Predix Edge Apps and Services 2.7.0

Documentation

We want to hear from you. If you have any comments, questions, or suggestions about our documentation, send them to the following email address:

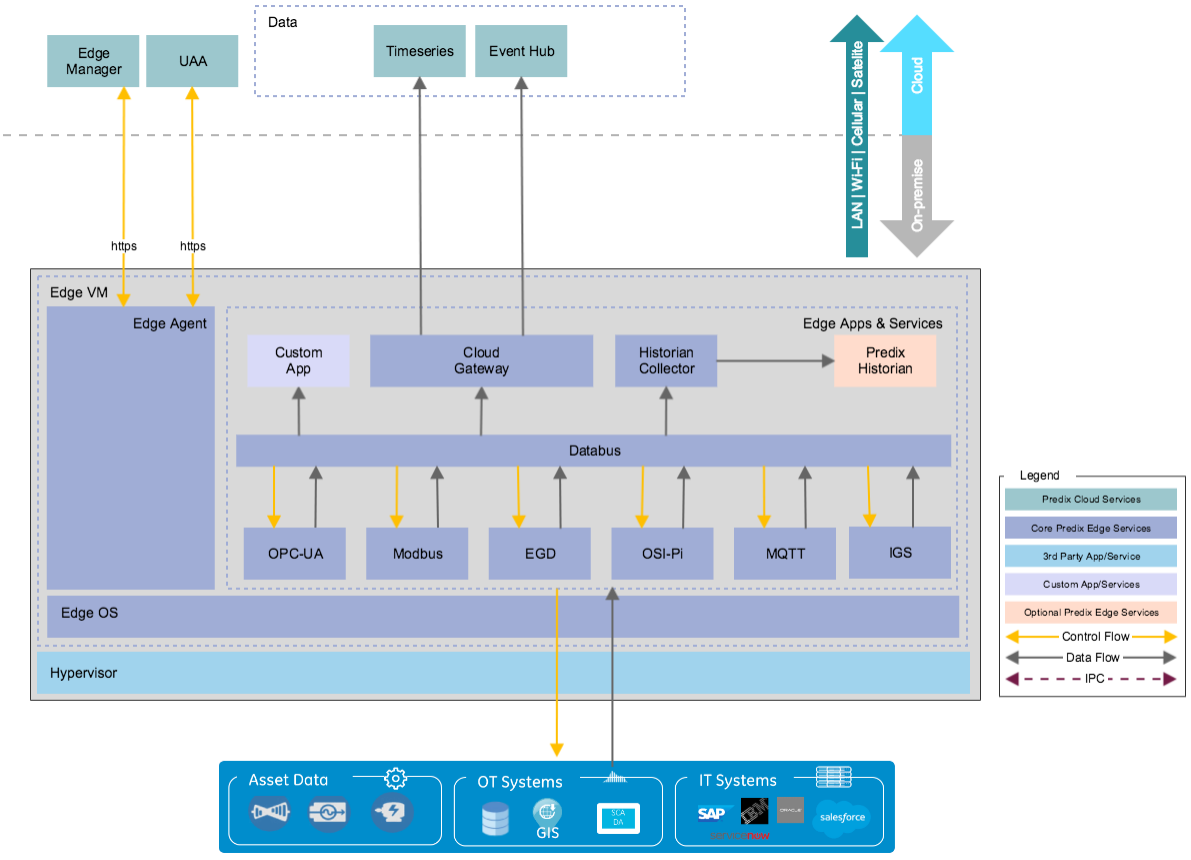
doc@ge.com

Predix Edge Apps and Services Overview

About Predix Edge Applications

Architecture

The following diagram depicts the core components available for applications in Predix Edge and how they interact with each other over the Predix Edge Broker.



Things to Know About Edge Applications

- All Predix Edge applications are deployed as Docker containers.
- Any development language can be used as long as it can be deployed as a Docker container in a Linux environment and communicate with MQTT. Most modern languages you would consider include MQTT libraries.

