows to skip or combine to single value. Used when header-Method is either `skip_rows` or `pass_in_datapoint`.
 - **‘variableCols’**: **type: boolean default: ‘false’**: It should be set true, when the columns in every row of CSV are not fixed. For example If you have a file like this

```
a,b,c
2,3,,23
4
```

Then you should set it true.

Note: Only one reading will be ingested at a time in this case. If you want to increase the rate then increase `readingPerSec` parameter in advanced plugin configuration.

- **‘columnMethod’**: **type: enumeration default: ‘pick_from_file’**: If variable Columns is false then it indicates how columns are considered.
 1. `pick_from_file` : The columns will be picked using a row index given.
 2. `explicit` : Specify the columns inside `useColumns` parameter.
- **‘autoGeneratePrefix’**: **type: string default: ‘column’**: If variable Columns is set true then data points will be generated using the prefix. For example if there is row like this 1,,2 and we chose `autoGeneratePrefix` to be `column`, then we will get data points like this `column_1: 1, column_3: 2`. Empty values will be ignored.
- **‘useColumns’**: **type: string default: ‘:’**: Format **column1:type,column2:type**

The data types supported are: int, float, str, datetime, bool

We can perform three tasks with this config parameter.

1. The column name will get renamed in the reading if different name is used other than present in CSV file.
2. We can select a subset of columns from total columns.
3. We can convert the data type of each column.

Example if the file is like the following

```
id,value,status
1,2.5,'OK'
2,2.7,'OK'
```

Then we can give

1. `id:int,temperature:float,status:str`

The column value will be renamed to temperature.

2. `id:int,value:float`

Only two columns will be selected here.

3. id:int,temperature:int,status:str

The data type will be converted to integer. Also column will be renamed.

Row index for column names	<input type="text" value="0"/>
Ingest mode	<input type="button" value="burst"/>
Sample rate	<input type="text" value="8000"/>
Burst interval (ms)	<input type="text" value="1000"/>
Timestamp processing mode	<input type="button" value="current time"/>
Timestamp column name	<input type="text"/>
Timestamp format	<input type="text" value="%Y-%m-%d %H:%M:%S.%f%z"/>
Ignore or report for NaN	<input type="button" value="ignore"/>
Post process method	<input type="button" value="continue_playing"/>
Suffix name	<input type="text" value=".tmp"/>

- **‘rowIndexForColumnNames’**: **type: integer default: ‘0’**: If column method is pick_from_file then it is the index from where column names are taken.
- **‘ingestMode’**: **type: enumeration default: ‘burst’**: Burst or continuous mode for ingestion.
- **‘sampleRate’**: **type: integer default: ‘8000’**: No of readings per second to ingest.
- **‘burstInterval’**: **type: integer default: ‘1000’**: Used for burst mode. Time interval between consecutive bursts in milliseconds.
- **‘timestampStyle’**: **type: enumeration default: ‘current time’**: Controls how to give timestamps to reading. Works in four ways:
 1. current time: The timestamp in the readings is whatever the local time in the machine.
 2. copy csv value: Copy the timestamp present in the CSV file.
 3. move csv value: Used when we do not want to include timestamps from files in actual readings.
 4. use csv sample delta: Pick the delta between two readings in the file and construct the timestamp of reading using this delta. Assuming the delta remains constant through out the file.)
- **‘timestampCol’**: **type: string default: ‘’**: The timestamp column to pick from the file. Used only when timestampStyle is not ‘current time’.
- **‘timestampFormat’**: **type: string default: ‘%Y-%m-%d %H:%M:%S.%f%z’**: The timestamp format that will be used to parse the time stamps present in the file. Used only when timestampStyle is not ‘current time’.
- **‘ignoreNaN’**: **type: enumeration default: ignore**: Pandas takes the white spaces and missing values as NaN’s. These NaN’s cause problem while ingesting into database. It is left to the user to ensure there

are no missing values in CSV file. However if the option selected is report. Then plugin will check for NaN's and report error to user. This can serve as a way to check the CSV file for missing values. However the user has to take action on what to do with NaN values. The default action is to ignore them. When error is reported the user must delete the south service and try again with clean CSV file.

- **'postProcessMethod': type: enumeration default: 'continue_playing'**: It is the method to process the CSV file once all rows are ingested. It could be:
 1. continue_playing
Play the file again if finished.
 2. delete
Delete the played file once finished.
 3. rename
Rename the file with suffix after playing.
- **'suffixName': type: string default: '.tmp'**: The suffix name for renaming the file if postProcess method is rename.

Execution

Assuming you have a csv file named vibration.csv inside FLIR_ROOT/data/csv_data (Can give a pattern like vib. The plugin will search for all the files starting with vib and therefore find out the file named vibration.csv). The csv file has fixed number of columns per row. Also assuming the column names are present in the first line. The plugin will rename the file with suffix .tmp after playing. Here is the cURL command for that.

```
res=$(curl -sX POST http://localhost:8081/flir/service -d @- << EOF | jq '.')
{
  "name": "csv_player",
  "type": "south",
  "plugin": "csvplayback",
  "enabled": false,
  "config": {
    "assetName": {"value": "My_csv_asset"},
    "csvDirName": {"value": "FLIR_DATA/csv_data"},
    "csvFileName": {"value": "vib"},
    "headerMethod": {"value": "do_not_skip"},
    "variableCols": {"value": "false"},
    "columnMethod": {"value": "pick_from_file"},
    "rowIndexForColumnNames": {"value": "0"},
    "ingestMode": {"value": "burst"},
    "sampleRate": {"value": "8000"},
    "postProcessMethod": {"value": "rename"},
    "suffixName": {"value": ".tmp"}
  }
}
EOF
)

echo $res
```

Poll Vs Async

The plugin also works in async mode. Though the default mode is poll. The async mode is faster but suffers with memory growth when sample rate is too high for the machine configuration.

Use the following sed operation for async and start the plugin again. The second sed operation, in similar way, can be used if you want to revert back to poll mode. Restart for the plugin service is required.

```
plugin_path=$FLIR_ROOT/python/flir/plugins/south/csvplayback/csvplayback.py
value='s/POLL_MODE=True/POLL_MODE=False/'
sudo sed -i $value $plugin_path

# for reverting back to poll the commands will be
plugin_path=$FLIR_ROOT/python/flir/plugins/south/csvplayback/csvplayback.py
value='s/POLL_MODE=False/POLL_MODE=True/'
sudo sed -i $value $plugin_path
```

Behaviour under various modes

Table 1: Behaviour of CSV playback plugin

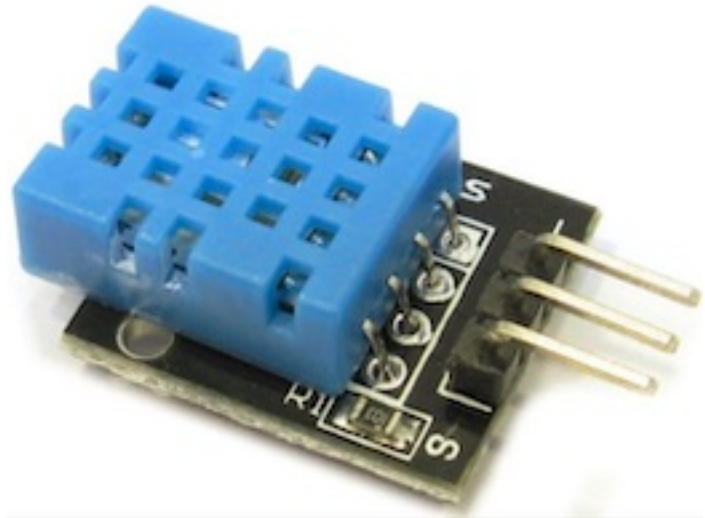
Plugin mode	Ingest mode	Behaviour
poll	burst	No memory growth. Resembles the way sensors give data in real life. However the timestamps of readings won't differ by a fixed delta.
poll	continuous	No memory growth. Readings differ by a constant delta. However it is slow in performance.
async	continuous	Similar to poll continuous but faster. However memory growth is observed over time.
async	burst	Similar to poll burst. Not used generally.

For using poll mode in continuous setting increase the readingPerSec category to the sample rate.

```
sampling_rate=8000
curl -sX PUT http://localhost:8081/flir/category/csv_playerAdvanced -d '{
  ↪ "bufferThreshold": "'$sampling_rate'", "readingsPerSec": "'$sampling_rate'" }' | jq
```

It is advisable to increase the buffer threshold to atleast half the sample rate for good performance. (As done in above command)

14.1.8 DHT11 (C version)



The *flir-south-dht* plugin implements a temperature and humidity sensor using the DHT11 sensor module. Two versions of plugins for the DHT11 are available and are used as the example for . The other DHT11 plugin is *flir-south-dht11* and is a .

The DHT11 and the associated DHT22 sensors may be used, however they have slightly different characteristics;

	DHT11	DHT22
Voltage	3 to 5 Volts	3 to 5 Volts
Current	2.5mA	2.5mA
Humidity Range	0-50 % humidity 5% accuracy	0-100% humidity 2.5% accuracy
Temperature Range	0-50 +/- 2 degrees C	-40 to 80 +/- 0.5 degrees C
Sampling Frequency	1Hz	0.5Hz

Note: Due to the requirement for attaching to GPIO pins this plugin is only available for the Raspberry Pi platform.

To create a south service with the DHT11 plugin

- Click on *South* in the left hand menu bar
- Select *dht11_V2* from the plugin list
- Name your service and click *Next*

1 Plugin & Service Name 2 Review Configuration 3 Done

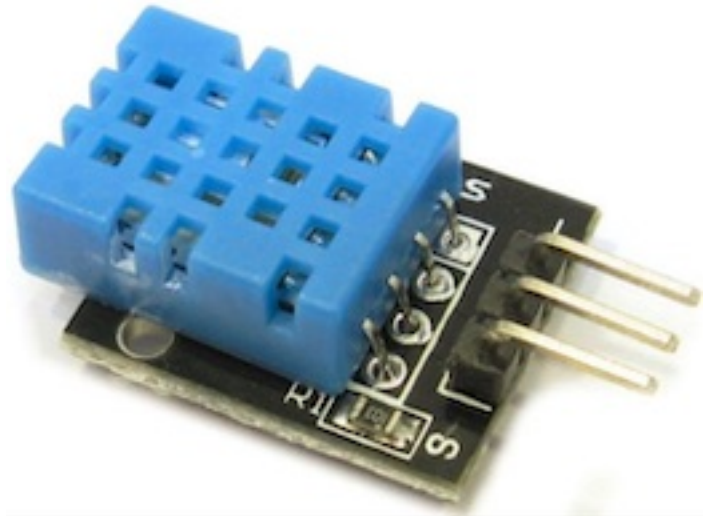
Asset Name: dht11

Rpi Pin: 7

Previous Next

- Configure the plugin
 - **Asset Name:** The asset name which will be used for all data read.
 - **Rpi Pin:** The GPIO pin on the Raspberry Pi to which the DHT11 serial pin is connected.
- Click *Next*
- Enable the service and click on *Done*

14.1.9 DHT11 (Python version)



The *flir-south-dht11* plugin implements a temperature and humidity sensor using the DHT11 sensor module. Two versions of plugins for the DHT11 are available and are used as the example for . The other DHT11 plugin is *flir-south-dht* and is a .

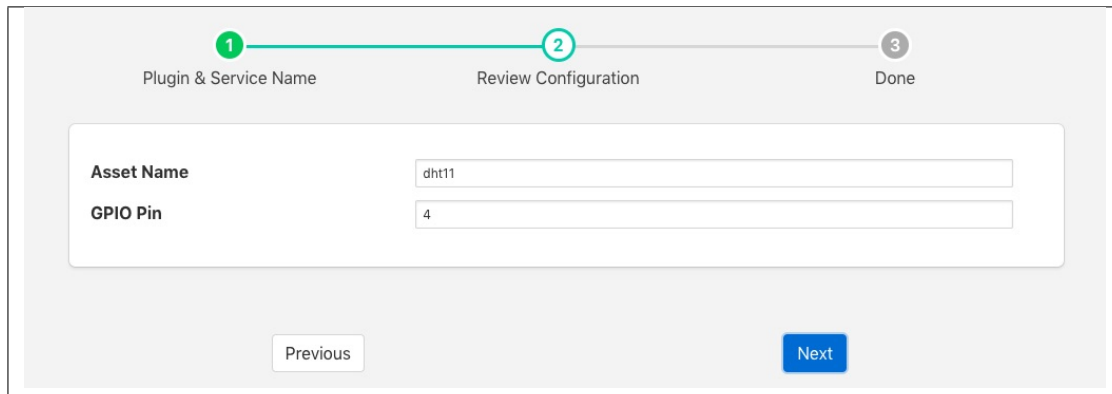
The DHT11 and the associated DHT22 sensors may be used, however they have slightly different characteristics;

	DHT11	DHT22
Voltage	3 to 5 Volts	3 to 5 Volts
Current	2.5mA	2.5mA
Humidity Range	0-50 % humidity 5% accuracy	0-100% humidity 2.5% accuracy
Temperature Range	0-50 +/- 2 degrees C	-40 to 80 +/- 0.5 degrees C
Sampling Frequency	1Hz	0.5Hz

Note: Due to the requirement for attaching to GPIO pins this plugin is only available for the Raspberry Pi platform.

To create a south service with the DHT11 plugin

- Click on *South* in the left hand menu bar
- Select *dht11* from the plugin list
- Name your service and click *Next*



- Configure the plugin
 - **Asset Name:** The asset name which will be used for all data read.
 - **GPIO Pin:** The GPIO pin on the Raspberry Pi to which the DHT11 serial pin is connected.
- Click *Next*
- Enable the service and click on *Done*

14.1.10 Digiducer Vibration Sensor



The *flir-south-digiducer* plugin allows a 333D01 USB Digital Accelerometer to be attached to FLIR Bridge for the collection of vibration data. The Digiducer is a piezoelectric accelerometer housed in a rugged enclosure complete with a data conditioning and acquisition interface that only requires a USB port on the FLIR Bridge device for connectivity.

The plugin allows for two modes of operation; continuous reading of the vibration data or sampled reading of the vibration data. In sampled mode the user configures a sample period and interval. The plugin will then read data for the sample period and forward it to the FLIR Bridge storage service. It will then pause collection for the sample interval before again collecting data. This repeats indefinitely.

To create a south service with the Digiducer

- Click on *South* in the left hand menu bar
- Select *digiducer* from the plugin list
- Name your service and click *Next*

- Configure the plugin
 - **Asset Name:** The name of the asset that will be created in FLIR Bridge.
 - **Sample Rate:** The rate at which data will be sampled. A number of frequencies are supported in the range 8KHz to 48KHz.

- **Block size:** To aid efficiency the plugin collects data in blocks, this allows the block size to be tuned. The value should be a power of 2.
- **Continuous Sampling:** This toggle supports the selection of continuous verses sampled collection.
- **Sample Period:** The duration of each sample period in seconds.
- **Sample Interval:** The time in seconds between each sample being taken.
- **Channel:** Select collection of the 10G Peak channel, the 20G Peak channel or both channels

Channel

✓ 20 g pk

10 g pk

both

- Click on *Next*
- Enable your south service and click on *Done*

14.1.11 DNP3 Master Plugin

The *flir-south-dnp3* allows FLIR Bridge to act as a DNP3 master and gather data from a DNP3 Out Station. The plugin will fetch all data types from the DNP3 Out Station and create assets for each in FLIR Bridge. The DNP3 plugin also handles unsolicited messages transmitted by the outstation.

1

2

3

Plugin & Service Name

Review Configuration

Done

Asset Name prefix

dnp3_

Master link Id

1

Outstation address

127.0.0.1

Outstation port

20000

Outstation link Id

10

Data scan

☐

Scan interval

30

Network timeout

5

Previous

Next

- **Asset Name prefix:** An asset name prefix that is prepended to the DNP3 objects retrieved from the DNP3 outstations to create the FLIR Bridge asset name.
- **Master link id:** The master link id FLIR Bridge uses when implementing the DNP3 protocol.
- **Outstation address:** The IP address of the DNP3 Out Station to be connected.
- **Outstation port:** The port on the Out Station to which the connection is established.

- **Outstation link Id:** The Out Station link id.
- **Data scan:** Enable or disable the scanning of all objects and values in the Out Station. This is the Integrity Poll for all Classes.
- **Scan interval:** The interval between data scans of the Out Station.
- **Network timeout:** Timeout for fetching data from the Out Station expressed in seconds.

DNP3 Out Station Testing

The opendnp3 package contains a demo Out Station that can be used for test purposes. After building the opendnp3 package on your machine run the demo program as follows;

```
$ cd opendnp3/build
$ ./outstation-demo
```

This demo application listens on any IP address, port 20001 and has link Id set to 10. It also assumes master link Id is 1. Configuring your FLIR Bridge plugin with these parameters should allow FLIR Bridge to connect to this Out Station.

Once started it logs traffic and waits for use input to send unsolicited messages:

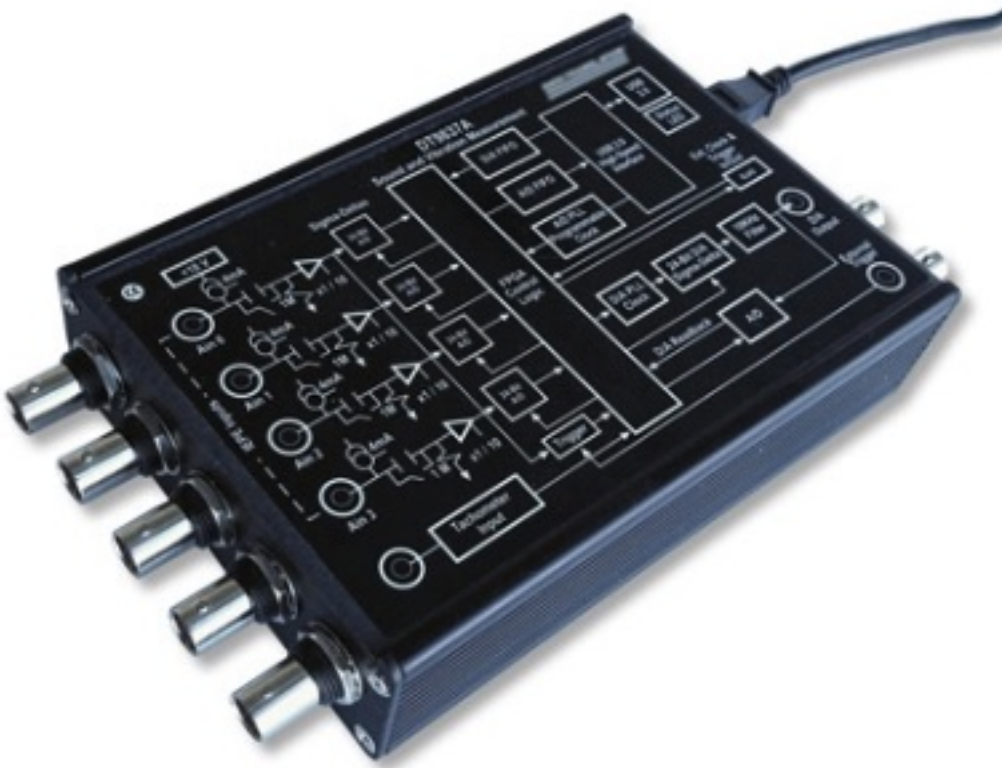
```
Enter one or more measurement changes then press <enter>
c = counter, b = binary, d = doublebit, a = analog, o = octet string, 'quit' = exit
```

Another option is the use of a DNP3 Out Station simulator, as an example:

<http://freyrscada.com/dnp3-ieee-1815-Client-Simulator.php#Download-DNP3-Development-Bundle>

Once the bundle has been downloaded, the **DNPOutstationSimulator.exe** application under the “Simulator” folder can be installed and run on a Windows 32bit platform.

14.1.12 Data Translation DT9837 Series



The *flir-south-dt9837* plugin is a south plugin that is designed to gather data from a Data Translation DT9873 Series DAQ.

To create a south service with the DT9837

- Click on *South* in the left hand menu bar
- Select *dt9837* from the plugin list
- Name your service and click *Next*

1 Plugin & Service Name 2 Review Configuration 3 Done

Asset Name

Scan Rate

Input Mode

Range

First Channel

Last Channel

Sensitivity

IEPE Ch. 0 ☐

IEPE Ch. 1 ☐

IEPE Ch. 2 ☐

IEPE Ch. 3 ☐

Coupling Ch. 0

Coupling Ch. 1

Coupling Ch. 2

Coupling Ch. 3

- Configure the plugin
 - **Asset Name:** The name of the asset that will be created with the values read from the DT9837
 - **Scan Rate:** The rate at which each channel is read. This may be expressed as a numeric value, in which case it is the number of samples per second, or it may be expressed in KHz or MHz.
 - **Input Mode:** Defines how the input is treated, it may be either a differential pair or a single ended value with a reference ground.

Scan Rate

Input Mode

Range

- **Range:** This defines the voltage range for all channels. It may be defined as a bipolar value, in which case it is expected the signal can swing between + and - the specified voltage. A uni-polar

value, in which case the voltage swing is between ground and the specified voltage. Or it is a 0 to 20mA current loop.

Range

First Channel

Last Channel

Sensitivity

IEPE Ch. 0

IEPE Ch. 1

IEPE Ch. 2

IEPE Ch. 3

Coupling Ch. 0

Coupling Ch. 1

Coupling Ch. 2

Coupling Ch. 3

✓ BiPolar 60 Volts

BiPolar 30 Volts

BiPolar 20 Volts

BiPolar 15 Volts

BiPolar 10 Volts

BiPolar 5 Volts

BiPolar 4 Volts

BiPolar 3 Volts

BiPolar 2.5V Volts

BiPolar 2 Volts

BiPolar 1.25 Volts

BiPolar 1 Volt

BiPolar 625 mVolts

BiPolar 500 mVolts

BiPolar 250 mVolts

BiPolar 312 mVolts

BiPolar 200 mVolts

BiPolar 156 mVlt

BiPolar 125 mVolts

BiPolar 100 mVolts

BiPolar 78 mVolts

BiPolar 50 mVolts

BiPolar 10 mVolts

BiPolar 5 mVolts

60 Volts

30 Volts

20 Volts

15 Volts

10 Volts

5 Volts

4 Volts

2.5 Volts

2 Volts

1.25 Volts

1 Volt

625 mVolts

500 mVolts

250 mVolts

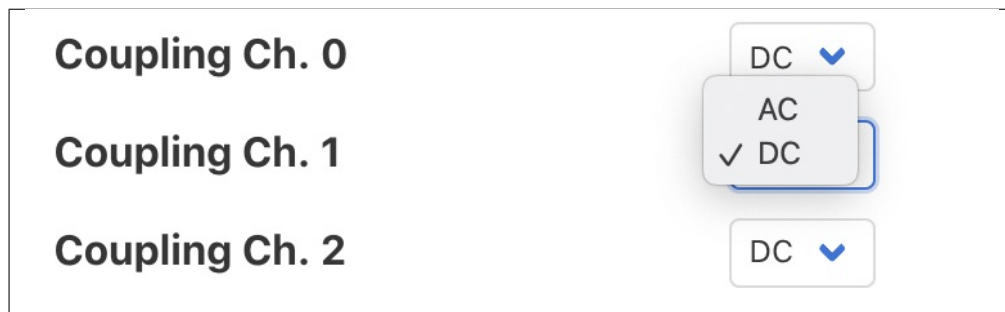
200 mVolts

125 mVolts

▼

Previous

- **First Channel:** The DT9837 can scan a number of channel in a single operation, these must however be adjunct channels. This option sets the lowest numbered channel to be scanned.
- **Last Channel:** The DT9837 can scan a number of channel in a single operation, these must however be adjunct channels. This option sets the highest numbered channel to be scanned.
- **Sensitivity:** This sets the sensor sensitivity for IEPE sensors attached to any of the channels.
- **IEPE Ch. 0:** Specifies that a IEPE compatible sensor is attached to channel 0.
- **IEPE Ch. 1:** Specifies that a IEPE compatible sensor is attached to channel 1.
- **IEPE Ch. 2:** Specifies that a IEPE compatible sensor is attached to channel 2.
- **IEPE Ch. 3:** Specifies that a IEPE compatible sensor is attached to channel 3.
- **Coupling Ch. 0:** Specifies the input coupling to use for channel 0. This setting has no effect if the channel has been setup for IEPE as IEPE always uses AC coupling.



- **Coupling Ch. 1:** Specifies the input coupling to use for channel 1. This setting has no effect if the channel has been setup for IEPE as IEPE always uses AC coupling.
 - **Coupling Ch. 2:** Specifies the input coupling to use for channel 2. This setting has no effect if the channel has been setup for IEPE as IEPE always uses AC coupling.
 - **Coupling Ch. 3:** Specifies the input coupling to use for channel 3. This setting has no effect if the channel has been setup for IEPE as IEPE always uses AC coupling.
- Click on *Next*
 - Enable your service and click on *Done*

14.1.13 Edge ML Plugin

The plugin takes a video frame from a camera or stream , sends that to edgectl cluster running somewhere else. The Edge ML cluster returns a response in the form of json which contains information about detected objects, their bounding boxes and confidence score. This information is overlayed on the frame and saved onto disk in the form of images. The results are also streamed on a browser.

1
Plugin & Service Name

2
Review Configuration

3
Done

?

Source of Data Generation	<div style="border: 1px solid #ccc; padding: 2px 5px;">stream ▼</div>
Directory Name	<div style="border: 1px solid #ccc; padding: 2px 5px;">Directory For Using Images</div>
Camera ID	<div style="border: 1px solid #ccc; padding: 2px 5px;">0</div>
RTSP URL	<div style="border: 1px solid #ccc; padding: 2px 5px;">rtsp://localhost:8554/clip</div>
Frames Per Minute	<div style="border: 1px solid #ccc; padding: 2px 5px;">1000</div>
Inference Choice	<div style="border: 1px solid #ccc; padding: 2px 5px;">k8 ▼</div>
Deployment Name	<div style="border: 1px solid #ccc; padding: 2px 5px;">mledge-deployment</div>
The URL to post the images.	<div style="border: 1px solid #ccc; padding: 2px 5px;">http://localhost:30163/v1/vision/detection</div>

- **‘source’:** type: enumeration default: **‘stream’**: Source of data being generated. Could be camera if camera is attached, stream if rtsp stream is to be used and directory if we have a directory of images.
- **‘sourceDirName’:** type: string default: **‘Directory For Using Images’**: If source is directory then the directory which contains images.
- **‘cameraId’:** type: integer default: **0**: If camera is to be used then enter the device id of camera. If you use 0 then the following command should be successful.

```
v4l2-ctl -list-formats-ext -device /dev/video0 .
```

In case you dont get output use camera id 1, 2 and so on.
- **‘rtspUrl’:** type: string default: **rtsp://localhost:8554/clip**: If source is stream, then enter the url of the rtsp stream.
- **‘fpm’:** type: integer default: **‘1000’**: No of frames to process in a minute.
- **‘inferenceChoice’:** type: enumeration default: **‘k8’**: If the edgeml cluster is running inside microk8’s on the same machine then use k8 or use URL if you want to send inference request to some other machine or some k8 cluster other than microk8’s.
- **‘deploymentName’:** type: string default: **‘mledge-deployment’**: If inferenceChoice is k8 then the name of the deployment inside microk8’s. The plugin will pick the ip and port from the deployment name itself.
- **‘restUrl’:** type: string default: **http://localhost:30163/v1/vision/detection**: If inferenceChoice is URL, then the URL where the post request will be sent.

Stream Results	<input checked="" type="checkbox"/>
Stream Port	<input type="text" value="8085"/>
Output Asset	<input type="text" value="Detection Results"/>
Destination Directory	<input type="text" value="detection"/>
Rotate after	<input type="text" value="120"/>
Rotation Data Amount	<input type="text" value="10"/>

Previous
Next

- **‘streamResults’**: type: boolean default: **‘true’**: Whether to stream detection results over HTTP(s)
- **‘streamPort’**: type: integer default: **‘8085’**: The port over which we can display detection results in browser.
- **‘outputAsset’**: type: string default: **‘Detected Results’**: The name of asset which contains detected results.
- **‘destinationDir’**: type: string default: **‘detection’**: The directory where resultant images will be stored.
- **‘rotateAfterMinutes’**: type: integer default: **‘120’**: The amount of time (in minutes) after which source images (with bounding boxes) are deleted”.
- **‘rotateDataMinutes’**: type: integer default: **‘10’**: The amount of jpeg files (with bounding boxes) in minutes to be rotated.

Installation

Part 1: Get the video feed

There are two ways to get the video feed.

1. Camera

To see the supported configuration of the camera run the following command.

```
$ v4l2-ctl --list-formats-ext --device /dev/video0
You will see something like
'YUYV' (YUYV 4:2:2)
  Size: Discrete 640x480
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 720x480
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 1280x720
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 1920x1080
    Interval: Discrete 0.067s (15.000 fps)
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 2592x1944
    Interval: Discrete 0.067s (15.000 fps)
  Size: Discrete 0x0
```

Above example uses Camera ID 0 to indicate use of /dev/video0 device, please use the applicable value for your setup

2. Network RTSP stream

To create a network stream follow the following steps

1. Install vlc

```
$ sudo add-apt-repository ppa:videolan/master-daily
$ sudo apt update
$ apt show vlc
$ sudo apt install vlc qtwayland5
$ sudo apt install libavcodec-extra
```

2. Download some sample files from here.

```
$ git clone https://github.com/intel-iot-devkit/sample-videos.git
```

3. Either stream a file using the following

```
$ vlc <name_of_file>.mp4 --sout '#gather:transcode{vcodec=h264,vb=512,
↪scale=Auto,width=640,height=480,acodec=none,scodec=none}:rtp{sdp=rtsp://
↪<ip_of_machine_streaming>:8554/clip}' --no-sout-all --sout-keep --loop --
↪no-sout-audio --sout-x264-profile=baseline
```

Note : fill the <ip_of_the_machine> with ip of the machine which will be used to stream video. Also fill <name_of_file> with the name of mp4 file.

4. You can also stream from a camera using the following

```
$ vlc v4l2:///dev/video<index_of_video_device> --sout '#gather:transcode
↪{vcodec=h264,vb=512,scale=Auto,width=<supported_width_of_camera_image>,
↪height=<supported_height_of_camera_image>,acodec=none,scodec=none}:rtp
↪{sdp=rtsp://<ip_of_the_machine>:8554/clip}' --no-sout-all --sout-keep -
↪no-sout-audio --sout-x264-profile=baseline
```

Fill the following :

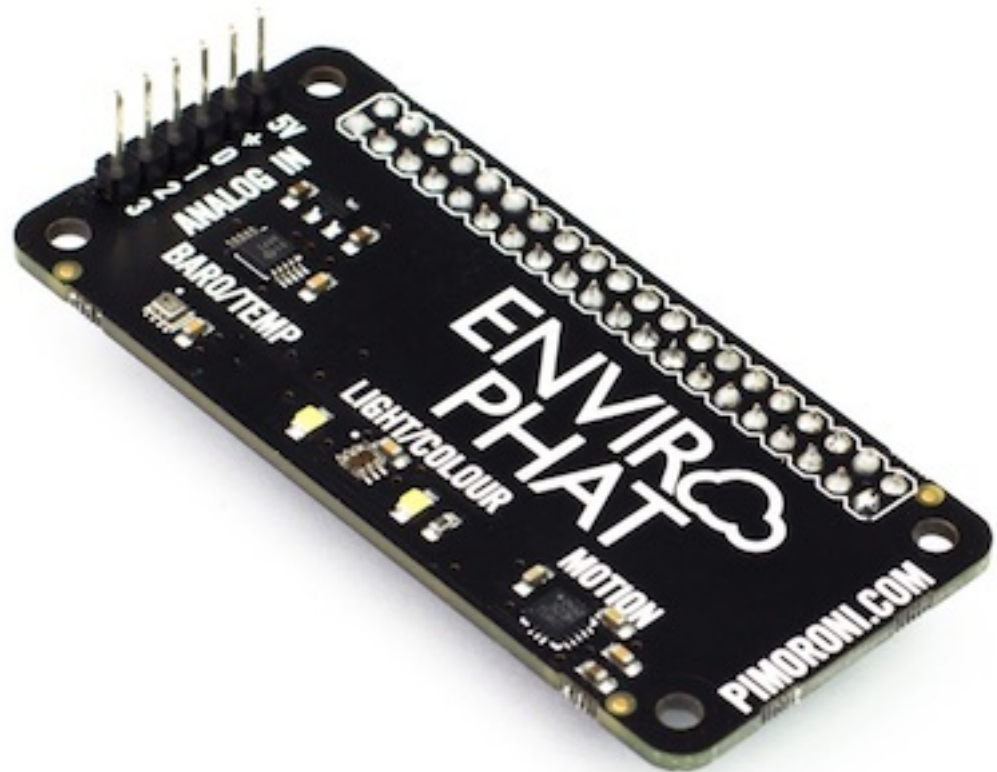
- <index_of_video_device> The index with which you ran the v4l2 command mentioned above. for example video0.
- <supported_height_of_camera_image> Height you get when you ran v4l2 command mentioned above. For example Discrete 640x480. Here 480 is height.
- <supported_width_of_camera_image> Width you get when you ran v4l2 command mentioned above. For example Discrete 640x480. Here 640 is width.
- <ip_of_the_machine> ip of the machine which will be used to stream video.

Part 2: Start the Edge ML cluster

For starting the Edge ML cluster you should follow this [README](#) file.

Now run the plugin by filling parameters according to your setup.

14.1.14 Enviro pHAT Plugin



The *flir-south-enviophat* is a plugin that uses the Pimoroni Enviro pHAT sensor board. The Enviro pHAT board is an environmental sensing board populated with multiple sensors, the plugin pulls data from the;

- RGB light sensor
- Magnetometer
- Accelerometer
- Temperature/pressure Sensor

Individual sensors can be enabled or disabled separately in the configuration. Separate assets are created for each sensor within FLIR Bridge with individual controls over the naming of these assets.

Note: The Enviro pHAT plugin is only available on the Raspberry Pi as it is specific the GPIO pins of that device.

To create a south service with the Enviro pHAT

- Click on *South* in the left hand menu bar
- Select *enviophat* from the plugin list
- Name your service and click *Next*

1 Plugin & Service Name 2 Review Configuration 3 Done

Asset Name Prefix

RGB Sensor ☒

RGB Sensor Name

Magnetometer Sensor ☒

Magnetometer Sensor Name

Accelerometer Sensor ☒

Accelerometer Sensor Name

Weather Sensor ☒

Weather Sensor Name

- Configure the plugin
 - **Asset Name Prefix:** An optional prefix to add to the asset names. The asset names created by the plugin are; rgb, magnetometer, accelerometer and weather. Using the prefix you can add an identifier to the front of each such that it becomes easier to differentiate between multiple instances of the sensor.
 - **RGB Sensor:** A toggle control to turn on or off collection of RGB light level information
 - **RGB Sensor Name:** Set a name for the RGB sensor asset
 - **Magnetometer Sensor:** A toggle control to turn on or off collection of magnetometer data
 - **Magnetometer Sensor Name:** Set a name for the magnetometer sensor asset
 - **Accelerometer Sensor:** A toggle to turn on or off collection of accelerometer data
 - **Accelerometer Sensor Name:** Set a name for the accelerometer sensor asset
 - **Weather Sensor:** A toggle to turn on or off collection of weather data
 - **Weather Sensor Name:** Set a name for the weather sensor asset
- Click *Next*
- Enable the service and click on *Done*

14.1.15 Expression South Plugin

The *flir-south-expression* plugin is a plugin that is used to generate synthetic data using a mathematical expression to generate data that changes over time. The user may configure the plugin with an expression of their choice and define a period in terms of samples put period of the output and the increment between each sample.

1 Plugin & Service Name 2 Review Configuration 3 Done

Asset Name	Expression
Expression	<code>clamp(-1.0, sin(2 * pi * x) + cos(x / 2 * pi), +1.0)</code>
Minimum Value	-5
Maximum Value	5
Step Value	0.001

Previous Next

The parameters that can be configured are;

- **Asset Name:** The name of the asset to be created inside FLIR Bridge.
- **Expression:** The expression that should be evaluated to create the asset value, see below.
- **Minimum Value:** The minimum value of x , where x is the value that sweeps over time.
- **Maximum Value:** The maximum value of x , where x is the value that sweeps over time.
- **Step Value:** The step in x for each call to the expression evaluation.

Expression Support

The *flir-south-expression* plugin makes use of the library to do run time expression evaluation. This library provides a rich mathematical operator set, the most useful of these in the context of this plugin are;

- Mathematical operators (+, -, *, /, %, ^)
- Functions (min, max, avg, sum, abs, ceil, floor, round, roundn, exp, log, log10, logn, pow, root, sqrt, clamp, inrange, swap)
- Trigonometry (sin, cos, tan, acos, asin, atan, atan2, cosh, cot, csc, sec, sinh, tanh, d2r, r2d, d2g, g2d, hyp)

14.1.16 Flir Thermal Imaging Cameras

The *flir-south-Flir* plugin is a south plugin that reads temperature data from a Flir thermal imaging device.

Using the Plugin

To create a south service with the Flir plugin

- Click on *South* in the left hand menu bar
- Select *Flir* from the plugin list
- Name your service and click *Next*

- Configure the plugin
 - **Asset Name:** The name of the asset to use.
 - **Discovered Cameras:** A drop down list of all the cameras discovered on the local network. Select the camera you wish to use of *Manual* if the camera does not appear in the list.

- **Address:** The IP address of the Flir device. Use this field to enter a manual camera address if your camera does not appear in the list of discovered cameras. This field is only activated if the *Manual* option is selected in the list of discovered cameras.

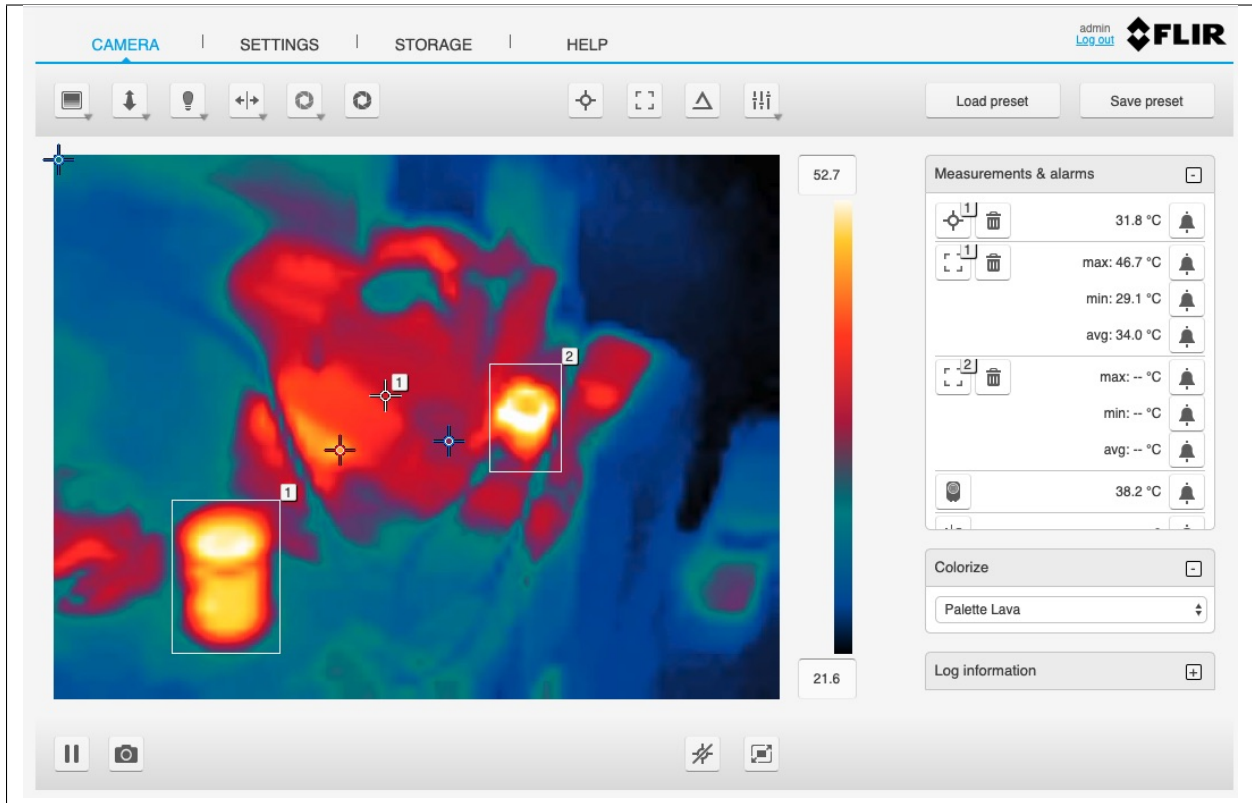
- **Area Labels:** The labels to use for each of the numbered areas defined within the device. Simply replace the number of your chosen area with the name to assign to that area in the asset data read

14.1.17 Flir AX8 Thermal Imaging Camera



The *flir-south-FlirAX8* plugin is a south plugin that enables temperature data to be collected from Flir Thermal Imaging Devices, in particular the AX8 and other A Series cameras. The camera provides a number of temperatures for both spots and boxes defined within the field of view of the camera. In addition it can also provide deltas between two temperature readings.

The bounding boxes and spots to read are configured by connecting to the web interface of the camera and dropping the spots on a thermal imaging or pulling out rectangles for the bounding boxes. The camera will return a minimum, maximum and average temperature within each bounding box.



In order to configure a south service to obtain temperature data from a Flir camera select the *South* option from the left-hand menu bar and click on the Add icon in the top right corner of the South page that appears. Select the *FlirAX8* plugin from the list of south plugins, name your service and click on *Next*.

The screen that appears is the configuration screen for the *FlirAX8* plugin.

The screenshot shows the configuration screen for the FlirAX8 plugin. At the top, there's a progress bar with three steps: 1. Plugin & Service Name, 2. Review Configuration, and 3. Done. The main area contains a form with the following fields:

- Asset Name:** AX8
- Server Address:** 192.168.0.48
- Port:** 502
- Slave ID:** 1

At the bottom, there are two buttons: 'Previous' and 'Next'.

There are four configuration parameters that can be set, usually it is only necessary to change the first two however;

- **Asset Name:** This is the asset name that the temperature data will be written to FLIR Bridge using. A single asset is used that will contain all of the values read from the camera.
- **Server Address:** This is the address of the Modbus server within the camera. This is the same IP address that is used to connect to the user interface of the camera.
- **Port:** The TCP port on which the cameras listens for Modbus requests. Unless changed in the camera the default port of 502 should be used.
- **Slave ID:** The Modbus Slave ID of the camera. By default the cameras are supplied with this set to 1, if changed within your camera setup you must also change the value here to match.

Once entered click on *Next*, enable the service on the next page and click on *Done*.

This will create a single asset that contains values for all boxes and spots that may be define. A filter *flir-filter-FlirValidity* can be added to the south service to remove data for boxes and spots not switched on in the camera user interface. See . This filter also allows you to name the boxes and hence have more meaningful names in the data points within the asset.

14.1.18 FLIR GW65 Vibration Sensors

The *flir-south-gw65* plugin provides a mechanism to connect the FLIR vibration sensors, via the GW65 gateway to FLIR Bridge. The plugin allows the GW65 to be used to connect sets of the SV87 vibration sensors. Raw vibration data is then collected from the sensors and may be process by one or more of the filters that offer vibration analysis.

The connection between the sensors and the plugin, via the GW65 gateway as established using the FLIR mobile application, before starting this process however you must install and configure the *flir-south-gw65* plugin. The plugin may be installed either by the user interface or by using the package manager of your Linux system to install it manually from a package.

You must also have an MQTT broker configured and running on your network. This should be configured to allow MQTTS and also have a username and password for the FLIR gateway to use.

Creating the GW65 South Service

Using the normal procedure for creating a new south service in FLIR Bridge,

- Select the *South* item from the menu on the left hand side of the screen
- Click on the *Add +* link on the top left of the South service screen
- In the list of available plugins choose the *gw65* entry, if it is not in the list click on the *available plugins* link and install it.
- Enter a name for your service and click on *Next*
- The configuration page will appear

The screenshot shows a configuration wizard with three steps: 1. Plugin & Service Name, 2. Review Configuration (active), and 3. Done. In the 'Review Configuration' step, there are two input fields: 'Asset Name' containing 'gw65' and 'MQTT Broker' containing 'localhost'. A 'Previous' button is located at the bottom left, and a 'Next' button is at the bottom right.


- **Asset Name:** The asset name to use as a fallback asset. This is normally unused.
- **MQTT Broker:** The address of the MQTT broker that will be used to communicate with the GW65 gateway.
- Enter the address of your MQTT broker, if you running the broker on the same machine as FLIR Bridge then you may use the default of 127.0.0.1.
- Click on *Next*
- Enable the service and click on *Done*




You may now proceed to configure the GW65 using the FLIR mobile application.



- Open the FLIR application and select the *LOCAL SERVER* option.



- Click on the *Add Server* link

14:59 

 **Add Server** 

To configure, please enter the Server IP,
User Name, Password and Port.

Server IP

192.168.0.34

User name

flir

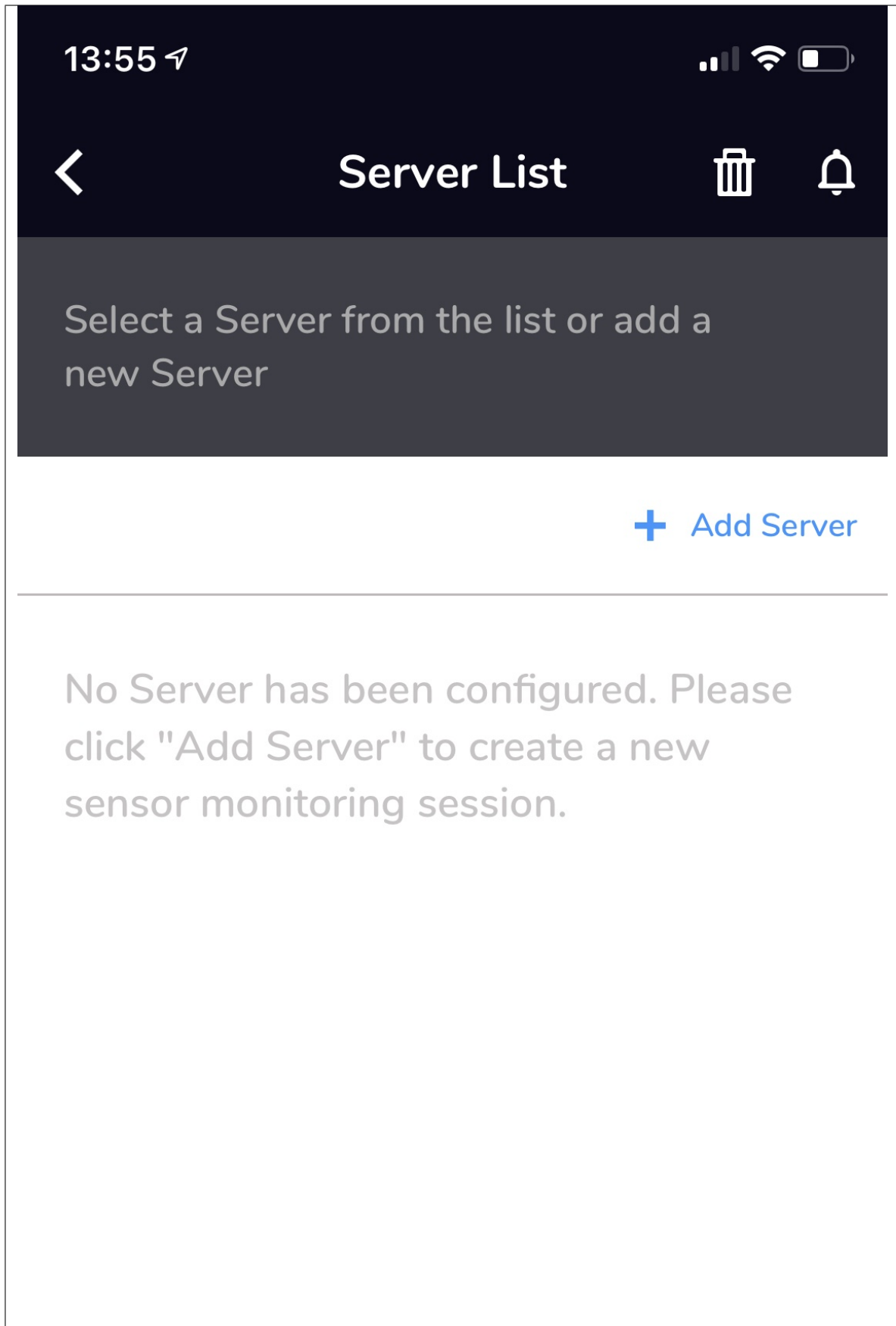
Password

123456

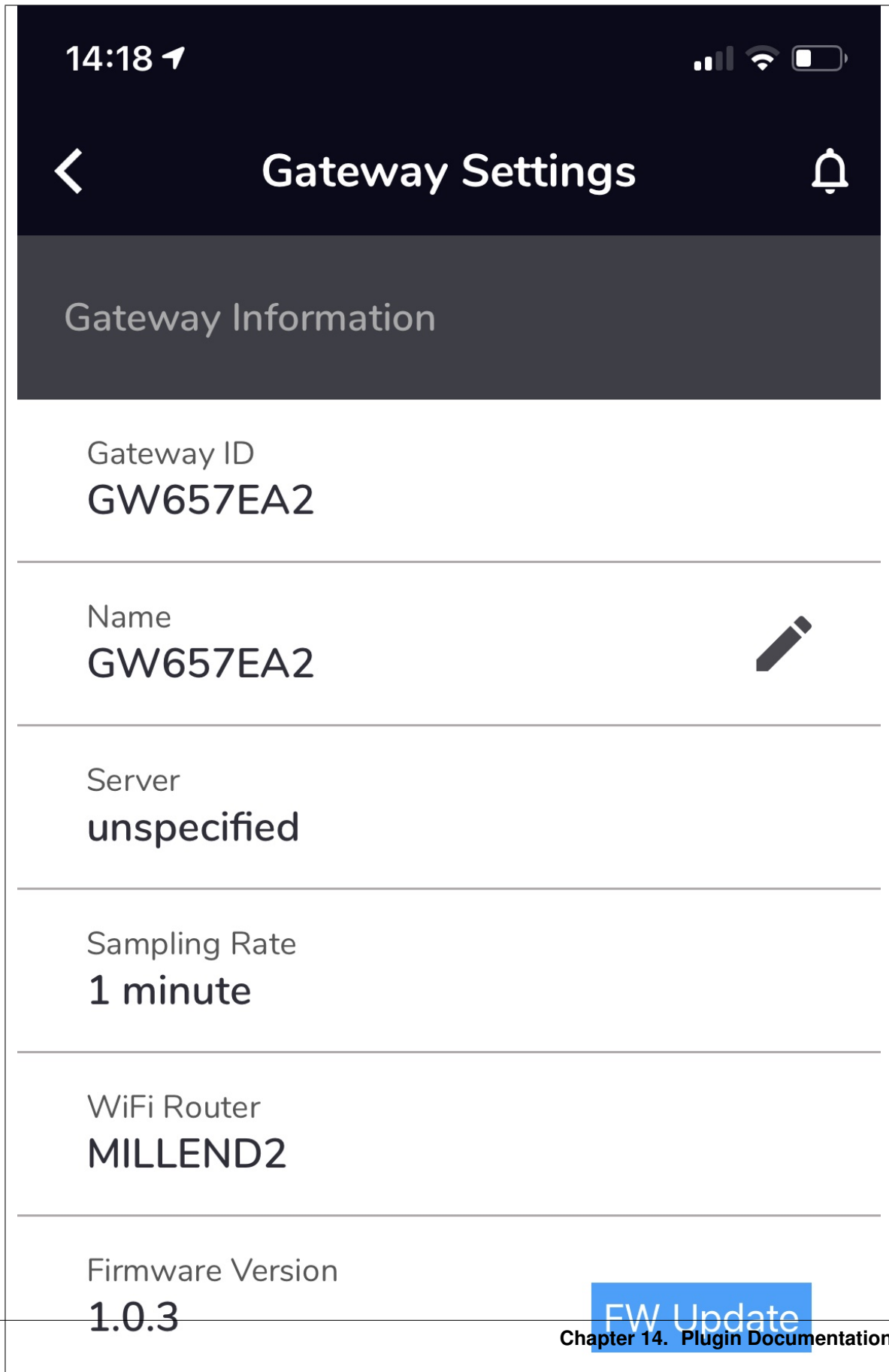
Port

8883

- **Server IP:** This is the IP address of your MQTT server. If you used the default 127.0.0.1 address in the south service then this should be the external address of your FLIR Bridge machine and not that address/
- **User name:** The user name you configured in your MQTT server.
- **Password:** The password assigned to the above user.
- **Port:** Leave this as the default value 8883
- Click on *Continue*



- Click on the name of the discovered gateway and follow the steps to setup a connection to the WiFi network
- You will see the details of your GW65 gateway



- Click on *Continue* and follow the instructions to add your sensors

Installing an MQTT Broker

You may use any compatible MQTT broker with the plugin and FLIR GW65 gateway, during testing the Mosquitto MQTT broker was used, a package exists that allows this to be installed and configured for use with gateway, this package is called *flir-mqtt-broker*.

To use the package simply use your package manager to install the package, for example on a *apt* based system such as Ubuntu

```
apt install flir-mqtt-broker
```

This will

- Install the mosquitto MQTT service
- Configure it with a certificate
- Add a user with the username *flir*
- Start it listening on port 8883 for MQTTS

Alternatively you can manually configure the Mosquitto MQTT browser by using the following steps

- Edit the configuration file */etc/mosquitto/mosquitto.conf* and adding the following lines

```
# Start of MQTTS support
listener 8883
cafile /etc/mosquitto/certs/ca.crt
certfile /etc/mosquitto/certs/client.crt
keyfile /etc/mosquitto/certs/client.key

password_file /etc/mosquitto/passwordfile
# End of MQTTS support
```

- Create a password file by running the command

```
touch /etc/mosquitto/passwordfile
```

- Create the *flir* user

```
mosquitto_passwd -b /etc/mosquitto/passwordfile flir 123456
```

- Generate the required certificates

```
mkdir /etc/mosquitto/certs
cd /etc/mosquitto/certs
openssl req -new -x509 -days 365 -extensions v3_ca -keyout ca.key -out ca.crt -
↳subj "/C=RO/ST=Home/L=Home/O=Dianomic/OU=FLIR Bridge/CN=dianomic.com" -passout
↳pass:flir
openssl genrsa -out client.key 2048
openssl req -new -out client.csr -key client.key -subj "/C=RO/ST=H/L=Home/O=MQTT
↳Broker/OU=MQTT Broker/CN=mqtt-broker.local"
openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out
↳client.crt -days 365 pass:flir
openssl rsa -in client.key -out client.key
```

- Set permissions for Mosquitto MQTT

```
chown -R mosquitto:/etc/mosquitto
chmod 700 /etc/mosquitto/certs
```

- Then restart your MQTT broker

```
systemctl restart mosquitto
```

14.1.19 South HTTP

The *flir-south-http* plugin allows data to be received from another FLIR Bridge instance or external system using a REST interface. The FLIR Bridge which is sending the data to the corresponding north task with the HTTP north plugin installed. There are two options for the HTTP north or , these serve the dual purpose of providing a data path between FLIR Bridge instances and also as examples of how other systems might use the REST interface from C/C++ or Python. The plugin supports both HTTP and HTTPS transport protocols and sends a JSON payload of reading data in the internal FLIR Bridge format.

The primary purpose of this plugin is for FLIR Bridge to FLIR Bridge communication however, there is no reason to prevent other applications that wish to send data into a FLIR Bridge system to not use this plugin also. The only requirement is that the application that is sending the data uses the same JSON payload structure as FLIR Bridge uses for passing reading data between different instances. Data should be sent to the URL defined in the configuration of the plugin using a POST request. The caller may choose to send one or many readings within a single POST request and those readings may be for multiple assets.

To create a south service you, as with any other south plugin

- Select *South* from the left hand menu bar.
- Click on the + icon in the top left
- Choose *http_south* from the plugin selection list
- Name your service
- Click on *Next*
- Configure the plugin

1 Plugin & Service Name 2 Review Configuration 3 Done

Host	0.0.0.0
Port	6683
URI	sensor-reading
Asset Name Prefix	http-
Enable HTTP	<input checked="" type="checkbox"/>
HTTPS Port	6684
Certificate Name	fledge

Previous Next

- **Host:** The host name or IP address to bind to. This may be left as default, in which case the plugin binds to any address. If you have a machine with multiple network interfaces you may use this parameter to select one of those interfaces to use.
- **Port:** The port to listen for connection from another FLIR Bridge instance.
- **URL:** URI that the plugin accepts data on. This should normally be left to the default.
- **Asset Name Prefix:** A prefix to add to the incoming asset names. This may be left blank if you wish to preserve the same asset names.
- **Enable HTTP:** This toggle specifies if HTTP connections should be accepted or not. If the toggle is off then only HTTPS connections can be used.
- **Certificate Name:** The name of the certificate to use for the HTTPS encryption. This should be the name of a certificate that is stored in the FLIR Bridge .

- Click *Next*
- Enable your service and click *Done*

JSON Payload

The payload that is expected by this plugin is a simple JSON presentation of a set of reading values. A JSON array is expected with one or more reading objects contained within it. Each reading object consists of a timestamp, an asset name and a set of data points within that asset. The data points are represented as name value pair JSON properties within the reading property.

The fixed part of every reading contains the following

Name	Description
times-tamp	The timestamp as an ASCII string in ISO 8601 extended format. If no time zone information is given it is assumed to indicate the use of UTC.
asset	The name of the asset this reading represents.
read-ings	A JSON object that contains the data points for this asset.

The content of the *readings* object is a set of JSON properties, each of which represents a data value. The type of these values may be integer, floating point, string, a JSON object or an array of floating point numbers.

A property

```
"voltage" : 239.4
```

would represent a numeric data value for the item *voltage* within the asset. Whereas

```
"voltageUnit" : "volts"
```

Is string data for that same asset. Other data may be presented as arrays

```
"acceleration" : [ 0.4, 0.8, 1.0 ]
```

would represent acceleration with the three components of the vector, x, y, and z. This may also be represented as an object

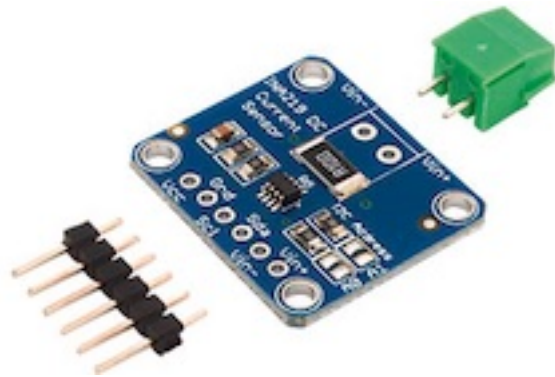
```
"acceleration" : { "X" : 0.4, "Y" : 0.8, "Z" : 1.0 }
```

both are valid formats within FLIR Bridge.

An example payload with a single reading would be as shown below

```
[
  {
    "timestamp" : "2020-07-08 16:16:07.263657+00:00",
    "asset"      : "motor1",
    "readings"   : {
      "voltage"  : 239.4,
      "current"  : 1003,
      "rpm"      : 120147
    }
  }
]
```

14.1.20 INA219 Voltage & Current Sensor



The *flir-south-ina219* plugin is a south plugin that uses an INA219 breakout board to measure current and voltage. The Texas Instruments INA219 is capable of measuring voltages up to 26 volts and currents up to 3.2 Amps. It connects via the I2C bus of the host and multiple sensors may be daisy chain on a single I2C bus. Breakout boards that mount the chip and its associate shunt resistor and connectors and easily available and attached to hosts with I2C buses.

The INA219 support three voltage/current ranges

- 32 Volts, 2 Amps
- 32 Volts, 1 Amp
- 16 Volts, 400 mAmps

Choosing the smallest range that is sufficient for your application will give you the best accuracy.

Note: This plugin is only available for the Raspberry Pi as it requires to be interfaced to the I2C bus on the Raspberry Pi GPIO header socket.

To create a south service with the INA219

- Click on *South* in the left hand menu bar
- Select *ina219* from the plugin list
- Name your service and click *Next*

1 Plugin & Service Name 2 Review Configuration 3 Done

Asset Name:

I2C Address:

Voltage Range:

Previous Next

- Configure the plugin
 - **Asset Name:** The asset name of the asst that will be written
 - **I2C Address:** The address of the INA219 device
 - **Voltage Range:** The voltage range that is to be used. This may be one of 32V2A, 32V1A or 16V400mA
- Click *Next*
- Enable the service and click on *Done*

Wiring The Sensor

The INA219 uses the I2C bus on the Raspberry PI, which requires two wires to connect the bus, it also requires power taking the total to four wires

INA219 Pin	Raspberry Pi Pin
Vin	3V3 pin 1
GND	GND pin 9
SDA	SDA pin 3
SCL	SCL pin 5

14.1.21 Lathe Simulation

The *flir-south-lathesim* plugin is a south plugin that simulates a lathe with a number of attached sensors. The purpose of this plugin is for test and demonstration only as it does not attach to any real device.

The plugin simulates four sensor devices attached to the virtual lathe

- The PLC controlling the lathe that gives details such as cutting depth, tool position, motor speed
- A current sensor that measures the current draw from the lathe
- A vibration sensor giving the RMS value of the vibration and the dominant vibration frequency
- A thermal imaging device that takes temperature readings every second from the motor, gearbox, headstock, tailstock and tool on the lathe

The vibration sensor reports at half the rate of the other sensors attached to the lathe in order to simulate handling data that is related to the same physical device but not available at the same rate as the other sensors.

The simulation runs a repeated pattern of operations;

- A spin-up period where the lathe spins up to speed from idle.
- A period where the lathe is doing some cutting of a work piece.
- A spin-down period where the lathe is slowing to a stop.
- An idle period where the work piece is removed and replaced with a new billet.

During the spin up period the lathe speed, expressed in revolutions per minute, will linearly increase from 0 to the maximum defined.

When the lathe is cutting the speed will remain predominantly constant, with a small random variation, whilst the depth of cut and X position of the cutting tool will change.

The lathe then spins down to rest and will remain idle for a short time whilst the worked item is removed and a new billet of material is installed.

During the cutting period the current draw and vibration will alter as load is applied to the piece.

Configuring the PLC

There are a number of configuration options that can be applied to the simulation.

Lathe1 South Service

Lathe Name	lathe
Spin up time	5
Cutting time	45
Idle time	15
Spin down time	6
RPM	500
Current	750
Enabled	<input checked="" type="checkbox"/>

[Show Advanced Config](#)

Applications +

Cancel Save

- **Lathe Name:** The name of the lathe in this configuration. This name is used to derive the assets returned from the three sets of sensors. The PLC data is returned with an asset name that machines the lathe name. The current data has *Current* appended to the lathe name and the asset id of the vibration name is the lathe name with *Vibration* appended to it. The temperature data uses the asset with the name of the lathe and *IR* appended to it.
- **Spin up time:** The time in seconds it takes the lathe to spin up to working speed from idle.
- **Cutting time:** The time in seconds for which the lathe is cutting material.
- **Spin Down time:** The time in seconds for which the lathe is spinning down from operating speed to stop.
- **Idle time:** The time in seconds for which the lathe is idle between jobs.

- **RPM:** The operating speed of the lathe, expressed in revolutions per minute.
- **Current:** The nominal operating current draw of the lathe.

14.1.22 Modbus South Plugin

The *flir-south-modbus-c* plugin is a south plugin that supports both TCP and RTU variants of Modbus. The plugin provides support for reading Modbus coils, input bits, registers and input registers, a flexible mechanism is provided to create a mapping between the Modbus registers and coils and the assets within FLIR Bridge. Multiple registers can be combined to allow larger values than register width to be mapped from devices that represent data in this way. Support is also included for floating point representation within the Modbus registers.

Configuration Parameters

A Modbus south service is added in the same way as any other south service in FLIR Bridge,

- Select the *South* menu item
- Click on the + icon in the top right
- Select *ModbusC* from the plugin list
- Enter a name for your Modbus service
- Click *Next*
- You will be presented with the following configuration page

Asset Name

modbus

Protocol

RTU

Server Address

127.0.0.1

Port

2222

Device

Baud Rate

9600

Number Of Data Bits

8

Number Of Stop Bits

1

Parity

none

Slave ID

1

Register Map

1

{

2

"values": [

3

{

4

"name": "temperature",

5

"slave": 1,

6

"assetName": "Booth1",

7

"register": 0,

8

"scale": 0.1,

9

"offset": 0

10

},

11

{

12

"name": "humidity",

}

},

]

}

Timeout

0.5

Control

None

Control Map

1

{

2

"values": []

3

}

- **Asset Name:** This is the name of the asset that will be used for the data read by this service. You can override this within the Modbus Map, so this should be treated as the default if no override is given.

- **Protocol:** This allows you to select either the *RTU* or *TCP* protocol. Modbus RTU is used whenever you have a serial connection, such as RS485 for connecting to your device. The TCP variant is used where you have a network connection to your device.
- **Server Address:** This is the network address of your Modbus device and is only valid if you selected the *TCP* protocol.
- **Port:** This is the port to use to connect to your Modbus device if you are using the TCP protocol.
- **Device:** This is the device to open if you are using the RTU protocol. This would be the name of a Linux device in `/dev`, for example `/dev/SERIAL0`
- **Baud Rate:** The baud rate used to communicate if you are using a serial connection with Modbus RTU.
- **Number of Data Bits:** The number of data bits to send on serial connections.
- **Number of Stop Bits:** The number of stop bits to send on the serial connections.
- **Parity:** The parity setting to use on the serial connection.
- **Slave ID:** The slave ID of the Modbus device from which you wish to pull data.
- **Register Map:** The register map defines which Modbus registers and coils you read, and how to map them to FLIR Bridge assets. The map is a complex JSON object which is described in more detail below.
- **Timeout:** The request timeout when communicating with a Modbus TCP client. This can be used to increase the timeout when a slow Modbus device or network is used.
- **Control:** Which register map should be used for mapping control entities to modbus registers.



If no control is required then this may be set to *None*. Setting this to *Use Register Map* will cause all the registers that are being read to also be targets for control. Setting this to *Use Control Map* will cause the separate *Control Map* to be used to map the control set points to modbus registers.

- **Control Map:** The register map that is used to map the set point names into Modbus registers for the purpose of set point control. The control map is the same JSON format document as the register map and uses the same set of properties.

Register Map

The register map is the most complex configuration parameter for this plugin and over time has supported a number of different variants. We will only document the latest of these here although previous variants are still supported. This latest variant is the most flexible to date and is thus the recommended approach to adopt.

The map is a JSON object with a single array *values*, each element of this array is a JSON object that defines a single item of data that will be stored in FLIR Bridge. These objects support a number of properties and values, these are

Property	Description
name	The name of the value that we are reading. This becomes the name of the data point with the asset. This may be either the default asset name defined plugin or an individual asset if an override is given.
slave	The Modbus slave ID of the device if it differs from the global Slave ID defined for the plugin. If not given the default Slave ID will be used.
asset-Name	This is an optional property that allows the asset name define for the plugin to be overridden on an individual basis. Multiple values in the values array may share the same AssetName, in which case the values read from the Modbus device are placed in the same asset. Note: This is unused in a control map.
register	This defines the Modbus register that is read. It may be a single register, in which case the value is the register number or it may be multiple registers in which case the value is a JSON array of numbers. If an array is given then the registers are read in the order of that array and combined into a single value by shifting each value up 16 bits and performing a logical OR operation with the next register in the array.
coil	This defines the number of the Modbus coil to read. Coils are single bit Modbus values.
input	This defines the number of the Modbus discrete input. Coils are single bit Modbus values.
inputRegister	This defines the Modbus input register that is read. It may be a single register, in which case the value is the register number or it may be multiple registers in which case the value is a JSON array of numbers. If an array is given then the registers are read in the order of that array and combined into a single value by shifting each value up 16 bits and performing a logical OR operation with the next register in the array.
scale	A scale factor to apply to the data that is read. The value read is multiplied by this scale. This is an optional property.
offset	An optional offset to add to the value read from the Modbus device.
type	This allows data to be cast to a different type. The only support type currently is <i>float</i> and is used to interpret data read from the one or more of the 16 bit registers as a floating point value. This property is optional.
swap	This is an optional property used to byte swap values read from a Modbus device. It may be set to one of <i>bytes</i> , <i>words</i> or <i>both</i> to control the swapping to apply to bytes in a 16 bit value, 16 bit words in a 32 bit value or both bytes and words in 32 bit values.

Every *value* object in the *values* array must have one and only one of *coil*, *input*, *register* or *inputRegister* included as this defines the source of the data in your Modbus device. These are the Modbus object types and each has an address space within a typical Modbus device.

Object Type	Size	Address Space	Map Property
Coil	1 bit	00001 - 09999	coil
Discrete Input	1 bit	10001 - 19999	input
Input Register	16 bits	30001 - 39999	inputRegister
Holding Register	16 bits	40001 - 49999	register

The values in the map for coils, inputs and registers are relative to the base of the address space for that object type rather than the global address space and each is 0 based. A map value that has the property *“coil”* : 10 would return the values of the tenth coil and *“register”* : 10 would return the tenth register.

Example Maps

In this example we will assume we have a cooling fan that has a Modbus interface and we want to extract three data items of interest. These items are

- Current temperature that is in Modbus holding register 10

- Current speed of the fan that is stored as a 32 bit value in Modbus holding registers 11 and 12
- The active state of the fan that is stored in a Modbus coil 1

The Modbus Map for this example would be as follow:

```
{
  "values" : [
    {
      "name"      : "temperature",
      "register"   : 10
    },
    {
      "name"      : "speed",
      "register"   : [ 11, 12 ]
    },
    {
      "name"      : "active",
      "coil"      : 1
    }
  ]
}
```

Since none of these values have an `assetName` defined all there values will be stored in a single asset, the name of which is the default asset name defined for the plugin as a whole. This asset will have three data points within it; *temperature*, *speed* and *active*.

Set Point Control

The *flir-south-modbus-c* plugin supports the FLIR Bridge set point control mechanisms and allows a register map to be defined that maps the set point attributes to the underlying modbus registers. As an example a control map as follows

```
{
  "values" : [
    {
      "name" : "active",
      "coil" : 1
    }
  ]
}
```

Defines that a set point write operation can be instigated agisnt the set point named *active* and this will map to the Modbus coil 1.

Set points may be defined for Modbus coils and registers, the rad only input bits and input registers can not be used for set point control.

The *Control Map* can use the same swapping, scaling and offset properties as modbus *Register Map*, it can also map multiple registers to a single set point and floatign point values.

14.1.23 South MQTT

The *flir-south-mqtt-readings* plugin allows to create an MQTT subscriber service. MQTT Subscriber reads messages from topics on the MQTT broker.

To create a south service you, as with any other south plugin

- Select *South* from the left hand menu bar

- Click on the + icon in the top right
- Choose mqtt-readings from the plugin selection list
- Name your service
- Click on *Next*
- Configure the plugin

The screenshot shows a configuration window with a progress bar at the top indicating three steps: 1. Plugin & Service Name, 2. Review Configuration (current step), and 3. Done. The configuration fields are as follows:

Field	Value
MQTT Broker host	localhost
MQTT Broker Port	1883
Keep Alive Interval	60
Topic To Subscribe	Room1/conditions
QoS Level	0
Asset Name	mqtt-

At the bottom of the window, there are two buttons: "Previous" and "Next".

- **MQTT Broker host:** Hostname or IP address of the broker to connect to.
 - **MQTT Broker Port:** The network port of the broker.
 - **Keep Alive Interval:** Maximum period in seconds allowed between communications with the broker. If no other messages are being exchanged, this controls the rate at which the client will send ping messages to the broker.
 - **Topic To Subscribe:** The subscription topic to subscribe to receive messages.
 - **QoS Level:** The desired quality of service level for the subscription.
 - **Asset Name:** Name of Asset.
- Click *Next*
 - Enable your service and click *Done*

Message Payload

The content of the message payload published to the topic, to which the service is configured to subscribe, should be parsable to a JSON object.

e.g. `{“humidity”: 93.29, “temp”: 16.82}`

```
$ mosquitto_pub -h localhost -t "Room1/conditions" -m '{"humidity": 93.29, "temp": 16.82}'
```

The `mosquitto_pub` client utility comes with the `mosquitto` package, and a great tool for conducting quick tests and troubleshooting. https://mosquitto.org/man/mosquitto_pub-1.html

14.1.24 MQTT Sparkplug B

The *flir-south-mqtt-sparkplug* plugin implements the Sparkplug B payload format with an MQTT (Message Queue Telemetry Transport) transport. The plugin will subscribe to a configured topic and will process the Sparkplug B payloads, creating FLIR Bridge assets from those payloads. Sparkplug is an open source software specification of a payload format and set of conventions for transporting sensor data using MQTT as the transport mechanism.

Note: Sparkplug is bi-directional, however this plugin will only read data from the Sparkplug device.

To create a south service with the MQTT Sparkplug B plugin

- Click on *South* in the left hand menu bar
- Select *mqtt_sparkplug* from the plugin list
- Name your service and click *Next*

The screenshot displays the configuration screen for the MQTT Sparkplug B plugin. At the top, a progress bar indicates the current step is 'Review Configuration' (step 2 of 3). The configuration fields are as follows:

Field	Value
Asset Name	mqtt
MQTT Host	chariot.groov.com
MQTT Port	1883
Username	opto
Password	*****
Topic	spBv1.0/Opto22/DDATA/groovEPIC_workshop/Strategy

Navigation buttons 'Previous' and 'Next' are located at the bottom of the form.

- Configure the plugin
 - **Asset Name:** The asset name which will be used for all data read.
 - **MQTT Host:** The MQTT host to connect to, this is the host that is running the MQTT broker.
 - **MQTT Port:** The MQTT port, this is the port the MQTT broker uses for unencrypted traffic, usually 1883 unless modified.
 - **Username:** The user name to be used when authenticating with the MQTT subsystem.
 - **Password:** The password to use when authenticating with the MQTT subsystem.
 - **Topic:** The MQTT topic to which the plugin will subscribe.
- Click *Next*
- Enable the service and click on *Done*

14.1.25 MQTT South with Payload Scripting

The *flir-south-mqtt-scripted* plugin uses MQTT to receive messages via an MQTT broker from sensors or other sources. It then uses an optional script, written in Python, that converts the message into a JSON document and pushes data to the FLIR Bridge System.

If the payload of the MQTT message is a JSON document with simple key/value pairs, e.g.

```
{ "temperature" : 23.1, "humidity" : 47.2 }
```

Then no translation script is required. Also if the payload is a simple numeric value the plugin will accept this and create an asset with the data point name matching the topic on which the value was given in the payload.

If the message format is not a simple JSON document or a single value, or is in some other format then a Python script should be provided that turns the message into a JSON format.

An example script, assuming the payload in the message is simply a value, might be as follows

```
def convert(message, topic):  
    return {  
        'temperature' : float(message)  
    }
```

Note that the message and topic are passed as strings and the data we wish to ingest into FLIR Bridge in this case is assumed to be a floating point value. The example above of course is unnecessary as the plugin can consume this data without the need of a script.

The script could return either one or two values.

The script should return the JSON document as a Python DICT in the case of a single value.

The script should return a string and a JSON document as a Python DICT in the case of two values, the first of these values is the name of the asset to use and overrides the default asset naming defined in the plugin configuration.

First case sample:

```
def convert(message, topic):  
    return {"temperature_1": 10.2}
```

Second case sample:

```
def convert(message, topic):  
    return "ExternalTEMP", {"temperature_3": 11.3}
```

Configuration

When adding a south service with this plugin the same flow is used as with any other south service. The configuration page for the plugin is as follows.

1 Plugin & Service Name 2 Review Configuration 3 Done

Asset Name

MQTT Broker

Username

Password

Trusted Certificate

Client Certificate

Private Key

Key Password

Topic

Object Policy

Script

1	

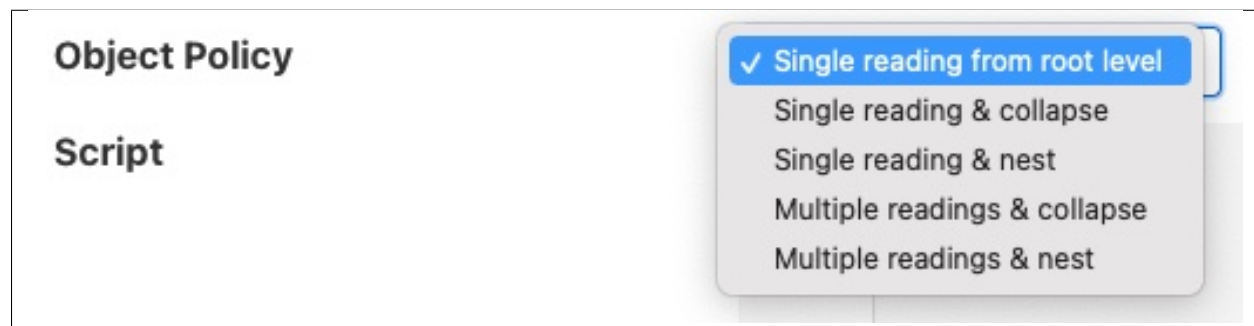
No file chosen

- **Asset Name:** The name of the asset the plugin will create for each message, unless the convert function returns an explicit asset name to be used.
- **MQTT Broker:** The IP address/hostname of the MQTT broker to use. Note FLIR Bridge requires an external MQTT broker is run currently and does not provide an internal broker in the current release.
- **Username:** The username to be used if required for authentication. This should be left blank if authentication is not required.
- **Password:** The password to use if username is to be used.
- **Trusted Certificate:** The trusted certificate of the MQTT broker. If MQTTS communication is not required then this can be left blank.
- **Client Certificate:** The certificate that will be used by the MQTT plugin.
- **MQTTS Key:** The private key of the MQTT plugin. If the key is included in the PEM file of the client certificate this may be left blank.

- **Key Password:** The password used to encrypted the private key. This may be left blank if the private key was not encrypt.
- **Topic:** The MQTT topic to which to subscribe. The topic may include the usual MQTT wildcards; + for a single level wildcard and # for a multi-level wildcard
- **Object Policy:** Controls how the plugin deals with nested objects within the JSON payloads it receives or the return from the script that is executed. See below for a description of the various object policy values.
- **Time Format:** The format to both pass the timestamps into the query parameters using and also to interpret the timestamps returned in the payload.
- **Timezone:** The timezone to use for the start and end times that are sent in the API request and also when timestamps are read from the API response. Timezone is expressed as an offset in hours and minutes from UTC for the local timezone of the API. E.g. -08:00 for PST time zones.
- **Script:** The Python script to execute for message processing. Initially a file must be uploaded, however once uploaded the user may edit the script in the box provided. A script is optional.

14.1.26 Object Policy

The object policy is used by the plugin to determine how it deals with nested objects within the JSON that is in the MQTT payload or the JSON that is returned from the script that is executed, if present.



- **Single reading from root level:** This is the simple behavior of the plugin, it will only take numeric and string values that are in the root of the JSON document and ignore any objects contained in the root.
- **Single reading & collapse:** The plugin will create a single reading form the payload that will contain the string and numeric data in the root level. The plugin will also recursively traverse any child objects and add the string and numeric data from those to the reading as data points of the reading itself.
- **Single reading & nest:** As above, the plugin will create a single reading form the payload that will contain the string and numeric data in the root level. The plugin will also recursively traverse any child objects and add the string and numeric data from those objects and add them as nested data points.
- **Multiple readings & collapse:** The plugin will create one reading that contains any string and numeric data in the root of the JSON. It will then create one reading for each object in the root level. Each of these readings will contain the string and numeric data from those child objects along with the data found in the children of those objects. Any child data will be collapse into the base level of the readings.
- **Multiple readings & nest:** As above, but any data in the children of the readings found below the first level, which defines the reading names, will be created as nested data points rather than collapsed.

As an example of how the policy works assume we have an MQTT payload with a message as below

```

{
  "name" : "pump47",
  "motor" : {
    "current" : 0.75,
    "speed" : 1496
  },
  "flow" : 1.72,
  "temperatures" : {
    "bearing" : 21.5,
    "impeller" : 16.2,
    "motor" : {
      "casing" : 24.6,
      "gearbox" : 28.2
    }
  }
}

```

If the policy is set to *Single reading from root level* then a reading would be created, with the asset name given in the configuration of the plugin, that contained two data points *name* and *flow*.

If the policy is set to *Single reading & collapse* then the reading created would now have 8 data points; *name*, *current*, *speed*, *flow*, *bearing*, *impeller*, *casing* and *gearbox*. These would all be in a reading with the asset name defined in the configuration and in a flat structure.

If the policy is set to *Single reading & nest* there would still be a single reading, with the asset name set in the configuration, which would have data points for *name*, *motor*, *flow* and *temperature*. The *motor* data point would have two child data points called *current* and *speed*, the *temperature* data point would have three child data points called *bearing*, *impeller* and *motor*. This *motor* data point would itself have two children call *casing* and *gearbox*.

If the policy is set to *Multiple readings & collapse* there would be three readings created from this payload; one that is names as per the asset name in the configuration, a *motor* reading and a *temperature* reading. The first of these readings would have data points called *name* and *flow*, the *motor* reading would have data points *current* and *speed*. The *temperatures* reading would have data points *bearing*, *impeller*, *casing* and *gearbox*.

If the policy is set to *Multiple readings & nest* there would be three readings created from this payload; one that is names as per the asset name in the configuration, a *motor* reading and a *temperature* reading. The first of these readings would have data points called *name* and *flow*, the *motor* reading would have data points *current* and *speed*. The *temperatures* reading would have data points *bearing*, *impeller* and *motor*, the *motor* data point would have two child data points *casing* and *gearbox*.

Timestamp Treatment

The default timestamp for a reading collected via this plugin will be the time at which the reading was taken, however it is possible for the API that is being called to include a different timestamp.

Returning a data point called whose name is defined in the *Timestamp* configuration option will result in the value of that data point being used as the timestamp. This data point will not be added to the reading. The default name of the timestamp is *timestamp*.

The timestamp data point should be a string and the timestamp should be formatted to match the definition given in the *Time format* configuration parameter. The format is based on the standard Linux strptime formatting options and is discussed below in the section discussing the :ref:ref::time_format selection method.

The timezone may be set by using the *Timezone* configuration parameter to set the offset of the timezone in which the API is running.

Time Format

The format of the timestamps read in the message payload or by the script returned are defined by the *Time Format* configuration parameter and uses the standard Linux mechanism to define a time format. The following character sequences are supported.

%% The % character.

%a or %A The name of the day of the week according to the current locale, in abbreviated form or the full name.

%b or %B or %h

The month name according to the current locale, in abbreviated form or the full name.

%c The date and time representation for the current locale.

%C The century number (0–99).

%d or %e The day of month (1–31).

%D Equivalent to %m/%d/%y. (This is the American style date, very confusing to non- Americans, especially since %d/%m/%y is widely used in Europe. The ISO 8601 standard format is %Y-%m-%d.)

%H The hour (0–23).

%I The hour on a 12-hour clock (1–12).

%j The day number in the year (1–366).

%m The month number (1–12).

%M The minute (0–59).

%n Arbitrary white space.

%p The locale’s equivalent of AM or PM. (Note: there may be none.)

%r The 12-hour clock time (using the locale’s AM or PM). In the POSIX locale equivalent to %I:%M:%S %p. If t_fmt_ampm is empty in the LC_TIME part of the current locale, then the behavior is undefined.

%R Equivalent to %H:%M.

%S The second (0–60; 60 may occur for leap seconds; earlier also 61 was allowed).

%t Arbitrary white space.

%T Equivalent to %H:%M:%S.

%U The week number with Sunday the first day of the week (0–53). The first Sunday of January is the first day of week 1.

%w The ordinal number of the day of the week (0–6), with Sunday = 0.

%W The week number with Monday the first day of the week (0–53). The first Monday of January is the first day of week 1.

%x The date, using the locale’s date format.

%X The time, using the locale’s time format.

%y The year within century (0–99). When a century is not otherwise specified, values in the range 69–99 refer to years in the twentieth century (1969–1999); values in the range 00–68 refer to years in the twenty-first century (2000–2068).

%Y The year, including century (for example, 1991).

14.1.27 OPC/UA South Plugin

The *flir-south-opcua* plugin allows FLIR Bridge to connect to an OPC/UA server and subscribe to changes in the objects within the OPC/UA server.

A south service to collect OPC/UA data is created in the same way as any other south service in FLIR Bridge.

- Use the *South* option in the left hand menu bar to display a list of your South services
- Click on the + add icon at the top right of the page
- Select the *opcua* plugin from the list of plugins you are provided with
- Enter a name for your south service
- Click on *Next* to configure the OPC/UA plugin

1 Plugin & Service Name 2 Review Configuration 3 Done

Asset Name

OPCUA Server URL

OPCUA Object Subscriptions

```

1 {
2   "subscriptions": [
3     "ns=5;s=85/0:Simulation"
4   ]
5 }

```

Subscribe By ID ☒

Min Reporting Interval

[Previous](#) [Next](#)

The configuration parameters that can be set on this page are;

- **Asset Name:** This is a prefix that will be applied to all assets that are created by this plugin. The OPC/UA plugin creates a separate asset for each data item read from the OPC/UA server. This is done since the OPC/UA server will deliver changes to individual data items only. Combining these into a complex asset would result in assets that do only contain one of many data points in each update. This can cause upstream systems problems with the every changing asset structure.

- **OPCUA Server URL:** This is the URL of the OPC/UA server from which data will be extracted. The URL should be of the form `opc.tcp://.../`
- **OPCUA Object Subscriptions:** The subscriptions are a set of locations in the OPC/UA object hierarchy that defined which data is subscribed to in the server and hence what assets get created within FLIR Bridge. A fuller description of how to configure subscriptions is shown below.
- **Subscribe By ID:** This toggle determines if the OPC/UA objects in the subscription are using names to identify the objects in the OPC/UA object hierarchy or using object ID's.
- **Min Reporting Interval:** This control the minimum interval between reports of data changes in subscriptions. It sets an upper limit to the rate that data will be ingested into the plugin and is expressed in milliseconds.

Subscriptions

Subscriptions to OPC/UA objects are stored as a JSON object that contents an array named “subscriptions”. This array is a set of OPC/UA nodes that will control the subscription to variables in the OPC/UA server.

The array may be empty, in which case all variables are subscribed to in the server and will create assets in FLIR Bridge. Although simply subscribing to everything will return a lot of data that may not be of use.

If the *Subscribe By ID* option is set then this is an array of node Id's. Each node Id should be of the form `ns=...;s=...`. Where `ns` is a namespace index and `s` is the node id string identifier. A subscription will be created with the OPC/UA server for the object with the specified node id and its children, resulting in data change messages from the server for those objects. Each data change received from the server will create an asset in FLIR Bridge with the name of the object prepended by the value set for *Asset Name*. An integer identifier is also supported by using a node Id of the form `ns=...;i=...`.

If the *Subscribe By ID* option is not set then the array is an array of browse names. The format of the browse names is `<namespace>:<name>`. If the namespace is not required then the name can simply be given, in which case any name that matches in any namespace will have a subscription created. The plugin will traverse the node tree of the server from the *ObjectNodes* root and subscribe to all variables that live below the named nodes in the subscriptions array.

Configuration examples

```
{"subscriptions":["5:Simulation","2:MyLevel"]}
```

We subscribe to

- 5:Simulation is a node name under ObjectsNode in namespace 5
- 2:MyLevel is a variable under ObjectsNode in namespace 2

```
{"subscriptions":["5:Sinusoid1","2:MyLevel","5:Sawtooth1"]}
```

We subscribe to

- 5:Sinusoid1 and 5:Sawtooth1 are variables under ObjectsNode/Simulation in namespace 5
- 2:MyLevel is a variable under ObjectsNode in namespace 2

```
{"subscriptions":["2:Random.Double","2:Random.Boolean"]}
```

We subscribe to

- Random.Double and Random.Boolean are variables under ObjectsNode/Demo both in namespace 2

It's also possible to specify an empty subscription array:

```
{"subscriptions":[]}
```

Note: Depending on OPC/UA server configuration (number of objects, number of variables) this empty configuration might take a long time to create the subscriptions and hence delay the startup of the south service. It will also result in a large number of assets being created within FLIR Bridge.

Object names, variable names and NamespaceIndexes can be easily retrieved browsing the given OPC/UA server using OPC UA clients, such as .

14.1.28 Person Detection Plugin

The *flir-south-person-detection* detects a person on a live video feed from either a camera or on a network stream. It uses Google's Mobilenet SSD v2 to detect a person. The bounding boxes and confidence scores are displayed on the same video frame itself. Also FPS (frames per second) are also displayed on the same frame. The detection results are also converted into readings. The readings have mainly three things:

1. *Count* : The number of people detected.
2. *Coordinates* : It consists of coordinates (x,y) of top-left and bottom right corners of bounding box for each detected person.
3. *Confidence* : Confidence with which the model detected each person.

- **TFlite Model File:** This is the name of the tflite model file that should be placed in `python/flir/plugins/south/person_detection/model` directory. Its default value is `detect_edgetpu.tflite`. If a Coral Edge TPU is not being used, the file name will be different (i.e. `detect.tflite`).
- **Labels File:** This is the name of the labels file that was used when training the above model, this file should also be placed in same directory as the model.
- **Asset Name:** The name of the asset used for the readings generated by this plugin.

- **Enable Edge TPU:** Indicates whether to use edge TPU for inference. If you don't want to use Coral Edge TPU then disable this configuration parameter. Also ensure to change the name of the model file to detect.tflite if disabled. Default is set to enabled.
- **Minimum Confidence Threshold:** The detection results from the model will be filtered out, if the score is below this value.
- **Source:** Either use a stream over a network or use a local camera device. Default is set to stream.
- **Streaming URL:** The URL of the RTSP stream, if stream is to be used. Only RTSP streams are supported for now.
- **OpenCV Backend:** The backend required by OpenCV to process the stream, if stream is to be used. Default is set to ffmpeg.
- **Streaming Protocol:** The protocol over which live frames are being transported over the network, if stream is to be used. Default is set to udp.
- **Camera ID:** The number associated with your video device. See /dev in your filesystem you will see video0 or video1. It is required when source is set to camera. Default is set to 0.
- **Enable Detection Window:** Show detection results in a native window. Default is set to disabled.

rtrr South Service

Stream Protocol

udp

Camera ID

0

Enable Detection Window

☐

Enable Web Streaming

☒

Web Streaming Port

8085

Enabled

☐

[Show Advanced Config](#)

Applications

Cancel

Save

?

Export Readings

Delete Service

- **Enable Web Streaming:** Whether to stream the detected results in a browser or not. Default is set to enabled.
- **Web Streaming Port:** Port number where web streaming server should run, if web streaming is enabled. Default is set to 8085.

Installation

1. First run requirements.sh

There are two ways to get the video feed.

1. **Camera**

To see the supported configuration of the camera run the following command.

```
$ v4l2-ctl --list-formats-ext --device /dev/video0
You will see something like
'YUYV' (YUYV 4:2:2)
  Size: Discrete 640x480
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 720x480
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 1280x720
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 1920x1080
    Interval: Discrete 0.067s (15.000 fps)
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 2592x1944
    Interval: Discrete 0.067s (15.000 fps)
  Size: Discrete 0x0
```

Above example uses Camera ID 0 to indicate use of /dev/video0 device, please use the applicable value for your setup

2. Network RTSP stream

To create a network stream follow the following steps

1. Install vlc

```
$ sudo add-apt-repository ppa:videolan/master-daily
$ sudo apt update
$ apt show vlc
$ sudo apt install vlc qtwayland5
$ sudo apt install libavcodec-extra
```

2. Download some sample files from here.

```
$ git clone https://github.com/intel-iot-devkit/sample-videos.git
```

3. Either stream a file using the following

```
$ vlc <name_of_file>.mp4 --sout '#gather:transcode{vcodec=h264,vb=512,
↪scale=Auto,width=640,height=480,acodec=none,scodec=none}:rtp{sdp=rtsp://
↪<ip_of_machine_streaming>:8554/clip}' --no-sout-all --sout-keep --loop --
↪no-sout-audio --sout-x264-profile=baseline
```

Note : fill the <ip_of_the_machine> with ip of the machine which will be used to stream video. Also fill <name_of_file> with the name of mp4 file.

4. You can also stream from a camera using the following

```
$ vlc v4l2:///dev/video<index_of_video_device> --sout '#gather:transcode
↪{vcodec=h264,vb=512,scale=Auto,width=<supported_width_of_camera_image>,
↪height=<supported_height_of_camera_image>,acodec=none,scodec=none}:rtp
↪{sdp=rtsp://<ip_of_the_machine>:8554/clip}' --no-sout-all --sout-keep --
↪no-sout-audio --sout-x264-profile=baseline
```

Fill the following :

<index_of_video_device> The index with which you ran the v4l2 command mentioned above. for example video0.

<supported_height_of_camera_image> Height you get when you ran v4l2 command mentioned above. For example Discrete 640x480. Here 480 is height.

<supported_width_of_camera_image> Width you get when you ran v4l2 command mentioned above. For example Discrete 640x480. Here 640 is width.

<ip_of_the_machine> ip of the machine which will be used to stream video.

Once you have run the plugin by filling appropriate parameters Now go to your browser and enter *ip_where_flir_is_running:the_port_for_web_streaming*

14.1.29 PI Web API south Plugin

The *flir-south-piwebapi* plugin is a south plugin that reads a PI Point and the related attributes from PI Web API. The plugin extracts the last value stored in the PI Point.

Using the Plugin

To create a south service with the PI Web API plugin

- Click on *South* in the left hand menu bar
- Select *PIWebAPI* from the plugin list
- Name your service and click *Next*

1 Plugin & Service Name
2 Review Configuration
 3 Done

Hostname

Server port, 0=use the default

Authentication Method

User Id

Password

PIPoint

Attributes

PI Server type

Server instance name

Database to use

Path on the server

basic
 ▼

.....
 👁

```

1 {
2   "items": []
3 }
            
```

PI Asset Framework
 ▼

Previous
Next

- Configure the plugin
 - **Hostname:** The name or IP address of the PI Web API server.
 - **Server port:** The port on which the PI Web API server is listening. 0 means to use the default 443 port.
 - **Authentication Method:** The authentication method requested by the PI Web API server, it could be either *basic* or *anonymous*, if *basic* is selected *user id* and *password* are required.

Authentication Method

User Id

basic
 ▼

anonymous

basic

- **User Id:** The user id on the PI Web API server to allow the *basic* authentication.
- **Password:** The password associated to the user on the PI Web API server.
- **PIPoint:** The name of the PI Point on PI Web API for which the data should be extracted.
- **Attributes:** The attributes of the PI Point to extract. It can be either a single attribute or multiple attributes expressed as a json array, an example:

Attributes	<pre> 1 { 2 "items": [3 "attr_4_1", 4 "attr_4_2", 5 "attr_4_3" 6] 7 }</pre>
-------------------	---

- **Server type:** It allows to select the PI Server type either PI Asset Framework or PI Data Archive.

PI Server type	<div>PI Asset Framework ▼</div> <div>PI Asset Framework</div> <div>PI Data Archive</div>
Server instance name	

- **Server instance name:** It specifies server instance to be used.
- **Database to use:** Available only in case of PI Asset Framework, it specifies the Asset Framework database from which the data should be extracted.
- **Path on the server:** Available only in case of PI Asset Framework, the path of the PI Web API hierarchy that should be traversed to identify the position from which the data should be to extracted, an example:

Path on the server	foglamp/r4dev2_4758
---------------------------	---------------------

14.1.30 Playback Plugin

The *flir-south-playback* plugin is a feature rich plugin for playing back comma separated variable (CSV) files. It supports features such as;

- Header rows
- User defined column names
- Use of historic or current timestamps
- Multiple timestamp formats
- Pick and optionally rename columns

- Looped or single pass readings of the data

To create a south service with the playback plugin

- Click on *South* in the left hand menu bar
- Select *playback* from the plugin list
- Name your service and click *Next*

The screenshot shows the 'Review Configuration' step of a configuration wizard. At the top, a progress bar indicates three steps: 1. Plugin & Service Name, 2. Review Configuration (highlighted), and 3. Done. The main configuration area contains the following fields and options:

- Asset Name:** sample
- CSV file name with extension:** some.csv
- Header Row:** ☒
- Header columns:** None
- Cherry pick column with same/new name:** 1, {}
- Historic timestamps:** ☐
- Pick timestamp delta from file:** ☐
- Timestamp column name:** ts
- Timestamp format:** %Y-%m-%d %H:%M:%S.%f
- Ingest mode:** batch (dropdown menu)
- Sample Rate:** 100
- Burst Interval (ms):** 1000
- Burst size:** 1
- Read file in a loop:** ☐

At the bottom, there are 'Previous' and 'Next' buttons.

- Configure the plugin
 - **Asset Name:** An asset name to use for the content of the file.
 - **CSV file name with extension:** The name of the file that is to be processed, the file must be located in the flir data directory.
 - **Header Row:** Toggle to indicate the first row is a header row that contains the names that should be used for the data points within the asset.
 - **Header Columns:** Only used if *Header Row* is not enabled. This parameter should a column separated list of column names that will be used to name the data points within the asset.
 - **Cherry pick column with same/new name:** This is a JSON document that can define a set of columns to include and optionally names to give those columns. If left empty then all columns, are included.

- **Historic timestamps:** A toggle field to control if the timestamp data should be the current time or a date and time taken from the file itself.
 - **Pick timestamp delta from file:** If current timestamps are used then this option can be used to maintain the same relative times between successive timestamps added to the data as it is ingested.
 - **Timestamp column name:** The name of the column that should be used for reading timestamp value. This must be given if either historic timestamps are used or the interval between readings is to be maintained.
 - **Timestamp format:** The format of the timestamp within the file.
 - **Ingest mode:** Determine if ingest should be in batch or burst mode. In burst mode data is ingested as a set of bursts of rows, defined by *Burst size*, every *Burst Interval*, this allows simulation of sensors that have internal buffering within them. Batch mode is the normal, regular rate ingest of data.
 - **Sample Rate:** The data sampling rate that should be used, this is defined in readings per second.
 - **Burst Interval (ms):** The time interval between consecutive bursts when burst mode is used.
 - **Burst size:** The number of readings to be sent in each burst.
 - **Read file in a loop:** Once the end of the file is reached then the plugin will go back to the start and resend the data if this toggle is on.
- Click *Next*
 - Enable the service and click on *Done*

Picking Columns

The *Cherry pick column with same/new name* entry is a JSON document with a set of key/value pairs. The key is the name of the column in the file and the value is the name which should appear in the final asset. To illustrate this let's assume we have a CSV file as follows

```
X, Y, Z, Amps, Volts
1.3, 0.1, 0.3, 2.1, 240
1.2, 0.3, 0.2, 2.2, 235
....
```

We want to create an asset that has the *X* and *Y* values, *Amps* and *Volts* but we want to name them *X*, *Y*, *Current*, *Voltage*. We can do this by creating a JSON document that maps the columns.

```
{
  "X" : "X",
  "Y" : "Y",
  "Amps" : "Current",
  "Volts" : "Voltage"
}
```

Since we only mention the columns *X*, *Y*, *Amps* and *Volts*, only these will be included in the asset and we will not include the column *Z*. We map the column name *X* to *X*, so it will be unchanged. As will the column *Y*, the column *Amps* will become the data point *Current* and *Volts* will become *Voltage*.