- Each application communicates with other applications through publishing and subscribing to messages on the Predix Edge Broker. Predix Edge Broker is included when you download and install Predix Edge.

1. The OPC-UA Protocol Adapter is configured to retrieve tag data and publish it to the broker on a topic named **opcua_data**.
2. Your custom app running as a container subscribes to the **opcua_data** topic, manipulates the data in some way and publishes the results back to the broker on the topic **timeseries_data**.
3. The Time Series Cloud Gateway application subscribes to the **timeseries_data** topic and sends the data to Predix Time Series.

You can use the same “pub/sub“ data flow to route data to other applications such as Event Hub Cloud Gateway, Predix Historian or other custom applications.

For more information on the applications provided with Edge refer to:

- Predix Edge Protocol Adapters Overview *(page 8)*
- About Predix Edge Cloud Gateway *(page 88)*
- Predix Historian *(page)*

Additional Information

Hello World - A Local Microservice

Setup Predix Edge Applications

Predix Edge Applications

Predix Edge includes the following applications for acquiring, publishing and storing data at the Edge. The applications are stored in Artifactory. Use the following information to ensure you can access the applications.

For GE Employees

To access Artifactory downloads, those using a GE email address must first be logged into Artifactory.

Note: If you attempt to download from Artifactory without first logging into Artifactory, you will be asked to **Sign in**, which will not work.

1. Go to Artifactory.
2. Click the **Log In** button.
3. Click the **SAML SSO** icon.
4. Use your SSO to log in.

5. You can then return to the documentation link to access Artifactory.

For Predix Users

To access Artifactory links in the Predix Edge documentation, you must first create an account on predix.io. Your predix.io account sign in credentials will be used to access Artifactory.

When you click an Artifactory link, enter your predix.io username (email address) and password in Artifactory's **Sign In** dialog.

Application	Configuration	Description
Pre-Packaged Cloud Gateway (AArch64/ARM64) Pre-Packaged Cloud Gateway (AMD64/Intel64)	Sample	Sends data from the Predix Edge Broker to Time Series or Event Hub instance(s)
Modbus (AArch64/ARM64) Modbus (AMD64/Intel64)	Sample	Acquires Modbus data and publishes it to the Predix Edge Broker
EGD (AArch64/ARM64) EGD (AMD64/Intel64)	Sample	Acquires EGD data and publishes it to the Predix Edge Broker
MQTT (AArch64/ARM64) MQTT (AMD64/Intel64)	Sample	Acquires data from an external MQTT broker and publishes it to the Predix Edge Broker
OSI-Pi (AArch64/ARM64) OSI-Pi (AMD64/Intel64)	Sample	Acquires data from an OSI-PI server and publishes it to the Predix Edge Broker
OPC-UA (AArch64/ARM64) OPC-UA (AMD64/Intel64)	Sample	Acquires data from an OPC-UA server and publishes it to the Predix Edge Broker
Predix Historian (licensed separately from Predix Edge)	Sample	Historian database and RESTful query engine for storing and extracting data
Predix Historian Collector	Sample	Sends data from the Predix Edge Broker to Predix Historian

Installing an Application

Download and install a Predix Edge application.

1. Click the application link to download the application to your machine.

2. Upload the file to your Edge Manager Repository as a Predix Edge application.
3. Deploy the application to an enrolled Predix Edge device.

Configuring an Application

Configure a Predix Edge application.

1. Download and extract the sample configuration ZIP for the application.
2. Modify the settings in the sample config file for your environment.
3. Re-zip the file.
4. Upload the new ZIP file to the Predix Edge Manager Repository as a Predix Edge configuration.
5. Deploy the configuration to the corresponding application running on your Predix Edge device.

Related concepts
Predix Edge Protocol Adapters Overview *(page 8)*
About Predix Edge Cloud Gateway *(page 88)*

Related tasks
Uploading Software and Configuration Packages to the Predix Edge Manager Repository *(page)*

Predix Edge Protocol Adapters

Protocol Adapters Overview

Predix Edge Protocol Adapters Overview

Learn about where to download the Predix Edge Protocol Adapters and their corresponding sample configurations and understand how the `blocks` section is used.