14.1.31 PT100 Temperature Sensor



The *flir-south-pt100* is a south plugin for the PT-100 temperature sensor. The PT100 is a resistance temperature detectors (RTDs) consist of a fine wire (typically platinum) wrapped around a ceramic core, exhibiting a linear increase in resistance as temperature rises. The sensor connects via a MAX31865 converter to a GPIO pins for I2C bus and a chip select pin.

Note: This plugin is only available for the Raspberry Pi as it requires to be interfaced to the I2C bus on the Raspberry Pi GPIO header socket.

To create a south service with the PT100

- Click on *South* in the left hand menu bar
- Select *pt100* from the plugin list
- Name your service and click *Next*

 The screenshot shows a configuration interface for the PT100 sensor. At the top, there is a progress bar with three steps: 1. Plugin & Service Name, 2. Review Configuration (currently active), and 3. Done. Below the progress bar, there are two input fields: 'Asset Name Prefix' with the value 'PT100/' and 'GPIO Pin' with the value '8'. At the bottom, there are two buttons: 'Previous' and 'Next'.

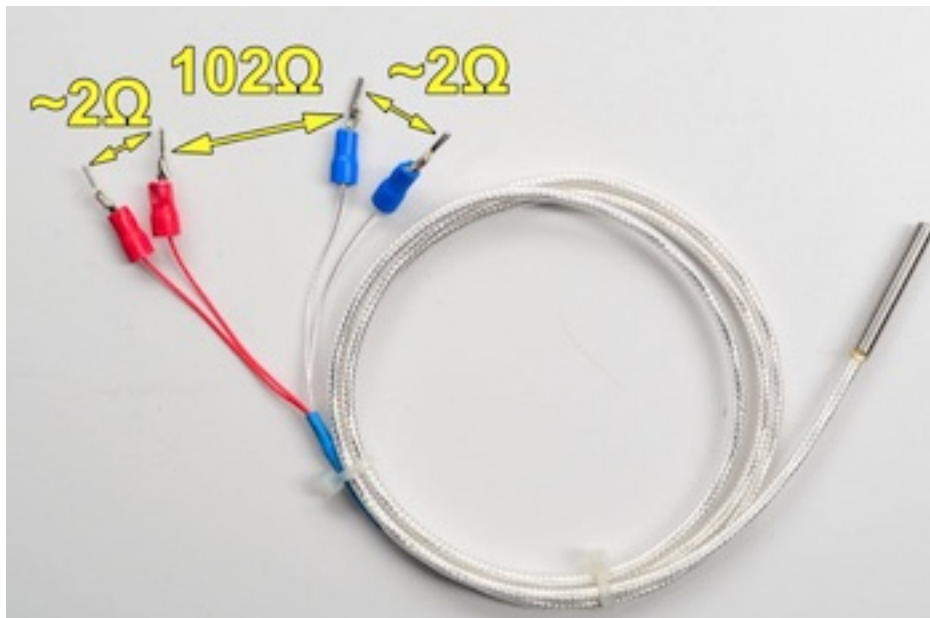
- Configure the plugin
 - **Asset Name Prefix:** A prefix to add to the asset name
 - **GPIO Pin:** The GPIO pin on the Raspberry PI to which the MAX31865 chip select is connected.
- Click *Next*
- Enable the service and click on *Done*

Wiring The Sensor

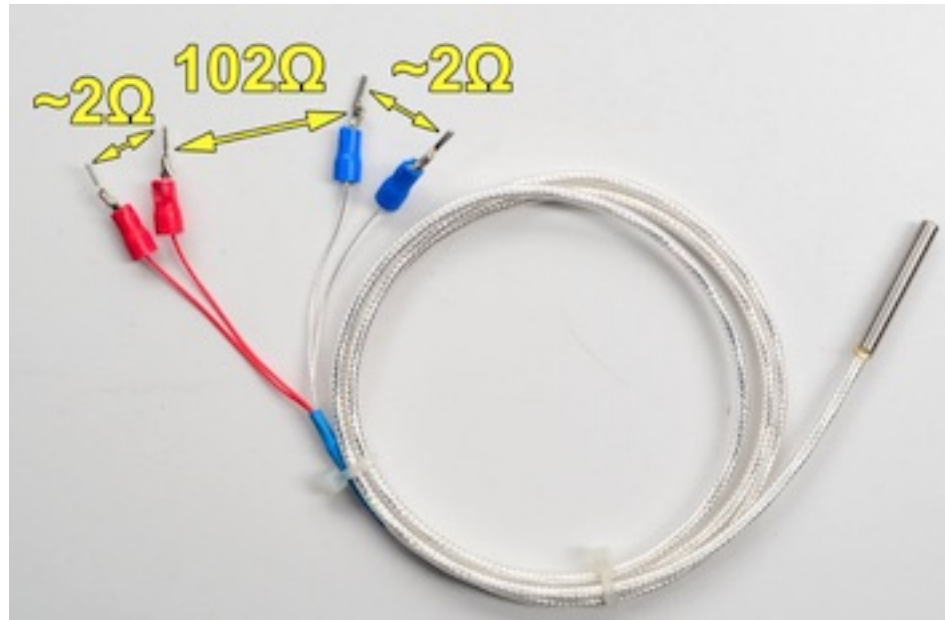
The MAX31865 uses the I2C bus on the Raspberry PI, which requires three wires to connect the bus, it also requires a chip select pin to be wired to a general GPIO pin and power.

MAX 31865 Pin	Raspberry Pi Pin
Vin	3V3
GND	GND
SDI	MOSI
SDO	MISO
CLK	SCLK
CS	GPIO (default GPIO8)

There are two options for connecting a PT100 to the MAX31865, a three wire PT100 or a four wire PT100.



To connect a four wire PT100 to the MAX 31865 the wires are connected in pairs, the two red wires are connected to the RTD- connector pair on the MAX31865 and the two remaining wires are connected to the RTD+ connector pair. If your PT100 does not have red wires or you wish to verify the colours are correct use a multimeter to measure the resistance across the pair of wires. Each pair should show 2 ohms between them and the difference between the two pairs should be 102 ohms, but will vary with temperature.



To connect a three wire sensor connect the red pair of wires across the RTD+ pair of connectors and the third wire on the RTD- block. If your PT100 does not have a pair of red wires, or you wish to verify the colours and have access to a multimeter, the resistance between the red wires should be 2 ohms. ~The resistance to the third wire, from the red pair, will be approximately 102 ohms but will vary with temperature.

If using the 3 wire sensor you must also modify the jumpers on the MAX31865.



Create a solder bridge across the 2/3 Wire jumper, outlined in red in the picture above.

You must also cut the thin wire trace on the jumper block outlined in yellow that runs between the 2 and 4.

Then create a new connection between the 4 and 3 side of this jumper block. This is probably best done with a solder bridge.

14.1.32 Random

The *flir-south-random* plugin is a plugin that will create random data.

To create a south service with the Random plugin

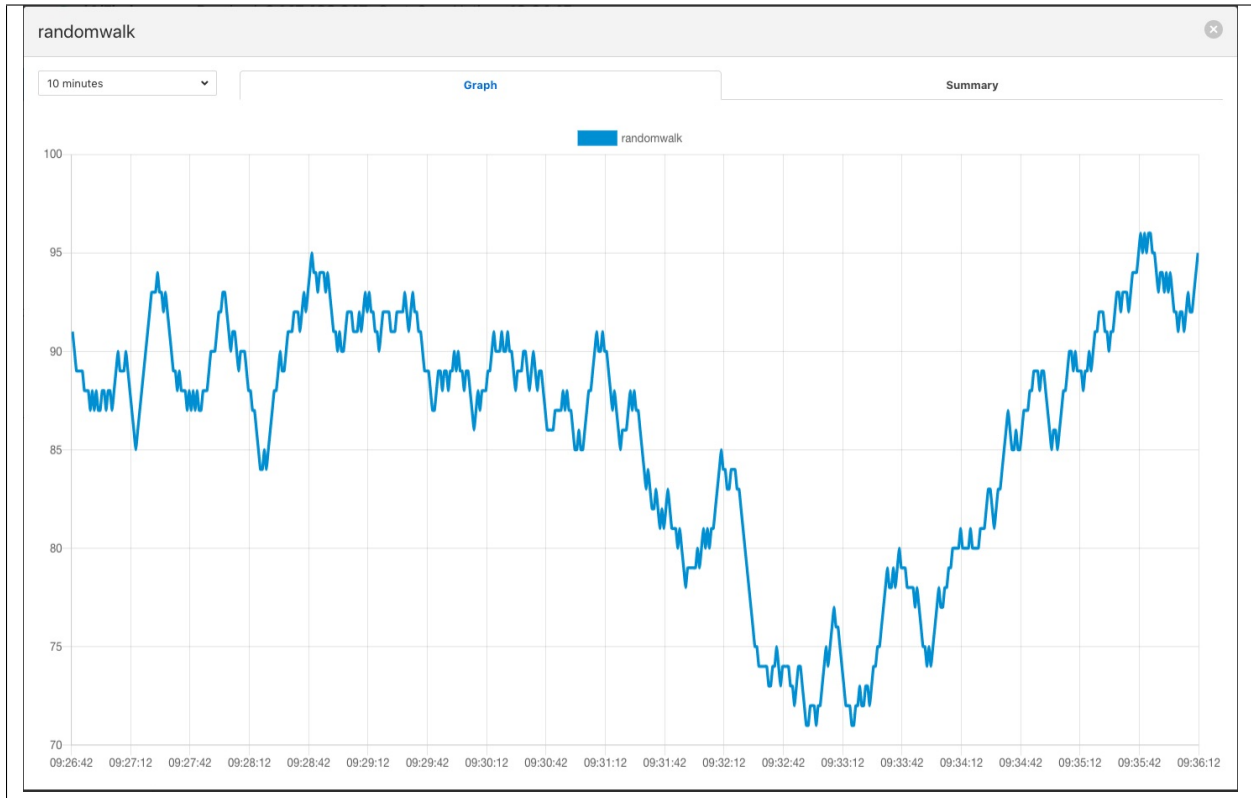
- Click on *South* in the left hand menu bar
- Select *Random* from the plugin list
- Name your service and click *Next*

The screenshot shows a three-step configuration process. Step 1, 'Plugin & Service Name', is active and highlighted with a green circle and line. It contains a text input field labeled 'Asset Name' with the value 'Random' entered. Step 2, 'Review Configuration', is indicated by a green circle and line. Step 3, 'Done', is indicated by a grey circle and line. At the bottom, there are 'Previous' and 'Next' buttons. The 'Next' button is highlighted in blue.

- Configure the plugin
 - **Asset name:** The name of the asset that will be created
- Click *Next*
- Enable the service and click on *Done*

14.1.33 Random Walk

The *flir-south-randomwalk* plugin is a plugin that will create random data between a pair of values. Each new value is based on a random increment or decrement of the previous. This results in an output that appears as follows



To create a south service with the Random Walk plugin

- Click on *South* in the left hand menu bar
- Select *randomwalk* from the plugin list
- Name your service and click *Next*

1
2
3

Plugin & Service Name
Review Configuration
Done

Asset name
randomwalk

Minimum Value
10

Maximum Value
100

Previous
Next

- Configure the plugin
 - **Asset name:** The name of the asset that will be created
 - **Minimum Value:** The minimum value to include in the output

- **Maximum Value:** The maximum value to include in the output
- Click *Next*
- Enable the service and click on *Done*

14.1.34 OPC/UA Safe & Secure South Plugin

The *flir-south-s2opcua* plugin allows FLIR Bridge to connect to an OPC/UA server and subscribe to changes in the objects within the OPC/UA server. This plugin is very similar to the *flir-south-opcua* plugin but is implemented using a different underlying OPC/UA open source library, from Systerel. The major difference between the two is the ability of this plugin to support secure endpoints with the OPC/UA server.

A south service to collect OPC/UA data is created in the same way as any other south service in FLIR Bridge.

- Use the *South* option in the left hand menu bar to display a list of your South services
- Click on the + add icon at the top right of the page
- Select the *s2opcua* plugin from the list of plugins you are provided with
- Enter a name for your south service
- Click on *Next* to configure the OPC/UA plugin

The screenshot shows a configuration window for an OPCUA plugin. At the top, there are three steps: 1. Plugin & Service Name, 2. Review Configuration (current), and 3. Done. The configuration fields are as follows:

- Asset Name:** s2opcua
- OPCUA Server URL:** opc.tcp://localhost:53530/OPCUA/SimulationServer
- OPCUA Object Subscriptions:** A code editor showing a JSON array:


```

1 {
2   "subscriptions": [
3     "ns=3;i=1001",
4     "ns=3;i=1002"
5   ]
6 }
      
```
- Min Reporting Interval (millisec):** 1000
- Security mode:** None
- Security policy:** None
- User authentication policy:** anonymous
- Username:** (empty field)
- Password:** password
- CA certificate authority:** cacert
- Server public key:** OPCUAServer
- Client public key:** clientcert
- Client private key:** clientkey
- Certificate revocation list:** cacrl

At the bottom, there are 'Previous' and 'Next' buttons.

The configuration parameters that can be set on this page are;

- **Asset Name:** This is a prefix that will be applied to all assets that are created by this plugin. The OPC/UA plugin creates a separate asset for each data item read from the OPC/UA server. This is done since the OPC/UA server will deliver changes to individual data items only. Combining these into a complex asset would result in assets that do only contain one of many data points in each update. This can cause upstream systems problems with the every changing asset structure.
- **OPCUA Server URL:** This is the URL of the OPC/UA server from which data will be extracted. The URL should be of the form `opc.tcp://.../`
- **OPCUA Object Subscriptions:** The subscriptions are a set of locations in the OPC/UA object hierarchy that defined which data is subscribed to in the server and hence what assets get created within FLIR Bridge. A fuller description of how to configure subscriptions is shown below.
- **Min Reporting Interval:** This control the minimum interval between reports of data changes in subscriptions. It sets an upper limit to the rate that data will be ingested into the plugin and is expressed in milliseconds.

- **Security Mode:** Specify the OPC/UA security mode that will be used to communicate with the OPC/UA server.

- **Security Policy:** Specify the OPC/UA security policy that will be used to communicate with the OPC/UA server.

- **User authentication policy:** Specify the user authentication policy that will be used when authenticating the connection to the OPC/UA server.
- **Username:** Specify the username to use for authentication. This is only used if the *User authentication policy* is set to *username*.
- **Password:** Specify the password to use for authentication. This is only used if the *User authentication policy* is set to *username*.
- **CA certificate authority:** The name of the root certificate authorities certificate in DER format. This is the certificate authority that forms the root of trust and signs the certificates that will be trusted. If using self signed certificates this should be left blank.
- **Server public key:** The name of the public key of the OPC/UA server specified in the *OPCUA Server URL*. This should be a DER format certificate signed by the certificate authority.
- **Client public key:** The name of the public key of the client application, i.e. the key to use for this plugin. This should be a DER format certificate signed by the certificate authority.
- **Client private key:** The name of the private key of the client application, i.e. the private key the plugin will use. This should be a PEM format key.
- **Certificate revocation list:** The name of the certificate authority's Certificate Revocation List. This is a DER format certificate. If using self signed certificates this should be left blank.

Subscriptions

Subscriptions to OPC/UA objects are stored as a JSON object that contains an array named “subscriptions”. This array is a set of OPC/UA nodes that will control the subscription to variables in the OPC/UA server. Each element in the array is an OPC/UA node id, if that node is the id of a variable then that single variable will be added to the subscription list. If the node id is not a variable, then the plugin will recurse down the object tree below that node and add every variable it finds in this tree to the subscription list.

A subscription list which gives the root node of the OPC/UA server will cause all variables within the server to be added to the subscription list. Care however should be taken as this may be a large number of assets.

Subscription examples

```
{"subscriptions":["5:Simulation","2:MyLevel"]}
```

We subscribe to

- 5:Simulation is a node name under ObjectsNode in namespace 5
- 2:MyLevel is a variable under ObjectsNode in namespace 2

```
{"subscriptions":["5:Sinusoid1","2:MyLevel","5:Sawtooth1"]}
```

We subscribe to

- 5:Sinusoid1 and 5:Sawtooth1 are variables under ObjectsNode/Simulation in namespace 5
- 2:MyLevel is a variable under ObjectsNode in namespace 2

```
{"subscriptions":["2:Random.Double","2:Random.Boolean"]}
```

We subscribe to

- Random.Double and Random.Boolean are variables under ObjectsNode/Demo both in namespace 2

Object names, variable names and namespace indices can be easily retrieved browsing the given OPC/UA server using OPC UA clients, such as .

Certificate Management

The configuration described above uses the names of certificates that will be used by the plugin, these certificates must be loaded into the FLIR Bridge certificate store as a manual process and named to match the names used in the configuration before the plugin is started.

Typically the certificate authorities certificate is retrieved and uploaded to the certificate store along with the certificate from the OPC/UA server that has been signed by that certificate authority. A public/private key pair must also be created for the plugin and signed by the certificate authority. These are uploaded to the FLIR Bridge certificate store.

Openssl may be used to generate and convert the keys and certificates required, and to do this is available as part of the underlying library.

14.1.35 Siemens S7 PLC



The *flir-south-s7* plugin is a south plugin that reads data from a Siemens S7 PLC using the S7 communication protocol. Data can be read from a number of sources within the PLC

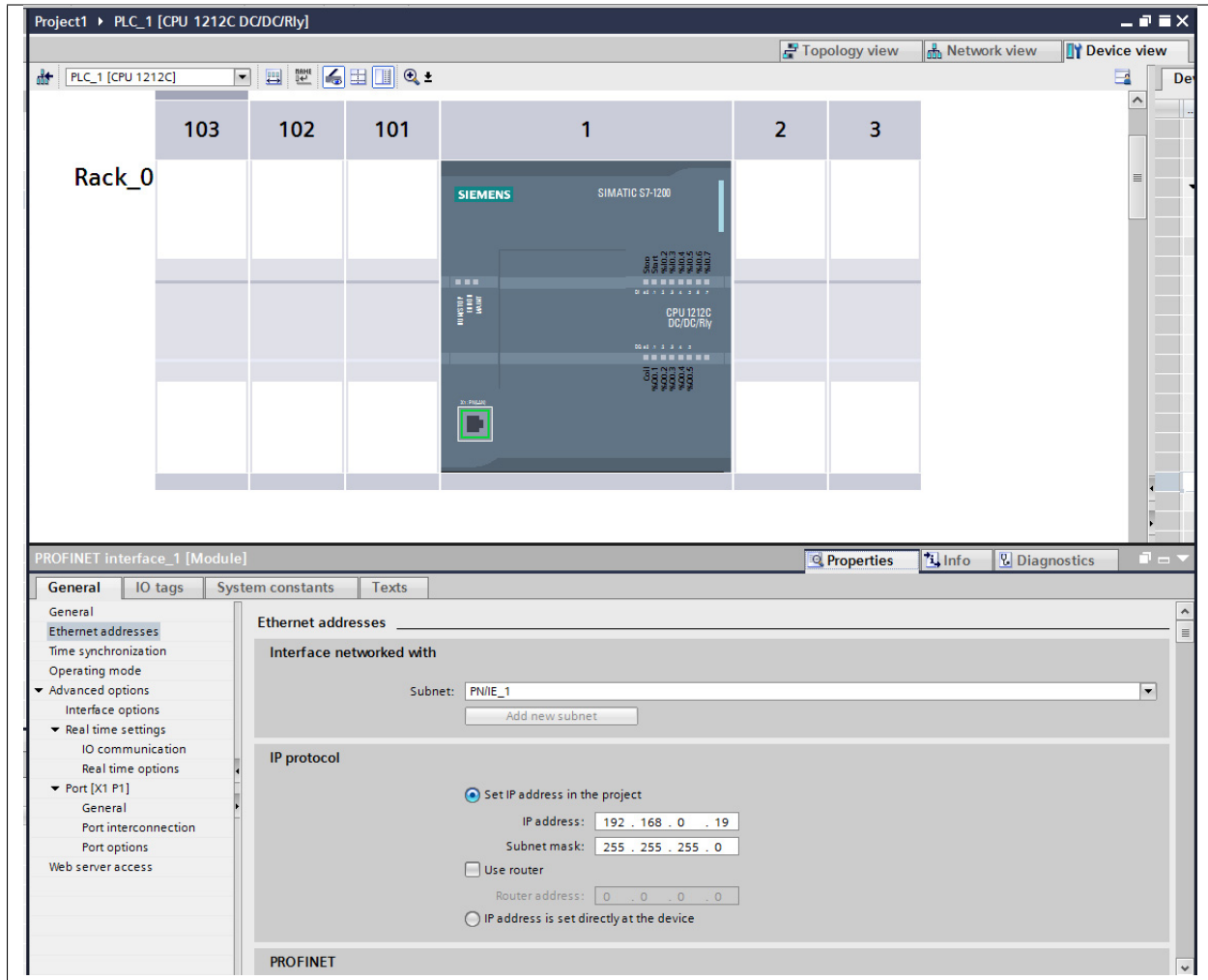
- Data blocks - The data blocks store the state of the PLC
- Inputs - Read the state of the inputs to the PLC
- Outputs - Read the state of the outputs from the PLC
- Merkers - Read from the single bit flag store
- Counters - Read a counter
- Timers - Read a timer

Configuring the PLC

There are a number of configuration steps that must be taken on the PLC itself to support the use of the S7 protocol.

Assigning an IP Address

Using the Siemens TIA console assign an IP address to your PLC. Connect to your PLC and locate the display of the PLC device. Double click on the network connector to bring up the properties for the network interface.

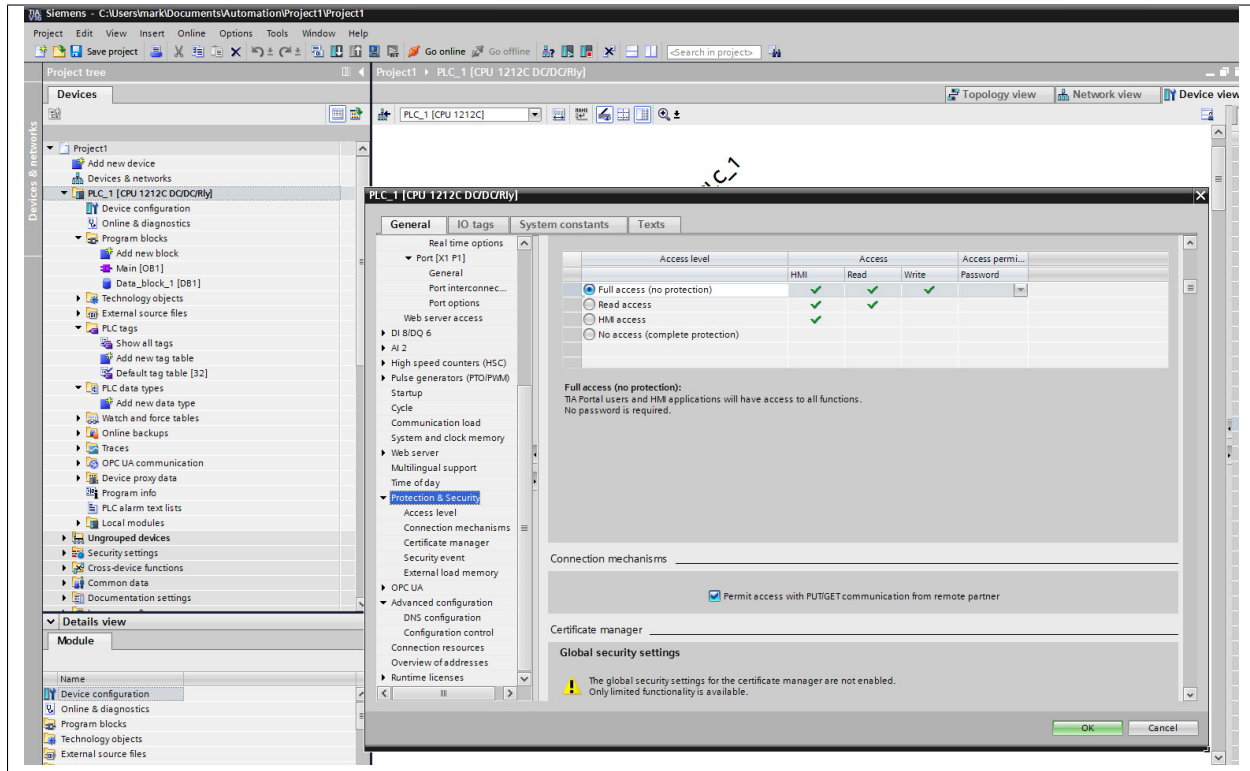


Assign an IP address to your interface and if you require it you may also assign a router to use.

Enable PUT/GET operations

The S7 1200 and 1500 series PLC's require the PUT/GET communication from partners to be enabled in order to retrieve data using the S7 protocol. To permit the PUT/GET network operations on your PLC use the Siemens VIA tool. Note you must be sure that you are offline when you do this. Locate you PLC in the tool and right click on the device select properties and the following dialog will be displayed.

The older S7-300 and S7-400 series do not require this to be done.



Select the protection tab and scroll down to find the checkbox that enables the use of GET/PUT operations. Make sure it is selected for your PLC.

Using the Plugin

To create a south service with the Siemens S7 plugin

- Click on *South* in the left hand menu bar
- Select *S7* from the plugin list
- Name your service and click *Next*

1 Plugin & Service Name 2 Review Configuration 3 Done

Default Asset Name: S7

PLC IP Address: 127.0.0.1

Rack: 0

Slot: 0

Map:

```

1 {
2   "items": [
3     {
4       "datapoint": "dbl",
5       "area": "DB",
6       "DBnumber": 1,
7       "start": 1,
8       "type": "byte"
9     }
10  ]
11 }

```

Previous Next

- Configure the plugin
 - **Default Asset Name:** The name of the asset to use if none is given in each of the data mapping items.
 - **PLC IP Address:** The IP address assigned to your PLC.
 - **Rack:** The rack number to address, usually this is 0 for a standalone PLC.
 - **Slot:** The slot within the rack, most CPU's are in slot 1 of the rack.
 - **Map:** The data mapping for the plugin. This tells the plugin what data to fetch from the PLC.

Map Format

The data mapping uses a JSON document to define the data that should be read. The document is an array of items to read from the PLC, each item is a datapoint within either the default asset or it may be defined as a different asset within the item. An item contains a number of properties

- **asset:** An optional property that can be used to put this item into an asset other than than one defined as the default asset for the service.
- **datapoint:** The name of the data point that the data will be placed in. All items must have a datapoint defined.
- **area:** The area in the PLC that data will be read from. There are a number of areas available
 - **PE:** Process Input - these are the inputs to the PLC
 - **PA:** Process Output - these are the outputs from the PLC
 - **MK:** Merker - a single bit memory used to store flags
 - **DB:** Data Block - the data blocks within the PLC used to store state within the PLC code
 - **CT:** Counter - The counters within the PLC

- **TM:** Timer - The timers within the PLC

You may use the abbreviated area name, e.g. *PA* or the longer name *Process Inputs* interchangeably in the map.

- **DBnumber:** The data block number, this is only required for data blocks and is used to define the block to read.
- **start:** The offset of the start of the item within the data block
- **type:** The type of the data item to read. A number of different types are supported
 - **bit:** A single bit value, mostly used to retrieve the state of a digital input to the PLC
 - **byte:** A 8 bit integer value.
 - **word:** A 16 bit integer value.
 - **dword:** A 32 bit integer value.
 - **real:** A 32 bit floating point value.
 - **counter:** A 16 bit counter.
 - **timer:** a 16 bit timer.

A simple data mapping that wanted to read the state of two digital inputs to the PLC, say DI0 and DI2, and wanted to labeled these as datapoints “Stop” and “Start” within the default asset would consist of two items as follows

```
{
  "items" : [
    {
      "datapoint": "Stop",
      "area": "PE",
      "start": 0,
      "type": "bit"
    },
    {
      "datapoint": "Start",
      "area": "PE",
      "start": 2,
      "type": "bit"
    }
  ]
}
```

In this case we set start to 0 for DI0 as it is the first digital input in the set. DI2 has a start of 2 as it is the second input. We use the type of *bit* to return a simple 0 or 1 to indicate the state of the input. We could use *byte* instead, this would return the 8 inputs states encoded as a binary number.

```
{
  "datapoint": "Inputs",
  "area": "PE",
  "start": 0,
  "type": "byte"
}
```

Since *start* is set to 0 and *type* is byte, then we return the state of the 8 inputs. We can do the same thing using the longer name form of the area as follows.

```
{
  "datapoint": "Inputs",
  "area": "Process Inputs",
  "start": 0,
  "type": "byte"
}
```

To add in a digital output, say DO4 and label that running, we would add another item to the map

```
{
  "datapoint": "Running",
  "area": "PA",
  "start": 4,
  "type": "bit"
}
```

If we assume we have a data block that we wish to read data from that appears as follows

	Name	Data type	Offset	Start value	Retain	Accessible f...	Writa...	Visible in ...	Setpoint	Comment
1	Static									
2	count	Int	0.0	0		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
3	state	Int	2.0	1		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
4	failures	DWord	4.0	3		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
5	rate	Real	8.0	1.3		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
6	running	Bool	12.0	false		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
7	downtime	Time	14.0	T#575_656MS		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	

Then we can setup a number of items in the map to retrieve these values and place them in data points. The items that would read this data block would be

```
{
  "datapoint": "count",
  "area": "DB",
  "DBnumber": 1,
  "start": 0,
  "type": "word"
},
{
  "datapoint": "state",
  "area": "DB",
  "DBnumber": 1,
  "start": 2,
  "type": "word"
},
{
  "datapoint": "failures",
  "area": "DB",
  "DBnumber": 1,
  "start": 4,
  "type": "dword"
},
{
  "datapoint": "rate",
  "area": "DB",
```

(continues on next page)

(continued from previous page)

```

    "DBnumber" : 1,
    "start": 8,
    "type": "word"
  },
  {
    "datapoint": "running",
    "area": "DB",
    "DBnumber" : 1,
    "start": 12,
    "type": "word"
  },
  {
    "datapoint": "downtime",
    "area": "DB",
    "DBnumber" : 1,
    "start": 14,
    "type": "timer"
  }
}

```

For clarity we have used the name in the data block as the datapoint name, but these need not be the same.

If there is an error in the map definition for a given item then that item is ignored and a message is written to the error log. For example if a bad area name is given

```

Jun 25 08:53:04 flir-18 FLIR Bridge S7[6121]: ERROR: Invalid area Data specified in ↵
↵device mapping for S7 db1-bad
Jun 25 08:53:04 flir-18 FLIR Bridge S7[6121]: ERROR: Discarded invalid item in map ↵
↵for datapoint db1-bad

```

If a Data Block is missing its DBnumber property then the following style of error will be produced.

```

Jun 25 08:39:07 flir-18 FLIR Bridge S7[6121]: ERROR: Missing data block number in map ↵
↵for S7, db1-bad. A data block number must be specified for a data block area read.
Jun 25 08:39:07 flir-18 FLIR Bridge S7[6121]: ERROR: Discarded invalid item in map ↵
↵for datapoint db1-bad

```

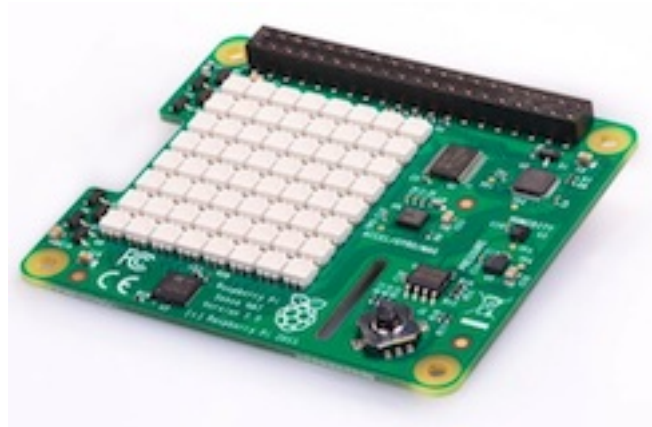
Other errors that can occur include

```

Jun 25 08:57:28 flir-18 FLIR Bridge S7[6121]: ERROR: Missing start in map for ↵
↵datapoint db1-bad
Jun 25 08:57:46 flir-18 FLIR Bridge S7[6121]: ERROR: Missing type in map for ↵
↵datapoint db1-bad

```

14.1.36 SenseHAT



The *flir-south-sensehat* is a plugin that uses the Raspberry Pi Sense HAT sensor board. The Sense HAT has an 8x8 RGB LED matrix, a five-button joystick and includes the following sensors:

- Gyroscope
- Accelerometer
- Magnetometer
- Temperature
- Barometric pressure
- Humidity

In addition it has an 8x8 matrix for RGB LED's, these are not included in the devices the plugin supports.

Individual sensors can be enabled or disabled separately in the configuration. Separate assets are created for each sensor within FLIR Bridge with individual controls over the naming of these assets.

Note: The Sense HAT plugin is only available on the Raspberry Pi as it is specific the GPIO pins of that device.

To create a south service with the Sense HAT

- Click on *South* in the left hand menu bar
- Select *sensehat* from the plugin list
- Name your service and click *Next*

The screenshot shows a configuration window titled 'Review Configuration' (step 2 of 3). It contains a list of sensors with checkboxes and text input fields for their names. The 'Asset Name Prefix' is set to 'sensehat/'. The sensors listed are Pressure, Temperature, Humidity, Gyroscope, Accelerometer, Magnetometer, and Joystick, all of which are currently enabled. The 'Previous' button is disabled, and the 'Next' button is active.

Sensor Type	Enabled	Sensor Name
Asset Name Prefix		sensehat/
Pressure Sensor	<input checked="" type="checkbox"/>	pressure
Temperature Sensor	<input checked="" type="checkbox"/>	temperature
Humidity Sensor	<input checked="" type="checkbox"/>	humidity
Gyroscope Sensor	<input checked="" type="checkbox"/>	gyroscope
Accelerometer Sensor	<input checked="" type="checkbox"/>	accelerometer
Magnetometer Sensor	<input checked="" type="checkbox"/>	magnetometer
Joystick Sensor	<input checked="" type="checkbox"/>	joystick

- Configure the plugin
 - **Asset Name Prefix:** An optional prefix to add to the asset names.
 - **Pressure Sensor:** A toggle control to turn on or off collection of pressure information
 - **Pressure Sensor Name:** Set a name for the Pressure sensor asset
 - **Temperature Sensor:** A toggle control to turn on or off collection of temperature information
 - **Temperature Sensor Name:** Set a name for the temperature sensor asset
 - **Humidity Sensor:** A toggle control to turn on or off collection of humidity information
 - **Humidity Sensor Name:** Set a name for the humidity sensor asset
 - **Gyroscope Sensor:** A toggle control to turn on or off collection of gyroscope information
 - **Gyroscope Sensor Name:** Set a name for the gyroscope sensor asset
 - **Accelerometer Sensor:** A toggle to turn on or off collection of accelerometer data
 - **Accelerometer Sensor Name:** Set a name for the accelerometer sensor asset
 - **Magnetometer Sensor:** A toggle control to turn on or off collection of magnetometer data
 - **Magnetometer Sensor Name:** Set a name for the magnetometer sensor asset
 - **Joystick Sensor:** A toggle control to turn on or off collection of joystick data
 - **Joystick Sensor Name:** Set a name for the joystick sensor asset
- Click *Next*
- Enable the service and click on *Done*

14.1.37 Simple REST with Payload Scripting

The *flir-south-simple-rest* plugin uses REST calls to receive API responses from sensors or other sources. The plugin make HTTP or HTTPS GET requests to retrieve API responses, HTTP header fields can also be added via the plugin configuration. It then uses an optional script, written in Python, that converts the message into a JSON document and pushes data to the FLIR Bridge System. However it also has a set of built in rules for interpreting some common payload formats which enable it to be used without providing a script in a large number of common cases.

Configuration

When adding a south service with this plugin the same flow is used as with any other south service. The configuration page for the plugin is as follows.

The screenshot shows a configuration form for a REST API asset. The form is organized into two main sections: configuration fields on the left and a list of headers and scripts on the right.

Configuration Fields:

- Asset Name:** A text input field containing the value "rest".
- URL:** A text input field containing the value "http://server/location".
- Headers:** A table with one row containing the value "{}".
- Selection Method:** A dropdown menu with the value "None".
- ID Parameter:** A text input field.
- Initial ID:** A text input field.
- ID Field:** A text input field.
- Start:** A text input field.
- End:** A text input field.
- Timestamp:** A text input field.
- Time Format:** A text input field.
- Timezone:** A text input field containing the value "+00:00".
- Collapse:** A checkbox that is checked.
- Asset Field:** A text input field.
- Script:** A text input field.

Buttons:

- Choose files:** A button located at the bottom right of the form.
- No file chosen:** A label next to the "Choose files" button.

- **Asset Name:** The name of the asset the plugin will create for each message, unless the convert function returns an explicit asset name to be used.
- **URL:** The URL of the REST API to be called. This should be a complete URL, including the http or https protocol to use.
- **Headers:** An optional set of headers to include in the REST API call. The headers are encoded as a JSON document as a set of name/value pairs within a JSON object.
- **Selection Method:** The plugin supports a number of methods for selecting the data should be returned. The choices are return all the data, return data based on an ID or return data based on time. See [Selection Method](#) for more details.

The image shows a configuration window with three labeled input fields: 'Selection Method', 'ID Parameter', and 'Initial ID'. A dropdown menu is open for the 'Selection Method' field, displaying three options: 'None' (which is selected and highlighted in blue with a checkmark), 'ID Based', and 'Time Based'.

- **ID Parameter:** An optional URL query parameter to add to each call to the URL. This is expected to be a numeric value that gets passed to the API and is used for implementing ID passing to calls. This is only valid if the selection method *ID Based* has been chosen.
- **Initial ID:** The initial value to pass for the query parameter. This may be used on the first call only if the *ID Based* selection method is chosen.
- **ID Field:** This defines a data field that is ingested that will be used for second and subsequent calls to the API as the new value of ID. This is only used with a selection method of *ID Based*.
- **Start:** The name of the query parameter to add to the URL to indicate the start time if a selection method of *Time Based* has been chosen.
- **End:** The name of the query parameter to add to the URL to indicate the end time if a selection method of *Time Based* has been chosen.
- **Time Format:** The format to both pass the timestamps into the query parameters using and also to interpret the timestamps returned in the payload.
- **Timezone:** The timezone to use for the start and end times that are sent in the API request and also when timestamps are read from the API response. Timezone is expressed as an offset in hours and minutes from UTC for the local timezone of the API. E.g. -08:00 for PST timezones.
- **Collapse:** Collapse the returned returning to a flat structure, if not enabled a nested reading will be produced.
- **Timestamp:** The name of the item in the response payload that should be treated as the timestamp for the reading.
- **Asset Field:** The name of a field in the response payload that should be treated as the asset name to use for the reading. If this is left empty or the data does not contain a field with this name then the default asset name configured in the *Asset Name* configuration item will be used.
- **Script:** The Python script to execute for message processing. Initially a file must be uploaded, however once uploaded the user may edit the script in the box provided. A script is optional.

Selection Method

The plugin supports two methods to select data to be retrieved from the API that is called, these methods are designed for use with an API that is maintaining historic data and provided a mechanism to present the same historic data being read multiple times. If your API does not store historic data then you may select the method *None* to simply retrieve all the data available via the API.

ID Based

The select mechanism *ID Based* is designed for API that give each value some form of ID that increases over time. When a call is made you pass the value of the ID for the next data item you wish to read. This method is used in

conjunction with 3 other parameters. These parameters are used to control the name of the query parameter to add to the URL, *ID Parameter*. This name will be used to pass in the ID to be read and is added to the URL that is configured. In the first call using the *ID Based* method the value of *Initial ID* will be passed as the value. In subsequent calls the maximum value of the data field name as per the *ID Field* configuration parameter will be used as the value of the ID parameter.

ID Parameter is the name of the parameter that is passed in the requests, it is appended to the configured *URL* along with the current value for the parameter.

For example if the *URL* is configured as `http://api-server.com/api/v1/data?user=dianomic` and the *ID Parameter* is defined as *requestID* with the *Initial ID* of 100, then the full URL that is used in the call will be

```
http://api-server.com/api/v1/data?user=dianomic&requestID=100
```

The URL used in the next call will be dependent on the setting of *ID Field*. If it is left empty then the value last used will be incremented for the next call, provided the previous call was successful. In our above example this would result in the next call using the URL

```
http://api-server.com/api/v1/data?user=dianomic&requestID=101
```

If the first call had failed, then the next call would use the same value for our parameter.

A more common case is when the data returned contains ID values for each returned value, in this case the *ID Field* configuration option is set and the values taken from the response will generate the next ID to use. For example, if the response payload returns sets of readings, each identified by a field called *id*, then set *ID Field* to *id*. A response payload that returned *id*'s 125, 126 & 127 would then cause the next request to send a value for the parameter of 128.

```
http://api-server.com/api/v1/data?user=dianomic&requestID=128
```

Time Based

The selection mechanism *Time Based* is designed for an API that returns values for a time window. It requires two parameters to be passed in the request, *Start* and *End*, to specify the time window to the server. The format of the timestamps passed to the server are defined by the *Time Format* configuration parameter.

%% The % character.

%a or %A The name of the day of the week according to the current locale, in abbreviated form or the full name.

%b or %B or %h

The month name according to the current locale, in abbreviated form or the full name.

%c The date and time representation for the current locale.

%C The century number (0–99).

%d or %e The day of month (1–31).

%D Equivalent to `%m/%d/%y`. (This is the American style date, very confusing to non- Americans, especially since `%d/%m/%y` is widely used in Europe. The ISO 8601 standard format is `%Y-%m-%d`.)

%H The hour (0–23).

%I The hour on a 12-hour clock (1–12).

%j The day number in the year (1–366).

%m The month number (1–12).

- %M** The minute (0–59).
- %n** Arbitrary white space.
- %p** The locale’s equivalent of AM or PM. (Note: there may be none.)
- %r** The 12-hour clock time (using the locale’s AM or PM). In the POSIX locale equivalent to **%I:%M:%S %p**. If **t_fmt_ampm** is empty in the **LC_TIME** part of the current locale, then the behavior is undefined.
- %R** Equivalent to **%H:%M**.
- %S** The second (0–60; 60 may occur for leap seconds; earlier also 61 was allowed).
- %t** Arbitrary white space.
- %T** Equivalent to **%H:%M:%S**.
- %U** The week number with Sunday the first day of the week (0–53). The first Sunday of January is the first day of week 1.
- %w** The ordinal number of the day of the week (0–6), with Sunday = 0.
- %W** The week number with Monday the first day of the week (0–53). The first Monday of January is the first day of week 1.
- %x** The date, using the locale’s date format.
- %X** The time, using the locale’s time format.
- %y** The year within century (0–99). When a century is not otherwise specified, values in the range 69–99 refer to years in the twentieth century (1969–1999); values in the range 00–68 refer to years in the twenty-first century (2000–2068).
- %Y** The year, including century (for example, 1991).

When using the *Time Based* selection mechanism two parameters may be appended. For example if the *URL* is configured as *http://api-server.com/api/v1/data?user=dianomic*, the configuration option *Start* is defined as *startTime* and *End* as *endTime*, with the *Time Format* set to be *%Y-%m-%dT%H:%M:%s*, then the full URL that is used in the call will be

```
http://api-server.com/api/v1/data?user=dianomic&startTime=2021-07-11T15:12:34&
↪endTime=2021-07-12T12:45:12
```

If this call succeeds then the next call will use the *endTime* from this call as the *startTime* for the next call. The *endTime* is always the current time.

Request URL Handling

The plugin makes HTTP (or HTTPS) GET requests to the configured URL, this may include parameter passing. Parameters used be encoded within the URL of in the plugin configuration, however if a *Selection Method* other than *None* is selected extra parameters will be added to the request URL.

Response Payload Handling

If the payload of the REST response is a JSON document with simple key/value pairs, e.g.

```
{ "temperature" : 23.1, "humidity" : 47.2 }
```

Then no translation script is required, each key/value pair will become a data point within an asset whose name is set in the configuration of the plugin. A working example of this is the `/flir/ping` API call of FLIR Bridge itself, it produces a response,

```
{
  "uptime": 27,
  "dataRead": 1063459,
  "dataSent": 617310,
  "dataPurged": 1063024,
  "authenticationOptional": true,
  "serviceName": "FLIR Bridge",
  "hostName": "flir-18",
  "ipAddresses": [
    "192.168.0.173"
  ],
  "health": "green",
  "safeMode": false,
  "version": "1.9.1"
}
```

This results in an asset which has data points for all the string, numeric and boolean items with the response. In this case the `ipAddresses` item is ignored as FLIR Bridge does not currently support string array type data.

If the response payload included nested JSON objects then these will be included also. For example if the response payload was

```
{
  "motor" : {
    "speed" : 12345,
    "current" : 1.4
  },
  "gearbox" : {
    "ratio" : 64,
  }
}
```

The three values would be extracted, *speed*, *current* and *ratio*. How these values are represented will depend on the setting of the *Collapse Data* configuration option. If this is set to true then a flat reading will be created for each of the three values. If it is set to false then a reading with two objects will be created, one for the motor and one for the gearbox. Within these they will contain the values for the appropriate object. The choice of flattening the data will depend on how the user wishes to use this data upstream within FLIR Bridge.

If the payload is a JSON document that is an array rather than an object, then it is interpreted as a set of readings. For example a payload that is an array of numbers such as

```
[
  12.4, 15.8, 18.2
]
```

Will result in a set of readings, one per value being created. The asset name and data point name will be taken for the *Asset* configuration option. This same rule applies for arrays of integers, floating point numbers, string or booleans.

The payload may also be an array of objects, in which case each object will be an asset, with the members of the object becoming the data points. These objects may be nested in which case they will follow the same rules as the motor and gearbox example above.

```
[
  {
```

(continues on next page)

(continued from previous page)

```

    "motor" : {
      "speed" : 12345,
      "current" : 1.4
    },
    "gearbox" : {
      "ratio" : 64,
    }
  },
  {
    "motor" : {
      "speed" : 12345,
      "current" : 1.4
    },
    "gearbox" : {
      "ratio" : 64,
    }
  }
},
]

```

This will create two assets with the name of the asset as the *Asset* configuration option.

The default asset naming can be overridden by setting a value for the configuration item *Asset Field*. This can be used to extract a value from the data returned by the API as the name to use for the resultant asset. If the *Asset Field* configuration item is set to *machine* and the payload returned by the API calls is as follows.

```

[
  {
    "machine" : "CNC14698",
    "motor" : {
      "speed" : 12345,
      "current" : 1.4
    },
    "gearbox" : {
      "ratio" : 64,
    }
  },
  {
    "machine" : "CNC15217",
    "motor" : {
      "speed" : 12345,
      "current" : 1.4
    },
    "gearbox" : {
      "ratio" : 64,
    }
  }
],
]

```

The result would be two assets called *CNC14698* and *CNC15217*.

Also if the payload is a simple numeric value the plugin will accept this and create an asset with the data point name matching the topic on which the value was given in the payload.

If the message format is not a JSON document that can be parsed using the built in rules or is in some other format then a Python script should be provided that turns the message into a JSON format.

An example script, assuming the payload in the message is simply a value, might be as follows

```
def convert(message) :  
    return {  
        'temperature' : float(message)  
    }
```

Note that the message is passed as a string and the data we wish to ingest into FLIR Bridge in this case is assumed to be a floating point value. The example above of course is unnecessary as the plugin can consume this data without the need of a script.

The script could return either one or two values.

The script should return the JSON document as a Python DICT in the case of a single value.

The script should return a string and a JSON document as a Python DICT in the case of two values, the first of these values is the name of the asset to use and overrides the default asset naming defined in the plugin configuration.

First case sample:

```
def convert(message) :  
    return {'temperature_1': 10.2}
```

Second case sample:

```
def convert(message) :  
    return "ExternalTEMP", {'temperature_3': 11.3}
```

A single API call can return reading data for multiple assets. In this case the script can return a more complex JSON document that contains both the asset name and the data points for that asset. The return document should return a single JSON object called *readings* and within that a set of readings, one per asset, expressed as a number of reading objects. The key of each of these objects becomes the asset name and the value of each is the data points within the asset.

As an example, if a single API call gives us back both data on a motor and on a machine tool, we can process that API response into two distinct assets; *motor* and *tool*, each with its own set of data points. In this case the *rpm* and *current* of the motor and the *temperature* and *coolant* flow rate for the machine tool.

```
{  
    "readings" :  
        {  
            "motor" : {  
                "rpm"      : 8450,  
                "current"  : 1.3  
            },  
            "tool" : {  
                "temperature" : 32.1,  
                "coolant"     : 147  
            }  
        }  
}
```

If a script is returning this more complex JSON object it should not return an asset name, if it does return an asset name with this JSON format then the asset name will be ignored.

Timestamp Treatment

The default timestamp for a reading collected via this plugin will be the time at which the reading was taken, however it is possible for the API that is being called to include a different timestamp.

Returning a data point called whose name is defined in the *Timestamp* configuration option will result in the value of that data point being used as the timestamp. This data point will not be added to the reading. The default name of the timestamp is *timestamp*.

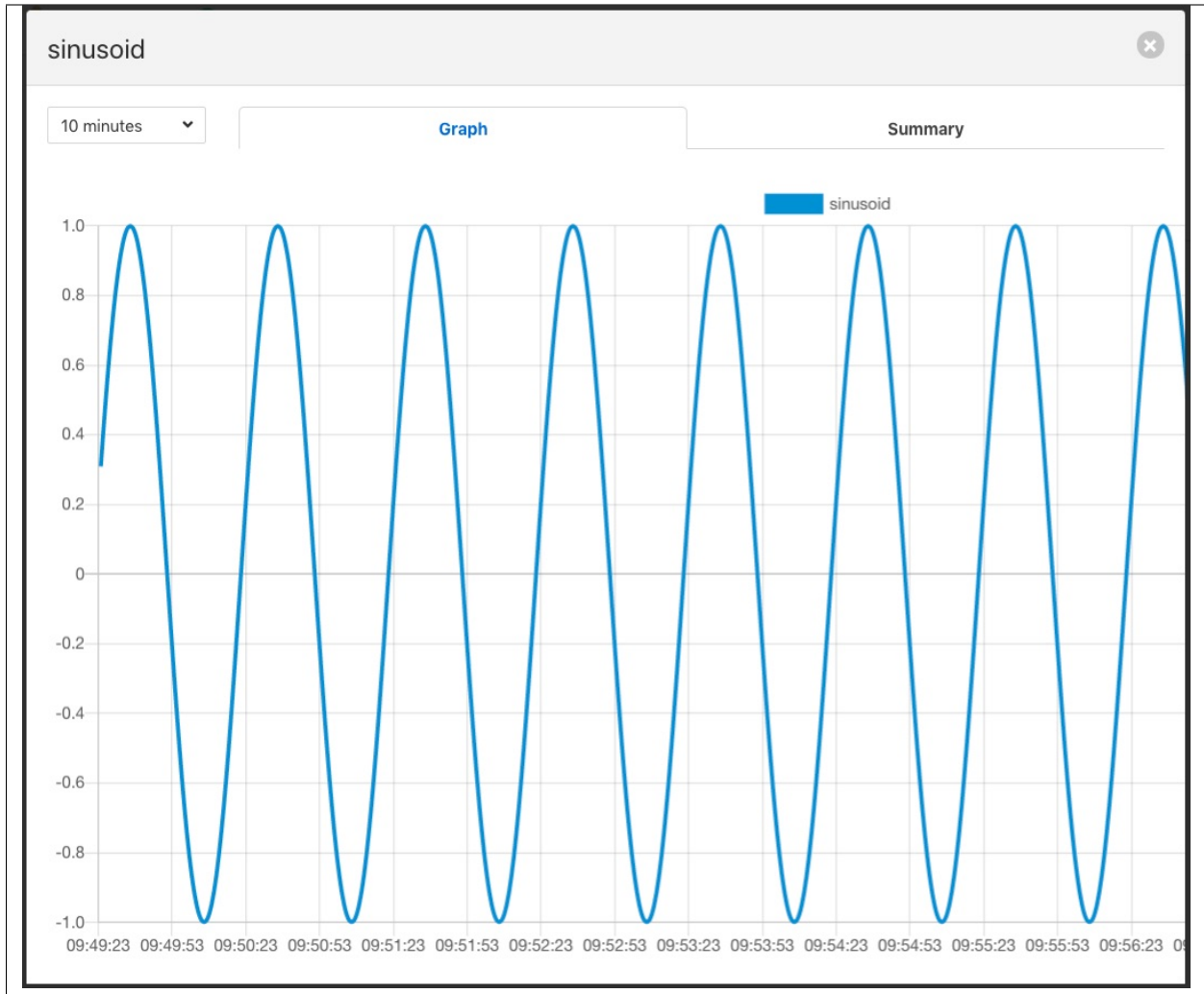
The timestamp data point should be a string and the timestamp should be formatted to match the definition given in the *Time format* configuration parameter. The format is based on the standard Linux `strptime` formatting options and is discussed above in the section discussing the `:ref:ref::time_based` selection method. It should be noted however that this timestamp handling in the payload is independent of the selection method chosen.

The timezone may be set by using the *Timezone* configuration parameter to set the offset of the timezone in which the API is running.

The plugin will automatically filter out a second or subsequent readings that have the same timestamp value as previous reading for that same asset. This allows an API which returns the timestamp of the data to be called multiple times and the data will only be ingested once for the given timestamp. The result is the polling rate of the south service can be set independently of the rate the data changes.

14.1.38 Sinusoid

The *flir-south-sinusoid* plugin is a south plugin that is primarily designed for testing purposes. It produces as it's output a simple sine wave, the period of which can be adjusted by changing the poll rate in the advanced settings of the south service into which it is loaded.



There is very little configuration required for the *sinusoid* plugin, merely the name of the asset that should be written. This can be useful if you wish to have multiple sinusoid in your FLIR Bridge system.

The frequency of the sinusoid can be adjusted by changing the poll rate of the sinusoid plugin. To do this select the

South item from the left-hand menu bar and then click on the name of your sinusoid service. You will see a link labeled *Show Advanced Config*, click on this to reveal the advanced configuration.

Sine South Service

Asset name

sinusoid

Hide Advanced Config

Maximum Reading Latency (mS)

5000

Maximum buffered Readings

100

Reading Rate

10

Throttle

☐

Reading Rate Per

second

Minimum Log Level

warning

Enabled

☒











































Applications

Cancel

Save

Amongst the advanced setting you will see one labeled *Reading Rate*. This defaults to 1 per second. The sinusoid takes 60 samples to complete one cycle of the sine wave, therefore it has a periodicity of 1 minute, or 0.0166Hz. If the *Reading Rate* is st to 60, then the frequency of the output becomes 1Hz.

14.1.39 System Information

Asset	Readings		
system/cpuUsage_all	94		
system/diskTraffic_loop0	94		
system/diskTraffic_loop1	94		
system/diskTraffic_loop2	94		
system/diskTraffic_sda	94		
system/diskTraffic_sdb	94		
system/diskUsage_dev/loop0	94		
system/diskUsage_dev/loop1	94		
system/diskUsage_dev/sda2	94		
system/diskUsage_dev/sdb1	94		
system/diskUsage_tmpfs	470		
system/diskUsage_udev	94		
system/hostname	94		
system/loadAverage	94		
system/memInfo	94		
system/networkTraffic_enp0s3	94		
system/networkTraffic_lo	94		
system/pagingAndSwappingEvents	94		
system/platform	94		
system/processes	94		
system/uptime	94		

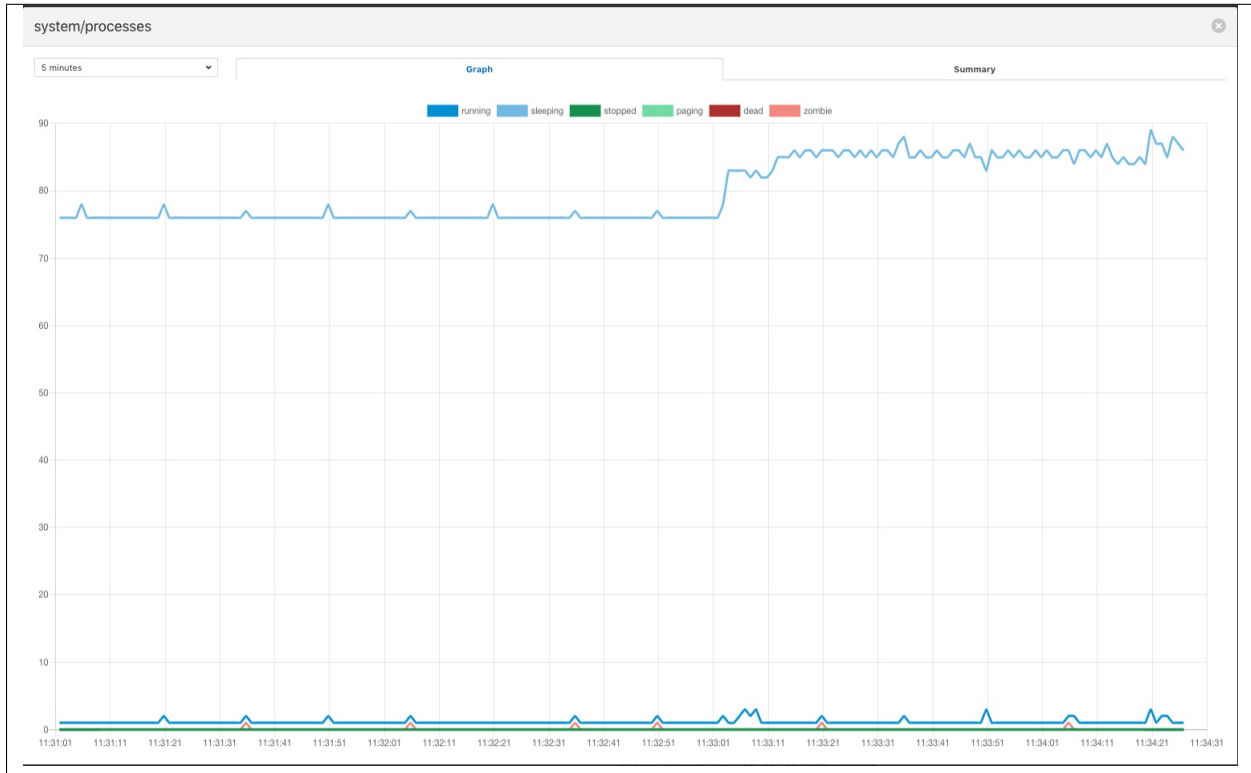
The *flir-south-systeminfo* plugin implements a that collects data about the machine that the FLIR Bridge instance is running on. The plugin is designed to allow the monitoring of the edge devices themselves to be included in the monitoring of the equipment involved in processing environment.

The plugin will create a number of assets, in general there are one or more assets per device connected in the case of disks and network interfaces. There are also some generic assets to measure;

- CPU Usage
- Host name
- Load Average
- Memory Usage

- Paging and swapping
- Process information
- System Uptime

A typical output for one of these assets, in this case the processes asset is shown below



To create a south service with the systeminfo plugin

- Click on *South* in the left hand menu bar
- Select *systeminfo* from the plugin list
- Name your service and click *Next*

- Configure the plugin
 - **Asset Name Prefix:** The asset name prefix for the assets created by this plugin. The plugin will create a number of assets, the exact number is dependent on the number of devices attached to

the machine.

- Click *Next*
- Enable the service and click on *Done*

14.1.40 Advantech USB-4704



The *flir-south-usb4704* plugin is a south plugin that is designed to gather data from an Advantech USB-4704 data acquisition module. The module supports 8 digital inputs and 8 analogue inputs. It is possible to configure the plugin to combine multiple digital input to create a single numeric data point or have each input as a boolean data point. Each analogue input, which is a 14 bit analogue to digital converter, becomes a single numeric data point in the range 0 to 16383, although a scale and offset may be applied to these values.

To create a south service with the USB-4704

- Click on *South* in the left hand menu bar
- Select *usb4704* from the plugin list
- Name your service and click *Next*

1 Plugin & Service Name 2 Review Configuration 3 Done

Asset Name

Connections

```

1 {
2   "analogue_example": {
3     "type": "analogue",
4     "pin": "AI0",
5     "name": "value1",
6     "scale": 0.1
7   },
8   "digital_example": {
9     "type": "digital",
10    "pins": [
11      "DI0",
12      "DI1",
13      "DI2"

```

- Configure the plugin
 - **Asset Name:** The name of the asset that will be created with the values read from the USB-4704
 - **Connections:** A JSON document that describes the connections to the USB-4704 and the data points within the asset that they map to. The JSON document is a set of objects, one per data point. The objects contain a number of key/value pairs as follow

Key	Description
type	The type of connection, this may be either digital or analogue.
pin	The analogue pin used for the connection.
pins	An array of pins for a digital connection, the first element in the array becomes the most significant bit of the numeric value created.
name	The data point name within the asset.
scale	An optional scale value that may be applied to the value.

- Click on *Next*
- Enable your service and click on *Done*

14.1.41 South Webcam Media Plugin

The plugin keeps on taking a video frame from directory or webcam and saves into a directory. It also appends the name of the saved files in the reading generated.

1 Plugin & Service Name 2 Review Configuration 3 Done

Asset Name

Media type

Media storage directory

File data format

Repeat loop ☐

Camera number

Frames per minute processed

- **‘assetName’**: type: string default: **‘WebcamImages’**: Name of Asset output.
- **‘mediaType’**: type: string default: **‘directory’**: Source from which the media files are generated
- **‘mediaDir’**: type: string default: **‘webcam’**: If the mediaType is camera then the directory where media will be stored. If the mediaType is directory then it is the name of directory inside FLIR_ROOT/data where images are stored.
- **‘dataFormat’**: type: enumeration default: **‘IMG’**: Format of files in ‘mediaDir’
- **‘repeatLoop’**: type: boolean default: **‘false’**: If the mediaType is directory is reload when you hit the end playing the images from directory.
- **‘cameraNumber’**: type: integer default: **‘0’**: Number associated with /dev/video in your file system. for example /dev/video0 then use 0.
- **‘fpm’**: type: float default: **‘10.0’**: frames to save per minute.

Execution

To run the south webcam media plugin you can either

1. Copy some images inside some directory in FLIR_ROOT/data. Let’s say the directory name is pics. Run the following command.

```
curl -sX POST http://localhost:8081/flir/service -d '{"name":"My_web_cam","type":
↳ "south","plugin":"webcam_media","enabled":true,"config":{"assetName":{"value":
↳ "WebcamImages"}, "imageDir":{"value":"pics"}, "mediaType":{"value":"directory"},
↳ "fpm":{"value":"10.0"}}}' |
```

2. Connect a camera to the machine and run the following command.

```
$ v4l2-ctl --list-formats-ext --device /dev/video0
You will see something like
'YUYV' (YUYV 4:2:2)
  Size: Discrete 640x480
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 720x480
```

(continues on next page)

(continued from previous page)

```

Interval: Discrete 0.033s (30.000 fps)
Size: Discrete 1280x720
Interval: Discrete 0.033s (30.000 fps)
Size: Discrete 1920x1080
Interval: Discrete 0.067s (15.000 fps)
Interval: Discrete 0.033s (30.000 fps)
Size: Discrete 2592x1944
Interval: Discrete 0.067s (15.000 fps)
Size: Discrete 0x0

```

Now we know that the id 0 is functional. If no output then try 1,2,3 and so on.

Finally launch the plugin using

```

curl -sX POST http://localhost:8081/flir/service -d '{"name":"My_web_cam","type":
↪"south","plugin":"webcam_media","enabled":true,"config":{"assetName":{"value":
↪"WebcamImages"}, "imageDir":{"value":"webcam"}, "mediaType":{"value":"camera"},
↪"cameraNumber":{"value":"0"}, "fpm":{"value":"10.0"}}}' |jq

```

14.2 FLIR Bridge North Plugins

14.2.1 OMF

The *OMF* north plugin is included in all distributions of the Flir core and provides the north bound interface to the OSIsoft data historians in all it forms; PI Server, Edge Data Store and OSIsoft Cloud Services.

PI Web API OMF Endpoint

To use the PI Web API OMF endpoint first ensure the OMF option was included in your PI Server when it was installed.

Now go to the Flir user interface, create a new North instance and select the “OMF” plugin on the first screen. The second screen will request the following information:

Endpoint	PI Web API
Server hostname	localhost
Server port, 0=use the default	0
Producer Token	omf_north_0001
Data Source	readings
Static Data	Location: Palo Alto, Company: Dianomic
Sleep Time Retry	1
Maximum Retry	3
HTTP Timeout	10
Integer Format	int64
Number Format	float64
Compression	<input type="checkbox"/>
Asset Framework hierarchies tree	/fledge/data_piwebapi/default
Asset Framework hierarchies rules	<div><div>1</div><div>{ }</div></div>
PI Web API Authentication Method	anonymous
PI Web API User Id	user_id
PI Web API Password
PI Web API Kerberos keytab file	piwebapi_kerberos_https.keytab

Select PI Web API from the Endpoint options.

- **Basic Information**

- **Endpoint:** Select what you wish to connect to, in this case PI Web API.
- **Send full structure:** Used to control if AF structure messages are sent to the PI server. If this is turned off then the data will not be placed in the asset framework.
- **Naming scheme:** Defines the naming scheme to be used when creating the PI points within the PI Server. See [Naming Scheme](#).
- **Server hostname:** The hostname or address of the PI Server.
- **Server port:** The port the PI Web API OMF endpoint is listening on. Leave as 0 if you are using the default port.
- **Data Source:** Defines which data is sent to the PI Server. The readings or Flir's internal statistics.
- **Static Data:** Data to include in every reading sent to PI. For example, you can use this to specify the location of the devices being monitored by the Flir server.

- **Asset Framework**

- **Asset Framework Hierarchies Tree:** The location in the Asset Framework into which the data will be inserted. All data will be inserted at this point in the Asset Framework unless a later rule overrides this.
- **Asset Framework Hierarchies Rules:** A set of rules that allow specific readings to be placed elsewhere in the Asset Framework. These rules can be based on the name of the asset itself or some metadata associated with the asset. See [Asset Framework Hierarchy Rules](#)

- **PI Web API authentication**

- **PI Web API Authentication Method:** The authentication method to be used, anonymous equates to no authentication, basic authentication requires a user name and password and Kerberos allows integration with your single sign on environment.
- **PI Web API User Id:** The user name to authenticate with the PI Web API.
- **PI Web API Password:** The password of the user we are using to authenticate.
- **PI Web API Kerberos keytab file:** The Kerberos keytab file used to authenticate.

- **Connection management (These should only be changed with guidance from support)**

- **Sleep Time Retry:** Number of seconds to wait before retrying the HTTP connection (Flir doubles this time after each failed attempt).
- **Maximum Retry:** Maximum number of times to retry connecting to the PI server.
- **HTTP Timeout:** Number of seconds to wait before Flir will time out an HTTP connection attempt.

- **Other (Rarely changed)**

- **Integer Format:** Used to match Flir data types to the data type configured in PI. This defaults to int64 but may be set to any OMF data type compatible with integer data, e.g. int32.
- **Number Format:** Used to match Flir data types to the data type configured in PI. The defaults is float64 but may be set to any OMF datatype that supports floating point values.
- **Compression:** Compress the readings data before sending it to the PI System.

EDS OMF Endpoint

To use the OSIsoft Edge Data Store first install Edge Data Store on the same machine as your Flir instance. It is a limitation of Edge Data Store that it must reside on the same host as any system that connects to it with OMF.

Now go to the Flir user interface, create a new North instance and select the “OMF” plugin on the first screen. The second screen will request the following information:

Endpoint	Edge Data Store
Server hostname	localhost
Server port, 0=use the default	0
Producer Token	omf_north_0001
Data Source	readings
Static Data	Location: Palo Alto, Company: Dianomic
Sleep Time Retry	1
Maximum Retry	3
HTTP Timeout	10
Integer Format	int64
Number Format	float64
Compression	<input type="checkbox"/>
Asset Framework hierarchies tree	/fledge/data_piwebapi/default
Asset Framework hierarchies rules	<div>1</div> <div>{ }</div>
PI Web API Authentication Method	anonymous
PI Web API User Id	user_id
PI Web API Password
PI Web API Kerberos keytab file	piwebapi_kerberos_https.keytab
14.2. FLIR Bridge North Plugins	
OCS Namespace	name_space
OCS Tenant ID	ocs_tenant_id

Select Edge Data Store from the Endpoint options.

- **Basic Information**

- **Endpoint:** Select what you wish to connect to, in this case Edge Data Store.
- **Naming scheme:** Defines the naming scheme to be used when creating the PI points within the PI Server. See [Naming Scheme](#).
- **Server hostname:** The hostname or address of the PI Server. This must be the localhost for EDS.
- **Server port:** The port the Edge Datastore is listening on. Leave as 0 if you are using the default port.
- **Data Source:** Defines which data is sent to the PI Server. The readings or Flir’s internal statistics.
- **Static Data:** Data to include in every reading sent to PI. For example, you can use this to specify the location of the devices being monitored by the Flir server.

- **Connection management (These should only be changed with guidance from support)**

- **Sleep Time Retry:** Number of seconds to wait before retrying the HTTP connection (Flir doubles this time after each failed attempt).
- **Maximum Retry:** Maximum number of times to retry connecting to the PI server.
- **HTTP Timeout:** Number of seconds to wait before Flir will time out an HTTP connection attempt.

- **Other (Rarely changed)**

- **Integer Format:** Used to match Flir data types to the data type configured in PI. This defaults to int64 but may be set to any OMF data type compatible with integer data, e.g. int32.
- **Number Format:** Used to match Flir data types to the data type configured in PI. The defaults is float64 but may be set to any OMF datatype that supports floating point values.
- **Compression:** Compress the readings data before sending it to the PI System.

OCS OMF Endpoint

Go to the Flir user interface, create a new North instance and select the “OMF” plugin on the first screen. The second screen will request the following information:

omf

Endpoint

OSIsoft Cloud Services

Server hostname

pi-server

Server port, 0=use the default

0

Producer Token

uid=5ced49c3-3a55-40e7-983f-c6cdcd5c5fd1&crt=20180620084'

Data Source

readings

Static Data

Location: Palo Alto, Company: Dianomic

Sleep Time Retry

1

Maximum Retry

3

HTTP Timeout

10

Integer Format

int64

Number Format

float64

Compression

Asset Framework hierarchies tree

/fledge/data_piwebapi/default

Asset Framework hierarchies rules

1

{}

PI Web API Authentication Method

anonymous

PI Web API User Id

user_id

PI Web API Password

....

PI Web API Kerberos

piwebapi_kerberos_https.keytab

Select OSIsoft Cloud Services from the Endpoint options.

- **Basic Information**

- **Endpoint:** Select what you wish to connect to, in this case OSIsoft Cloud Services.
- **Naming scheme:** Defines the naming scheme to be used when creating the PI points within the PI Server. See [Naming Scheme](#).
- **Data Source:** Defines which data is sent to the PI Server. The readings or Flir's internal statistics.
- **Static Data:** Data to include in every reading sent to PI. For example, you can use this to specify the location of the devices being monitored by the Flir server.

- **Authentication**

- **OCS Namespace:** Your namespace within the OSIsoft Cloud Services.
- **OCS Tenant ID:** Your OSIsoft Cloud Services tenant ID for your account.
- **OCS Client ID:** Your OSIsoft Cloud Services client ID for your account.
- **OCS Client Secret:** Your OCS client secret.

- **Connection management (These should only be changed with guidance from support)**

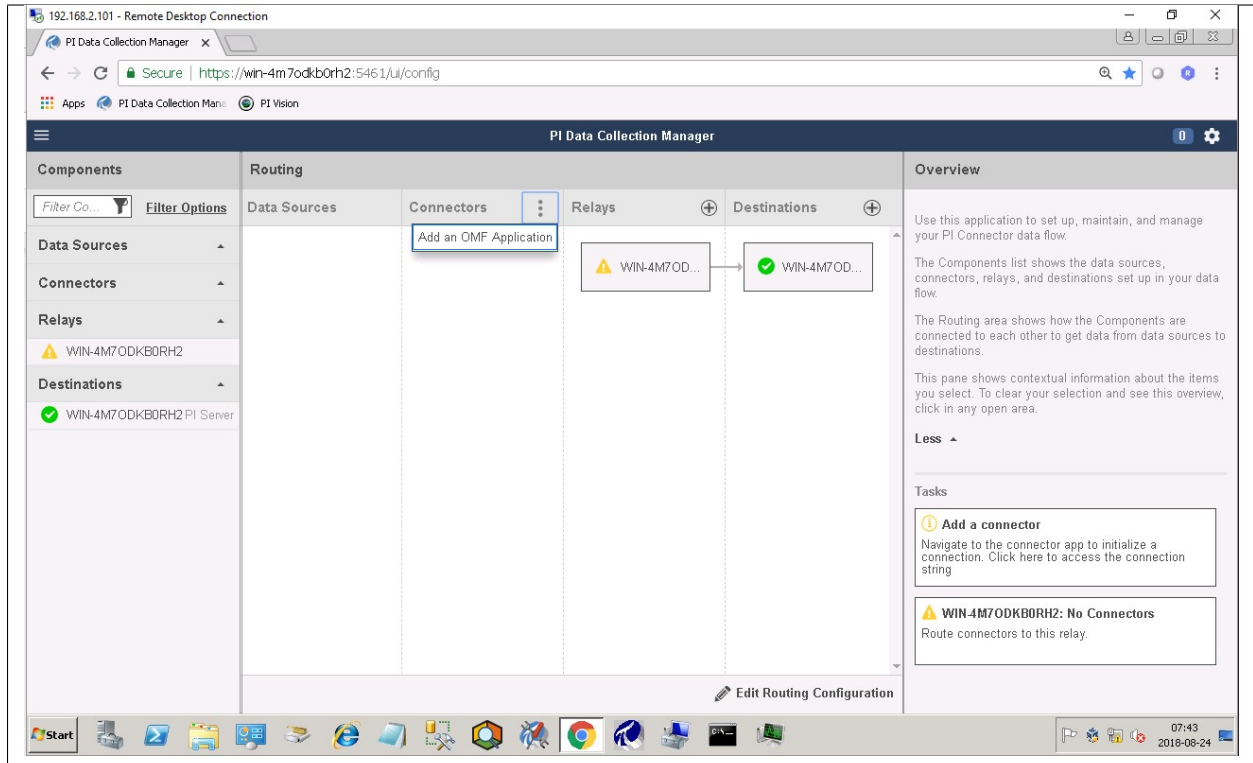
- **Sleep Time Retry:** Number of seconds to wait before retrying the HTTP connection (Flir doubles this time after each failed attempt).
- **Maximum Retry:** Maximum number of times to retry connecting to the PI server.
- **HTTP Timeout:** Number of seconds to wait before Flir will time out an HTTP connection attempt.

- **Other (Rarely changed)**

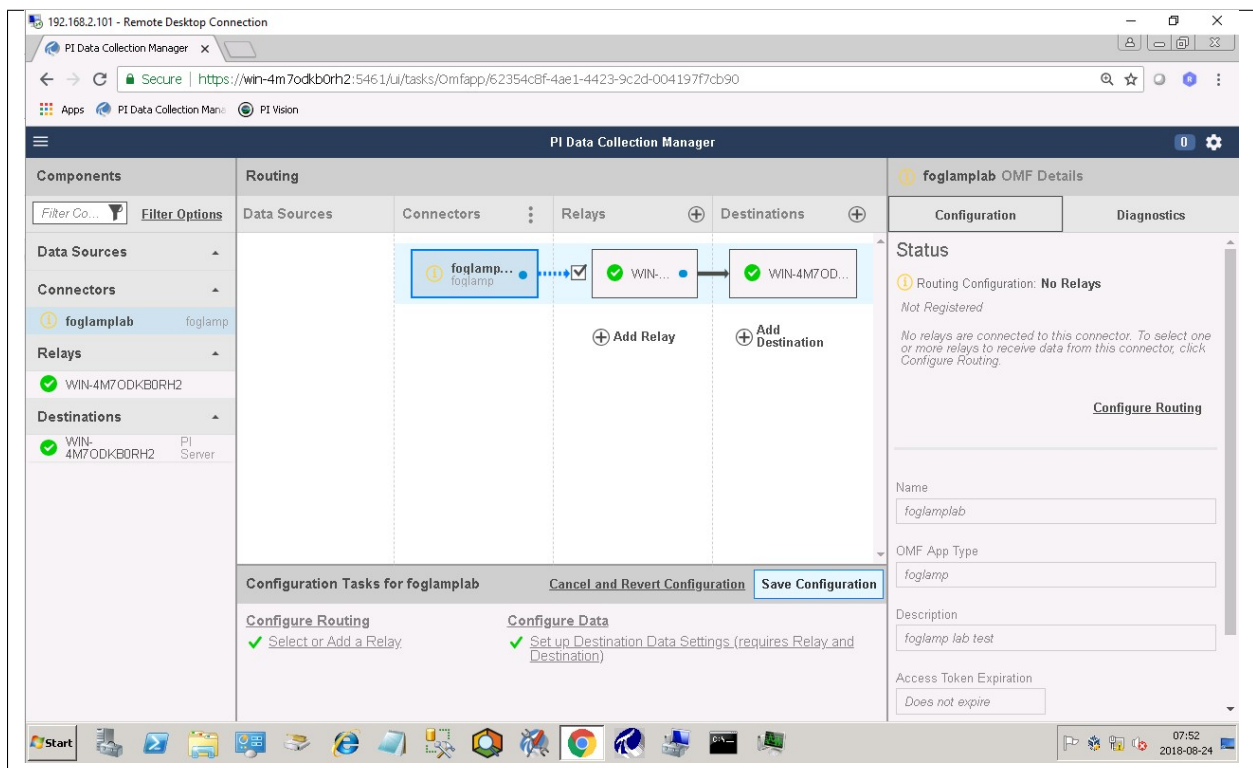
- **Integer Format:** Used to match Flir data types to the data type configured in PI. This defaults to int64 but may be set to any OMF data type compatible with integer data, e.g. int32.
- **Number Format:** Used to match Flir data types to the data type configured in PI. The default is float64 but may be set to any OMF datatype that supports floating point values.
- **Compression:** Compress the readings data before sending it to the PI System.

PI Connector Relay

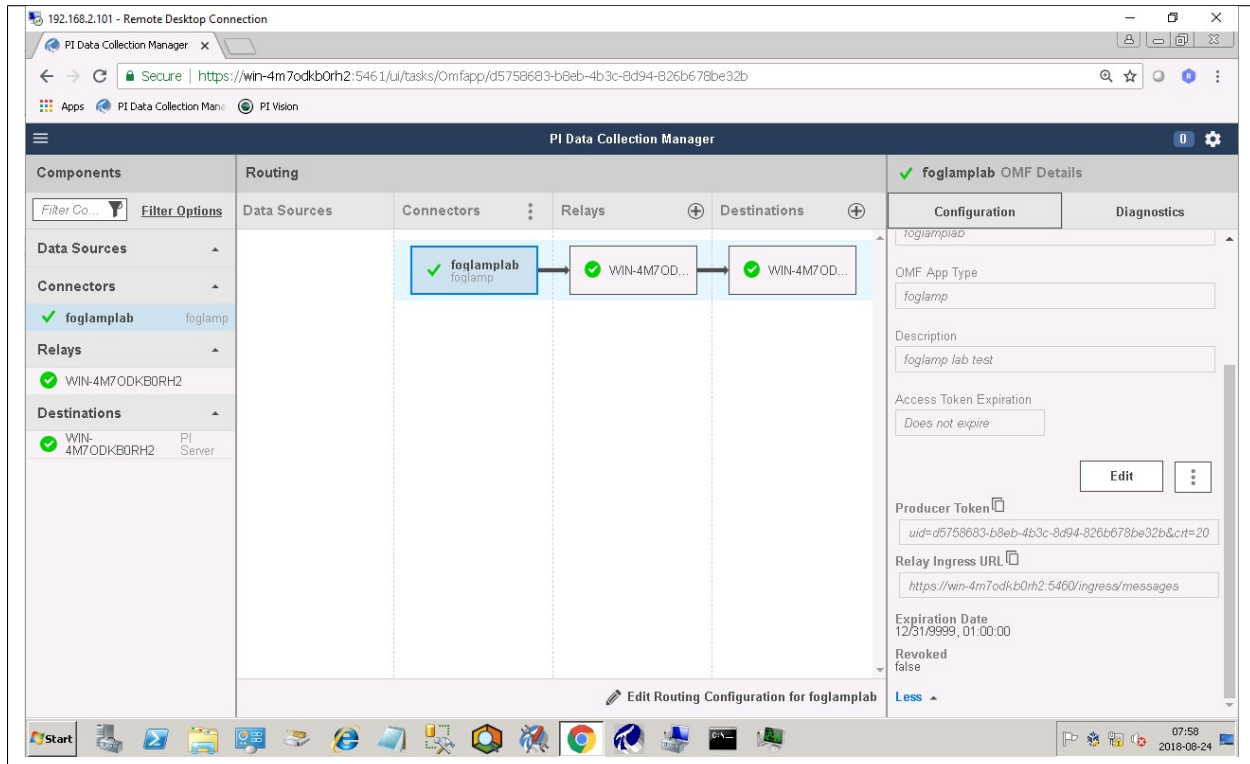
The PI Connector Relay was the original mechanism by which OMF data could be ingested into a PI Server, this has since been replaced by the PI Web API OMF endpoint. It is recommended that all new deployments should use the PI Web API endpoint as the Connector Relay has now been discontinued by OSIsoft. To use the Connector Relay, open and sign into the PI Relay Data Connection Manager.



To add a new connector for the Flir system, click on the drop down menu to the right of “Connectors” and select “Add an OMF application”. Add and save the requested configuration information.



Connect the new application to the OMF Connector Relay by selecting the new Flir application, clicking the check box for the OMF Connector Relay and then clicking “Save Configuration”.



Finally, select the new Flir application. Click “More” at the bottom of the Configuration panel. Make note of the Producer Token and Relay Ingress URL.

Now go to the Flir user interface, create a new North instance and select the “OMF” plugin on the first screen. The second screen will request the following information:

Endpoint	Connector Relay		
Server hostname	localhost		
Server port, 0=use the default	0		
Producer Token	omf_north_0001		
Data Source	readings		
Static Data	Location: Palo Alto, Company: Dianomic		
Sleep Time Retry	1		
Maximum Retry	3		
HTTP Timeout	10		
Integer Format	int64		
Number Format	float64		
Compression	<input type="checkbox"/>		
Asset Framework hierarchies tree	/fledge/data_piwebapi/default		
Asset Framework hierarchies rules	<table border="1"> <tr> <td>1</td> <td>{ }</td> </tr> </table>	1	{ }
1	{ }		
PI Web API Authentication Method	anonymous		
PI Web API User Id	user_id		
PI Web API Password		
PI Web API Kerberos keytab file	piwebapi_kerberos_https.keytab		
14.2. FLIR Bridge North Plugins OCS Namespace	name_space		
OCS Tenant ID	ocs_tenant_id		

- **Basic Information**

- **Endpoint:** Select what you wish to connect to, in this case the Connector Relay.
- **Server hostname:** The hostname or address of the Connector Relay.
- **Server port:** The port the Connector Relay is listening on. Leave as 0 if you are using the default port.
- **Producer Token:** The Producer Token provided by PI
- **Data Source:** Defines which data is sent to the PI Server. The readings or Flir's internal statistics.
- **Static Data:** Data to include in every reading sent to PI. For example, you can use this to specify the location of the devices being monitored by the Flir server.

- **Connection management (These should only be changed with guidance from support)**

- **Sleep Time Retry:** Number of seconds to wait before retrying the HTTP connection (Flir doubles this time after each failed attempt).
- **Maximum Retry:** Maximum number of times to retry connecting to the PI server.
- **HTTP Timeout:** Number of seconds to wait before Flir will time out an HTTP connection attempt.

- **Other (Rarely changed)**

- **Integer Format:** Used to match Flir data types to the data type configured in PI. This defaults to int64 but may be set to any OMF data type compatible with integer data, e.g. int32.
- **Number Format:** Used to match Flir data types to the data type configured in PI. The defaults is float64 but may be set to any OMF datatype that supports floating point values.
- **Compression:** Compress the readings data before sending it to the PI System.

Naming Scheme

The naming of objects in the asset framework and of the attributes of those objects has a number of constraints that need to be understood when storing data into a PI Server using OMF. An important factor in this is the stability of your data structures. If, in your environment you have objects are liable to change, i.e. the types of attributes change or the number of attributes change between readings, then you may wish to take a different naming approach to if they do not.

This occurs because of a limitation of the OMF interface to the PI server. Data is sent to OMF in a number of stages, one of these is the definition of the types for the AF Objects. OMF let's a type be defined, but once defined it can not be changed. A new type must be created rather than changing the existing type. This means a new asset framework object is created each time a type changes.

The OMF plugin names objects in the asset framework based upon the asset name in the reading within Flir. Asset names are typically added to the readings in the south plugins, however they may be altered by filters between the south ingest and the north egress points in the data pipeline. Asset names can be overridden using the *OMF Hints* mechanism described below.

The attribute names used within the objects in the PI System are based on the names of the data points within each reading within Flir. Again *OMF Hints* can be used to override this mechanism.

The naming used within the objects in the Asset Framework is controlled by the *Naming Scheme* option

Concise No suffix or prefix is added to the asset name and property name when creating the objects in the AF framework and Attributes in the PI server. However if the structure of an asset changes a new AF Object will be created which will have the suffix `-type*x*` appended to it.

Use Type Suffix The AF Object names will be created from the asset names by appending the suffix -type*x* to the asset name. If and when the structure of an asset changes a new object name will be created with an updated suffix.

Use Attribute Hash Attribute names will be created using a numerical hash as a prefix.

Backward Compatibility The naming reverts to the rules that were used by version 1.9.1 and earlier of Flir, both type suffices and attribute hashes will be applied to the naming.

Asset Framework Hierarchy Rules

The asset framework rules allow the location of specific assets within the PI Asset Framework to be controlled. There are two basic type of hint;

- Asset name placement, the name of the asset determines where in the Asset Framework the asset is placed
- Meta data placement, metadata within the reading determines where the asset is placed in the Asset Framework

The rules are encoded within a JSON document, this document contains two properties in the root of the document; one for name based rules and the other for metadata based rules

```
{
  "names" :
  {
    "asset1" : "/Building1/EastWing/GroundFloor/Room4",
    "asset2" : "Room14"
  },
  "metadata" :
  {
    "exist" :
    {
      "temperature" : "temperatures",
      "power" : "/Electrical/Power"
    },
    "nonexist" :
    {
      "unit" : "Uncalibrated"
    }
    "equal" :
    {
      "room" :
      {
        "4" : "ElecticalLab",
        "6" : "FluidLab"
      }
    }
    "notequal" :
    {
      "building" :
      {
        "plant" : "/Office/Environment"
      }
    }
  }
}
```

The name type rules are simply a set of asset name and AF location pairs. The asset names must be complete names, there is no pattern matching within the names.

The metadata rules are more complex, four different tests can be applied;

- **exists:** This test looks for the existence of the named datapoint within the asset.
- **nonexist:** This test looks for the lack of a named datapoint within the asset.
- **equal:** This test looks for a named data point having a given value.
- **notequal:** This test looks for a name data point having a value different from that specified.

The *exist* and *nonexist* tests take a set of name/value pairs that are tested. The name is the datapoint name to examine and the value is the asset framework location to use. For example

```
"exist" :
{
  "temperature" : "temperatures",
  "power"       : "/Electrical/Power"
}
```

If an asset has a data point called *temperature* it will be stored in the AF hierarchy *temperatures*, if the asset had a data point called *power* the asset will be placed in the AF hierarchy */Electrical/Power*.

The *equal* and *notequal* tests take an object as a child, the name of the object is data point to examine, the child nodes are sets of values and locations. For example

```
"equal" :
{
  "room" :
  {
    "4" : "ElectricalLab",
    "6" : "FluidLab"
  }
}
```

In this case if the asset has a data point called *room* with a value of *4* then the asset will be placed in the AF location *ElectricalLab*, if it has a value of *6* then it is placed in the AF location *FluidLab*.

If an asset matches multiple rules in the ruleset it will appear in multiple locations in the hierarchy, the data is shared between each of the locations.

If an OMF Hint exists within a particular reading this will take precedence over generic rules.

The AF location may be a simple string or it may also include substitutions from other data points within the reading. For example if the reading has a data point called *room* that contains the room in which the readings were taken, an AF location of */BuildingA/\${room}* would put the reading in the asset framework using the value of the room data point. The reading

```
"reading" : {
  "temperature" : 23.4,
  "room"       : "B114"
}
```

would be put in the AF at */BuildingA/B114* whereas a reading of the form

```
"reading" : {
  "temperature" : 24.6,
  "room"       : "2016"
}
```

would be put at the location */BuildingA/2016*.

It is also possible to define defaults if the referenced data point is missing. Therefore in our example above if we used the location `/BuildingA/${room:unknown}` a reading without a `room` data point would be place in `/BuildingA/unknown`. If no default is given and the data point is missing then the level in the hierarchy is ignore. E.g. if we use our original location `/BuildingA/${room}` and we have the reading

```
"reading" : {
  "temperature" : 22.8,
}
```

this reading would be stored in `/BuildingA`.

OMF Hints

The OMF plugin also supports the concept of hints in the actual data that determine how the data should be treated by the plugin. Hints are encoded in a specially name data point within the asset, *OMFHint*. The hints themselves are encoded as JSON within a string.

Number Format Hints

A number format hint tells the plugin what number format to insert data into the PI Server as. The following will cause all numeric data within the asset to be written using the format *float32*.

```
"OMFHint" : { "number" : "float32" }
```

The value of the *number* hint may be any numeric format that is supported by the PI Server.

Integer Format Hints

an integer format hint tells the plugin what integer format to insert data into the PI Server as. The following will cause all integer data within the asset to be written using the format *integer32*.

```
"OMFHint" : { "number" : "integer32" }
```

The value of the *number* hint may be any numeric format that is supported by the PI Server.

Type Name Hints

A type name hint specifies that a particular name should be used when defining the name of the type that will be created to store the object in the Asset Framework. This will override the *Naming Scheme* currently configured.

```
"OMFHint" : { "typeName" : "substation" }
```

Type Hint

A type hint is similar to a type name hint, but instead of defining the name of a type to create it defines the name of an existing type to use. The structure of the asset *must* match the structure of the existing type with the PI Server, it is the responsibility of the person that adds this hint to ensure this is the case.

```
"OMFHint" : { "type" : "pump" }
```

Tag Name Hint

Specifies that a specific tag name should be used when storing data in the PI server.

```
"OMFHint" : { "tagName" : "AC1246" }
```

Datapoint Specific Hint

Hints may also be targeted to specific data points within an asset by using the datapoint hint. A *datapoint* hint takes a JSON object as it's value, this object defines the name of the datapoint and the hint to apply.

```
"OMFHint" : { "datapoint" : { "name" : "voltage:", "number" : "float32" } }
```

The above hint applies to the datapoint *voltage* in the asset and applies a *number format* hint to that datapoint.

Asset Framework Location Hint

An asset framework location hint can be added to a reading to control the placement of that asset within the Asset Framework. An asset framework hint would be as follow

```
"OMFHint" : { "AFLocation" : "/UK/London/TowerHill/Floor4" }
```

Adding OMF Hints

An OMF Hint is implemented as a string data point on a reading with the data point name of *OMFHint*. It can be added at any point in the processing of the data, however a specific plugin is available for adding the hints, the .

14.2.2 Google Cloud Platform North Plugin

The *flir-north-gcp* plugin provide connectivity from a FLIR Bridge system to the Google Cloud Platform. The plugin connects to the IoT Core in Google Cloud using MQTT and is fully compliant with the security features of the Google Cloud Platform. See for a tutorial on setting up a FLIR Bridge system and getting it to send data to Google Cloud.