Protocol-Specific Information

Specific protocol adapter information is available for:

• EGD *(page 21)*
• MQTT *(page 36)*
• Modbus *(page 40)*
• OPC-UA *(page 55)*
• OSI-Pi *(page 81)*

Download the Adapters

The Protocol Translator Apps and sample configurations in the table below are stored in Artifactory. Use the following information to ensure you can access the apps.

For GE Employees

To access Artifactory downloads, those using a GE email address must first be logged into Artifactory.

Note: If you attempt to download from Artifactory without first logging into Artifactory, you will be asked to **Sign in**, which will not work.

1. Go to Artifactory.
2. Click the **Log In** button.
3. Click the **SAML SSO** icon.
4. Use your SSO to log in.
5. You can then return to the documentation link to access Artifactory.

For Predix Users

To access Artifactory links in the Predix Edge documentation, you must first create an account on predix.io. Your predix.io account sign in credentials will be used to access Artifactory.

When you click an Artifactory link, enter your predix.io username (email address) and password in Artifactory's **Sign In** dialog.

Adapter App	Config
OSI-Pi (AMD64/Intel64)	Sample
OPC-UA (AMD64/Intel64)	Sample
MQTT (AMD64/Intel64)	Sample
Modbus (AMD64/Intel64)	Sample
EGD (AMD64/Intel64)	Sample

Deployment

Go to your specific adapter documentation for a sample `docker-compose.yml` file, and use it as you follow the Packaging and Deployment instructions.

Configuration

Each adapter requires a configuration file, and a sample is available in the documentation for each adapter. The samples are named `config.json`, but the file name can be changed, as long as the change is reflected in the `docker-compose.yml`. The configuration file is required as part of the Packaging and Deployment process.

The configuration file is a JSON file that contains two sections—`blocks` and `mappings`. The format is as follows:

```
{
 "blocks": {
  ...
 },
 "mappings": {
  ...
 }
}
```

The Blocks Section

The `blocks` section is used to initialize the blocks that will be used by the protocol translator. Declaring a block in this section will instantiate it but it will not wire it to any other blocks.

Every block must have a `type` and `config` field in the configuration file. The `type` field defines what type of block is to be instantiated. The `config` section defines the configuration fields for that block and will look different depending on the `type` of the block. The `config` section is passed to the block when it is instantiated.

In the example below, two blocks are defined—one named `block1` of type `fooblock`, and another named `block2` of type `barblock`.

```
{
    "blocks": {
        "block1": {
            "type": "fooblock",
            "config": {
                ...
            }
        },
        "block2": {
            "type": "barblock",
            "config": {
                ...
            }
        },
        ...
    },
    "mappings": {
```

```
        ...
    }
}
```

Mapping the Blocks

The `mappings` section is used to connect the blocks that were initialized in the blocks section.

If a block name referenced in this section does not exist in the blocks section, the protocol translator will log an error and ignore the mapping. By default, most blocks have an implicit `input` and `output` port that can be connected to a port from a different block so that data flows between the blocks.

In the example below, two blocks are mapped together. The `output` port of the block named `block1` is connected to the `input` port of the block named `block2`. Therefore, data will flow from `block1` to `block2`.

```
{
    "blocks": {
        ...
    },
    "mappings": {
        "block1:output":"block2:input",
        ...
    }
}
```

Flat JSON to Time Series Conversion Block

Table 1.

Type
'flattotimeseries

This block is used to convert data formatted as Flat JSON into the Predix Time Series data format. A description of this format is in Pushing Time Series Data *(page)*.

Table 2.

Field	Type	Required	Default
attributes	Object		
log_level	String		'off'
log_name	String		<block_name>

attributes

The `attributes` field is an object with key/value pairs that will be directly injected into all output data as Time Series attributes.

log_level and log_name

For details about the `log_level` and `log_name` fields, see Generic CDP Blocks *(page 15)*.