Prerequisites

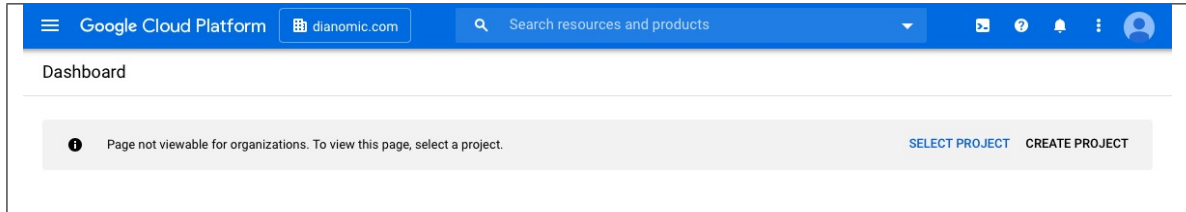
A number of things must be done in the Google Cloud before you can create your north connection to GCP. You must

- Create a GCP IoT Core project
- Download the *roots.pem* certificate from your GCP account
- Create a registry
- Create a device ID and configure a key pair for that device
- Upload the certificates to the FLIR Bridge certificate store

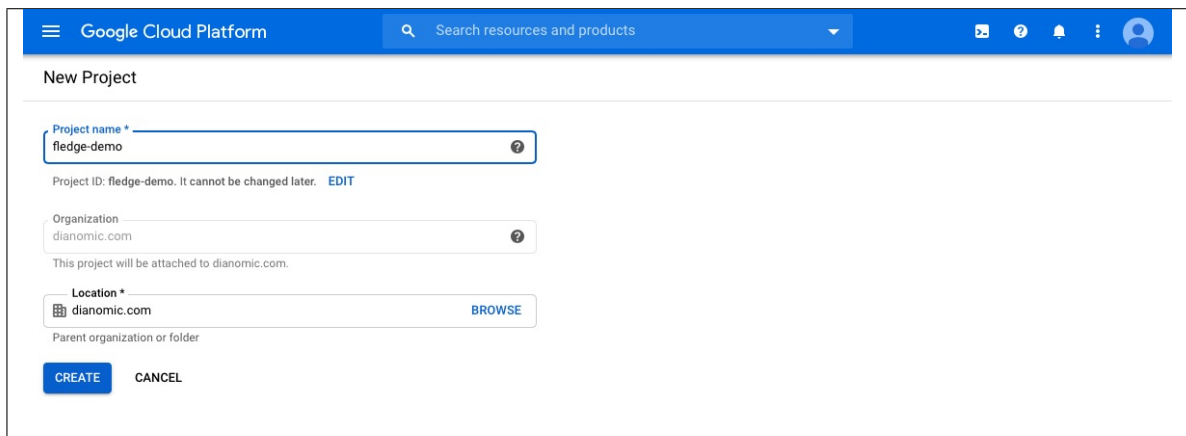
Create GCP IoT Core Project

To create a new project

- Goto the
- Select the Projects page and select the *Create New Project* option



- Enter your project details



Download roots.pem

To download the roots.pem security certificate

- From the command line shell of your machine run the command

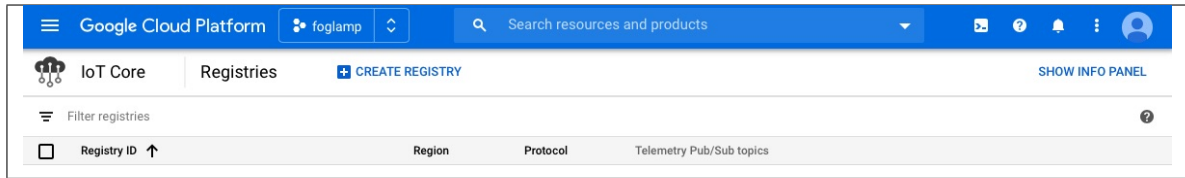
```
$ wget https://pki.goog/roots.pem
```

Create a Registry

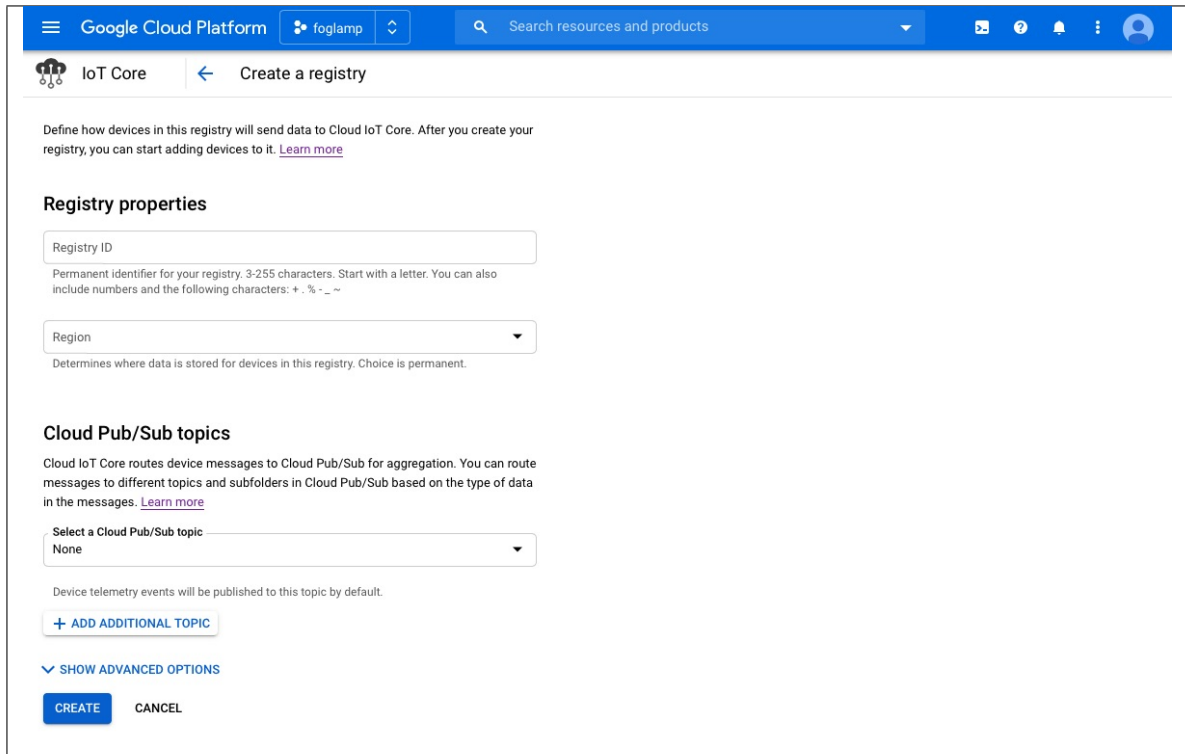
To create a registry in your project

- Goto the
- Click on the menu icon in the top left corner of the page
- Select the *Create Registry* option





- A new screen is shown that allows you to create a new registry



- Note the Registry ID and region as you will need these later
- Select an existing telemetry topic or create a new topic (for example, projects/[YOUR_PROJECT_ID]/topics/[REGISTRY_ID])
- Click on *Create*

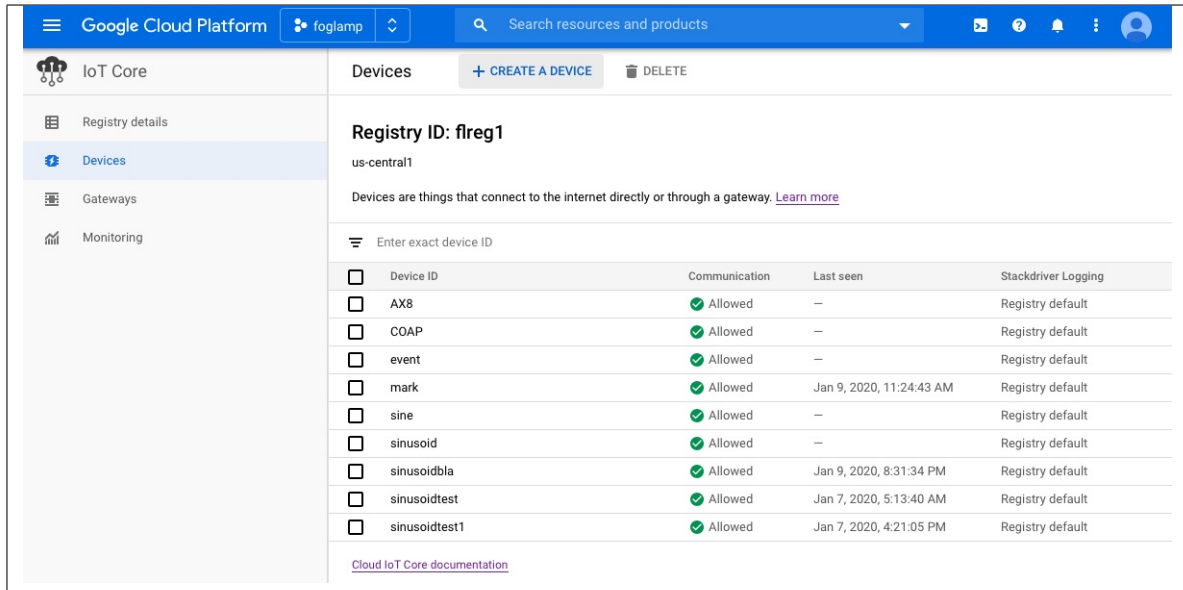
Create a Device ID

To create a device in your Google Cloud Project

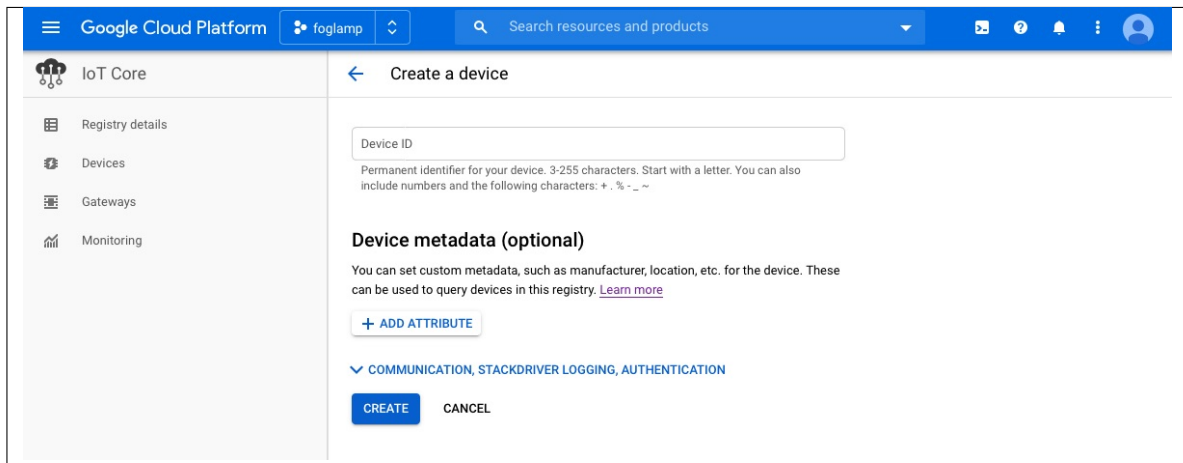
- Create an RSA public/private key pair on your local machine

```
openssl genpkey -algorithm RSA -out rsa_flir.pem -pkeyopt rsa_keygen_bits:2048
openssl rsa -in rsa_flir.pem -pubout -out rsa_flir.pem
```

- Goto the
- In the left pane of the IoT Core page in the Cloud Console, click Devices



- At the top of the Devices page, click *Create a device*

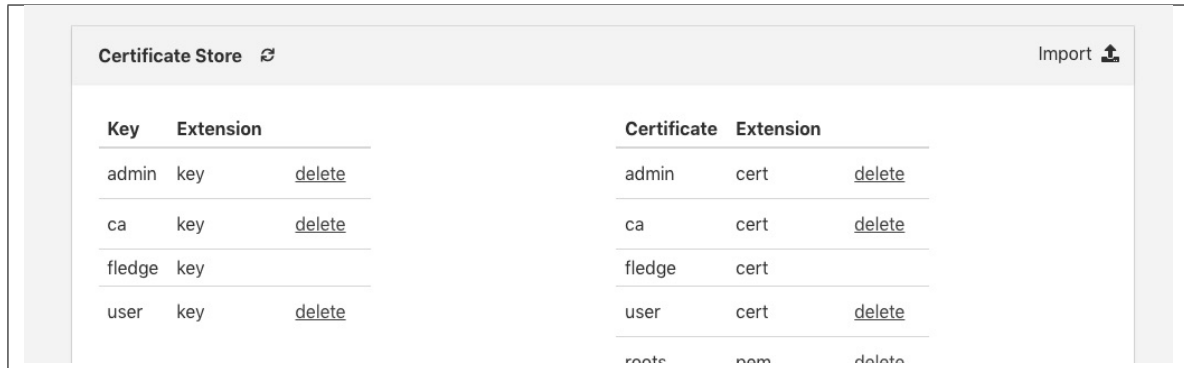


- Enter a device ID, you will need to add this in the north plugin configuration later
- Click on the *ADD ATTRIBUTE COMMUNICATION, STACKDRIVER LOGGING, AUTHENTICATION* link to open the remainder of the inputs
- Make sure the public key format matches the type of key that you created in the first step of this section (for example, RS256)
- Paste the contents of your public key in the Public key value field.

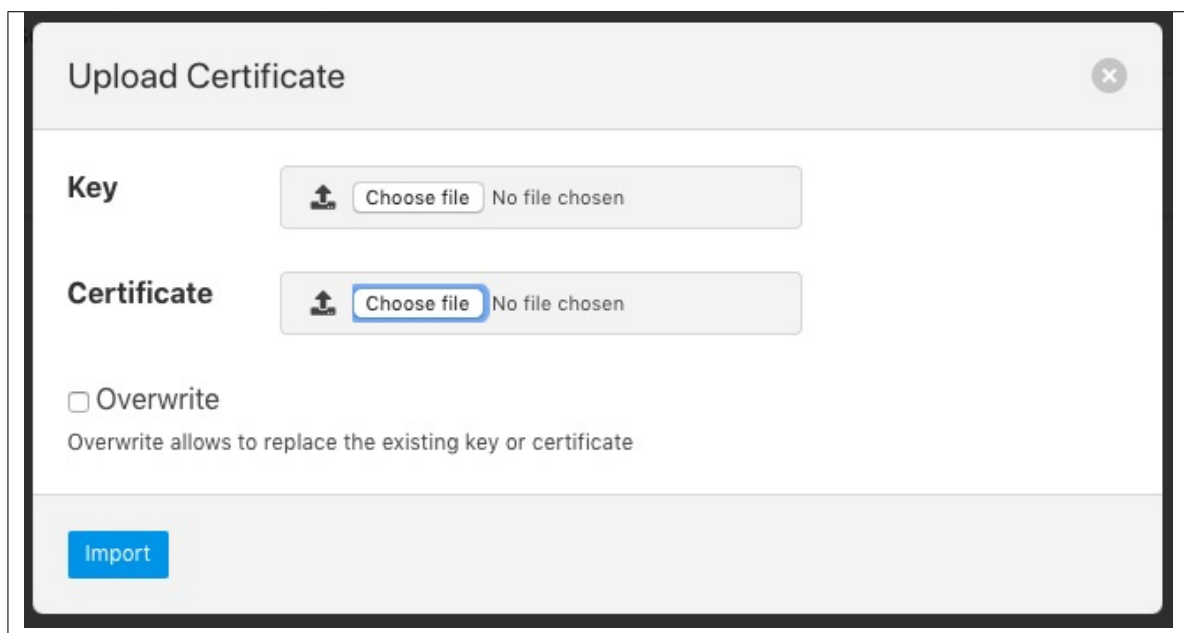
Upload Your Certificates

You should upload your certificates to FLIR Bridge

- From the FLIR Bridge user interface select the *Certificate Store* from the left-hand menu bar



- Click on the Import option in the top left corner



- In the Certificate option select the *Choose file* option and select your roots.pem and click on open
- Repeat the above for your device key and certificate

Create Your North Task

Having completed the pre-requisite steps it is now possible to create the north task to send data to GCP.

- Select the *North* option from the left-hand menu bar.
- Select GCP from the North Plugin list
- Name your North task and click on *Next*

The screenshot shows a configuration wizard with three steps: 1. Plugin & Name, 2. Review Configuration (current), and 3. Done. The 'Review Configuration' step contains a form with the following fields and values:

Field	Value
Project ID	fledge-demo
The GCP Region	us-central1
Registry ID	fledge
Device ID	demo
Key Name	fledge_private
JWT Algorithm	RS256
Data Source	readings

At the bottom of the form, there is a 'Back' button on the left and a 'Next' button on the right.

- Configure your GCP plugin
 - **Project ID:** Enter the project ID you created in GCP
 - **The GCP Region:** Select the region in which you created your registry
 - **Registry ID:** The Registry ID you created should be entered here
 - **Device ID:** The Device ID you created should be entered here
 - **Key Name:** Enter the name of the device key you uploaded to the certificate store
 - **JWT Algorithm:** Select the algorithm that matches the key you created earlier
 - **Data Source:** Select the data to send to GCP, this may be readings or FLIR Bridge statistics
- Click on *Next*
- Enable your plugin and click on *Done*

14.2.3 Graphite

The *flir-north-graphite* plugin provides a means to send data from FLIR Bridge to the Carbon storage within Graphite to allow data to be graphed.

To create the connection to Graphite

- Select *North* from the left hand menu bar.
- Click on the + icon in the top left
- Choose *graphite* from the plugin selection list
- Name your task
- Click on *Next*
- Configure the plugin

The screenshot shows a configuration window for a plugin. At the top, there is a progress bar with three steps: 1. Plugin & Name, 2. Review Configuration (highlighted), and 3. Done. Below the progress bar is a form with the following fields:

- Host:** graphite.local
- Port:** 3000
- Asset Root:** foglamp
- Source:** readings (dropdown menu)

At the bottom of the form, there are two buttons: 'Back' and 'Next'.

- **Host:** The host where Graphite is running.
- **Port:** The carbon listening port of your Graphite Carbon engine.
- **Asset Root:** The root of the asset structure to use with Graphite.
- Click on *Next*
- Enable your north task and click on *Done*

14.2.4 North HTTP

The *flir-north-http* plugin allows data to be sent from the north of one FLIR Bridge instance into the south of another FLIR Bridge instance. It allows hierarchies of FLIR Bridge instances to be built. The FLIR Bridge to which the data is sent must run the corresponding in order for data to flow between the two FLIR Bridge instances. The plugin supports both HTTP and HTTPS transport protocols and sends a JSON payload of reading data in the internal FLIR Bridge format.

The plugin may also be used to send data from FLIR Bridge to another system, the receiving system should implement a REST end point that will accept a POST request containing JSON data. The format of the JSON payload is described below. The required REST endpoint path is defined in the configuration of the plugin.

Filters may be applied to the connection in either the north task that loads this plugin or the receiving south service on the up stream FLIR Bridge.

A of this plugin exists also that performs the same function as this plugin, the pair are provided for purposes of comparison and the user may choose whichever they prefer to use.

To create a north task to send to another FLIR Bridge you should first create the that will receive the data. Then create a new north tasks by;

- Selecting *North* from the left hand menu bar.
- Click on the + icon in the top left
- Choose `http_north` from the plugin selection list
- Name your task
- Click on *Next*
- Configure the plugin

The screenshot shows a configuration window for a FLIR Bridge plugin. At the top, a progress bar indicates three steps: 1. Plugin & Name, 2. Review Configuration (the current step), and 3. Done. The main configuration area contains the following fields:

- URL:** A text input field containing `http://localhost:6683/sensor-reading`.
- Source:** A dropdown menu currently set to `readings`.
- Verify SSL:** An unchecked checkbox.
- Apply Filter:** An unchecked checkbox.
- Filter Rule:** A text input field containing `.[]`.

At the bottom of the window, there are two buttons: a 'Back' button and a 'Next' button.

- **URL:** The URL of the receiving , the address and port should match the service in the up stream FLIR Bridge. The URL can specify either HTTP or HTTPS protocols.
- **Source:** The data to send, this may be either the reading data or the statistics data
- **Verify SSL:** When HTTPS rather the HTTP is used this toggle allows for the verification of the certificate that is used. If a self signed certificate is used then this should not be enabled.
- **Apply Filter:** This allows a simple jq format filter rule to be applied to the connection. This should not be confused with FLIR Bridge filters and exists for backward compatibility reason only.
- **Filter Rule:** A jq filter rule to apply. Since the introduction of FLIR Bridge filters in the north task this has become deprecated and should not be used.

- Click *Next*
- Enable your task and click *Done*

JSON Payload

The payload that is sent by this plugin is a simple JSON presentation of a set of reading values. A JSON array is sent with one or more reading objects contained within it. Each reading object consists of a timestamp, an asset name and a set of data points within that asset. The data points are represented as name value pair JSON properties within the reading property.

The fixed part of every reading contains the following

Name	Description
times-tamp	The timestamp as an ASCII string in ISO 8601 extended format. If no time zone information is given it is assumed to indicate the use of UTC.
asset	The name of the asset this reading represents.
read-ings	A JSON object that contains the data points for this asset.

The content of the *readings* object is a set of JSON properties, each of which represents a data value. The type of these values may be integer, floating point, string, a JSON object or an array of floating point numbers.

A property

```
"voltage" : 239.4
```

would represent a numeric data value for the item *voltage* within the asset. Whereas

```
"voltageUnit" : "volts"
```

Is string data for that same asset. Other data may be presented as arrays

```
"acceleration" : [ 0.4, 0.8, 1.0 ]
```

would represent acceleration with the three components of the vector, x, y, and z. This may also be represented as an object

```
"acceleration" : { "X" : 0.4, "Y" : 0.8, "Z" : 1.0 }
```

both are valid formats within FLIR Bridge.

An example payload with a single reading would be as shown below

```
[
  {
    "timestamp" : "2020-07-08 16:16:07.263657+00:00",
    "asset"      : "motor1",
    "readings"   : {
      "voltage"  : 239.4,
      "current"  : 1003,
      "rpm"      : 120147
    }
  }
]
```

14.2.5 North HTTP-C

The *flir-north-http-c* plugin allows data to be sent from the north of one FLIR Bridge instance into the south of another FLIR Bridge instance. It allows hierarchies of FLIR Bridge instances to be built. The FLIR Bridge to which the data is sent must run the corresponding in order for data to flow between the two FLIR Bridge instances. The plugin supports both HTTP and HTTPS transport protocols and sends a JSON payload of reading data in the internal FLIR Bridge format.

Additionally this plugin allows for two URL's to be configured, a primary URL and a secondary URL. If the connection to the primary URL fails then the plugin will switch over to using the secondary URL. It will switch back if the connection to the secondary fails or if when the north task completes and a new north task is later run.

The plugin may also be used to send data from FLIR Bridge to another system, the receiving system should implement a REST end point that will accept a POST request containing JSON data. The format of the JSON payload is described below. The required REST endpoint path is defined in the configuration of the plugin.

Filters may be applied to the connection in either the north task that loads this plugin or the receiving south service on the up stream FLIR Bridge.

A plugin exists also that performs the same function as this plugin, the pair are provided for purposes of comparison and the user may choose whichever they prefer to use.

To create a north task to send to another FLIR Bridge you should first create the that will receive the data. Then create a new north tasks by;

- Selecting *North* from the left hand menu bar.

- Click on the + icon in the top left
- Choose httpc from the plugin selection list
- Name your task
- Click on *Next*
- Configure the HTTP-C plugin

The screenshot displays the 'Review Configuration' step of the HTTP-C plugin setup. The form is organized into several sections:

- URL:** A text field containing 'http://localhost:6683/sensor-reading'.
- Secondary URL:** An empty text field.
- Source:** A dropdown menu set to 'readings'.
- Headers:** A table with one row:

1	{}
---	----
- Sleep Time Retry:** A text field with the value '1'.
- Maximum Retry:** A text field with the value '3'.
- Http Timeout (in seconds):** A text field with the value '10'.
- Verify SSL:** An unchecked checkbox.
- Apply Filter:** An unchecked checkbox.
- Filter Rule:** A text field containing '[]'.

At the bottom of the form are 'Back' and 'Next' buttons.

- **URL:** The URL of the receiving , the address and port should match the service in the up stream FLIR Bridge. The URL can specify either HTTP or HTTPS protocols.
- **Secondary URL:** The URL to failover to if the connection to the primary URL fails. If failover is not required then leave this field empty.
- **Source:** The data to send, this may be either the reading data or the statistics data
- **Headers:** An optional set of header fields to send in every request. The headers are defined as a JSON document with the name of each item in the document as header field name and the value the value of the header field.
- **Sleep Time Retry:** A tuning parameter used to control how often a connection is retried to the up stream FLIR Bridge if it is not available. On every retry the time will be doubled.
- **Maximum Retry:** The maximum number of retries to make a connection to the up stream FLIR Bridge. When this number is reached the current execution of the task is suspended until the next scheduled run.
- **Http Timeout (in seconds):** The timeout to set on the HTTP connection after which the connection will be closed. This can be used to tune the response of the system when communication links are unreliable.

- **Verify SSL:** When HTTPS rather the HTTP is used this toggle allows for the verification of the certificate that is used. If a self signed certificate is used then this should not be enabled.
- **Apply Filter:** This allows a simple jq format filter rule to be applied to the connection. This should not be confused with FLIR Bridge filters and exists for backward compatibility reason only.
- **Filter Rule:** A jq filter rule to apply. Since the introduction of FLIR Bridge filters in the north task this has become deprecated and should not be used.

- Click *Next*
- Enable your task and click *Done*

Header Fields

Header fields can be defined if required using the *Headers* configuration item. This is a JSON document that defines a set of key/value pairs for each header field. For example if a header field *token* was required with the value of *sfe93rjfk93rj* then the *Headers* JSON document would be as follows

```
{
  "token" : "sfe93rjfk93rj"
}
```

Multiple header fields may be set by specifying multiple key/value pairs in the JSON document.

JSON Payload

The payload that is sent by this plugin is a simple JSON presentation of a set of reading values. A JSON array is sent with one or more reading objects contained within it. Each reading object consists of a timestamp, an asset name and a set of data points within that asset. The data points are represented as name value pair JSON properties within the reading property.

The fixed part of every reading contains the following

Name	Description
times-tamp	The timestamp as an ASCII string in ISO 8601 extended format. If no time zone information is given it is assumed to indicate the use of UTC.
asset	The name of the asset this reading represents.
read-ings	A JSON object that contains the data points for this asset.

The content of the *readings* object is a set of JSON properties, each of which represents a data value. The type of these values may be integer, floating point, string, a JSON object or an array of floating point numbers.

A property

```
"voltage" : 239.4
```

would represent a numeric data value for the item *voltage* within the asset. Whereas

```
"voltageUnit" : "volts"
```

Is string data for that same asset. Other data may be presented as arrays

```
"acceleration" : [ 0.4, 0.8, 1.0 ]
```

would represent acceleration with the three components of the vector, x, y, and z. This may also be represented as an object

```
"acceleration" : { "X" : 0.4, "Y" : 0.8, "Z" : 1.0 }
```

both are valid formats within FLIR Bridge.

An example payload with a single reading would be as shown below

```
[
  {
    "timestamp" : "2020-07-08 16:16:07.263657+00:00",
    "asset" : "motor1",
    "readings" : {
      "voltage" : 239.4,
      "current" : 1003,
      "rpm" : 120147
    }
  }
]
```

14.2.6 InfluxDB Time Series Database

The *flir-north-influxdb* plugin is designed to send data from FLIR Bridge to the open source time series database.

The process of creating a North InfluxDB is similar to any other north setup

- Selecting the *North* option in the left-hand menu bar
- Click on the add icon in the top right corner.
- In the *North Plugin* list select the influxdb option.
- Click *Next*
- Configure your InfluxDB plugin

The screenshot displays the 'Review Configuration' step of the InfluxDB plugin setup. The progress bar at the top indicates the current step is 2 out of 3. The configuration form includes the following fields:

- Host:** influxdb.local
- Port:** 8086
- Database:** foglamp
- Username:** (empty field)
- Password:** password
- Source:** readings (dropdown menu)

At the bottom of the configuration area, there are 'Back' and 'Next' buttons.

- **Host:** The hostname or IP address of the machine where your InfluxDB server is running.
- **Port:** The port on which your InfluxDB server is listening.

- **Database:** The database in your InfluxDB server into which to write data.
 - **Username:** The username if any to use to authenticate with your InfluxDB server.
 - **Password:** The password to use to authenticate with your InfluxDB server.
 - **Source:** The source of data to send, this may be either FLIR Bridge readings or the FLIR Bridge statistics
- Click *Next*
 - Enable your north task and click on *Done*

14.2.7 InfluxDB Cloud

The *flir-north-influxdbcloud* plugin is designed to send data from FLIR Bridge to the system for collection and analysis of data.

The process of creating a North InfluxDB Cloud connection is similar to any other north setup

- Selecting the *North* option in the left-hand menu bar
- Click on the add icon in the top right corner.
- In the *North Plugin* list select the *influxdbcloud* option.
- Click *Next*
- Configure your InfluxDB Cloud plugin

The screenshot displays the 'Review Configuration' step of the InfluxDB Cloud setup process. A progress bar at the top indicates the current step is 2 of 3. The form contains the following configuration details:

- URL:** `https://eu-central-1-1.aws.cloud2.influxdata.com`
- InfluxDB token:** (Empty field)
- Organisation ID:** (Empty field)
- Bucket:** (Empty field)
- Measurement:** `foglamp`
- Source:** `readings` (Selected from a dropdown menu)
- Filter Rule:** `.[]`
- Apply Filter:** (Unchecked checkbox)

Navigation buttons 'Back' and 'Next' are located at the bottom of the form.

- **URL:** The URL of the InfluxDB instance you are using
- **InfluxDB token:** an authorization token that has been generated by the InfluxDB Cloud
- **Organisation ID:** Your organization ID from the InfluxDB Cloud. You can find this by looking at the URL you use after connecting the InfluxDB Cloud.
- **Bucket:** The bucket in InfluxDB Cloud where you wish to store your data.

- **Measurement:** The measurement to use for the data you send to InfluxDB Cloud
 - **Source:** The source of data to send, this may be either FLIR Bridge readings or the FLIR Bridge statistics
 - * **Apply Filter:** This allows a simple jq format filter rule to be applied to the connection. This should not be confused with FLIR Bridge filters and exists for backward compatibility reasons only.
 - **Filter Rule:** A jq filter rule to apply. Since the introduction of FLIR Bridge filters in the north task this has become deprecated and should not be used.
- Click *Next*
 - Enable your north task and click on *Done*

14.2.8 Kafka Producer

The *flir-north-kafka* plugin sends data from FLIR Bridge to the an Apache Kafka. FLIR Bridge acts as a Kafka producer, sending reading data to Kafka. This implementation is a simplified producer that sends all data on a single Kafka topic. Each message contains an asset name, timestamp and set of readings values as a JSON document.

The configuration of the *Kafka* plugin is very simple, consisting of four parameters that must be set.

The screenshot displays the 'Review Configuration' step of the Kafka Producer setup. At the top, a progress bar indicates the sequence: 1. Plugin & Name, 2. Review Configuration (active), and 3. Done. The main configuration area contains four labeled fields: 'Bootstrap Brokers' with the value 'localhost:9092,kafka.local:9092', 'Kafka Topic' with 'Fledge', 'Send JSON' with a dropdown set to 'Strings', and 'Data Source' with a dropdown set to 'readings'. A help icon (?) is visible in the top right of the form. At the bottom, there are 'Back' and 'Next' buttons.

- **Bootstrap Brokers:** A comma separate list of Kafka brokers to use to establish a connection to the Kafka system.
- **Kafka Topic:** The Kafka topic to which all data is sent.
- **Send JSON:** This controls how JSON data points should be sent to Kafka. These may be sent as strings or as JSON objects.
- **Data Source:** Which FLIR Bridge data to send to Kafka; Readings or FLIR Bridge Statistics.

14.2.9 OPCUA Server

The *flir-north-opcua* plugin is a rather unusual north plugin as it does not send data to a system, but rather acts as a server from which other systems can pull data from FLIR Bridge. This is slightly at odds with the concept of short running tasks for sending north and does require a little more configuration when creating the North OPCUA server.

The process of creating a North OPCUA Server start as with any other north setup by selecting the *North* option in the left-hand menu bar, then press the add icon in the top right corner. In the *North Plugin* list select the *opcua* option.

1 Plugin & Name 2 Review Configuration 3 Done

North Plugin

- ocs_vz
- OMF
- opcu** OPCUA Server
- pi_server
- pi_server_v2

[available plugins](#)

Name

Repeat (Interval)

In addition to setting a name for this task it is recommended to run the OPCUA North as a service rather than a task. Running as a periodically restarted task will cause clients to be disconnected at regular intervals, when run as a service the disconnections do not occur. If run as a task set the *Repeat* interval to a higher value than the 30 second default as we will be later setting the maximum run time of the north task to a higher value. Once complete click on *Next* and move on to the configuration of the plugin itself.

1 Plugin & Name 2 Review Configuration 3 Done

Server Name: Fledge OPCUA

URL: opc.tcp://localhost:4840/fledge/server

URI: urn://fledge.dianomic.com

Namespace: http://fledge.dianomic.com

Source: readings

Object Root:

Hierarchy:

1	{ }
---	-----

Back Next

This second page allows for the setting of the configuration within the OPCUA server.

- **Server Name:** The name the OPCUA server will report itself as to any client that connects to it.
- **URL:** The URL that any client application will use to connect to the OPCUA server. This should always start `opc.tcp://`
- **URI:** The URI you wish to associate to your data, this is part of the OPCUA specification and may be set to any option you wish or can be left as default.
- **Namespace:** This defines the namespace that you wish to use for your OPCUA objects. If you are not employing a client that does namespace checking this is best left as the default.
- **Source:** What data is being made available via this OPCUA server. You may chose to make the reading data available or the FLIR Bridge statistics
- **Object Root:** This item can be used to define a root within the OPCUA server under which all objects are stored. If left empty then the objects will be created under the root.
- **Hierarchy:** This allows you to define a hierarchy for the OPCUA objects that is based on the meta data within the readings. See below for the definition of hierarchies.

Once you have completed your configuration click *Next* to move to the final page and then enable your north task and click *Done*.

The only step left is to modify the duration for which the task runs. This can only be done **after** it has been run for the first time. Enter your *North* task list again and select the OPCUA North that you just created. This will show the configuration of your North task. Click on the *Show Advanced Config* option to display your advanced configuration.

OPCUA Server

Server Name

Fledge OPCUA

URL

opc.tcp://localhost:4840/fledge/server

URI

urn://fledge.dianomic.com

Namespace

http://fledge.dianomic.com

Source

readings

Duration

60

Readings Block Size

500

Sleep Interval

1

Enabled

☒

Exclusive

☒

Interval

0

01:00:00

Hide Advanced Config

Applications

The *Duration* option controls how long the north task will run before stopping. Each time it stops any client connected to the FLIR Bridge OPCUA server will be disconnected, in order to reduce the disconnect/reconnect volumes it is advisable to set this to a value greater than the 60 second default. In our example here we set the repeat interval to one hour, so ideally we should set the duration to an hour also such that there is no time when an OPCUA server is not running. *Duration* is set in seconds, so should be 3600 in our example.

Hierarchy Definition

The hierarchy definition is a JSON document that defines where in the object hierarchy data is placed. The placement is controlled by meta data attached to the readings.

Assuming that we attach meta data to each of the assets we read that give a plant name and building to each asset using the names *plant* and *building* on those assets. If we wanted to store all data for the same plant in a single location in the OPCUA object hierarchy and have each building under the plant, then we can define a hierarchy as follows

```
{
  "plant" :
```

(continues on next page)

(continued from previous page)

```
{
    "building" : ""
}
```

If we had the following 4 assets with the metadata as defined

```
{
  "asset_code" : "A",
  "plant"      : "Bolton",
  "building"   : "10"
  ....
}
{
  "asset_code" : "B",
  "plant"      : "Bolton",
  "building"   : "7"
  ....
}
{
  "asset_code" : "C",
  "plant"      : "Milan",
  "building"   : "A"
  ....
}
{
  "asset_code" : "D",
  "plant"      : "Milan",
  "building"   : "C"
  ....
}
{
  "asset_code" : "General",
  "plant"      : "Milan",
  ....
}
```

The data would be shown in the OPCUA server in the following structure

```
Bolton
    10
        A
    7
        B
Milan
    A
        C
    C
        D
    General
```

Any data that does not fit this structure will be stored at the root.

14.2.10 Splunk Data Collector

The *flir-north-splunk* plugin is designed to send data from FLIR Bridge to the system for collecting and analysis of data.

The process of creating a North Splunk is similar to any other north setup

- Selecting the *North* option in the left-hand menu bar
- Click on the add icon in the top right corner.
- In the *North Plugin* list select the splunk option.
- Click *Next*
- Configure your Splunk plugin

The screenshot displays the 'Review Configuration' step of the Splunk Data Collector setup. At the top, a progress bar indicates the sequence: 1. Plugin & Name, 2. Review Configuration (active), and 3. Done. The main configuration area contains the following fields:

- URL:** A text input field containing 'http://splunk:8088/services/collector/event'.
- Source:** A dropdown menu currently set to 'readings'.
- Splunk authorisation token:** A text input field containing '42b66064-ee25-407c-857c-3ded78d21b38'.
- Apply Filter:** An unchecked checkbox.
- Filter Rule:** A text input field containing 'jq format filter rule'.

At the bottom of the configuration area, there are two buttons: 'Back' and 'Next'.

- **URL:** The URL of the splunk collector for events
- **Source:** The source of data to send, this may be either FLIR Bridge readings or the FLIR Bridge statistics
- **Splunk authorisation token:** an authorisation token that has been issued by the splunk data collector
- * **Apply Filter:** This allows a simple jq format filter rule to be applied to the connection. This should not be confused with FLIR Bridge filters and exists for backward compatibility reasons only.
- **Filter Rule:** A jq filter rule to apply. Since the introduction of FLIR Bridge filters in the north task this has become deprecated and should not be used.

- Click *Next*
- Enable your north task and click on *Done*

14.2.11 ThingSpeak

The *flir-north-thingspeak* plugin provides a mechanism to , allowing an easy route to send data from an FLIR Bridge environment into MATLAB.

In order to send data to ThingSpeak you must first create a channel to receive it.

- Login to your account

- From the menu bar select the *Channels* menu and the *My Channels* option

ThingSpeak™ Channels Apps Support Commercial Use How to Buy MR

My Channels

[New Channel](#)

Name	Created	Updated
sinusoid <div> Private Public Settings Sharing API Keys Data Import / Export </div>	2018-08-08	2019-07-26 15:04

Help

Collect data in a ThingSpeak channel from a device, from another channel, or from the web.

Click **New Channel** to create a new ThingSpeak channel.

Click on the column headers of the table to sort by the entries in that column or click on a tag to show channels with that tag.

Learn to [create channels](#), explore and transform data.

Learn more about [ThingSpeak Channels](#).

Examples

- [Arduino](#)
- [Arduino MKR1000](#)
- [ESP8266](#)
- [Raspberry Pi](#)
- [Netduino Plus](#)

Upgrade

Need to send more data faster?

Need to use ThingSpeak for a commercial project?

[Upgrade](#)

- Click on *New Channel* to create a new channel

[Channels](#)
[Apps](#)
[Support](#)

[Commercial Use](#)
[How to Buy](#)
[MR](#)

New Channel

Name

Description

Field 1 ☒

Field 2 ☐

Field 3 ☐

Field 4 ☐

Field 5 ☐

Field 6 ☐

Field 7 ☐

Field 8 ☐

Metadata

Tags

(Tags are comma separated)

Link to External Site

Link to GitHub

Elevation

Show Channel Location ☐

Latitude

Longitude

Show Video ☐

☒ YouTube
 ☐ Vimeo

Video URL

Show Status ☐

Help

Channels store all the data that a ThingSpeak application collects. Each channel includes eight fields that can hold any type of data, plus three fields for location data and one for status data. Once you collect data in a channel, you can use ThingSpeak apps to analyze and visualize it.

Channel Settings

- **Percentage complete:** Calculated based on data entered into the various fields of a channel. Enter the name, description, location, URL, video, and tags to complete your channel.
- **Channel Name:** Enter a unique name for the ThingSpeak channel.
- **Description:** Enter a description of the ThingSpeak channel.
- **Field#:** Check the box to enable the field, and enter a field name. Each ThingSpeak channel can have up to 8 fields.
- **Metadata:** Enter information about channel data, including JSON, XML, or CSV data.
- **Tags:** Enter keywords that identify the channel. Separate tags with commas.
- **Link to External Site:** If you have a website that contains information about your ThingSpeak channel, specify the URL.
- **Show Channel Location:**
 - **Latitude:** Specify the latitude position in decimal degrees. For example, the latitude of the city of London is 51.5072.
 - **Longitude:** Specify the longitude position in decimal degrees. For example, the longitude of the city of London is -0.1275.
 - **Elevation:** Specify the elevation position meters. For example, the elevation of the city of London is 35.052.
- **Video URL:** If you have a YouTube™ or Vimeo® video that displays your channel information, specify the full path of the video URL.
- **Link to GitHub:** If you store your ThingSpeak code on GitHub®, specify the GitHub repository URL.

Using the Channel

You can get data into a channel from a device, website, or another ThingsSpeak channel. You can then visualize data and transform it using ThingSpeak [Apps](#).

See [Get Started with ThingSpeak®](#) for an example of measuring dew point from a weather station that acquires data from an Arduino® device.

[Learn More](#)

- Enter the details for your channel, in particular name and the set of fields. These field names should match the asset names you are going to send from FLIR Bridge.
- When satisfied click on *Save Channel*
- You will need the channel ID and the API key for your channel. To get this for a channel, on the *My Channels* page click on the *API Keys* box for your channel

sinusoid
Channel ID: 556345 | Test channel
Author: markdianomic
Access: Private

Private View Public View Channel Settings Sharing API Keys Data Import / Export

Write API Key

Key:

[Generate New Write API Key](#)

Read API Keys

Key:

Note:

[Save Note](#) [Delete API Key](#)

[Add New Read API Key](#)

Help

API keys enable you to write data to a channel or read data from a private channel. API keys are auto-generated when you create a new channel.

API Keys Settings

- Write API Key:** Use this key to write data to a channel. If you feel your key has been compromised, click **Generate New Write API Key**.
- Read API Keys:** Use this key to allow other people to view your private channel feeds and charts. Click **Generate New Read API Key** to generate an additional read key for the channel.
- Note:** Use this field to enter information about channel read keys. For example, add notes to keep track of users with access to your channel.

API Requests

Write a Channel Feed

```
GET https://api.thingspeak.com/update?api_key=APIKEY&field1=0
```

Read a Channel Feed

```
GET https://api.thingspeak.com/channels/556345/feeds.json?api_key=APIKEY&results=2
```

Read a Channel Field

```
GET https://api.thingspeak.com/channels/556345/fields/1.json?api_key=APIKEY&results=2
```

Read Channel Status Updates

```
GET https://api.thingspeak.com/channels/556345/status.json?api_key=APIKEY
```

[Learn More](#)

Once you have created your channel on you may create your north task on FLIR Bridge to send data to this channel

- Select *North* from the left hand menu bar.
- Click on the + icon in the top left
- Choose ThingSpeak from the plugin selection list
- Name your task
- Click on *Next*
- Configure the plugin

The screenshot shows a configuration window for a plugin. At the top, there's a progress bar with three steps: 1. Plugin & Name, 2. Review Configuration (highlighted), and 3. Done. The main area contains several input fields and a code editor. The 'URL' field is filled with 'https://api.thingspeak.com/channels'. The 'API Key' field is empty. The 'Source' dropdown is set to 'readings'. The 'Fields' section shows a JSON array:

```
[{"asset": "sinusoid", "reading": "sinusoid"}]
```

. The 'Channel ID' field is filled with '0'. At the bottom, there are 'Back' and 'Next' buttons.

- **URL:** The URL of the ThingSpeak server, this can usually be left as the default.
- **API Key:** The write API key from the ThingSpeak channel you created
- **Source:** Controls if readings data or FLIR Bridge statistics are to be send to ThingSpeak
- **Fields:** Allows you to select what fields to send to ThingSpeak. It's a JSON document that contains a single array called elements. Each item of the array is a JSON object that has two properties, asset and reading. The asset should match the asset you wish to send and the reading the data point name.
- **Channel ID:** The channel ID of your ThingSpeak Channel

- Click on *Next*
- Enable your north task and click on *Done*

14.3 FLIR Bridge Filter Plugins

14.3.1 Asset Filter

The *flir-filter-asset* is a filter that allows for assets to be included, excluded or renamed in a stream. It may be used either in *South* services or *North* tasks and is driven by a set of rules that define for each named asset what action should be taken.

Asset filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.

- Select the *asset* plugin from the list of available plugins.
- Name your asset filter.
- Click *Next* and you will be presented with the following configuration page

1 Plugin Name 2 Review Configuration

Asset rules

```

1 {
2   "rules": [
3     {
4       "asset_name"    : "temperature",
5       "action"       : "rename",
6       "new_asset_name": "abient"
7     }
8   ]
9 }

```

Enabled ☒

Previous Done

- Enter the *Asset rules*
- Enable the plugin and click *Done* to activate it

Asset Rules

The asset rules are an array of JSON objects which define the asset name to which the rule is applied and an action. Actions can be one of

- **include:** The asset should be forwarded to the output of the filter
- **exclude:** The asset should not be forwarded to the output of the filter
- **rename:** Change the name of the asset. In this case a third property is included in the rule object, "new_asset_name"

In addition a *defaultAction* may be included, however this is limited to *include* and *exclude*. Any asset that does not match a specific rule will have this default action applied to them. If the default action is not given it is treated as if a default action of *include* had been set.

A typical set of rules might be

```

{
  "rules": [
    {
      "asset_name": "Random1",

```

(continues on next page)

(continued from previous page)

```

        "action": "include"
      },
      {
        "asset_name": "Random2",
        "action": "rename",
        "new_asset_name": "Random92"
      },
      {
        "asset_name": "Random3",
        "action": "exclude"
      },
      {
        "asset_name": "Random4",
        "action": "rename",
        "new_asset_name": "Random94"
      },
      {
        "asset_name": "Random5",
        "action": "exclude"
      },
      {
        "asset_name": "Random6",
        "action": "rename",
        "new_asset_name": "Random96"
      },
      {
        "asset_name": "Random7",
        "action": "include"
      }
    ],
    "defaultAction": "include"
  }

```

14.3.2 Change Filter

The *flir-filter-change* filter is used to only send information about an asset onward when a particular datapoint within that asset changes by more than a configured percentage. Data is sent for a period of time before and after the change in the monitored value. The amount of data to send before and after the change is configured in milliseconds, with a value for the pre-change time and one for the post-change time.

It is possible to define a rate at which readings should be sent regardless of the monitored value changing. This provides an average of the values of the period defined, e.g. send a 1 minute average of the values every minute.

This filter only operates on a single asset, all other assets are passed through the filter unaltered.

Change filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *change* plugin from the list of available plugins.
- Name your change filter.
- Click *Next* and you will be presented with the following configuration page

The screenshot shows a configuration window with a progress bar at the top. Step 1 is 'Plugin Name' and Step 2 is 'Review Configuration'. The configuration form has the following fields:

Field	Value
Asset	fan1
Trigger	current
Required Change %	5
Pre-trigger time (mS)	500
Post-trigger time (mS)	500
Reduced collection rate	2
Rate Units	per hour
Enabled	<input checked="" type="checkbox"/>

At the bottom, there are 'Previous' and 'Done' buttons.

- Enter the configuration for your change filter
 - **Asset:** The asset to monitor and control with this filter. This asset is both the asset that is used to look for changes and also the only asset whose data is affected by the triggered or non-triggered state of this filter.
 - **Trigger:** The datapoint within the asset that is used to trigger the sending of data at full rate. This datapoint may be either a numeric value or a string. If it is a string then a change of value of the defined change percentage or greater will trigger the sending of data. If the value is a string then any change in value will trigger the sending of the data.
 - **Required Change %:** The percentage change required for a numeric value change to trigger the sending of data. If this value is set to 0 then any change in the trigger value will be enough to trigger the sending of data.
 - **Pre-trigger time:** The number of milliseconds worth of data before the change that triggers the sending of data will be sent.
 - **Post-trigger time:** The number of milliseconds after a change that triggered the sending of data will be sent. If there is a subsequent change while the data is being sent then this period will be reset and the sending of data will recommence.
 - **Reduced collection rate:** The rate at which to send averages if a change does not trigger full rate data. This is defined as a number of averages for a period defined in the rateUnit, e.g. 4 per hour.
 - **Rate Units:** The unit associated with the average rate above. This may be one of “per second”, “per minute”, “per hour” or “per day”.
- Enable the change filter and click on *Done* to activate your plugin

14.3.3 CSV Writer

The plugin collects the readings from south service into csv files and compresses them when limit set per file is exceeded. The files are collected in date-wise directories where a single directory contains all the files collected on

that day. The directories are rotated when the limit set is exceeded. We may collect data continuously, periodically or using a CRON string.

?

Input assets name	<input type="text"/>
Forward data	<input type="checkbox"/>
Destination directory	<input type="text" value="FOGLAMP_DATA/readings-out"/>
Subdirectory name	<input type="text" value="south-storage"/>
File type	<input type="text" value="csv"/>
Sampling rate	<input type="text" value="8000"/>
Cron mode	<input type="text" value="continuous"/>
Cron period starting time	<input type="text"/>
Recording period duration	<input type="text" value="100ms"/>
Event repetition time	<input type="text" value="16mins"/>
Pre event time	<input type="text" value="1min"/>
Event duration	<input type="text" value="1min"/>

- **‘inputAssets’:** type: string default: ‘’: The names of assets (comma separated) to be stored in the csv file. All the data points of these assets will become columns in the csv file. Other assets that are not included will be forwarded. If empty all asset names will be taken.
- **‘forwardData’:** type: boolean default: false: Forces data to be forwarded upstream, normally data is written to csv and not sent to storage.
- **‘destDir’:** type: string default: ‘FLIR_DATA/readings-out’: Destination directory inside \$FLIR_DATA e.g. /usr/local/flir/data/readings-out if FLIR_DATA/ is prefixed otherwise it will be created as specified. Default is FLIR_DATA/readings-out. If the path given without FLIR_DATA/ then directory will be created inside \$FLIR_ROOT/services/<path>, path can be recursive.
- **‘filterName’(Subdirectory name):** type: string default: ‘south-storage’: Name of the specific sub-directory under the “dest dir” where records are to be written. Useful when we have multiple instances of csv writer filter. Just for convention use the name source-destination to indicate that the filter is applied between source and destination. For example use filterName rms-database if the filter is applied between a rms filter and sqlite database.
- **‘fileType’:** type: string default: ‘csv’: The file type (csv or pickle) in order to store readings.
- **‘samplingRate’:** type: integer default: ‘8000’: The number of readings per second to be stored in the csv/pickle file.
- **‘cronMode’:** type: enumeration default: continuous: Cron style, either periodic, continuous, or table. In continuous mode files are continuously, in periodic mode the files are collected at every given interval of time (configurable). In table mode the files are collected according to a cron string similar to cron in Unix environments.
- **‘cronPeriodStart’:** type: string default: ‘’: The time at which the collection should start. If empty the collection will start immediately after the first reading. If a time stamp (a sample time stamp could be 2021-04-27 09:25:35.300875+00:00) is given then the plugin will start collection when the timestamp (will use user_ts of reading, if no user_ts then ts) of a reading that has arrived becomes greater than this value. Note cron-PeriodStart is useful for periodic mode and won’t be used when cronMode is table. For periodic mode we

can give cronPeriodStart either in the past or in the future.

- **‘cronPeriodDuration’**: type: string default: **100ms**: Amount of time records are written per file (in ms, sec[s], min[s], hr[s], day[s], week[s]). For example if cronPeriodDuration is 3 mins and sample rate is 8000, Then each file will have $3*60*8000=1440000$ records. Note: It won't be used when cronMode is table. The limit will be picked from cron string. It also won't be used for periodic mode.
- **‘eventRepetitionTime’**: type: string default: **16min**: Only for periodic mode. The time after which the collection starts again. (in ms, sec[s], min[s], hr[s], day[s], week[s]).
- **‘eventPreTime’**: type: string default: **1min**: Only for periodic mode. The amount of time to consider for collection before the desired event [eg., 1min].
- **‘eventDuration’**: type: string default: **1min**: Only for periodic mode. The duration for desired event [eg., 1min].

Post event time	1min
Rotate after	14days
Periodic collection spec	
Add timestamp to csv data	<input type="checkbox"/>
Enable compression	<input checked="" type="checkbox"/>
Compression type	bzip2
Encryption password	
Enabled	<input checked="" type="checkbox"/>

- **‘eventPostTime’**: type: string default: **1min**: Only for periodic mode. The amount of time to consider for collection after the desired event [eg., 1min]. Note the cronPeriodDuration for periodic mode is the sum eventPreTime, eventDuration and eventPostTime.
- **‘rotateAfter’**: type: string default: **10min**: Total time after which rotation will occur. (eg., 4wks). For example if rotateAfter is 7 days. Then on 12:00:00 AM of ninth day then the first directory will be deleted.
- **‘cronTabSpec’**: type: string default: **‘’**: This parameter controls the time at which collection takes place.

Note: It is only used when cronMode is table. It is a string which consists of seven parts separated by spaces. It takes the form ‘seconds(0-59) minute(0-59) hour(0-23) day-of-month (1-31) month(1-12/names) day-of-week(0-7 or names) duration(seconds[float]’

Examples:

- use string ‘0 0,15,30,45 * * * * 60’ if you want to collect at one minute worth of data zeroth, fifteenth, thirtieth, forty fifth minute of every hour.
- use string ‘0,15,30,35 * * * * * 5’ if you want to collect at five seconds worth of data zeroth, fifteenth, thirtieth, forty fifth second of every minute.

- **‘addTimestamp’**: type: boolean default: **false**: Add a timestamp to each csv entry.
- **‘enableCompress’**: type: boolean default: **true**: Compress files after they have reached their maximum size.
- **‘compressionType’**: enumeration [‘bzip2’, ‘gzip’, ‘7za’] default **bzip2**: Select compression type to be used when ‘enableCompress’ is true. if 7za is selected then the files will get encrypted.

- **‘encryptPw’**: **type:password default: ‘’**: The password used to to encrypt files if 7za compression is selected.
- **‘enable’**: **type: boolean default: ‘false’**: Enable / Disable plugin operation.

Execution

Part 1: Get some south service running

For starting a south service use any of the following commands.