Examples

The following is a sample configuration for this block.

```
"flat_to_time_series": {
 "type": "flattotimeseries",
 "config": {
  "attributes": {
   "source": "Albuquerque"
  }
 }
}
```

An example of input data (Flat JSON) and output data (Predix Time Series) resulting from the above configuration block is as follows. Its configuration fields are as follows.

Input Flat JSON

```
Input Flat JSON:
{
 "timestamp": "1504739531776",
 "data": {
  "temperature": {
   "val": 15,
   "type": "int16"
  },
  "pressure": {
   "val": 16.2,
   "type": "float"
  }
 }
}
```

Output Predix Time Series

```
{
 "messageId": "flex-pipe",
 "body": [
  {
   "name": "temperature",
```

```
    "datapoints": [
     [1504739531776, 15, 3]
    ],
    "attributes": {
     "source": "Albuquerque"
    }
   },
   {
    "name": "pressure",
    "datapoints": [
     [1504739531776, 16.2, 3]
    ],
    "attributes": {
     "source": "Albuquerque"
    }
   }
 ]
}
```

Since no configuration fields for this block are required, the entire ‘config’ field can be omitted:

```
"flat_to_time_series": {
 "type": "flattotimeseries"
}
```

Splitter Routing Block

Table 3.

Type
splitter

This block will receive data on a single input port and send that data on to any number of output ports specified by the `output_count` field. Its configuration fields are as follows:

Table 4.

Field	Type	Required	Default
output_count	Integer	yes	
log_level	String		'off'
log_name	String		<block_name>

output_count

The `output_count` field determines how many output ports will be created for this block. The block's output ports are named "output#" where "#" is a number starting at 1. The 'mappings' section in the example shows these output port names.

log_level and log_name

For details about the `log_level` and `log_name` fields, see Generic CDP Blocks *(page 15)*.

Example

The following is a sample configuration file that includes this block.

```
{
 "blocks": {
  "input_block": {
   "type": "dummy",
   "config": { "field": "value" }
  },
  "splitter": {
   "type": "splitter",
   "config": {
    "output_count": 3
   }
  },
  "out_block1": {
   "type": "dummy",
   "config": { "field": "value" }
  },
  "out_block2": {
   "type": "dummy",
   "config": { "field": "value" }
  },
  "out_block3": {
   "type": "dummy",
   "config": { "field": "value" }
  }
 },
 "mappings": {
  "input_block:output":"splitter:input",
  "splitter:output1":"out_block1:input",
  "splitter:output2":"out_block2:input",
  "splitter:output3":"out_block3:input"
 }
}
```

Generic CDP Blocks

These are the possible options for the configuration fields of the CDP In, CDP Out, and CDP Out Queue generic CDP blocks.

The block types CDP In, CDP Out, and CDP Out Queue are classified as generic CDP blocks because they directly transfer the information in their configuration to the CDP library. In these blocks, the protocol translator adds minimal functionality on top of the CDP library and directly calls the relevant CDP procedures with the parameters collected from the configuration block.

For details on protocol specific documentation, refer to specific protocol adapter documentation.

The following table shows the possible options for the configuration fields of these blocks. The sections after the below table go into more detail about each field.

Field	Type	Required	Default	CDP In	CDP Out	CDP Out Queue
`transport_addr`	String	yes		yes	yes	yes
`node_ref`	String	yes		yes	yes	yes
`method`	String	yes		yes	yes	yes
`interval`	Integer	yes if `method` is ‘get‘		yes		
`log_level`	String		‘off‘	yes	yes	yes
`log_name`	String		<block_name>	yes	yes	yes
`options`	Object			yes	yes	yes
`directory`	String	yes				yes
`max_cache_size`	Integer		90			yes
`max_cache_size_units`	String		‘%‘			yes

transport_addr

The `transport_addr` field determines the protocol and location of the endpoint the block will communicate with.