1. Use csvplayback

Assuming you have a csv file named vibration.csv inside FLIR_ROOT/data/csv_data (Can give a pattern like vib. The plugin will search for all the files starting with vib and therefore find out the file named vibration.csv). The csv file has fixed number of columns per row. Also assuming the column names are present in the first line. The plugin will rename the file with suffix .tmp after playing. Here is the curl command for that.

```
res=$(curl -sX POST http://localhost:8081/flir/service -d @- << EOF | jq '.')
{
  "name": "My_south",
  "type": "south",
  "plugin": "csvplayback",
  "enabled": false,
  "config": {
    "assetName": {"value": "My_csv_asset"},
    "csvDirName": {"value": "FLIR_DATA/csv_data"},
    "csvFileName": {"value": "vib"},
    "headerMethod": {"value": "do_not_skip"},
    "variableCols": {"value": "false"},
    "columnMethod": {"value": "pick_from_file"},
    "rowIndexForColumnNames": {"value": "0"},
    "ingestMode": {"value": "burst"},
    "sampleRate": {"value": "8000"},
    "postProcessMethod": {"value": "rename"},
    "suffixName": {"value": ".tmp"}
  }
}
EOF
)

echo $res
```

2. Use dt9837 plugin

Assuming you have connected accelerometers to the DAQ, run the following command. This command uses 4 channel data.

```
curl -sX POST http://localhost:8081/flir/service -d '{"name": "My_south",
  "type": "south", "plugin": "dt9837", "enabled": "true", "config": {"range":
  {"value": "BiPolar 10 Volts"}, "lowChannel": {"value": "0"}, "highChannel":
  {"value": "3"}}}' | jq
```

Part 2: Add the filter & attach to service

```
curl -sX POST http://localhost:8081/flir/filter -d '{"name":"csv_writer_
↪continuous","plugin":"csv_writer","filter_config":{"samplingRate":"8000",
↪"enable":"true","enableCompress":"true","cronTabSpec":"","addTimestamp":
↪"true","filterName":"continuous","cronMode":"continuous",
↪"cronPeriodDuration":"5min","rotateAfter":"7days","forwardData":"true"}}' |jq
curl -sX PUT 'http://localhost:8081/flir/filter/My_south/pipeline?allow_
↪duplicates=true&append_filter=true' -d '{"pipeline":["csv_writer_continuous
↪"]}' |jq
```

Modes

Periodic

The following command will collect data after every 16 minutes and will collect 3 minutes (pre + post + event duration) worth of data in every file. The data will get rotated after 14 + 1 = 15 days. The collection will start at 2021-07-05 11:00:00.000000+00:00 (subtract the pre time of 1 minutes). If this time is of the past the plugin will calculate the time accordingly. Note times are considered in utc. The plugin will convert the time zone to utc.

```
# assign start time to a variable.
start_time="2021-07-05 11:01:00.000000+00:00"
curl -sX POST http://localhost:8081/flir/filter -d '{"name":"csv_writer_periodic",
↪"plugin":"csv_writer","filter_config":{"samplingRate":"8000","enable":"true",
↪"enableCompress":"true","cronTabSpec":"","addTimestamp":"true","filterName":
↪"periodic","cronPeriodStart":"'$start_time'", "cronMode":"periodic",
↪"eventRepetitionTime":"16min","eventDuration":"1min","eventPreTime":"1min",
↪"eventPostTime":"1min","rotateAfter":"14days","forwardData":"true"}}' |jq
curl -sX PUT 'http://localhost:8081/flir/filter/My_south/pipeline?allow_
↪duplicates=true&append_filter=true' -d '{"pipeline":["csv_writer_periodic"]}' |jq
```

Some sample files will be as follows:

```
flir@flir:~/usr/local/flir/data/readings-out/periodic/2021-07-05.d$ ls
2021-07-05-11-00-00-0000.csv.bz2
2021-07-05-11-16-00-0000.csv.bz2
2021-07-05-11-32-00-0000.csv.bz2
..
..
..
```

Continuous

The following command collects files continuously. Each files has 5 minutes worth of data.

```
curl -sX POST http://localhost:8081/flir/filter -d '{"name":"csv_writer_continuous",
↪"plugin":"csv_writer","filter_config":{"samplingRate":"8000","enable":"true",
↪"enableCompress":"true","cronTabSpec":"","addTimestamp":"true","filterName":
↪"continuous","cronMode":"continuous","cronPeriodDuration":"5min","rotateAfter":
↪"7days","forwardData":"true"}}' |jq
curl -sX PUT 'http://localhost:8081/flir/filter/My_south/pipeline?allow_
↪duplicates=true&append_filter=true' -d '{"pipeline":["csv_writer_continuous"]}' |jq
```

Some sample files will be as follows:

```
flir@flir:~/usr/local/flir/data/readings-out/continuous/2021-05-07.d$ ls
2021-05-07-10-00-00-0000.csv.bz2
2021-05-07-10-05-00-0000.csv.bz2
2021-05-07-10-10-00-0000.csv.bz2
..
..
..
```

Cron style collection

The following command collects 5 minutes of data in every two hours.

```
curl -sX POST http://localhost:8081/flir/filter -d '{"name":"csv_writer_discontinuous",
  ↪ "plugin":"csv_writer","filter_config":{"samplingRate":"8000","enable":"true",
  ↪ "enableCompress":"true","cronTabSpec":"0 0 0,2,4,6,8,10,12,14,16,18,20,22 * * * 300
  ↪ ","addTimestamp":"true","filterName":"discontinuous","cronMode":"table",
  ↪ "rotateAfter":"4weeks","forwardData":"true"}}' |jq
curl -sX PUT 'http://localhost:8081/flir/filter/My_south/pipeline?allow_
  ↪ duplicates=true&append_filter=true' -d '{"pipeline":["csv_writer_discontinuous"]}' |jq
  ↪ |jq
```

The sample files will be like

```
flir@flir:~/usr/local/flir/data/readings-out/discontinuous/2021-05-07.d$ ls
2021-05-07-10-00-00-0000.csv.bz2
2021-05-07-12-00-00-0000.csv.bz2
2021-05-07-14-00-00-0000.csv.bz2
..
..
..
```

Cascading CSV writer filter

We can apply multiple instances of csv writer filter. Let's say we want to apply three filters. Then we need to keep forwardData of first two filters to be true. The third filter's forwardData may or may not be true.

Consider the following example

```
# continuous
curl -sX POST http://localhost:8081/flir/filter -d '{"name":"csv_writer_continuous",
  ↪ "plugin":"csv_writer","filter_config":{"samplingRate":"8000","enable":"true",
  ↪ "enableCompress":"true","cronTabSpec":"","addTimestamp":"true","filterName":
  ↪ "continuous","cronMode":"continuous","cronPeriodDuration":"5m","rotateAfter":
  ↪ "7days","forwardData":"true"}}' |jq
curl -sX PUT 'http://localhost:8081/flir/filter/My_south/pipeline?allow_
  ↪ duplicates=true&append_filter=true' -d '{"pipeline":["csv_writer_continuous"]}' |jq

# discontinuous
curl -sX POST http://localhost:8081/flir/filter -d '{"name":"csv_writer_discontinuous",
  ↪ "plugin":"csv_writer","filter_config":{"samplingRate":"8000","enable":"true",
  ↪ "enableCompress":"true","cronTabSpec":"0 0 0,2,4,6,8,10,12,14,16,18,20,22 * * * 300
  ↪ ","addTimestamp":"true","filterName":"discontinuous","cronMode":"table",
  ↪ "rotateAfter":"4weeks","forwardData":"true"}}' |jq
curl -sX PUT 'http://localhost:8081/flir/filter/My_south/pipeline?allow_
  ↪ duplicates=true&append_filter=true' -d '{"pipeline":["csv_writer_discontinuous"]}' |jq
  ↪ |jq
```

(continues on next page)

(continued from previous page)

```
# periodic
# assigning the start time to a variable.
start_time="2021-07-05 11:01:00.000000+00:00"
curl -sX POST http://localhost:8081/flir/filter -d '{"name":"csv_writer_periodic",
↪ "plugin":"csv_writer","filter_config":{"samplingRate":"8000","enable":"true",
↪ "enableCompress":"true","cronTabSpec":"","addTimestamp":"true","filterName":
↪ "periodic" , "cronPeriodStart":"'$start_time'" , "cronMode":"periodic",
↪ "eventRepetitionTime":"16min","eventDuration":"1min","eventPreTime":"1min",
↪ "eventPostTime":"1min" , "rotateAfter":"14days", "forwardData":"true"}}' |jq
curl -sX PUT 'http://localhost:8081/flir/filter/My_south/pipeline?allow_
↪ duplicates=true&append_filter=true' -d '{"pipeline":["csv_writer_periodic"]}' |jq
```

If forwardData of first filter is false then only the first filter will collect data.

If forwardData of first filter is true and second filter is false only first and second filter will collect data.

The forwardData of third filter may or may not be true. It is advisable to switch it off to prevent ingesting into database.

The following table sums it up.

Table 2: **Two CSV filters cascaded together**

Filter 1 forwardData	Filter 2 forwardData	Behaviour
True	True	Both filters collect data and data is ingested into database.
True	False	Both filters collect data and data is NOT ingested into database.
False	True	Only filter 1 collects data and data is NOT ingested into database.
False	False	Only filter 1 collects data and data is NOT ingested into database.

Behaviour on restart and reconfigure

After restart the collection will resume normally which means collection will begin in the same directory as it was earlier. However it may happen that the plugin was writing a file and the file is uncompressed. This uncompressed file will be compressed when the plugin will restart.

Note that the plugin won't wait for compression as it would be offloaded to some other thread for compression.

If this is a csv file and is empty it will be deleted.

It may also happen the directory name is changed inside configuration of the plugin. Then collection will begin inside different directory without deleting existing files.

On reconfigure the plugin will behave similar to restart.

How data is rotated?

The plugin picks rotateAfter config parameter and converts into days. Since each day collection has got its own directory therefore when the number of directories exceed this number the first directory will get deleted and so on. (Assuming we already had transferred these files to somewhere else before rotation.)

Note: If rotateAfter is 1week, then limit calculated will be 8. (We are talking one more day to compensate the case when collection was started at let's say at 2 PM on first day. Had we taken 7 days then this is actually 6 days data.) Now at 12:00:00 AM at the ninth day the first directory will get deleted and so on.

Decryption

If you had selected 7z for compression then you will obtain encrypted files.

Use the following command to decrypt the file.

```
7za x -p<password> <file_name>

# example 7za x -ppassword123 vibration.7z
# assuming password123 is password and file name is vibration.7z.
```

For bzip2 and gzip compression use -d flag to uncompress the file.

```
bzip2 -d <file_name>
gzip -d <file_name>
```

14.3.4 Delta Filter

The *flir-filter-delta* is a filter that only forwards data that changes by more than a configurable percentage. It is used to remove duplicate data values from an asset stream. The definition of duplicate however allows for some noise in the reading value by requiring a delta percentage.

By defining a minimum rate it is possible to force readings to be sent at that defined rate when there is no change in the value of the reading. Rates may be defined as per second, per minute, per hour or per day.

Delta filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *delta* plugin from the list of available plugins.
- Name your delta filter.
- Click *Next* and you will be presented with the following configuration page

The screenshot shows a configuration window titled 'Review Configuration' (step 2 of 2). It contains the following fields:

- Tolerance %**: Input field with value 0.
- Minimum Rate**: Input field with value 0.
- Minimum Rate Units**: Dropdown menu with 'per second' selected.
- Individual Tolerances**: A text area containing a JSON object: `{ }`.
- Enabled**: A checkbox that is currently unchecked.

At the bottom, there are two buttons: 'Previous' and 'Done'.

- Configure the parameters of the delta filter
 - **Tolerance %**: The percentage tolerance when comparing reading data. Only values that differ by more than this percentage will be considered as different from each other.
 - **Minimum Rate**: The minimum rate at which readings should be sent. This is the rate at which readings will appear if there is no change in value.
 - **Minimum Rate Units**: The units in which minimum rate is define (per second, minute, hour or day)
 - **Individual Tolerances**: A JSON document that can be used to define specific tolerance values for an asset. This is defines as a set of name/value pairs for those assets that should use a tolerance percentage other than the global tolerances specified above. The following example would set the tolerance for the temperature asset to 15% and for the pressure asset to 5%. All other assets would use the tolerance specified in *Tolerance %*.

```
{
  "temperature" : 15,
  "pressure" : 5
}
```

- Enable the filter and click *Done* to complete the process of adding the new filter.

14.3.5 Down Sample Filter

The *flir-filter-downsample* filter is a mechanism to reduce the amount of data ingested, it allows the effective data rate to be reduced by a given factor, for example to have the data rate you select a down sample factor of 2, to get a third the rate you select a down sample factor of 3. There are a number of algorithms available to select the value to be sent.

- Sample - the first value in the sample is used as the value for the sample set.
- Mean - the average value in the down sampled set is sent as the down sampled value.
- Median - the mathematical median value is sent as the down sampled value. This is the number found by sorting the sample and choosing the mid point of the sample.
- Mode - the mathematical mode value is sent as the down sampled value. This is the number that appears most often in the sample.
- Minimum - the minimum value in the sample is sent forward.
- Maximum - the maximum value in the sample is sued as the sample value.

Downsample filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *downsample* plugin from the list of available plugins.
- Name your downsample filter.
- Click *Next* and you will be presented with the following configuration page

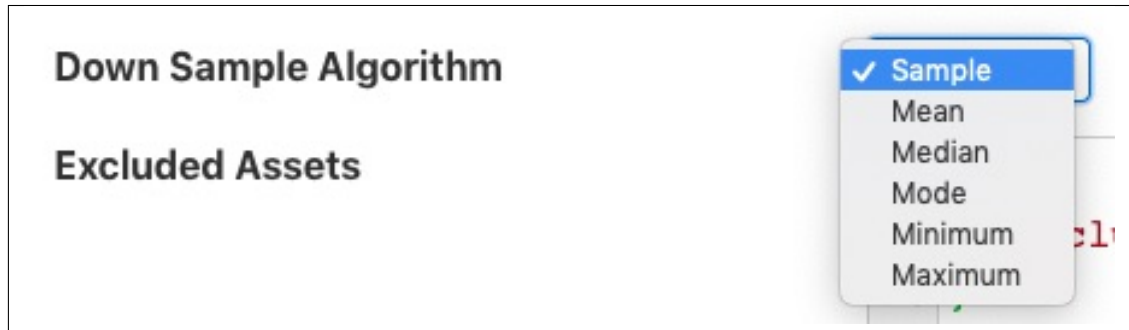
The screenshot shows a configuration window titled 'Sine South Service'. At the top, there is a progress bar with two steps: '1 Plugin Name' and '2 Review Configuration'. The 'Review Configuration' step is currently active. Below the progress bar, the configuration form includes:

- Down Sample Factor:** A text input field containing the value '2'.
- Down Sample Algorithm:** A dropdown menu with 'Sample' selected.
- Excluded Assets:** A code editor showing a JSON snippet:


```
1 {
2   "exclusions": []
3 }
```
- Enabled:** A checkbox that is currently unchecked.

At the bottom of the window, there are two buttons: 'Previous' and 'Done'.

- Configure your downsample filter
 - **Down Sample Factor:** The number of incoming values in each sample set.
 - **Down Sample Algorithm:** The algorithm used to determine the value for the sample.



- **Excluded Assets:** A list of assets that are excluded from the down sampling process.
- Enable your filter and click *Done*

14.3.6 Edge ML Filter Plugin

The plugin takes a image saved by south plugin named webcam media, sends that to Edge ML cluster running somewhere else. The Edge ML cluster returns a response in the form of json which contains information about detected objects, their bounding boxes and confidence score. This information is appended to the readings generated from south service.

- **‘assetName’:** type: string default: **‘edgeML’**: Name of asset to listen on; readings have path names of images to analyze. The plugin will pick these path names to read these images.
- **‘outAssetName’:** type: string default: **‘edgeMLInference’**: Name of asset to write ML inferences on.
- **‘deploymentName’:** type: string default: **“”**: Name of Kubernetes deployment for ML model

- **‘edgeMLUrl’:** type: string default: ‘’: REST URL for ML model which analyzes images; dynamically discovered if empty.
- **‘forwardData’:** type: boolean default: **‘true’**: Forward data as well as inferences.
- **‘rmFile’:** type: string default: **‘false’**: Remove source files after inference.
- **‘enable’:** type: boolean default: **‘true’**: Enable/ Disable the plugin.

Installation

To run the plugin you must follow these prerequisites.

1. **Run the south webcam media plugin.** To run the south webcam media plugin you can either

1. Copy some images inside some directory in FLIR_ROOT/data. Let’s say the directory name is pics. Run the following command.

```
curl -sX POST http://localhost:8081/flir/service -d '{"name":"My_
↪web_cam","type":"south","plugin":"webcam_media","enabled":false,
↪"config":{"assetName":{"value":"WebcamImages"},"imageDir":{"
↪"value":"pics"},"mediaType":{"value":"directory"},"fpm":{"value
↪":"10.0"}}}' |
```

2. Connect a camera to the machine and run the following command.

```
$ v4l2-ctl --list-formats-ext --device /dev/video0
You will see something like
'YUYV' (YUYV 4:2:2)
  Size: Discrete 640x480
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 720x480
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 1280x720
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 1920x1080
    Interval: Discrete 0.067s (15.000 fps)
    Interval: Discrete 0.033s (30.000 fps)
  Size: Discrete 2592x1944
    Interval: Discrete 0.067s (15.000 fps)
  Size: Discrete 0x0
```

Now we know that the id 0 is functional. If no output then try 1,2,3 and so on.

Finally launch the plugin using

```
curl -sX POST http://localhost:8081/flir/service -d '{"name":"My_
↪web_cam","type":"south","plugin":"webcam_media","enabled":false,
↪"config":{"assetName":{"value":"WebcamImages"},"imageDir":{"
↪"value":"webcam"},"mediaType":{"value":"camera"},"cameraNumber
↪":{"value":"0"},"fpm":{"value":"10.0"}}}' |jq
```

2. **Start the Edge ML cluster.** For starting the Edge ML cluster you should follow this [README](#) file.
3. Add the filter Edge ML.

```
curl -sX POST http://localhost:8081/flir/filter -d '{"name":"edge_ml_filter",
↪ "plugin":"edgectl","filter_config":{"deploymentName":"edgectl-deployment",
↪ "assetName":"WebcamImages","outAssetName":"DetectionResults","enable":"true"
↪ "forwardData":"true", "rmFile":"false"}}' |jq
curl -sX PUT 'http://localhost:8081/flir/filter/My_web_cam/pipeline?allow_
↪ duplicates=true&append_filter=true' -d '{"pipeline":["edge_ml_filter"]}' |jq
↪ |jq
```

4. Finally Enable the schedule.

```
curl -sX PUT http://localhost:8081/flir/schedule/enable -d '{"schedule_name":
↪ "My_web_cam"}' |jq
```

14.3.7 Exponential Moving Average

The *flir-filter-ema* plugin implements an exponential moving average across a set of data. It also forms an example of how to write a filter plugin purely in Python. Filters written in Python have the same functionality and set of entry points as any other filter.

The `plugin_info` entry point that returns details of the plugin and the default configuration

```
def plugin_info():
    """ Returns information about the plugin
    Args:
    Returns:
        dict: plugin information
    Raises:
    """
    return {
        'name': 'ema',
        'version': '1.9.2',
        'mode': "none",
        'type': 'filter',
        'interface': '1.0',
        'config': _DEFAULT_CONFIG
    }
```

The `plugin_init` entry point that initialises the plugin

```
def plugin_init(config, ingest_ref, callback):
    """ Initialise the plugin
    Args:
        config: JSON configuration document for the Filter plugin configuration_
↪ category
        ingest_ref:
        callback:
    Returns:
        data: JSON object to be used in future calls to the plugin

    ...
    return data
```

The `plugin_reconfigure` entry point that is called whenever the configuration is changed

```
def plugin_reconfigure(handle, new_config):
    """ Reconfigures the plugin
```

(continues on next page)

(continued from previous page)

```

    Args:
        handle: handle returned by the plugin initialisation call
        new_config: JSON object representing the new configuration category for the_
↪category
    Returns:
        new_handle: new handle to be used in the future calls
    """
    global rate, datapoint
    ...
    return new_handle

```

The `plugin_shutdown` entry point called to terminate the plugin

```

def plugin_shutdown(handle):
    """ Shutdowns the plugin doing required cleanup.
    Args:
        handle: handle returned by the plugin initialisation call
    Returns:
        plugin shutdown
    """

```

And the `plugin_ingest` call that is called to do the actual data processing

```

def plugin_ingest(handle, data):
    """ Modify readings data and pass it onward
    Args:
        handle: handle returned by the plugin initialisation call
        data: readings data
    """

```

Python filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *ema* plugin from the list of available plugins.
- Name your ema filter.
- Click *Next* and you will be presented with the following configuration page

Sine South Service

1 Plugin Name 2 Review Configuration

EMA datapoint

Rate

Enabled ☐

Previous Done

- Configure your ema filter
 - **EMA datapoint:** The name of the data point to create within the asset
 - **Rate:** The rate controls the rate of the average generated, in this case it is the percentage the current value contribute to the average value.
- Enable your plugin and click *Done*

14.3.8 Event Rate Filter

The *flir-filter-eventrate* is a filter plugin that has been explicitly designed to work with the notification server, mechanism and the north service. It can be used to reduce the rate a reading is sent northwards until an interesting event occurs. The filter will read data at full rate from the input side and buffer data internally, sending out averages for each value over a time frame determined by the filter configuration.

The user will provide two strings and a notification asset name that will be used to form a trigger for the filter. One trigger string will set the trigger and the other will clear it. When the trigger is set then the filter will no longer average the data over the configured time period, but will instead send the full bandwidth data out of the filter.

The trigger strings are values in the event data point of the notification asset that is named in the configuration. If the string given in the trigger is found within the event data point then the trigger is deemed to have fired. String matching is case sensitive, but strings given for trigger do not need to be the entire event reason, sub string searching is used to evaluate the trigger.

The filter also allows a pre-trigger time to be configured. In this case it will buffer this much data internally and when the trigger is initially set this pre-buffered data will be sent. The pre-buffered data is discarded if the trigger is not set and the data gets to the defined age for holding pre-trigger information.

Event rate filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *eventrate* plugin from the list of available plugins.
- Name your event rate filter.
- Click *Next* and you will be presented with the following configuration page

Sine South Service

1

2

Plugin Name

Review Configuration

Event asset

event

Trigger Reason

Terminate on

Event

Stop Reason

Full rate time (mS)

0

Pre-trigger time (mS)

1

Reduced collection rate

0

Rate Units

per second

Exclusions

1

2

3

{

"exclusions": []

}

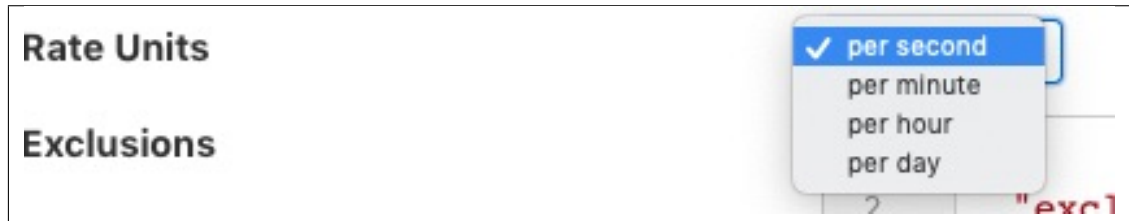
Enabled

Previous

Done

- Configure your event rate filter
 - **Event asset:** The asset used to trigger the full rate sending of data. This is the asset that is inserted by the plugin.
 - **Trigger Reason:** A trigger reason to set the trigger for full rate data
 - **Terminate on:** A switch to control if the end condition is a trigger or time based
 - **Step Reason:** An untrigger reason to clear the trigger for full rate data, if left blank this will simply be the trigger filter evaluating to false
 - **Full rate time (ms):** A full rate time after which the reduce rate is again active
 - **Pre-trigger time (mS):** An optional pre-trigger time expressed in milliseconds
 - **Reduced collection rate:** The nominal data rate to send data out. This defines the period over which is outgoing data item is averaged.

- **Rate Units:** This defines the units used for the above rate. This can be per second, per minute, per hour or per day.



- **Exclusions:** A set of asset names that are excluded from the rate limit processing and always sent at full rate

- Enable your plugin and click *Done*

14.3.9 Expression Filter

The *flir-filter-expression* allows an arbitrary mathematical expression to be applied to data values. The expression filter allows user to augment the data at the edge to include values calculate from one or more asset to be added and acted upon both within the FLIR Bridge system itself, but also forwarded on to the up stream systems. Calculations can range from very simply manipulates of a single value to convert ranges, e.g. a linear scale to a logarithmic scale, or can combine multiple values to create composite value. E.g. create a power reading from voltage and current or work out a value that is normalized for speed.

Expression filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *expression* plugin from the list of available plugins.
- Name your expression filter.
- Click *Next* and you will be presented with the following configuration page

- Configure the expression filter
 - **Datapoint Name:** The name of the new data point into which the new value will be stored.
 - **Expression to apply:** This is the expression that will be evaluated for each asset reading. The expression will use the data points within the reading as symbols within the asset. See [Expressions](#) below.

- Enable the plugin and click *Done* to activate your filter

Expressions

The *flir-filter-expression* plugin makes use of the library to do run time expression evaluation. This library provides a rich mathematical operator set, the most useful of these in the context of this plugin are;

- Logical operators (and, nand, nor, not, or, xor, xnor, mand, mor)
- Mathematical operators (+, -, *, /, %, ^)
- Functions (min, max, avg, sum, abs, ceil, floor, round, roundn, exp, log, log10, logn, pow, root, sqrt, clamp, inrange, swap)
- Trigonometry (sin, cos, tan, acos, asin, atan, atan2, cosh, cot, csc, sec, sinh, tanh, d2r, r2d, d2g, g2d, hyp)

Within the expression the data points of the asset become symbols that may be used; therefore if an asset contains values “voltage” and “current” the expression will contain those as symbols and an expression of the form

```
voltage * current
```

can be used to determine the power in Watts.

When the filter is used in an environment in which more than one asset is passing through the filter then symbols are created of the form <asset name>.<data point>. As an example if you have one asset called “electrical” that has data points of “voltage” and “current” and another asset called “speed” that has a data point called “rpm” then you can write an expression to obtain the power per 1000 RPM’s of the motor as follows;

```
(electrical.voltage * electrical.current) / (speed.rpm / 1000)
```

14.3.10 Fast Fourier Transform Filter

The *flir-filter-fft* filter is designed to accept some periodic data such as a sample electrical waveform, audio data or vibration data and perform a Fast Fourier Transform on that data to supply frequency data about that waveform.

Data is added as a new asset which is named as the sampled asset with “FFT” append. This FFT asset contains a set of data points that each represent the a band of frequencies, or as a frequency spectrum in a single array data point. The band information that is returned by the filter can be chosen by the user. The options available to represent each band are;

- the average in the band,
- the peak
- the RMS
- or the sum of the band.

The bands are created by dividing the frequency space into a number of equal ranges after first applying a low and high frequency filter to discard a percentage of the low and high frequency results. The bands are not created if the user instead opts to return the frequency spectrum.

If the low Pass filter is set to 15% and the high Pass filter is set to 10%, with the number of bands set to 5, the lower 15% of results are discarded and the upper 10% are discarded. The remaining 75% of readings are then divided into 5 equal bands, each of which representing 15% of the original result space. The results within each of the 15% bands are then averaged to produce a result for the frequency band.

FFT filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.

- Select the *fft* plugin from the list of available plugins.
- Name your FFT filter.
- Click *Next* and you will be presented with the following configuration page

Waveform South Service

1 Plugin Name 2 Review Configuration

Asset to analysis: wave

Result Data: average

Frequency Bands: 10

Band Prefix: Band

No. of samples per FFT: 8194

Low Frequency Reject %: 0

High Frequency Reject %: 0

Enabled: ☒

Previous Done

- Configure your FFT filter
 - **Asset to analysis:** The name of the asset that will be used as the input to the FFT algorithm.

Result Data

Frequency Bands

Band Prefix

average

peak

sum

rms

spectrum

- **Result Data:** The data that should be returned for each band. This may be one of average, sum, peak, rms or spectrum. Selecting average will return the average amplitude within the band, sum returns the sum of all amplitudes within the frequency band, peak the greatest amplitude and rms the root mean square of the amplitudes within the band. Setting the output type to be spectrum will result in the full FFT spectrum data being written. Spectrum data however can not be sent to all north destinations as it is not supported natively on all the systems FLIR Bridge can send data to.

- **Frequency Bands:** The number of frequency bands to divide the resultant FFT output into
- **Band Prefix:** The prefix to add to the data point names for each band in the output
- **No. of Samples per FFT:** The number of input samples to use. This must be a power of 2.
- **Low Frequency Reject %:** A percentage of low frequencies to discard, effectively reducing the range of frequencies to examine
- **High Frequency Reject %:** A percentage of high frequencies to discard, effectively reducing the range of frequencies to examine

14.3.11 Flir Validity Filter

The *flir-filter-Flir-Validity* plugin is a simple filter that filters out unused boxes and spot temperatures in the Flir temperature data stream. The filter also allows the naming of the boxes such that the data points added to the asset will use these names rather than the default box1, box2 etc.

Adding the filter to a south plugin you will receive a configuration screen as below

AX8 South Service

1

2

Plugin Name

Review Configuration

Area Labels

1

2

3

4

5

6

7

8

9

10

11

12

```
{
  "areas": [
    "1",
    "2",
    "3",
    "4",
    "5",
    "6",
    "7",
    "8",
    "9",
    "10",
  ]
}
```

Enabled

☒

Previous

Done

The JSON document *Area Labels* can be used to set the labels to use for each of the boxes and replace the min1, min2 etc. The value of this configuration option is a JSON document that has a single element called *areas* which is a JSON array. Each element in that area is the name to assign to the particular box. The default values would set the name of box1 to simply be 1, box2 to 2 etc.

If we assume we are monitoring a lathe with the camera and taking the temperature of the motor, the bearing and cutting bit using the boxes 1, 2, and 3 in the camera. We wish to rename the first box to be called *Motor*, the second box to be called *Bearing* and the third to be called *Tool*, setting an *areas* array as follows would achieve this.

```
{
  "areas" : [
    "Motor",
    "Bearing",
    "Tool",
    "4",
    "5",
    "6",
    "7",
    "8",
    "9",
    "10"
  ]
}
```

Note that we do not change the boxes 4 to 10 as these are not in use and have not been defined within the area interface. Using the above configuration setting for areas will result in asset names of *minMotor*, *maxMotor* and *averageMotor* being generated for the motor temperature. Similarly the bearing temperatures would be *minBearing*, *maxBearing* and *averageBearing*. The tool would have asset names of *minTool*, *maxTool* and *averageTool*.

14.3.12 Log Filter

The *flir-filter-log* plugin is a simple filter that converts data to a logarithmic scale.

When adding a scale filter to either the south service or north task, via the *Add Application* option of the user interface, a configuration page for the filter will be shown as below;

The screenshot displays a configuration window for the 'flir-filter-log' plugin. At the top, a progress bar indicates two steps: '1 Plugin Name' and '2 Review Configuration'. The 'Review Configuration' step is active. Below this, there is a white box containing the configuration options. The 'Asset filter' is represented by a text input field. The 'Enabled' option is a checkbox that is currently checked. At the bottom of the window, there are two buttons: 'Previous' and 'Done'.

The *Asset Filter* entry is a regular expression that can be used to limit the assets that the filter will effect. To change

all assets leave this entry blank.

14.3.13 Metadata Filter

The *flir-filter-metadata* filter allows data to be added to assets within FLIR Bridge. Metadata takes the form of fixed data points that are added to an asset used to add context to the data. Examples of metadata might be unit of measurement information, location information or identifiers for the piece of equipment to which the measurement relates.

A metadata filter may be added to either a south service or a north task. In a south service it will be adding data for just those assets that originate in that service, in which case it probably relates to a single machine that is being monitored and would add metadata related to that machine. In a north task it causes metadata to be added to all assets that the FLIR Bridge is sending to the up stream system, in which case the metadata would probably related to that particular FLIR Bridge instance. Adding metadata in the north is particularly useful when a hierarchy of FLIR Bridge systems is used and an audit trail is required with the data or the individual FLIR Bridge systems related to some physical location information such as building, floor and/or site.

To add a metadata filter

- Click on the Applications add icon for your service or task.
- Select the *metadata* plugin from the list of available plugins.
- Name your metadata filter.
- Click *Next* and you will be presented with the following configuration page

1 Plugin Name 2 Review Configuration

Metadata to add

```

1 {
2   "floor": "Third",
3   "location": "AirIntake",
4   "unit": "Celsius",
5   "serialNo": "A73953-42492-3229"
6 }

```

Enabled ☒

- Enter your metadata in the JSON array shown. You may add multiple items in a single filter by separating them with commas. Each item takes the format of a JSON key/value pair and will be added as data points within the asset.
- Enable the filter and click on *Done* to activate it

Example Metadata

Assume we are reading the temperature of air entering a paint booth. We might want to add the location of the paint booth, the booth number, the location of the sensor in the booth and the unit of measurement. We would add the following configuration value

```
{
  "value": {
    "floor": "Third",
    "booth": 1,
    "units": "C",
    "location": "AirIntake"
  }
}
```

In above example the filter would add “floor”, “booth”, “units” and “location” data points to all the readings processed by it. Given an input to the filter of

```
{ "temperature" : 23.4 }
```

The resultant reading that would be passed onward would become

```
{ "temperature" : 23.5, "booth" : 1, "units" : "C", "floor" : "Third", "location" :
  ↪ "AirIntake" }
```

This is an example of how metadata might be added in a south service. Turning to the north now, assume we have a configuration whereby we have several sites in an organization and each site has several building. We want to monitor data about the buildings and install a FLIR Bridge instance in each building to collect building data. We also install a FLIR Bridge instance in each site to collect the data from each individual FLIR Bridge instance per building, this allows us to then send the site data to the head office without having to allow each building FLIR Bridge to have access to the corporate network. Only the site FLIR Bridge needs that access. We want to label the data to say which building it came from and also which site. We can do this by adding metadata at each stage.

To the north task of a building FLIR Bridge, for example the “Pearson” building, we add the following metadata

```
{
  "value" : {
    "building": "Pearson"
  }
}
```

Likewise to the “Lawrence” building FLIR Bridge instance we add the following to the north task

```
{
  "value" : {
    "building": "Lawrence"
  }
}
```

These buildings are both in the “London” site and will send their data to the site FLIR Bridge instance. In this instance we have a north task that sends the data to the corporate headquarters, in this north task we add

```
{
  "value" : {
    "site": "London"
  }
}
```

If we assume we measure the power flow into each building in terms of current, and for the Pearson building we have a value of 117A at 11:02:15 and for the Lawrence building we have a value of 71.4A at 11:02:23, when the data is received at the corporate system we would see readings of

```
{ "current" : 117, "site" : "London", "building" : "Pearson" }  
{ "current" : 71.4, "site" : "London", "building" : "Lawrence" }
```

By adding the data like this it gives us more flexibility, if for example we want to change the way site names are reported, or we acquire a second site in London, we only have to change the metadata in one place.

14.3.14 OMF Hint Filter

The *flir-filter-omfhint* filter allows hints to be added to assets within FLIR Bridge that will be used by the plugin. These hints allow for individual configuration of specific assets within the OMF plugin.

A OMF hint filter may be added to either a south service or a north task. In a south service it will be adding data for just those assets that originate in that service. In a north task it causes OMF hints to be added to all assets that the FLIR Bridge is sending to the up stream system, it would normally only be used in a north that was using the OMF plugin, however it could be used in a north that is sending data to another FLIR Bridge that then forwards to OMF.

To add a OMF hints filter

- Click on the Applications add icon for your service or task.
- Select the *omfhint* plugin from the list of available plugins.
- Name your OMF hint filter.
- Click *Next* and you will be presented with the following configuration page

1 Plugin Name 2 Review Configuration

OMF Hint

```

1 {
2   "asset": {
3     "number": "float64"
4   }
5 }

```

Enabled ☐

Previous Done

- Enter your OMF Hints in the JSON editor shown. You may add multiple hints for multiple assets in a single filter instance. See [OMF Hint data](#)
- Enable the filter and click on *Done* to activate it

OMF Hint data

OMF Hints comprise of an asset name which the hint applies and a JSON document that is the hint. A hint is a name/value pair, the name is the hint type and the value is the value of that hint.

The asset name may be expressed as a regular expression, in which case the hint is applied to all assets that match that regular expression.

The following hint types are currently supported by

- *integer*: The format to use for integers, the value is a string and may be any of the PI Server supported formats; int64, int32, int16, uint64, uint32 or uint16
- *number*: The format to use for numbers, the value is a string and may be any of the PI Server supported formats; float64, float32 or float16
- *typeName*: Specify a particular type name that should be used by the plugin when it generates a type for the asset. The value of the hint is the name of the type to create.
- *tagName*: Specify a particular tag name that should be used by the plugin when it generates a tag for the asset. The value of the hint is the name of the tag to create.

- *type*: Specify a pre-existing type that should be used for the asset. In this case the value of the hint is the type to use. The type must already exist within your PI Server and must be compatible with the values within the asset.
- *datapoint*: Specifies that this hint applies to a single datapoint within the asset. The value is a JSON object that contains the name of the datapoint and one or more hints.

The following example shows a simple hint to set the number format to use for all numeric data within the asset names *supply*.

```
{
  "supply": {
    "number": "float32"
  }
}
```

To apply a hint to all assets, the single hint definition can be used with a regular expression.

```
{
  ".*": {
    "number": "float32"
  }
}
```

Regular expressions may also be used to select subsets of assets, in the following case only assets with the prefix OPCUA will have the hint applied.

```
{
  "OPCUA.*": {
    "number": "float32"
  }
}
```

To apply a hint to a particular data point the hint would be as follows

```
{
  "supply": {
    "datapoint": {
      "name": "frequency"
      "integer": "uint16"
    }
  }
}
```

This example sets the datapoint *frequency* within the *supply* asset to be stored in the PI server as a uint16.

Datapoint hints can be combined with asset hints

```
{
  "supply": {
    "number": "float32",
    "datapoint": {
      "name": "frequency"
      "integer": "uint16"
    }
  }
}
```

In this case all numeric data except for *frequency* will be stored as float32 and *frequency* will be stored as uint16.

14.3.15 Python 2.7 Filter

The *flir-filter-python27* filter allows snippets of Python to be easily written that can be used as filters in FLIR Bridge. A similar filter exists that uses Python 3.5 syntax, the filter. A Python code snippet will be called with sets of asset readings as they are read or processed in a filter pipeline. The data appears in the Python code as a JSON document passed as a Python Dict type.

The user should provide a Python function whose name matches the name given to the plugin when added to the filter pipeline of the south service or north task, e.g. if you name your filter *myPython* then you should have a function named *myPython* in the code you enter. This function is send a set of readings to process and should return a set of processed readings. The returned set of readings may be empty if the filter removes all data.

A general code syntax for the function that should be provided is;

```
def myPython(readings):
    for elem in list(readings):
        ...
    return readings
```

Each element that is processed has a number of attributes that may be accessed

Attribute	Description
asset_code	The name of the asset the reading data relates to.
timestamp	The data and time FLIR Bridge first read this data
user_timestamp	The data and time the data for the data itself, this may differ from the timestamp above
readings	The set of readings for the asset, this is itself an object that contains a number of key/value pairs that are the data points for this reading.

In order to access an data point within the readings, for example one named *temperature*, it is a simple case of extracting the value of with *temperature* as its key.

```
def myPython(readings):
    for elem in list(readings):
        reading = elem['readings']
        temp = reading['temperature']
        ...
    return readings
```

It is possible to write your Python code such that it does not know the data point names in advance, in which case you are able to iterate over the names as follows;

```
def myPython(readings):
    for elem in list(readings):
        reading = elem['readings']
        for attribute in reading:
            value = reading[attribute]
            ...
    return readings
```

A second function may be provided by the Python plugin code to accept configuration from the plugin that can be used to modify the behavior of the Python code without the need to change the code. The configuration is a JSON document which is again passed as a Python Dict to the *set_filter_config* function in the user provided Python code. This function should be of the form

```
def set_filter_config(configuration):
    config = json.loads(configuration['config'])
    value = config['key']
    ...
    return True
```

Python27 filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *python27* plugin from the list of available plugins.
- Name your python27 filter, this should be the same name as the Python function you will provide.
- Click *Next* and you will be presented with the following configuration page

1 Plugin Name
2 Review Configuration

Python script

```

1  # generate exponential moving average
2
3  import json
4
5  # exponential moving average rate default value: include 7%
6  # of current value
7  rate = 0.07
8  # latest ema value
9  latest = None
10
11 # get configuration if provided.
12 # set this JSON string in configuration:
13 #     {"rate":0.07}

```

ema.py

No file chosen

Configuration

```

1  {"rate" : 0.75}

```

Enabled ☐

- Enter the configuration for your python27 filter
 - **Python script:** This is the script that will be executed. Initially you are unable to type in this area and must load your initial script from a file using the *Choose Files* button below the text area. Once a file has been chosen and loaded you are able to update the Python code in this page.

Note: Any changes made to the script in this screen will **not** be written back to the original file it was loaded from.

- **Configuration:** You may enter a JSON document here that will be passed to the `set_filter_config` function of your Python code.

- Enable the python27 filter and click on *Done* to activate your plugin

Example

The following example uses Python to create an exponential moving average plugin. It adds a data point called *ema* to every asset. It assumes a single data point exists within the asset, but it does not assume the name of that data point. A rate can be set for the EMA using the configuration of the plugin.

```
# generate exponential moving average

import json

# exponential moving average rate default value: include 7% of current value
rate = 0.07
# latest ema value
latest = None

# get configuration if provided.
# set this JSON string in configuration:
# {"rate":0.07}
def set_filter_config(configuration):
    global rate
    config = json.loads(configuration['config'])
    if ('rate' in config):
        rate = config['rate']
    return True

# Process a reading
def doit(reading):
    global rate, latest

    for attribute in list(reading):
        if not latest:
            latest = reading[attribute]
        else:
            latest = reading[attribute] * rate + latest * (1 - rate)
            reading[b'ema'] = latest

# process one or more readings
def ema(readings):
    for elem in list(readings):
        doit(elem['reading'])
    return readings
```

Examining the content of the Python, a few things to note are;

- The filter is given the name *ema*. This name defines the default method which will be executed, namely *ema()*.
- The function *ema* is passed 1 or more readings to process. It splits these into individual readings, and calls the function *doit* to perform the actual work.

- The function `doit` walks through each attribute in that reading, updates a global variable `latest` with the latest value of the `ema`. It then adds an `ema` attribute to the reading.
- The function `ema` returns the modified readings list which then is passed to the next filter in the pipeline.
- `set_filter_config()` is called whenever the user changes the JSON configuration in the plugin. This function will alter the global variable `rate` that is used within the function `doit`.

14.3.16 Python 3.5 Filter

The `flir-filter-python35` filter allows snippets of Python to be easily written that can be used as filters in FLIR Bridge. A similar filter exists that uses Python 2.7 syntax, the filter. A Python code snippet will be called with sets of asset readings as they are read or processed in a filter pipeline. The data appears in the Python code as a JSON document passed as a Python Dict type.

The user should provide a Python function whose name matches the name given to the plugin when added to the filter pipeline of the south service or north task, e.g. if you name your filter `myPython` then you should have a function named `myPython` in the code you enter. This function is send a set of readings to process and should return a set of processed readings. The returned set of readings may be empty if the filter removes all data.

A general code syntax for the function that should be provided is;

```
def myPython(readings):
    for elem in list(readings):
        ...
    return readings
```

Each element that is processed has a number of attributes that may be accessed

Attribute	Description
<code>asset_code</code>	The name of the asset the reading data relates to.
<code>timestamp</code>	The data and time FLIR Bridge first read this data
<code>user_timestamp</code>	The data and time the data for the data itself, this may differ from the timestamp above
<code>readings</code>	The set of readings for the asset, this is itself an object that contains a number of key/value pairs that are the data points for this reading.

In order to access an data point within the readings, for example one named `temperature`, it is a simple case of extracting the value of with `temperature` as its key.

```
def myPython(readings):
    for elem in list(readings):
        reading = elem['readings']
        temp = reading['temperature']
        ...
    return readings
```

It is possible to write your Python code such that it does not know the data point names in advance, in which case you are able to iterate over the names as follows;

```
def myPython(readings):
    for elem in list(readings):
        reading = elem['readings']
        for attribute in reading:
            value = reading[attribute]
```

(continues on next page)

(continued from previous page)

```
...  
return readings
```

A second function may be provided by the Python plugin code to accept configuration from the plugin that can be used to modify the behavior of the Python code without the need to change the code. The configuration is a JSON document which is again passed as a Python Dict to the `set_filter_config` function in the user provided Python code. This function should be of the form

```
def set_filter_config(configuration):  
    config = json.loads(configuration['config'])  
    value = config['key']  
    ...  
    return True
```

Python35 filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *python35* plugin from the list of available plugins.
- Name your python35 filter, this should be the same name as the Python function you will provide.
- Click *Next* and you will be presented with the following configuration page

1
2

Plugin Name
Review Configuration

Python script

```

1  # generate exponential moving average
2
3  import json
4
5  # exponential moving average rate default value: include 7%
  of current value
6  rate = 0.07
7  # latest ema value
8  latest = None
9
10 # get configuration if provided.
11 # set this JSON string in configuration:
12 #     {"rate":0.07}

```

ema.py

Choose Files

No file chosen

Configuration

```

1  {"rate" : 0.75}

```

Enabled

☐

Previous

Done

- Enter the configuration for your python35 filter
 - **Python script:** This is the script that will be executed. Initially you are unable to type in this area and must load your initial script from a file using the *Choose Files* button below the text area. Once a file has been chosen and loaded you are able to update the Python code in this page.

Note: Any changes made to the script in this screen will **not** be written back to the original file it was loaded from.

- **Configuration:** You may enter a JSON document here that will be passed to the *set_filter_config* function of your Python code.
- Enable the python35 filter and click on *Done* to activate your plugin

Example

The following example uses Python to create an exponential moving average plugin. It adds a data point called *ema* to every asset. It assumes a single data point exists within the asset, but it does not assume the name of that data point. A rate can be set for the EMA using the configuration of the plugin.


```

# generate exponential moving average

import json

# exponential moving average rate default value: include 7% of current value
rate = 0.07
# latest ema value
latest = None

# get configuration if provided.
# set this JSON string in configuration:
# {"rate":0.07}
def set_filter_config(configuration):
    global rate
    config = json.loads(configuration['config'])
    if ('rate' in config):
        rate = config['rate']
    return True

# Process a reading
def doit(reading):
    global rate, latest

    for attribute in list(reading):
        if not latest:
            latest = reading[attribute]
        else:
            latest = reading[attribute] * rate + latest * (1 - rate)
            reading[b'ema'] = latest

# process one or more readings
def ema(readings):
    for elem in list(readings):
        doit(elem['reading'])
    return readings

```

Examining the content of the Python, a few things to note are;

- The filter is given the name `ema`. This name defines the default method which will be executed, namely `ema()`.
- The function `ema` is passed 1 or more readings to process. It splits these into individual readings, and calls the function `doit` to perform the actual work.
- The function `doit` walks through each attribute in that reading, updates a global variable `latest` with the latest value of the `ema`. It then adds an `ema` attribute to the reading.
- The function `ema` returns the modified readings list which then is passed to the next filter in the pipeline.
- `set_filter_config()` is called whenever the user changes the JSON configuration in the plugin. This function will alter the global variable `rate` that is used within the function `doit`.

14.3.17 Rate Filter

The *flir-filter-rate* plugin that can be used to reduce the rate a reading is stored until an interesting event occurs. The filter will read data at full rate from the input side and buffer data internally, sending out averages for each value over a time frame determined by the filter configuration.

The user can provide either one or two simple expressions that will be evaluated to form a trigger for the filter. One

expressions will set the trigger and the other will clear it. When the trigger is set then the filter will no longer average the data over the configured time period, but will instead send the full bandwidth data out of the filter. If the second expression, the one that clears the full rate sending of data is omitted then the full rate is cleared as soon as the trigger expression returns false. Alternatively the filter can be configured to clear the sending of full rate data after a fixed time.

The filter also allows a pre-trigger time to be configured. In this case it will buffer this much data internally and when the trigger is initially set this pre-buffered data will be sent. The pre-buffered data is discarded if the trigger is not set and the data gets to the defined age for holding pre-trigger information.

Rate filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *rate* plugin from the list of available plugins.
- Name your rate filter.
- Click *Next* and you will be presented with the following configuration page

1 Plugin Name 2 Review Configuration

Trigger expression

Terminate on

End Expression

Full rate time (mS)

Pre-trigger time (mS)

Reduced collection rate

Rate Units

Exclusions

```

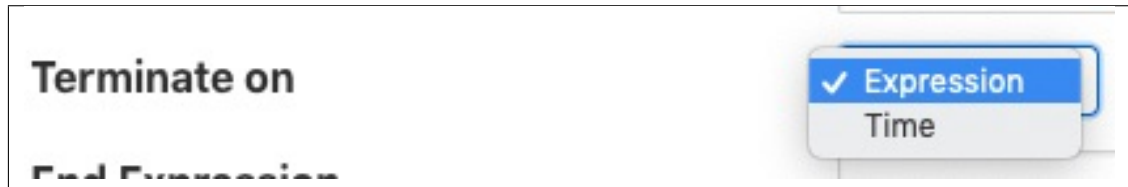
1 {
2   "exclusions": [ "speed" ]
3 }

```

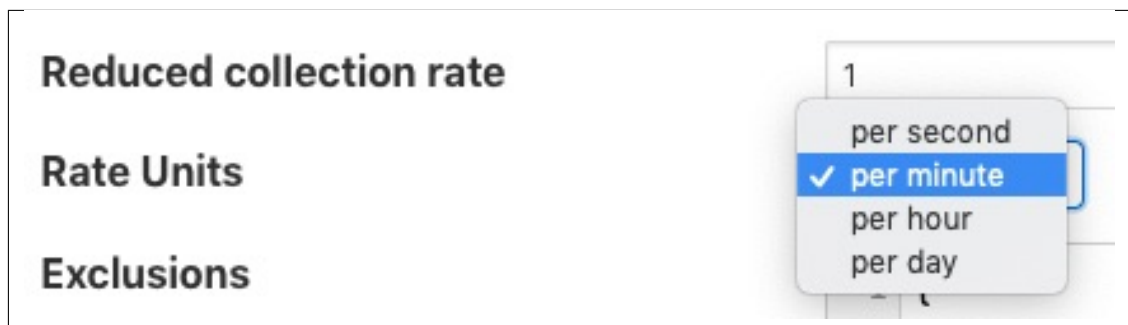
Enabled ☒

- Configure your rate filter
 - **Trigger Expression:** An expression to set the trigger for full rate data

- **Terminate ON:** The mechanism to stop full rate forwarding, this may be another expression or a time window



- **End Expression:** An expression to clear the trigger for full rate data, if left blank this will simply be the trigger filter evaluating to false
- **Full rate time (ms):** The time window, in milliseconds to forward data at the full rate
- **Pre-trigger time (ms):** An optional pre-trigger time expressed in milliseconds
- **Reduced collection rate:** The nominal data rate to send data out. This defines the period over which is outgoing data item is averaged.
- **Rate Units:** The units that the reduced collection rate is expressed in; per second, minute, hour or day



- **Exclusions:** A set of asset names that are excluded from the rate limit processing and always sent at full rate
- Enable your filter and click *Done*

For example if the filter is working with a SensorTag and it reads the tag data at 10ms intervals but we only wish to send 1 second averages under normal circumstances. However if the X axis acceleration exceed 1.5g then we want to send full bandwidth data until the X axis acceleration drops to less than 0.2g, and we also want to see the data for the 1 second before the acceleration hit this peak the configuration might be:

- **Nominal Data Rate:** 1, data rate unit “per second”
- **Trigger set expression:** $X > 1.5$
- **Trigger clear expression:** $X < 0.2$
- **Pre-trigger time (mS):** 1000

The trigger expression uses the same expression mechanism, as the , and plugins

Expression may contain any of the following...

- Mathematical operators (+, -, *, /, %, ^)
- Functions (min, max, avg, sum, abs, ceil, floor, round, roundn, exp, log, log10, logn, pow, root, sqrt, clamp, inrange, swap)
- Trigonometry (sin, cos, tan, acos, asin, atan, atan2, cosh, cot, csc, sec, sinh, tanh, d2r, r2d, d2g, g2d, hyp)

- Equalities & Inequalities (=, ==, <>, !=, <, <=, >, >=)
- Logical operators (and, nand, nor, not, or, xor, xnor, mand, mor)

Note: This plugin is designed to work with streams with a single asset in the stream, there is no mechanism in the expression syntax to support multiple asset names.

14.3.18 Rename Filter

The *flir-filter-rename* filter that can be used to modify the name of an asset, datapoint or both. It may be used either in *South* services or *North* services or *North* tasks.

To add a Rename filter

- Click on the Applications add icon for your service or task.
- Select the *rename* plugin from the list of available plugins.
- Name your Rename filter.
- Click *Next* and you will be presented with the following configuration page
- Configure the plugin

The screenshot displays the configuration interface for the Rename Filter. At the top, a progress bar indicates two steps: 'Plugin Name' (marked with a green circle 1) and 'Review Configuration' (marked with a green circle 2). The 'Review Configuration' section contains a form with the following fields:

- Operation:** A dropdown menu with 'asset' selected.
- Find:** A text input field containing 'assetName'.
- Replace With:** A text input field containing 'newAssetName'.
- Enabled:** A checkbox that is currently unchecked.

At the bottom of the form, there are two buttons: 'Previous' and 'Done'.

- **Operation:** Search and replace operation be performed on asset name, datapoint name or both
- **Find:** A regular expression to match for the given operation
- **Replace With:** A substitution string to replace the matched text with
- Enable the filter and click on *Done* to activate it

Example

The simplest following example perform on given below reading object

```
{
  "readings": {
    "sinusoid": -0.978147601,
    "a": {
      "sinusoid": "2.0"
    }
  },
  "asset": "sinusoid",
  "id": "albedea3-8d80-47e8-b256-63370ccfce5b",
  "ts": "2021-06-28 14:03:22.106562+00:00",
  "user_ts": "2021-06-28 14:03:22.106435+00:00"
}
```

1. To replace an asset apply a configuration would be as follows

- Operation : asset
- Find : sinusoid
- Replace With : sin

Output

```
{
  "readings": {
    "sinusoid": -0.978147601,
    "a": {
      "sinusoid": 2.0
    }
  },
  "asset": "sin",
  "id": "albedea3-8d80-47e8-b256-63370ccfce5b",
  "ts": "2021-06-28 14:03:22.106562+00:00",
  "user_ts": "2021-06-28 14:03:22.106435+00:00"
}
```

2. To replace a datapoint apply a configuration would be as follows

- Operation : datapoint
- Find : sinusoid
- Replace With : sin

Output

```
{
  "readings": {
    "sin": -0.978147601,
    "a": {
      "sin": 2.0
    }
  },
  "asset": "sinusoid",
  "id": "albedea3-8d80-47e8-b256-63370ccfce5b",
  "ts": "2021-06-28 14:03:22.106562+00:00",
  "user_ts": "2021-06-28 14:03:22.106435+00:00"
}
```

3. To replace both asset and datapoint apply a configuration would be as follows

- Operation : both
- Find : sinusoid
- Replace With : sin

Output

```
{
  "readings": {
    "sin": -0.978147601,
    "a": {
      "sin": 2.0
    }
  },
  "asset": "sin",
  "id": "albedea3-8d80-47e8-b256-63370ccfce5b",
  "ts": "2021-06-28 14:03:22.106562+00:00",
  "user_ts": "2021-06-28 14:03:22.106435+00:00"
}
```

14.3.19 Replace Filter

The *flir-filter-replace* is a filter that allows an be used to replace all occurrence of a set of characters with a single replacement character. This can be used to change reserved characters in the names of assets and datapoints.

The screenshot displays the configuration interface for the 'Replace Filter'. At the top, a progress bar indicates two steps: '1 Plugin Name' and '2 Review Configuration'. The main configuration area contains three fields: 'Replace' with the value '\?', 'With' with the value '-', and an 'Enabled' checkbox which is currently unchecked. A help icon (?) is visible in the top right of the configuration panel. At the bottom, there are two buttons: 'Previous' and 'Done'.

- **Replace:** The set of reserved characters to be replaced.
- **With:** The character to replace each occurrence of the above characters with

14.3.20 Root Mean Squared (RMS) Filter

The *flir-filter-rms* filter is designed to accept some periodic data such as a sample electrical waveform, audio data or vibration data and perform a Root Mean Squared, *RMS* operation on that data to supply power of the waveform. The

filter can also return the *peak to peak* amplitude of the waveform over the sampled period and the *crest* factor of the waveform.

Note: peak values may be less than individual values of the input if the asset value does not fall to or below zero. Where a data value swings between negative and positive values then the peak value will be greater than the maximum value in the data stream. For example if the minimum value of a data point in the sample set is 0.3 and the maximum is 3.4 then the peak value will be 3.1. If the maximum value is 2.4 and the minimum is zero then the peak will be 2.4. If the maximum value is 1.7 and the minimum is -0.5 then the peak value will be 2.2.

RMS, also known as the quadratic mean, is defined as the square root of the mean square (the arithmetic mean of the squares of a set of numbers).

Peak to peak, is the difference between the smallest value in the sampled data and the highest, this gives the maximum amplitude variation during the period sampled.

Crest factor is a parameter of a waveform, showing the ratio of peak values to the effective value. In other words, crest factor indicates how extreme the peaks are in a waveform. Crest factor 1 indicates no peaks, such as direct current or a square wave. Higher crest factors indicate peaks, for example sound waves tend to have high crest factors.

The user may also choose to include or not the raw data that is used to calculate the RMS values via a switch in the configuration.

Where a data stream has multiple assets within it the RMS filter may be limited to work only on those assets whose name matches a regular expression given in the configuration of the filter. The default for this expression is `.*`, i.e. all assets are processed.

RMS filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *rms* plugin from the list of available plugins.
- Name your RMS filter.
- Click *Next* and you will be presented with the following configuration page

Waveform South Service

1 Plugin Name 2 Review Configuration

Sample size: 10

RMS Asset name: %a RMS

Include Peak Values: ☐

Include Crest Values: ☐

Include Raw Data: ☐

Asset filter: .*

Enabled: ☐

Previous Done

- Configure your RMS filter
 - **Sample size:** The number of data samples to perform a calculation over.
 - **RMS Asset name:** The asset name to use to output the RMS values. “%a” will be replaced with the original asset name.
 - **Include Peak Values:** A switch to include peak to peak measurements for the same data set as the RMS measurement.
 - **Include Crest Values:** A switch to include crest measurements for the same data set as the RMS measurement.
 - **Include Raw Data:** A switch to include the raw input data in the output.
 - **Asset Filter:** A regular expression to limit the asset names on which this filter operations. Useful when multiple assets appear in the input data stream as it allows data which is not part of the periodic function that is being examined to be excluded.

14.3.21 Scale Filter

The *flir-filter-scale* plugin is a simple filter that allows a scale factor and an offset to be applied to numerical data. It’s primary uses are for adjusting values to match different measurement scales, for example converting temperatures from Centigrade to Fahrenheit or when a sensor reports a value in non-base units, e.g. 1/10th of a degree.

When adding a scale filter to either the south service or north task, via the *Add Application* option of the user interface, a configuration page for the filter will be shown as below;

The screenshot shows a configuration window titled "Sine South Service". At the top, there's a progress indicator with two steps: "1 Plugin Name" and "2 Review Configuration". The "Review Configuration" step is currently active. Below this, there's a form with the following fields:

- Scale Factor**: A text input field containing "100.0".
- Constant Offset**: A text input field containing "0.0".
- Asset filter**: A text input field that is currently empty.
- Enabled**: A checkbox that is checked.

At the bottom of the window, there are two buttons: "Previous" and "Done".

The configuration options supported by the scale filter are detailed in the table below

Setting	Description
Scale Factor	The scale factor to multiply the numeric values by
Constant Offset	A constant to add to all numeric values after applying the scale
Asset filter	This is useful when applying the filter in the north, it allows the filter to be applied only to those assets that match the regular expression given. If left blank then the filter is applied to all assets/

14.3.22 Scale Set Filter

The *flir-filter-scale-set* plugin is a filter that allows a scale factor and an offset to be applied to numerical data where an asset has multiple data points. It is very similar to the filter, which allows a single scale and offset to be applied to all assets and data points. It's primary uses are for adjusting values to match different measurement scales, for example converting temperatures from Centigrade to Fahrenheit or when a sensor reports a value in non-base units, e.g. 1/10th of a degree.

Scale set filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *scale-set* plugin from the list of available plugins.
- Name your scale-set filter.
- Click *Next* and you will be presented with the following configuration page

- Enter the configuration for your change filter
 - **Scale factors:** A JSON document that defines a set of factors to apply. It is an array of JSON objects that define the scale factor and offset, a regular expression that is matched against the asset name and another that matches the data point name within the asset.

Name	Description
asset	A regular expression to match against the asset name. The scale factor is only applied to assets whose name matches this regular expression.
data-point	A regular expression to match against the data point name within a matching asset. The scale factor is only applied to assets whose name matches this regular expression.
scale	The scale factor to apply to the numeric data.
off-set	The offset to add to the matching numeric data.

- Enable the scale-set filter and click on *Done* to activate your plugin

Example

In the following example we have an asset whose name is *environment* which contains two data points; *temperature* and *humidity*. We wish to allow two different scale factors and offsets to these two data points whilst not affecting assets of any other name in the data stream. We can accomplish this by using the following JSON document in the plugin configuration;

```
{
  "factors" : [
    {
```

(continues on next page)

(continued from previous page)

```

    "asset"      : "environment",
    "datapoint"  : "temperature",
    "scale"      : 1.8,
    "offset"     : 32
  },
  {
    "asset"      : "environment",
    "datapoint"  : "humidity",
    "scale"      : 0.1,
    "offset"     : 0
  }
]
}

```

If instead we had multiple assets that contain *temperature* and *humidity* we can accomplish the same transformation on all these assets, whilst not affecting any other assets, by changing the *asset* regular expression to something that matches more asset names;

```

{
  "factors" : [
    {
      "asset"      : ".*",
      "datapoint"  : "temperature",
      "scale"      : 1.8,
      "offset"     : 32
    },
    {
      "asset"      : ".*",
      "datapoint"  : "humidity",
      "scale"      : 0.1,
      "offset"     : 0
    }
  ]
}

```

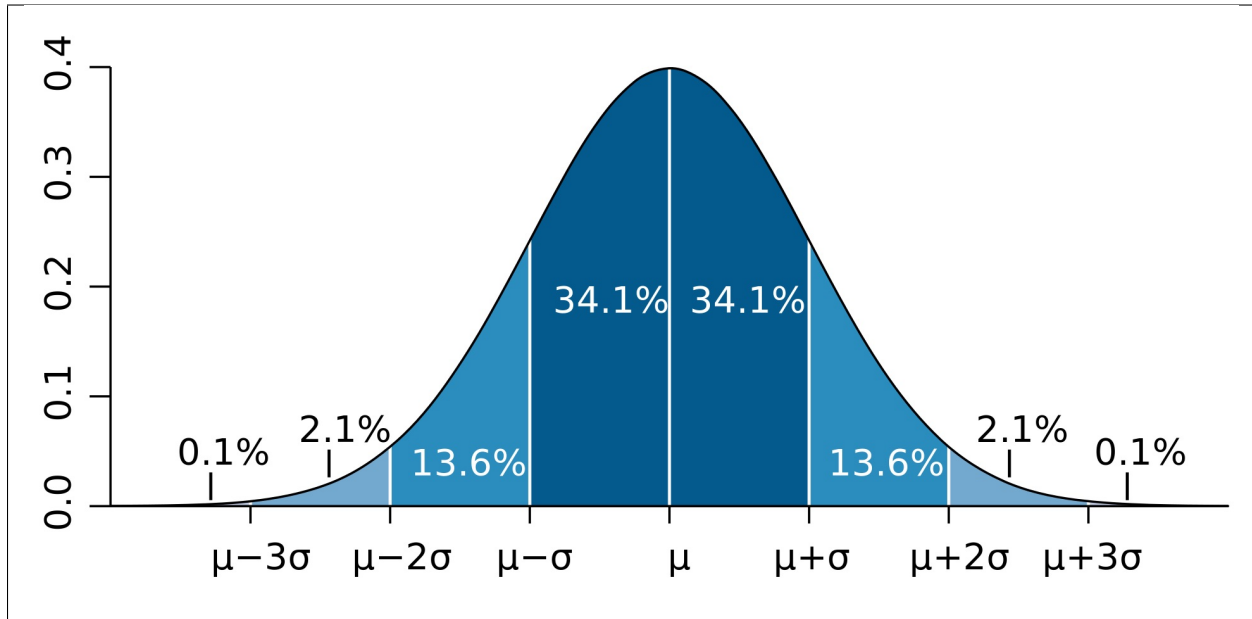
14.3.23 Sigma Data Cleansing Filter

The *flir-filter-sigmacleanse* filter is designed to cleanse data in a stream by removing outliers from the data stream. The method used to remove these outliers is to build an average and standard deviation for the data over time and remove any data that differs by more than a certain factor of the standard deviation from that average.

The plugin is designed to be used in situations when a sensor or item of equipment produces occasional anomalous results, these will be removed from the data passed onward within the system to provide a cleaner data stream. Care should be taken however that these values that are removed do represent sensor anomalies and are not the result of problems with the condition that is being monitored. If a sensor produces a high percentage of anomalous results then it should be considered for replacement.

In order to monitor the anomalous rates the plugin can optionally produce an hourly statistics report that will show the number of readings that have been forwarded as good and the number that have been discarded.

The method used to determine if a value is anomalous is based on the premise that data from a given sensor will follow a normal distribution from the mean value that is sampled over time. The probability of a value being valid reduces as the value differs more greatly from the mean value. This gives rise to the classical bell shaped distribution of values as shown below.



It can be seen from the diagram above how the probability drops as the values moves away from the mean, the sigma values here are the standard deviations observed for good data samples. Outlier values that are discarded do not contribute to the calculation of the standard deviation.

To add a sigma cleansing filter to your service:

- Click on the Applications add icon for your service or task.
- Select the *sigmacleanse* plugin from the list of available plugins.
- Name your cleansing filter.
- Click *Next* and you will be presented with the following configuration page

1 Plugin Name 2 Review Configuration

Sample Size (hours) 1

Sigma 3

Statistics Asset

Enabled ☐

Previous Done

- Configure your sigma cleanse filter
 - **Sample Size:** The number of hours over which an initial mean and standard deviation is built before any cleansing commences

- **Sigma:** The factor to apply to the standard deviation, the default is 3. Any value that differs from the mean by more than $3 * \text{sigma}$ will be removed.
- **Statistics Asset:** If this is not empty a statistics asset will be added every hour that details the number of readings that have been forwarded by the filter and the number removed. The name is that asset matches the value added here.
- Enable your filter and click *Done*

14.3.24 Simple Python Filter

The *flir-filter-simple-python* plugin allows very simple Python code to be used as a filter. A user may effectively write expressions in Python and have them execute in a filter.

The data is available within your Python code as a variable, named *reading* for each asset. You may access each data point within the asset by indexing the reading with the data point name. For example if your asset has two data points, *voltage* and *current*, then you would access these two values as

```
voltage = reading[b'voltage']
current = reading[b'current']
```

Using this type of filter it is possible to modify values of data points within an asset, remove data points in an asset or add new data points to an asset. It is **not** possible to remove assets or add new assets. The filter uses a Python 3 run time environment, therefore Python 3 syntax should be used.

The following examples show how to filter the readings data,

- Change datapoint value

```
reading[b'point_1'] = reading[b'point_1'] * 2 + 15
```

- Create a new datapoint while filtering

```
reading[b'temp_fahr'] = reading[b'temperature'] * 9 / 5 + 32
```

- Generate an exponential moving average (ema)

In this case we need to parse some data while filtering current data set the filter receives in input. A global 'user_data' empty dictionary is available to the Python interpreter and key values can be easily added. This illustrates the ability to maintain state within your filter.

```
global user_data
if not user_data:
    user_data['latest'] = None
for attribute in list(reading):
    if not user_data['latest']:
        user_data['latest'] = reading[attribute]
    user_data['latest'] = reading[attribute] * 0.07 + user_data['latest'] *
    ↪ (1 - 0.07)
    reading[b'ema'] = user_data['latest']
```

Simple Python filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *simple-python* plugin from the list of available plugins.
- Name your python filter.
- Click *Next* and you will be presented with the following configuration page

Sine South Service

1 Plugin Name 2 Review Configuration

Python code

```
1 reading[b'sinusoid'] = reading[b'sinusoid'] * 2 + 15
```

Enabled ☒

Previous Done

- Configure your simple Python filter
 - **Python Code:** Enter the code required for your filter.
- Enable your filter and click *Done*

14.3.25 Statistics Filter

The *flir-filter-statistics* filter is designed to accept data from one or more asset and produces statistics over specified time intervals, for example produce the mean, standard deviation and variance for 100 milliseconds samples of the data. The statistics that can be produced are;

- mean - the average of all the values in the time period calculated by adding up all the values and dividing by the number of values.
- mode - the number that appears most often in the time period.
- median - the median is found by sorting all the values in the time period and then choosing the middle number in this sorted set
- minimum - the minimum value that appears within the time period
- maximum - the maximum value that appears within the time period
- standard deviation - the standard deviation measures the spread of the numbers above and below the mean value
- variance - the variance is the average of the squared differences from the mean value calculated over the time period

Statistics filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *statistics* plugin from the list of available plugins.
- Name your statistics filter.
- Click *Next* and you will be presented with the following configuration page

Sine South Service

1 Plugin Name 2 Review Configuration

Sample Size (mS) 100

Mean ☒

Mode ☒

Median ☒

Minimum ☒

Maximum ☒

Standard Deviation ☒

Variance ☒

Enabled ☒

Previous Done

- Configure your statistics filter
 - **Mean:** A toggle that controls inclusion of the mean value
 - **Mode:** A toggle that controls inclusion of the mode value
 - **Median:** A toggle that controls inclusion of the median value
 - **Minimum:** A toggle that controls inclusion of the minimum value
 - **Maximum:** A toggle that controls inclusion of the maximum value
 - **Standard Deviation:** A toggle that controls inclusion of the standard deviation value
 - **Variance:** A toggle that controls inclusion of the variance value
- Enable your filter and click *Done*

14.3.26 Threshold Filter

The *flir-filter-threshold* plugin is a filter that is used to control the forwarding of data within FLIR Bridge. Its use is to only allow data to be stored or forwarded if a condition about that data is true. This can save storage or network bandwidth by eliminating data that is of no interest.

The filter uses an expression, that is entered by the user, to evaluate if data should be forwarded, if that expression evaluates to true then the data is forwarded, in the case of a south service this would be to the FLIR Bridge storage. In the case of a north task this would be to the upstream system.

Note: If the threshold filter is part of a chain of filters and the data is not forwarded by the threshold filter, i.e. the expression evaluates to false, then the following filters will not receive the data.

If an asset in the case of a south service, or data stream in the case of a north task, has other data points or assets that are not part of the expression, then they too are subject to the threshold. If the expression evaluates to false then no assets will be forwarded on that stream. This allows a single value to control the forwarding of data.

Another example use might be to have two north streams, one that uses a high cost, link to send data when some condition that requires close monitoring occurs and the other that is used to send data by a lower cost mechanism when normal operating conditions apply.

E.g. We have a temperature critical process, when the temperature is above 80 degrees it must be closely monitored. We use a high cost link to send data north wards in this case. We would have a north task setup that has the threshold filter with the condition:

```
temperature >= 80
```

We then have a second, lower cost link with a north task using the threshold filter with the condition:

```
temperature < 80
```

This way all data is sent once, but data is sent in an expedited fashion if the temperature is above the 80 degree threshold.

Threshold filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *threshold* plugin from the list of available plugins.
- Name your threshold filter.
- Click *Next* and you will be presented with the following configuration page

1 Plugin Name 2 Review Configuration

Expression speed > 10

Enabled ☒

Previous Done

- Enter the expression to control forwarding in the box labeled *Expression*
- Enable the filter and click on *Done* to activate it

Expressions

The *flir-filter-threshold* plugin makes use of the library to do run time expression evaluation. This library provides a rich mathematical operator set, the most useful of these in the context of this plugin are;

- Comparison operators (=, ==, <>, !=, <, <=, >, >=)
- Logical operators (and, nand, nor, not, or, xor, xnor, mand, mor)
- Mathematical operators (+, -, *, /, %, ^)
- Functions (min, max, avg, sum, abs, ceil, floor, round, roundn, exp, log, log10, logn, pow, root, sqrt, clamp, inrange, swap)
- Trigonometry (sin, cos, tan, acos, asin, atan, atan2, cosh, cot, csc, sec, sinh, tanh, d2r, r2d, d2g, g2d, hyp)

Within the expression the data points of the asset become symbols that may be used; therefore if an asset contains values “voltage” and “current” the expression will contain those as symbols and an expression of the form

```
voltage * current > 1000
```

can be used to determine if power (voltage * current) is greater than 1kW.

14.3.27 Vibration Features Filter

The *flir-filter-vibration_features* filter collects readings for configured observation interval, then calculates statistics on these readings and puts these statistics into a new reading.

- mean - the average of all the values in the time period calculated by adding up all the values and dividing by the number of values.
- median - the median is found by sorting all the values in the time period and then choosing the middle number in this sorted set
- standard deviation - the standard deviation measures the spread of the numbers above and below the mean value
- variance - the variance is the average of the squared differences from the mean value calculated over the time period
- RMS - the root mean squared of the waveform
- kurtosis - is a measure of the combined sizes of the two tails. It measures the amount of probability in the tails.

Vibration feature filters are added in the same way as any other filters.

- Click on the Applications add icon for your service or task.
- Select the *vibration_features* plugin from the list of available plugins.
- Name your vibration feature filter.
- Click *Next* and you will be presented with the following configuration page

Sine South Service

1 Plugin Name 2 Review Configuration

Asset name VibrationFeature

Observation interval (ms) 10

Enabled ☒

Previous Done

- Configure your vibration filter
 - **Asset name:** The name of the asset to create. This is the asset that will hold the vibration feature data.
 - **Observation interval (ms):** The interval over which the statistics are compiled.
- Enable your filter and click *Done*

14.4 FLIR Bridge Notification Rule Plugins

14.4.1 Threshold Rule

The threshold rule is used to detect the value of a data point within an asset going above or below a set threshold.

The configuration of the rule allows the threshold value to be set, the operation and the datapoint used to trigger the rule.

- **Asset name:** The name of the asset that is tested by the rule.
- **Datapoint Name:** The name of the datapoint in the asset used for the test.
- **Condition:** The condition that is being tested, this may be one of $>$, $>=$, $<=$ or $<$.
- **Trigger value:** The value used for the test.
- **Evaluation data:** Select if the data evaluate is a single value or a window of values.
- **Window evaluation:** Only valid if evaluation data is set to Window. This determines if the value used in the rule evaluation is the average, minimum or maximum over the duration of the window.
- **Time window:** Only valid if evaluation data is set to Window. This determines the time span of the window.

14.4.2 Moving Average Rule

The *flir-rule-average* plugin is a notification rule that is used to detect when a value moves outside of the determined average by more than a specified percentage. The plugin only monitors a single asset, but will monitor all data points within that asset. It will trigger if any of the data points within the asset differ by more than the configured percentage, an average is maintained for each data point separately.

During the configuration of a notification use the screen presented to choose the average plugin as the rule.

1 Notification Instance 2 Rule 3 Delivery Channel 4 Done

Rule Plugin

- Average (selected)
- OutOfBound
- SimpleExpression
- Threshold

Trigger if the current value deviates from the moving average by more than a defined percentage

[available plugins](#)

Previous Next

The next screen you are presented with provides the configuration options for the rule.

1 Notification Instance 2 Rule 3 Delivery Channel 4 Done

Asset temperature

Deviation % 10

Direction Both ▼

Average Simple Moving Average ▼

EMA Factor 10

Previous Next

The *Asset* entry field is used to define the single asset that the plugin should monitor.

The *Deviation %* defines how far away from the observed average the current value should be in order to be considered as triggering the rule.

Deviation %	<div> <div>Above Average</div> <div>Below Average</div> <div>✓ Both</div> </div>
Direction	

The *Direction* entry is used to define if the rule should trigger when the current value is above average, below average or in both cases.

Average	<div> <div>✓ Simple Moving Average</div> <div>Exponential Moving Average</div> </div>
EMA Factor	
	10

The *Average* entry is used to determine what type of average is used for the calculation. The average calculated may be either a simple moving average or an exponential moving average. If an exponential moving average is chosen then a second configuration parameter, *EMA Factor*, allows the setting of the factor used to calculate that average.

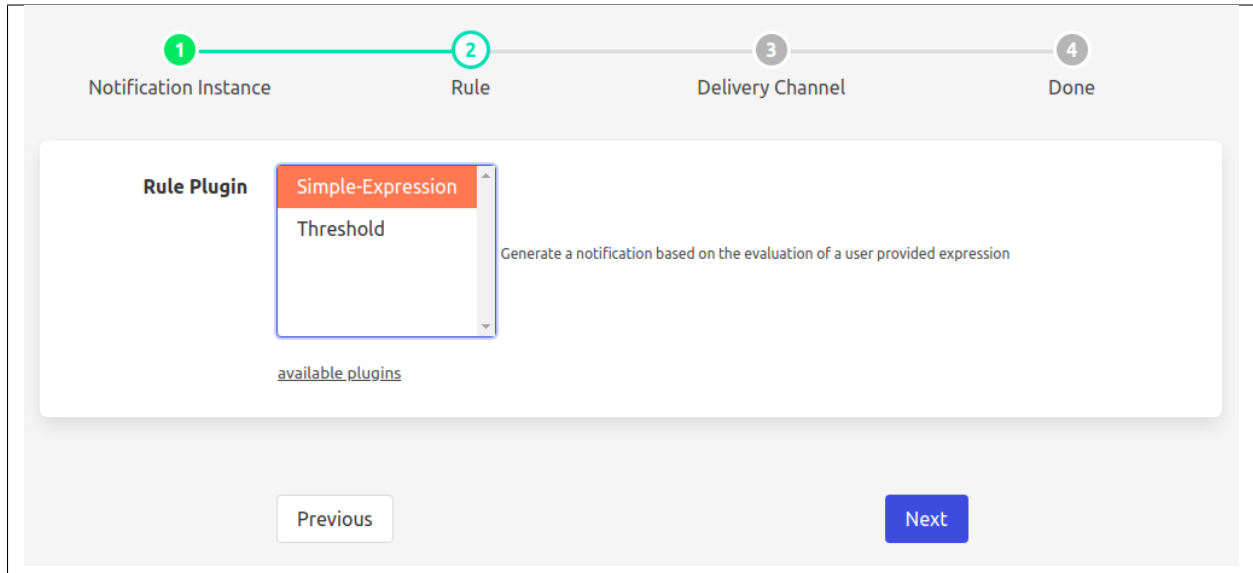
Exponential moving averages give more weight to the recent values compared to historical values. The smaller the EMA factor the more weight recent values carry. A value of 1 for *EMA Factor* will only consider the most recent value.

Note: The Average rule is not applicable to all data, only simple numeric values are considered and those values should not deviate with an average of 0 or close to 0 if good results are required. Data points that deviate wildly are also not suitable for this plugin.

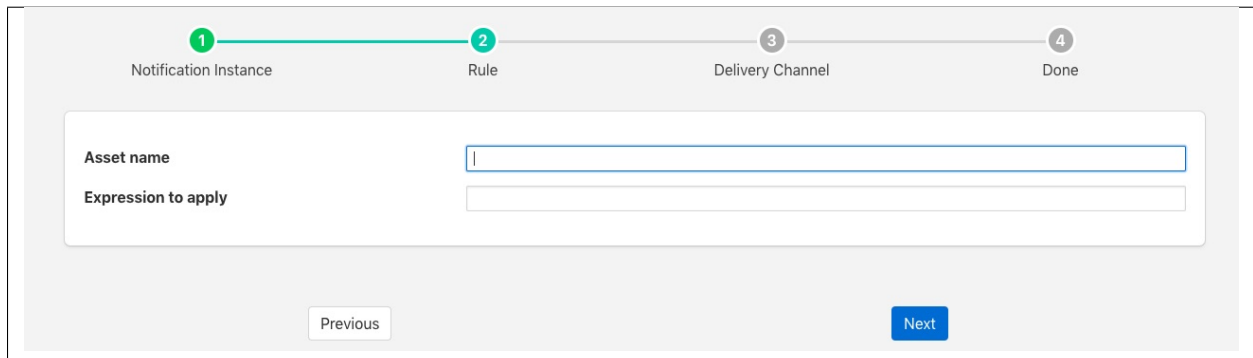
14.4.3 Expression Rule

The *flir-rule-simple-expression* is a notification rule plugin that evaluates a user defined function to determine if a notification has triggered or not. The rule will work with a single asset, but does allow access to all the data points within the asset.

During the configuration of a notification use the screen presented to choose the average plugin as the rule.



The next screen you are presented with provides the configuration options for the rule.



The *Asset* entry field is used to define the single asset that the plugin should monitor.

The *Expression to apply* defines the expression that will be evaluated each time the rule is checked. This should be a boolean expression that returns true when the rule is considered to have triggered. Each data point within the asset will become a symbol in the expression, therefore if your asset contains a data point called voltage, the symbol voltage can be used in the expression to obtain the current voltage reading. As an example to create an under voltage notification if the voltage falls below 48 volts, the expression to use would be;

```
voltage < 48
```

The trigger expression uses the same expression mechanism, as the , and plugins

Expression may contain any of the following. . .

- Mathematical operators (+, -, *, /, %, ^)
- Functions (min, max, avg, sum, abs, ceil, floor, round, roundn, exp, log, log10, logn, pow, root, sqrt, clamp, inrange, swap)
- Trigonometry (sin, cos, tan, acos, asin, atan, atan2, cosh, cot, csc, sec, sinh, tanh, d2r, r2d, d2g, g2d, hyp)
- Equalities & Inequalities (=, ==, <>, !=, <, <=, >, >=)

- Logical operators (and, nand, nor, not, or, xor, xnor, mand, mor)

14.4.4 Simple-Sigma Rule

The *flir-rule-simple-sigma* is a notification rule plugin that uses the principle of normal distribution to trigger a notification if a value is found that is outside of the normal distribution. The normal distribution is discovered by taking the mean of all the values over time and calculating the standard deviation, or sigma, from that mean. Until the rule has built up a reasonable sample of data on which to calculate the mean and standard deviation the rule will not trigger. This reasonable sample is defined as a time period, in hours, for which the rule will sample the data to determine the mean and sigma values. The default time period for this is 1 hour, however it may be overridden.

Once a mean and standard deviation have been determined the rule will move into a mode in which it will trigger. Whilst in triggering mode the rule will still refine the mean and standard deviation values. If a value is found in trigger mode that is more than a certain number of standard deviations from the mean, then the rule will trigger. The number of standard deviations is the sigma factor and defaults to 3.0, however the user can configure this to be more or less than 3.0.

To use the Simple-Sigma plugin create your notification rule as normal, when selecting the rule to use select the *Simple-Sigma* rule and click on next. You will be presented with a dialog as below

The screenshot shows a configuration dialog for the Simple-Sigma rule. The dialog is part of a four-step process: 1. Notification Instance, 2. Rule (current step), 3. Delivery Channel, and 4. Done. The configuration fields are as follows:

Field	Value
Asset name	Current
SigmaRule Factor	3.0
Sample Size (hours)	1

Navigation buttons at the bottom include 'Previous' and 'Next'.

Configure the Simple-Sigma rule

- **Asset name:** The asset name to monitor with the rule
- **Sigma Factor:** The factor to use for determining range, a factor of 3.0 will trigger when a value is more the 3.0 * Sigma from the current mean
- **Sample Size:** The number of hours to build a mean and standard deviation before the rule will trigger.

Click on *Next* and complete the configuration of your notification.

14.5 FLIR Bridge Notification Delivery Plugins

14.5.1 Amazon Alexa Notification

The *flir-notify-alexa* notification delivery plugin sends notifications via Amazon Alexa devices using the Alexa *NotifyMe* skill.

When you receive a notification Alexa will make a noise to say you have a new notification and the green light on your Alexa device will light to say you have waiting notifications. To hear your notifications simply say “Alexa, read my notifications”

To enable notifications on an Alexa device

- You must enable the NotifyMe skill on your Amazon Alexa device.
- Link this skill to your Amazon account
- NotifyMe will send you an access code that is required to configure this plugin.

Once you have created your notification rule and move on to the delivery mechanism

- Select the alexa plugin from the list of plugins
- Click *Next*

The screenshot shows a configuration window for the Alexa plugin. The window has a title bar and a main content area. At the top, there is a progress indicator with four steps: 1. Notification Instance, 2. Rule, 3. Delivery Channel (highlighted), and 4. Done. Below the progress indicator, there is a form with three fields: 'Access Code' (empty), 'Title' (containing the text 'The level in tank 15 is below 10%'), and 'Enabled' (checked). At the bottom of the form, there are two buttons: 'Previous' and 'Next'.

- Configure the plugin
 - **Access Code:** Paste the access code you received from the *NotifyMe* application here
 - **Title:** This is the title that the Alexa device will read to you
- Enable the plugin and click *Next*
- Complete your notification setup

When you notification triggers the Alexa device will read the title text to you followed by either “Notification has triggered” or “Notification has cleared”.

14.5.2 Asset Notification

The *flir-notify-asset* notification delivery plugin is unusual in that it does not notify an external system, instead it creates a new asset which is then processed like any other asset within FLIR Bridge. This plugin is useful to inform upstream systems that an event has occurred and allow them to take action or merely as a way to have a record of a condition occurring which may not require any further actions.

Once you have created your notification rule and move on to the delivery mechanism

- Select the asset plugin from the list of plugins
- Click *Next*

The screenshot shows a four-step configuration process: 1. Notification Instance, 2. Rule, 3. Delivery Channel, and 4. Done. Step 2, 'Rule', is currently active. Below the progress bar is a form with three fields: 'Asset' with the value 'event', 'Description' with the value 'Notification alert', and 'Enabled' which is an unchecked checkbox. At the bottom of the form are 'Previous' and 'Next' buttons.

- Now configure the asset delivery plugin
 - **Asset:** The name of the asset to create.
 - **Description:** A textual description to add to the asset
- Enable the plugin and click *Next*
- Complete your notification setup

The asset that will be created when the notification triggers will contain

- The timestamp of the trigger event
- Three data points
 - **rule:** The name of the notification that triggered this asset creation
 - **description:** The textual description entered in the configuration of the delivery plugin
 - **event:** This will be one of *triggered* or *cleared*. If the notification type was not set to be *toggled* then the *cleared* event will not appear. If *toggled* was set as the notification type then there will be a *triggered* value in the asset created when the rule triggered and a *cleared* value in the asset generated when the rule moved from the triggered to untriggered state.

14.5.3 Configuration Update

The *flir-notify-config* plugin is designed to allow a notification to alter the configuration of one of the configuration items within the local FLIR Bridge.

The plugin can be used to trigger changes to the way data is collected, for example by altering the *readingsPerSec* item in a south server Advanced category. It is not limited to this however and could equally be used to effect some configuration of a filter, for example to change a scale factor or threshold. It may also change configuration of notification rule or delivery plugins.

Once you have created your notification rule and moved on to the delivery mechanism

- Select the *config* plugin from the list of plugins
- Click *Next*

The screenshot shows a configuration wizard for a notification instance. The progress bar at the top indicates the current step is 'Delivery Channel' (Step 3). The form fields are as follows:

Field	Value
Category	SineAdvanced
Item	readingsPerSecond
Trigger Value	20
Cleared Value	1
Enabled	<input checked="" type="checkbox"/>

Navigation buttons at the bottom: 'Previous' and 'Next'.

- Configure the delivery plugin
 - **Category:** The name of the configuration category to be updated.
 - **Item:** The name of the item within the configuration category to be updated.
 - **Trigger Value:** The value to set the item to when an notification is triggered.
 - **Clear Value:** The value to set the item to when the notification is cleared. Note you must set the notification type to *toggled* if you wish to use a *Clear Value*.
- Enable the plugin and click *Next*
- Complete your notification setup

14.5.4 Email Notifications

The *flir-notify-email* delivery notification plugin allows notifications to be delivered as email messages. The plugin uses an SMTP server to send email and requires access to this to be configured as part of configuring the notification delivery method.

During the creation of your notification select the email notification plugin from the list of available notification mechanisms. You will be prompted with a configuration dialog in which to enters details of your SMTP server and of the email you wish to send.

The screenshot shows a configuration window for a notification rule. At the top, a progress bar indicates four steps: 1. Notification Instance, 2. Rule (current), 3. Delivery Channel, and 4. Done. The 'Rule' step contains a form with the following fields:

To address	alert.subscriber@dianomic.com
To	Notification alert subscriber
Subject	Fledge alert notification
From address	dianomic.alerts@gmail.com
From name	Notification alert
SMTP Server	smtp.gmail.com
SMTP Port	587
SSL/TLS	<input checked="" type="checkbox"/>
Username	dianomic.alerts@gmail.com
Password	pass
Enabled	<input type="checkbox"/>

At the bottom of the form are two buttons: 'Previous' and 'Next'.

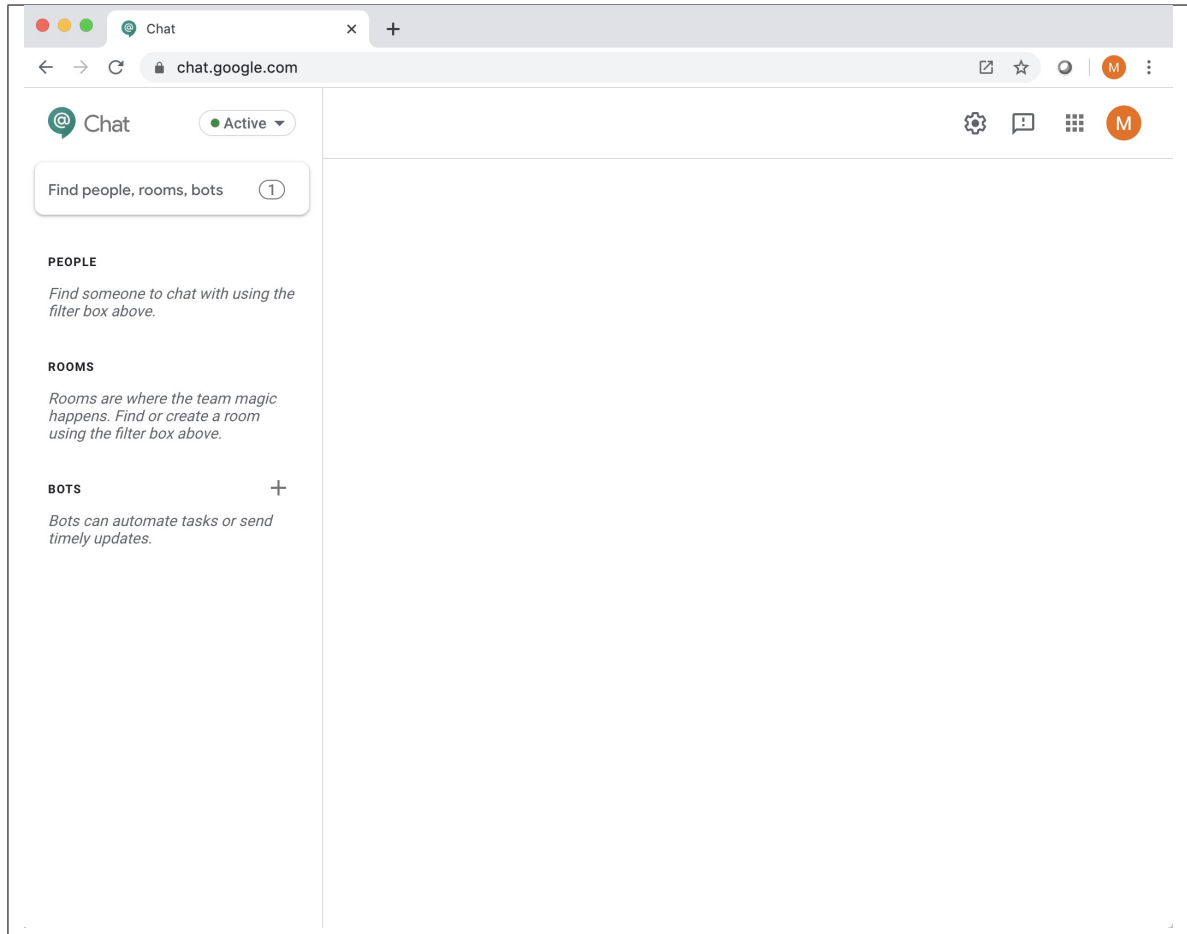
- **To address:** The email address to which the notification will be sent
- **To:** A textual name for the recipient of the email
- **Subject:** A Subject to put in the email message
- **From address:** A from address to use for the email message
- **From name:** A from name to include in the email
- **SMTP Server:** The address of the SMTP server to which to send messages
- **SMTP Port:** The port of your SMTP server
- **SSL/TLS:** A toggle to control if SSL/TLS encryption should be used when communicating with the SMTP server
- **Username:** A username to use to authenticate with the SMTP server
- **Password:** A password to use to authenticate with the SMTP server.

14.5.5 Google Chat

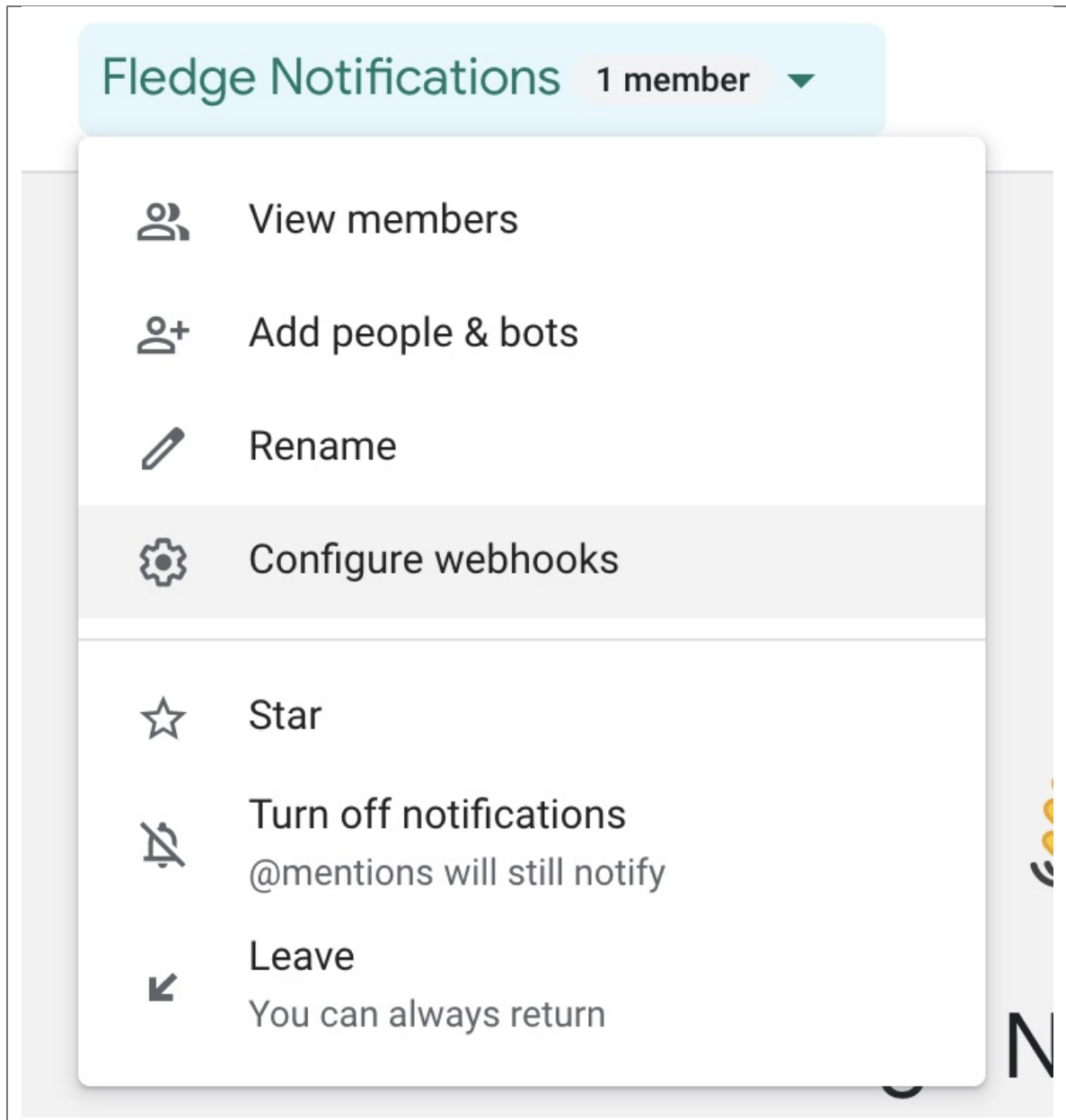
The *flir-notify-google-hangouts* plugin allows notifications to be delivered to the Google chat platform. The notification are delivered into a specific chat room within the application, in order to allow access to the chat room you must create a webhook for sending data to that chatroom.

To create a webhook

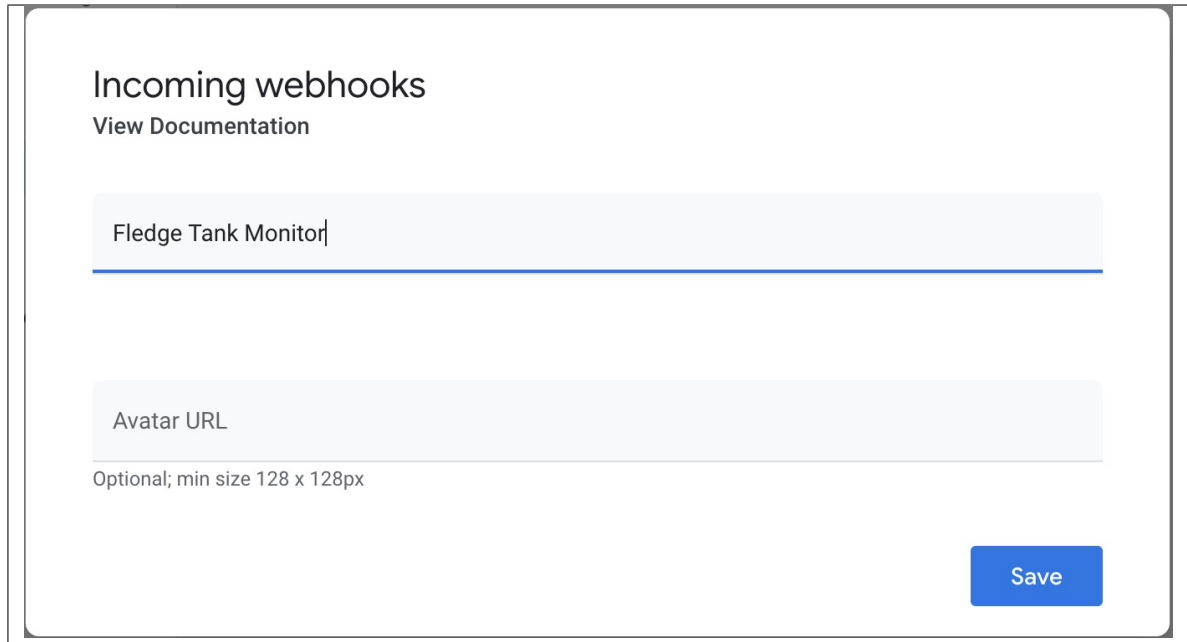
- Go to the page in your browser



- Select the chat room you wish to use or create a new chat room
- In the menu at the top of the screen select *Configure webhooks*



- Enter a name for your webhook and optional avatar and click *Save*



Incoming webhooks

[View Documentation](#)

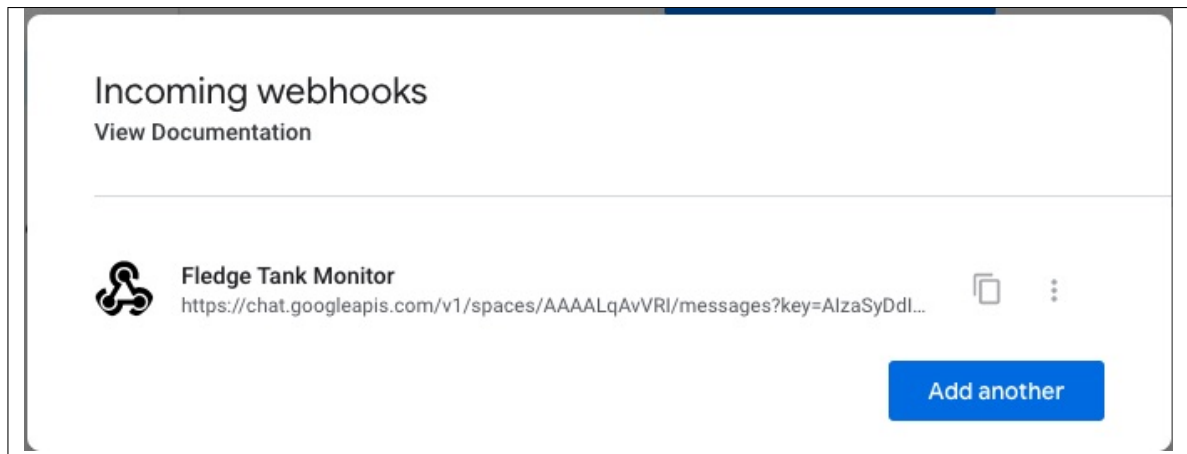
Fledge Tank Monitor

Avatar URL

Optional; min size 128 x 128px


Save

- Copy the URL that appears under your webhook name, you can use the copy icon next to the URL to place it in the clipboard



Incoming webhooks

[View Documentation](#)

 **Fledge Tank Monitor**
https://chat.googleapis.com/v1/spaces/AAAAALqAvVRI/messages?key=AlzaSyDdl...

Add another

- Close the webhooks window by clicking outside the window

Once you have created your notification rule and move on to the delivery mechanism

- Select the Hangouts plugin from the list of plugins
- Click *Next*

1 Notification Instance 2 Rule 3 Delivery Channel 4 Done

Google Hangout Webhook URL

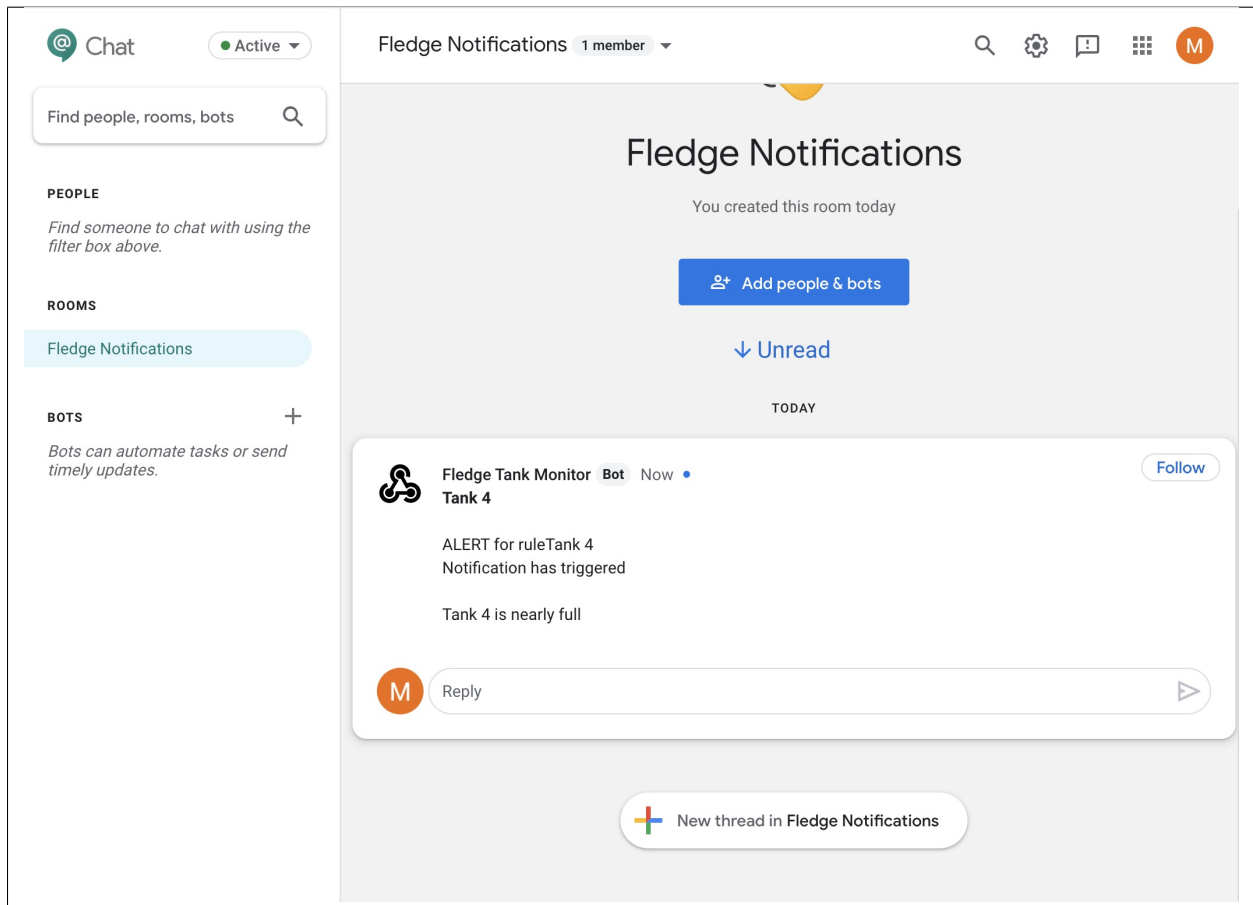
Message Text

Enabled ☒

[Previous](#) [Next](#)

- Now configure the asset delivery plugin
 - **Google Hangout Webhook URL:** Paste the URL obtain above here
 - **Message Text:** Enter the message text you wish to send
- Enable the plugin and click *Next*
- Complete your notification setup

A message will be sent to this chat room whenever a notification is triggered.

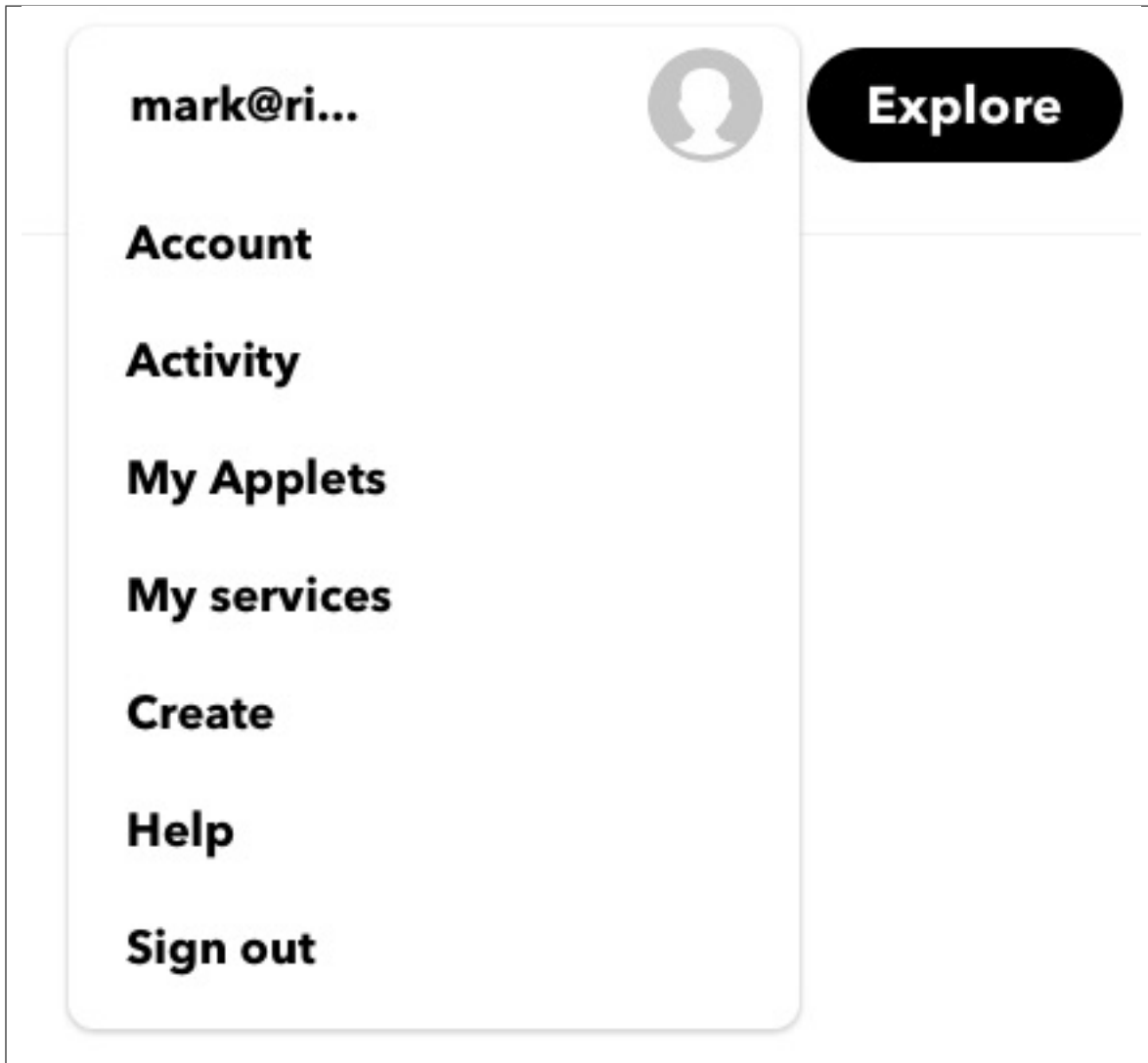


14.5.6 IFTTT Delivery Plugin

The *flir-notify-ifttt* is a notification delivery plugin designed to trigger an action on the *If This Than That* IoT platform. IFTTT allows the user to setup a webhook that can be used to trigger processing on the platform. The webhook could be sending an IFTTT notification to a destination not support by any FLIR Bridge plugin to controlling a device that is controllable via IFTTT.

In order to use the IFTTT webhook you must obtain a key from IFTTT by visiting your IFTTT account

- Select the “My Applets” page from your account pull down menu



- Select “New Applet”
- Click on the blue “+ this” logo
- Choose the service Webhooks
- Click on the blue box “Receive a web request”
- Enter an “Event Name”, this may be of your choosing and will be put in the configuration entry ‘Trigger’ for the FLIR Bridge plugin

- Click on the “+ that” logo
- Select the action you wish to invoke

Once you have setup your webhook on IFTTT you can now proceed to setup the FLIR Bridge delivery notification plugin. Create you notification, choose and configure your notification rule. Select the IFTTT delivery plugin and click on *Next*. You will be presented with the IFTTT plugin configuration page.

The screenshot shows the IFTTT plugin configuration interface. A progress bar at the top indicates the current step is 3, 'Delivery Channel'. The configuration form contains the following fields:

- IFTTT Trigger:** A text input field containing the value 'button_press'.
- IFTTT Key:** A text input field containing a long string of 'X' characters.
- Enabled:** A checkbox that is checked.

At the bottom of the form, there are two buttons: 'Previous' and 'Next'.

There are two important items to be configured

- **IFTTT Trigger:** This is the *Maker Event* that you used in IFTTT when defining the action that the webhook should trigger.
- **IFTTT Key:** This is the webhook key you obtain from the IFTTT platform.

Enable the delivery and click on *Next* to move to the final stage of completing your notification.

14.5.7 Jira Ticket Creation

The *flir-notify-jira* delivery notification plugin allows notifications to be used to create tickets within Jira. The tickets are created within a specified project with a summary, description and other information supplied by FLIR Bridge.

To obtain an API token from Jira

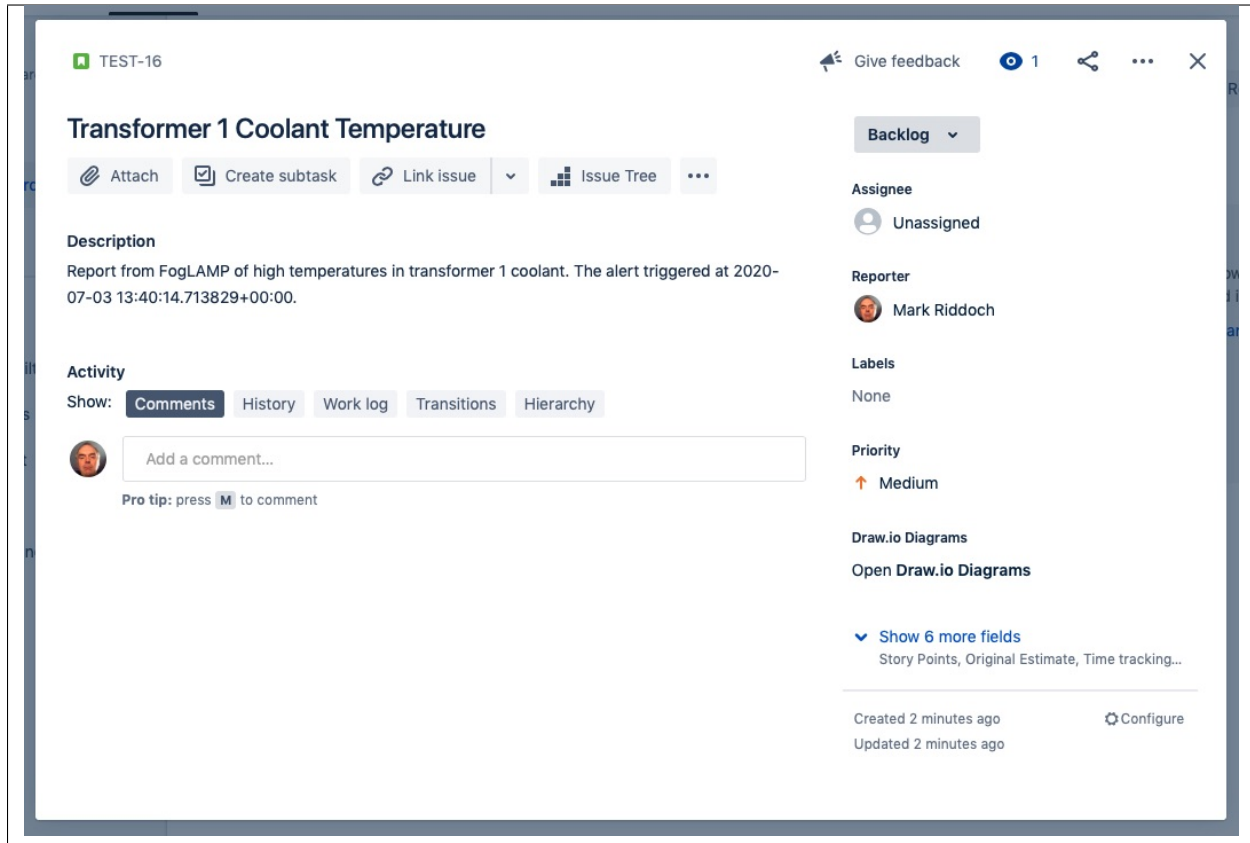
- Visit the page
- Select *Create API token*
- Enter a name for your application, this must be unique for each FLIR Bridge Jira application you create
- Click on Create

Once you have created your notification rule and move on to the delivery mechanism

- Select the *jira* plugin from the list of plugins
- Click *Next*

- Configure the delivery plugin
 - **Hostname:** The hostname where your Jira instance is installed. This may be a local instance or a cloud instance.
 - **Project:** The project into which you are creating the Jira tickets. The project name should be the one that appears as projectKey in the URL bar when browsing the Jira boards.
 - **User:** Your Jira user name, this is the name of the account you used to create the API token
 - **API Token:** The API token you created above
 - **Summary:** The text to add into the ticket summary, this may include text substitutions (see below).
 - **Description:** The text to add into the ticket description, this may include text substitution (see below).
 - **Type:** The issue type to create. This must be the name of one of the types that is valid for your Jira project.
 - **Additional Fields:** This is a JSON document that contains a number of key/value pairs, each of these pairs is a field name and content to add to the ticket. Text substitutions may be applied here also.
- Enable the plugin and click *Next*
- Complete your notification setup

When the notification rule triggers you a Jira ticket will be created.



Text Substitution

Text markers may be used to substitution text with the fields in the Jira ticket. The markers supported are

- **%MESSAGE%**: this is replaced with the message generated in the notification system
- **%REASON%**: this is replaced with the reason for the notification, it may be the string *triggered* or *cleared*.
- **%TIMESTAMP%**: this is replaced with the timestamp of the reading data that caused the notification to trigger.

14.5.8 JSON Configuration Update

The *flir-notify-jsonconfig* plugin is designed to allow a notification to alter the configuration of one of the JSON configuration items within the local FLIR Bridge.

The plugin can be used to trigger changes to the way data is collected by altering individual items within a complex JSON configuration items. The delivery plugin allows you to set a value when the notification is raised and a different value when it is cleared.

Once you have created your notification rule and moved on to the delivery mechanism

- Select the *config* plugin from the list of plugins
- Click *Next*

- Configure the delivery plugin
 - **Category:** The name of the configuration category to be updated.
 - **Item:** The name of the item within the configuration category to be updated.
 - **JSON Path:** The JSON path of the object that contains the item to be modified.
 - **Property:** The name of the JSON property to modify.
 - **Trigger Value:** The value to set the item to when an notification is triggered.
 - **Clear Value:** The value to set the item to when the notification is cleared. Note you must set the notification type to *toggled* if you wish to use a *Clear Value*.
- Enable the plugin and click *Next*
- Complete your notification setup

JSON Path

A subset of the full JSON Path expressions are supported in this plugins. Each path element is proceeded by a / character and may be one of

- **Literals:** A literal object name within the JSON document. E.g. `/a/b/c`
- **An Array Index:** An absolute index within an array. E.g. `a[2]`
- **A conditional test:** A property value to match within an array or object. `a[prop==value]`

To match the object under the *registers* element within the *map* element an expression would be of the form

```
/map/registers
```

To match the first element in the array called *assets* under the *exclusions* object the expression would be

```
/exclusions/assets[0]
```

To match the object that contains a property called *id* whose values in *QTE123* within the *connections* object the expression would be

```
/connections[id=="QTE123"]
```

14.5.9 Management Poll Notification

The *flir-notify-management* notification delivery plugin is designed to trigger the FogMan agent microservice of the current FLIR Bridge to poll its FogMan to retrieve any configuration updates for this FLIR Bridge.

Once you have created your notification rule and move on to the delivery mechanism

- Select the management plugin from the list of plugins
- Click *Next*
- There is no specific configuration for this plugin
- Enable the plugin and click *Next*
- Complete your notification setup

Plugin Uses

The plugin is designed for an environment whereby the updates of configuration of the FLIR Bridge are coordinated with the state of the equipment that is being monitored by the FLIR Bridge. This might be because you may wish to prevent updates from occurring during critical periods of operation or maybe because the FLIR Bridge is monitoring the network connectivity and you wish to synchronize updates with network availability.

14.5.10 MQTT Notification

The *flir-notify-mqtt* notification delivery plugin sends notifications via an MQTT broker. The MQTT topic and the payloads to send when the notification triggers or is cleared are configurable.

Once you have created your notification rule and move on to the delivery mechanism

- Select the mqtt plugin from the list of plugins
- Click *Next*

- Configure the plugin
 - **MQTT Broker:** The URL of your MQTT broker.
 - **Topic:** The MQTT topic on which to publish the messages.
 - **Trigger Payload:** The payload to send when the notification triggers
 - **Clear Payload:** The payload to send when the notification clears
- Enable the plugin and click *Next*
- Complete your notification setup

14.5.11 Conditional Forwarding

The *flir-notify-north* plugin is designed to allow conditional forwarding of data to an existing north application from within FLIR Bridge.

The scenario the plugin addresses is the need to send data to a system north of FLIR Bridge when a condition occurs. The sending is done via a standard FLIR Bridge north task and can use any plugin such as OMF, GCP, InfluxDB, etc. The condition used to send this data is monitored using the notification server, when the rule in the notification triggers we send data from the FLIR Bridge storage service to the specified north task.

The data that is send is based on the time the notification triggered and two configuration parameters, pre-trigger and post-trigger times. The pre-trigger setting control how long before the event the data is sent and the post-trigger for how long after the event data is sent.

The data that is sent may be anything that is buffered in the FLIR Bridge storage service. A list of assets to send may be configured as part of the plugin configuration.

Once you have created your notification rule and move on to the delivery mechanism

- Select the North plugin from the list of plugins
- Click *Next*

The screenshot shows a configuration interface for a notification rule. At the top, a progress bar indicates four steps: 1. Notification Instance, 2. Rule (current step), 3. Delivery Channel, and 4. Done. The main configuration area includes:

- North task name:** A text input field.
- Assets to send:** A JSON editor showing a structure with an "assets" array. The current content is:


```
1 {
2   "assets": []
3 }
```
- Pre-trigger time:** A numeric input field with the value 5.
- Post-trigger time:** A numeric input field with the value 0.
- Block Size:** A numeric input field with the value 500.
- Enabled:** A checkbox that is currently unchecked.

At the bottom, there are "Previous" and "Next" buttons.

- Configure the delivery plugin
 - **North task name:** This is the name of a north task to use for the sending of the data. The north task should have already been created but should be disabled.
 - **Assets to send:** A JSON structure that contains the list of assets that should be sent via the north task. This list is a simple JSON array of asset names.
 - **Pre-trigger time:** The length of time in seconds before the notification triggers for which data should be sent.
 - **Post-trigger time:** The length of time in seconds after the notification triggers for which data should be sent.
 - **Block size:** The size of the data block sent to the north service, this is a tuning parameter to throttle the data sent, under most circumstances it may be left as the default.
- Enable the plugin and click *Next*
- Complete your notification setup

14.5.12 Operation Notification

The *flir-notify-operation* notification delivery plugin is a mechanism by which a notification can be used to send a request to a south services to perform an operation.

Once you have created your notification rule and move on to the delivery mechanism

- Select the operation plugin from the list of plugins
- Click *Next*

1 Notification Instance 2 Rule 3 Delivery Channel 4 Done

Service

Trigger Value

```
1 {
2   "values": {
3     "name": "value"
4   }
5 }
```

Cleared Value

```
1 {
2   "values": {
3     "name": "value"
4   }
5 }
```

Enabled ☐

- Configure the plugin
 - **Service:** The name of the south service you wish to control
 - **Trigger Value:** The operation payload to send to the south service when the rule triggers. This is the name of the operation to perform and a set of name, value pairs which are the optional parameters to pass that operations.
 - **Cleared Value:** The operation payload to send to the south service when the rule clears. This is the name of the operation to perform and a set of name, value pairs which are the optional parameters to pass that operations.
- Enable the plugin and click *Next*
- Complete your notification setup

14.5.13 Python 3 Script

The *flir-notify-python35* notification delivery plugin allows a user supplied Python script to be executed when a notification is triggered or cleared. The script should be written in Python 3 syntax.

A Python script should be provided in the form of a function, the name of that function should match the name of the file the code is loaded from. E.g if you have a script to run which you have saved in a file called `alert_light.py` it should contain a function `alert_light`. ~that function is called with a message which is defined in notification itself as a simple string.

A second function may be provided by the Python plugin code to accept configuration from the plugin that can be used to modify the behavior of the Python code without the need to change the code. The configuration is a JSON document which is again passed as a Python Dict to the `set_filter_config` function in the user provided Python code. This function should be of the form


```
def set_filter_config(configuration):
    config = json.loads(configuration['config'])
    value = config['key']
    ...
    return True
```

Once you have created your notification rule and move on to the delivery mechanism

- Select the python35 plugin from the list of plugins
- Click *Next*

The screenshot shows a four-step progress bar at the top: 1 Notification Instance, 2 Rule (active), 3 Delivery Channel, and 4 Done. The main content area is divided into three sections: 'Python script', 'Configuration', and 'Enabled'. The 'Python script' section contains a code editor with the following code:

```
1 from time import sleep
2 from envirophat import leds
3
4 def flash_leds(message):
5     for count in range(4):
6         leds.on()
7         sleep(0.5)
8         leds.off()
9         sleep(0.5)
10
```

Below the code editor is a file selection interface showing 'flash_leds.py' with a 'Choose Files' button. The 'Configuration' section contains a JSON editor with the following content:

```
1 {}
```

At the bottom left, there is an 'Enabled' checkbox which is checked. At the bottom right, there are 'Previous' and 'Next' buttons.

- Configure the plugin
 - **Python Script:** This is the script that will be executed. Initially you are unable to type in this area and must load your initial script from a file using the *Choose Files* button below the text area. Once a file has been chosen and loaded you are able to update the Python code in this page.

Note: Any changes made to the script in this screen will be written back to the original file it was loaded from.

- **Configuration:** You may enter a JSON document here that will be passed to the *set_filter_config* function of your Python code.
- Enable the plugin and click *Next*
- Complete your notification setup

Example Script

The following is an example script that flashes the LEDs on the Enviro pHAT board on a Raspberry Pi

```
from time import sleep
from envirophat import leds
def flash_leds(message):
    for count in range(4):
        leds.on()
        sleep(0.5)
        leds.off()
        sleep(0.5)
```

This code imports some Python libraries and then in a loop will turn the leds on and then off 4 times.

Note: This example will take 4 seconds to execute, unless multiple threads have been turned on for notification delivery this will block any other notifications from being delivered during that time.

14.5.14 Set Point Control Notification

The *flir-notify-setpoint* notification delivery plugin is a mechanism by which a notification can be used to send set point control writes into south services which support set point control

Once you have created your notification rule and move on to the delivery mechanism

- Select the setpoint plugin from the list of plugins
- Click *Next*

The screenshot displays a configuration window for a notification rule, titled 'Rule' (step 2 of 4). The window is divided into several sections:

- Service:** A text input field.
- Trigger Value:** A code editor showing a JSON snippet:

```
1 {
2   "values": {
3     "name": "value"
4   }
5 }
```
- Cleared Value:** A code editor showing a JSON snippet:

```
1 {
2   "values": {
3     "name": "value"
4   }
5 }
```
- Enabled:** A checkbox, currently unchecked.

At the top of the window, a progress bar shows four steps: 1. Notification Instance, 2. Rule (current), 3. Delivery Channel, and 4. Done.

- Configure the plugin
 - **Service:** The name of the south service you wish to control
 - **Trigger Value:** The set point control payload to send to the south service. This is a list of name, value pairs to be set within the service. These are set when the notification rule triggers.
 - **Cleared Value:** The set point control payload to send to the south service. This is a list of name, value pairs to be set within the service. These are set when the notification rule clears.
- Enable the plugin and click *Next*
- Complete your notification setup

Trigger Values

The *Trigger Value* and *Cleared Value* are JSON documents that are sent to the set point entry point of the south service. The format of these is a set of name and value pairs that represent the data to write via the south service. A simple example would be as below

```
{
  "values": {
    "temperature" : "11",
    "rate"       : "245"
  }
}
```

In this example we are setting two variables in the south service, one named *temperature* and the other named *rate*. In this example the values are constants defined in the plugin configuration. It is possible however to use values that are in the data that triggered the notification.

As an example of this assume we are controlling the speed of a fan based on the temperature of an item of equipment. We have a south service that is reading the temperature of the equipment, let's assume this is in an asset called *equipment* which has a data point called *temperature*. We add a filter using the *flir-filter-expression* filter to calculate a desired fan speed. The expression we will use in this example is $desiredSpeed = temperature * 100$. This will cause the asset to have a second data point called *desiredSpeed*.

We create a notification that is triggered if the *desiredSpeed* is greater than 0. The delivery mechanism will be this plugin, *flir-notify-setpoint*. We want to set two values in the south plugin *speed* to set the speed of the fan and *run* which controls if the fan is on or off. We set the *Trigger Value* to the following

```
{
  "values" : {
    "speed" : "$equipment.desiredSpeed$",
    "run"   : "1"
  }
}
```

In this case the *speed* value will be substituted by the value of the *desiredSpeed* data point of the *equipment* asset that triggered the notification to be sent.

14.5.15 Slack Messages

The *flir-notify-slack* delivery notification plugin allows notifications to be delivered as instant messages on the Slack messaging platform. The plugin uses a Slack webhook to post the message.

To obtain a webhook URL from Slack

- Visit the page
- Select *Create New App*
- Enter a name for your application, this must be unique for each FLIR Bridge slack application you create
- Select your Slack workspace in which to deliver your notification. If not already logged in you may need to login to your workspace
- Click on Create
- Select *Incoming Webhooks*
- Activate your webhook
- Add your webhook to the workspace
- Select the channel or individual to send the notification to
- Authorize your webhook
- Copy the Webhook URL which you will need when configuring the plugin

Once you have created your notification rule and move on to the delivery mechanism

- Select the slack plugin from the list of plugins
- Click *Next*

1 Notification Instance 2 Rule 3 Delivery Channel 4 Done

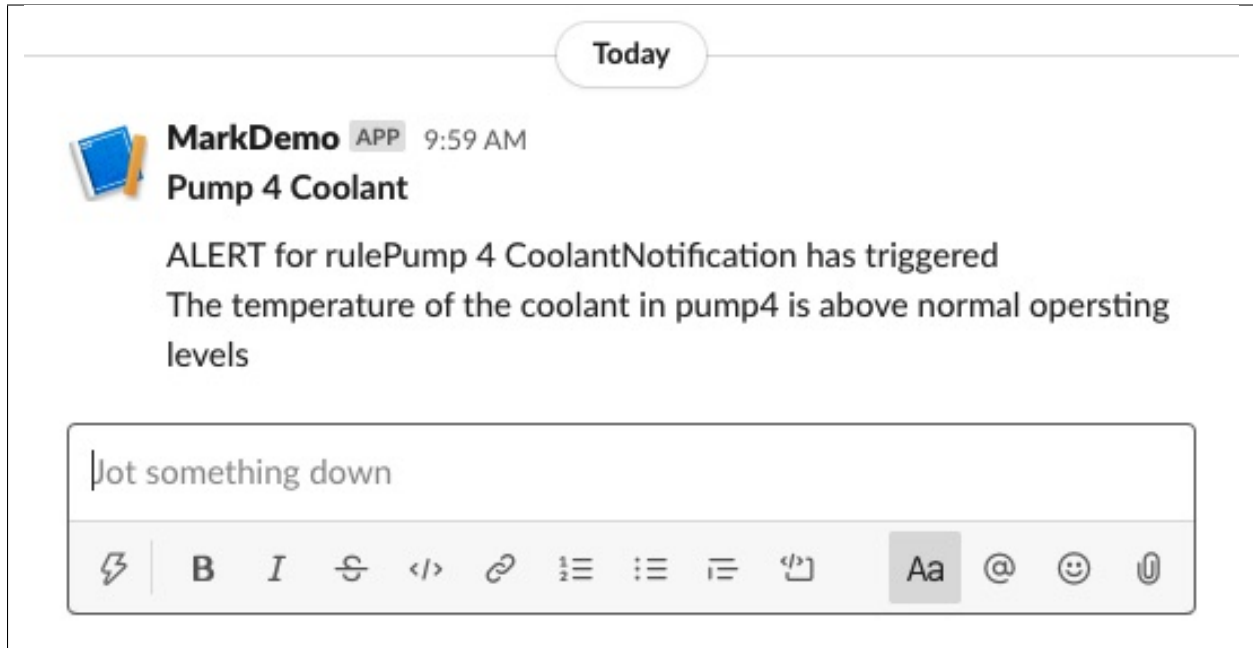
Slack Webhook URL:

Message Text:

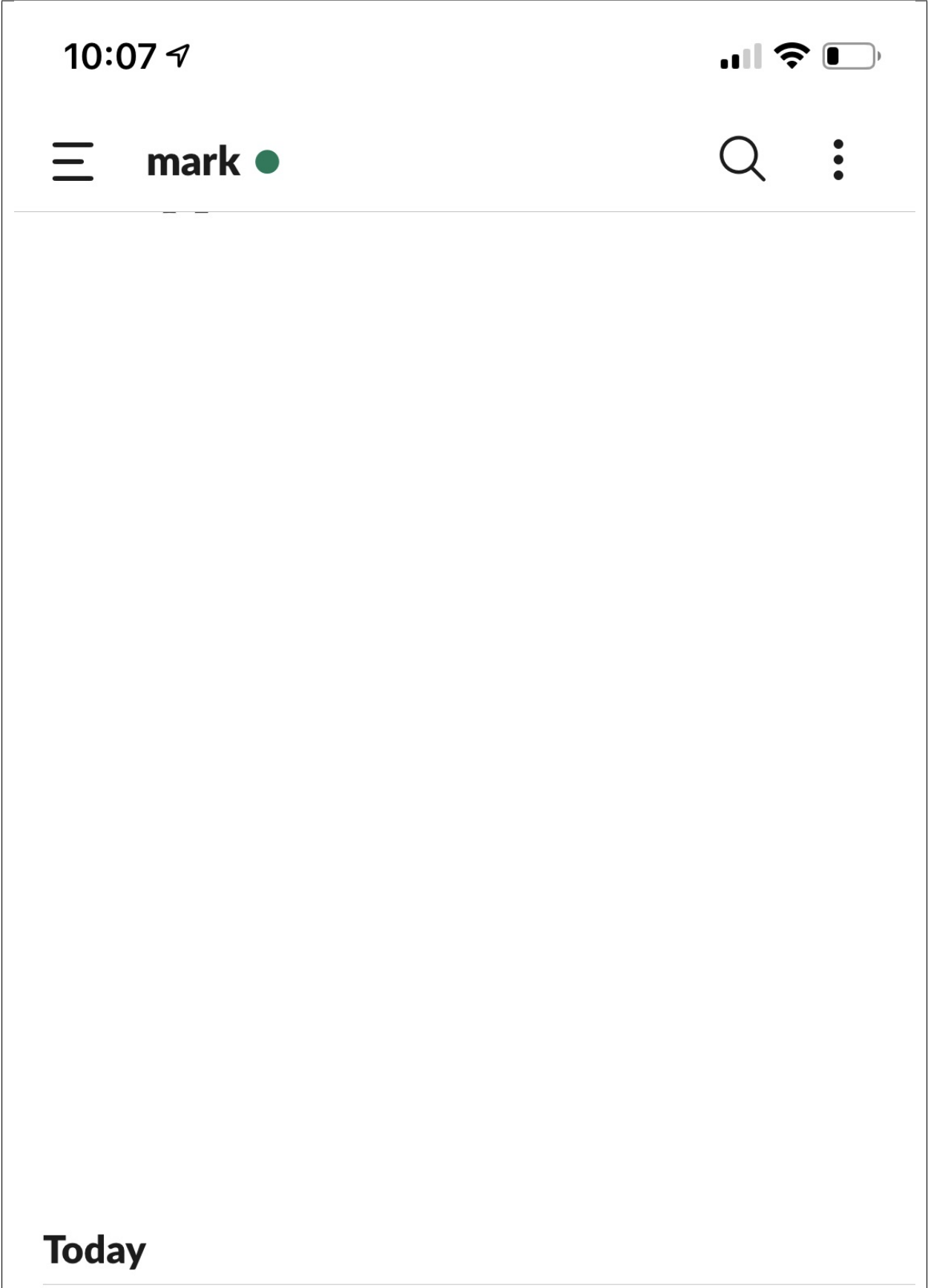
Enabled: ☒

- Configure the delivery plugin
 - **Slack Webhook URL:** Paste the URL you obtain above from the page
 - **Message Test:** Static text that will appear in the slack message you receive when the rule triggers
- Enable the plugin and click *Next*
- Complete your notification setup

When the notification rule triggers you will receive messages in you Slack client on your desk top

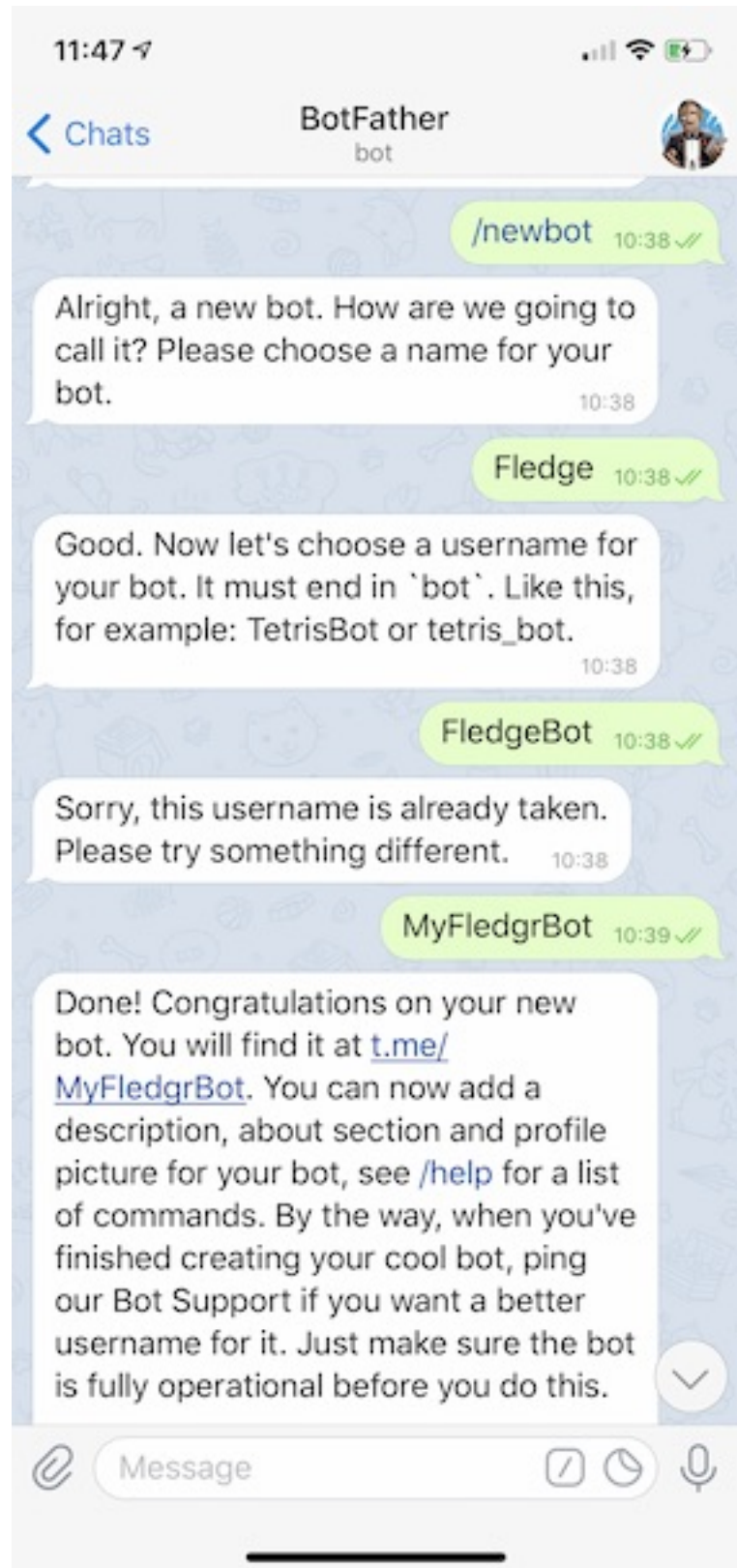


and/or your mobile devices



14.5.16 Telegram Messages

The *flir-notify-telegram* delivery notification plugin allows notifications to be delivered as instant messages on the Telegram messaging platform. The plugin uses Telegram BOT API, to use this you must create a BOT and obtain a token.



To obtain a Telegram BOT token

- Use the Telegram application to send a message to *botfather*.

- In your message send the text /start
- Then send the message /newbot
- Follow the instructions to name your BOT

- Copy your BOT token.

You now need to get a chat id

- In the Telegram application send a message to you chat BOT
- Run the following command at the your shell command line or use a web browser to go to the URL <https://api.telegram.org/bot<YourBOTToken>/getUpdates>

```
wget https://api.telegram.org/bot<YourBOTToken>/getUpdates
```

Examine the contents of the getUpdates file or the output from the web browser

- Extract the id from the “chat” JSON object

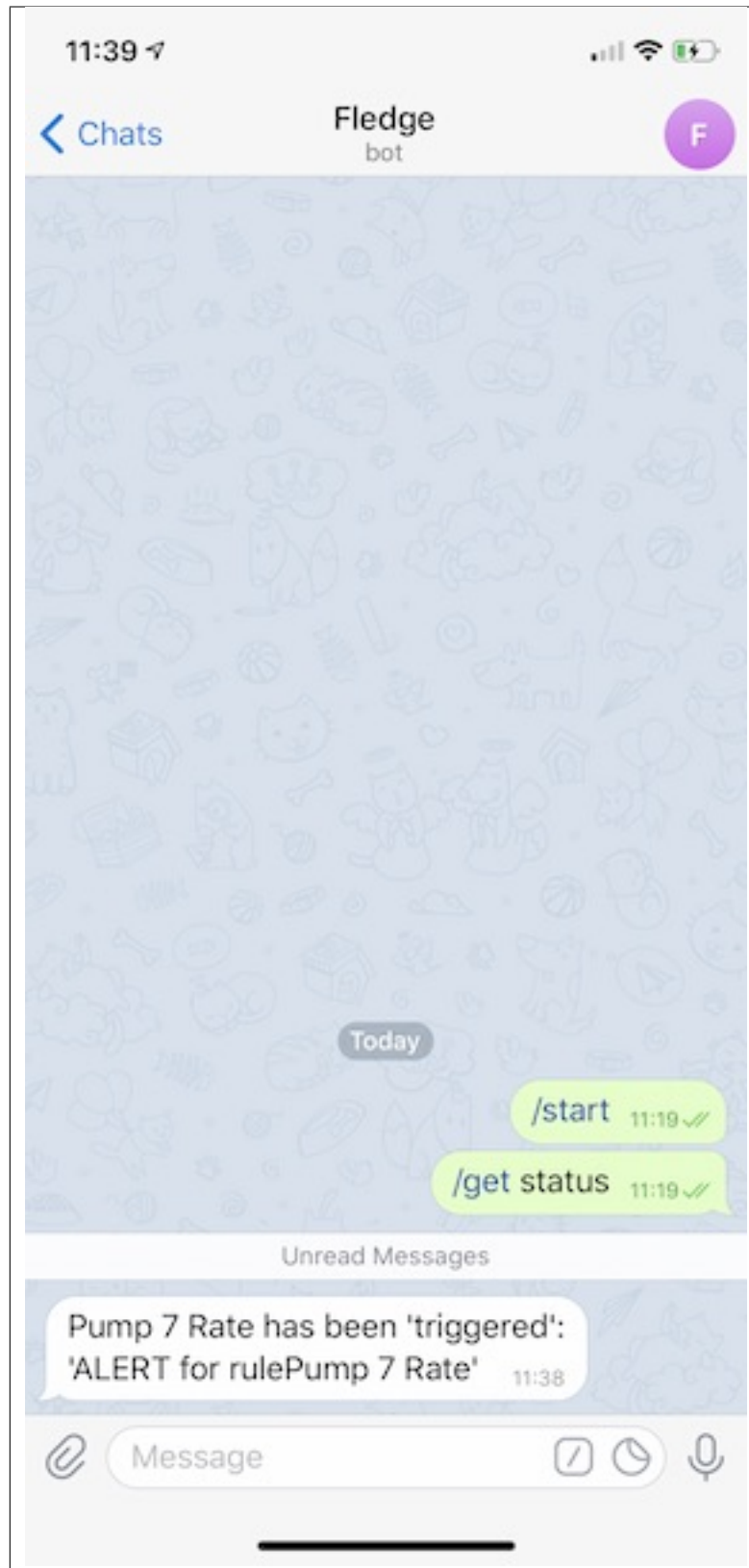
```
{
  "ok": true,
  "result": [
    {
      "update_id": 562812724,
      "message": {
        "message_id": 1,
        "from": {
          "id": 1166366214,
          "is_bot": false,
          "first_name": "Mark",
          "last_name": "Riddoch"
        },
        "chat": {
          "id": 1166366214,
          "first_name": "Mark",
          "last_name": "Riddoch",
          "type": "private"
        },
        "date": 1588328344,
        "text": "start",
        "entities": [
          {
            "offset": 0,
            "length": 6,
            "type": "bot_command"
          }
        ]
      }
    }
  ]
}
```

Once you have created your notification rule and move on to the delivery mechanism

- Select the Telegram plugin from the list of plugins
- Click *Next*

- Configure the delivery plugin
 - **Telegram BOT API token:** Paste the API token you received from botfather
 - **Telegram user chat_id:** Paste the id field form the chat
 - **Telegram BOT API url Prefix:** This is the fixed part of the URL used to send messages and should not be modified under normal circumstances.
- Enable the plugin and click *Next*
- Complete your notification setup

When the notification rule triggers you will receive messages Telegram application



14.5.17 Zendesk Ticket Creation

The *flir-notify-zendesk* delivery notification plugin allows notifications to be used to create tickets within Zendesk. The tickets are created within a specified project with a summary, description and other information supplied by FLIR Bridge.

To obtain an API token from Zendesk

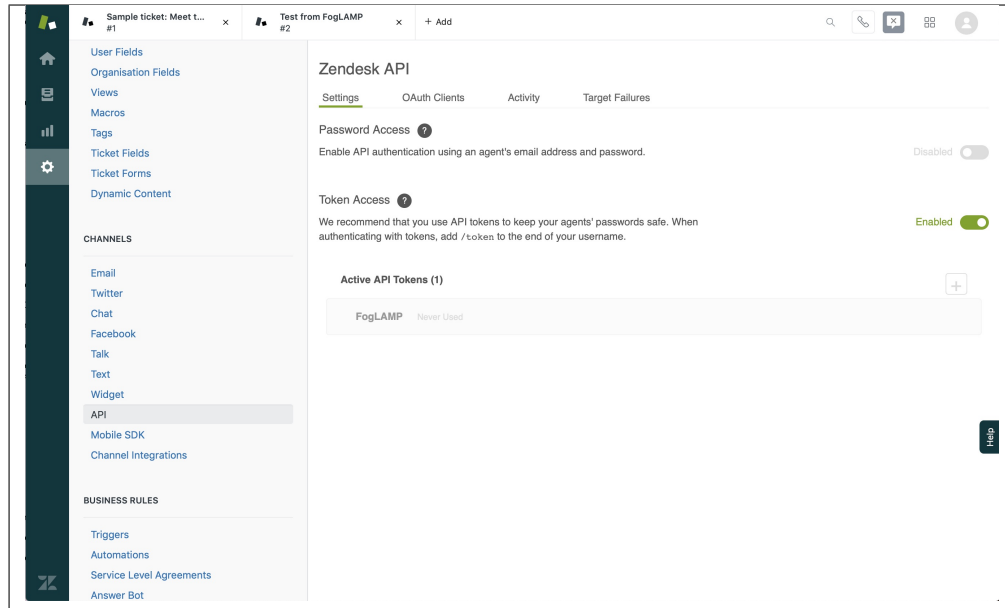
- Visit the page
- Select *Create API token*
- Enter a name for your application, this must be unique for each FLIR Bridge Zendesk application you create
- Click on Create

Once you have created your notification rule and move on to the delivery mechanism

- Select the *zendesk* plugin from the list of plugins
- Click *Next*

The screenshot displays the configuration page for the Zendesk delivery plugin. At the top, a progress bar indicates the sequence of steps: 1. Notification Instance, 2. Rule, 3. Delivery Channel (current), and 4. Done. The main form area includes input fields for 'Subdomain', 'Subject', 'Email', 'API Token' (with a 'password' placeholder and a toggle icon), and 'Comment'. Below these is an 'Additional Fields' section showing a single field with the number '1' and a code icon. An 'Enabled' checkbox is located at the bottom left of the form. At the very bottom of the interface are 'Previous' and 'Next' navigation buttons.

- Configure the delivery plugin
 - **Subdomain:** The subdomain where your Zendesk instance is installed.
 - **Subject:** The subject for the new ticket that is created.
 - **Email:** Your Zendesk registered email address, this is the name of the account you used to create the API token
 - **API Token:** The API token for your email address. You must enable API token in your Zendesk account and create a token for FLIR Bridge to use.



- **Comment:** The text to add into the comment of the ticket, this may include text substitutions (see below).
 - **Additional Fields:** This is a JSON document that contains a number of key/value pairs, each of these pairs is a field name and content to add to the ticket. Text substitutions may be applied here also.
- Enable the plugin and click *Next*
 - Complete your notification setup

When the notification rule triggers a Zendesk ticket will be created.

Text Substitution

Text markers may be used to substitution text with the fields in the Zendesk ticket. The markers supported are

- **%MESSAGE%:** this is replaced with the message generated in the notification system
- **%REASON%:** this is replaced with the reason for the notification, it may be the string *triggered* or *cleared*.
- **%TIMESTAMP%:** this is replaced with the timestamp of the reading data that caused the notification to trigger.