The prefix to the URI (for example, the "mqtt-tcp" in `mqtt-tcp://localhost:1883` determines what protocol to use:

Protocol	Possible Prefixes
EGD	`'egd'`
EGD Read Only	`'egd-ro'`

Protocol	Possible Prefixes
Modbus	`'modbus-tcp'`, `'modbus-rtu'`, `'modbus-ascii'`, `'modbus-rtu-tcp'`, `'modbus-ascii-tcp'`
MQTT	`'mqtt-tcp'`
OPC-UA	`'opc-tcp'`
OSI-PI	`'osipi-http'`, `'osipi-https'`
Predix Time Series	`'pxts'`, `'pxtss'`

node_ref

The `node_ref` field means a slightly different thing for each protocol but is essentially the specific node that is to be communicated with in the protocol specified by the `transport_addr`. For MQTT, this means the topic to publish to or subscribe from.

Note: This field is unused for the Timeseries CDP library, but is still a required field for the protocol translator to work properly.

method

The `method` field determines the method of the communication (synch/asynch, send/receive). This field's possible values are dependent on both the block type and protocol used, for example, some protocols support only the publish/subscribe **or** get/set type of communication.

Block Type	Possible Method Values
`cdpin`	`'get'`, `'sub'`
`cdpout`	`'set'`, `'pub'`
`cdpoutqueue`	`'set'`, `'pub'`

interval

The `interval` field is only relevant (and required) if the block is of type `cdpin` and the method field is `get`. This determines the interval (in milliseconds) at which the block will poll its endpoint for data.

For example, if this is set to 1000, the block will attempt to get data every second.

log_level

The `log_level` field determines which level of logs to output. If the field is not set to one of the following values, the block will not log anything.

Possible Values
`'debug'`, `'info'`, `'warn'`, `'err'`, `'critical'`

log_name

The `log_name` field defines a name to identify the block's logs. This is typically prepended to the log output. If unset, it defaults to the block name.

options

The options field is a JSON object whose contents are specific to the protocol specified in the `transport_addr` field.

This is not considered a required field, but some protocols may require it to function properly. For example, if the Time Series protocol (`'pxtss'`) is specified in the `transport_addr` field and the options field does not contain the `token_file` and `predix_zone_id`, the block will be unable to connect to the Time Series endpoint.

directory

The `directory` field is only used in the CDP Out Queue block. This field determines the directory where the block's store-and-forward files (a disk-backed queue of data to be sent) will be created and updated. If the directory does not exist, it will be created, but if its parent directory does not exist, an error will be thrown.

⚠ **Important:** Be careful about the directory you specify here. If the path in this field is the same as the path in the `config` of another block within the same Docker application, there will be a conflict between the two blocks' disk-backed queue (store-and-forward) files.

For example, if you have an OPC-UA Adapter container and a Cloud Gateway container both within the same Docker application, they will both receive the same `/data` mount directory. If `/data` is specified as the directory field for a block in both the OPC-UA and Cloud Gateway configuration files, they will both fail to maintain their own disk-backed queue files.

max_cache_size

The `max_cache_size` field determines the maximum amount of disk space allowed for this block to use for one of its disk-backed queues (store-and-forward file).

If left unset, this field's value will default to 90 and the `max_cache_size_units` field will default to `%`.

⚠ **Important:** Due to the way these files are managed, the block may consume up to 2 times the size of `max_cache_size` on disk. For example, a 100MB queue may consume up to 200MB of actual disk space.

max_cache_size_units

The `max_cache_size_units` field determines the units to use for the `max_cache_size` field.

This field's default value is `%`.

Value	Meaning
`'%'`	Percentage of available disk space
`'B'`	Bytes
`'KB'`	Kilobytes
`'MB'`	Megabytes
`'GB'`	Gigabytes

CDP In

Type
`'cdpin'`

This block is used to receive data from any transport on the CDP. Its configuration fields are as follows:

Field	Type	Required	Default
`transport_addr`	String	yes	
`node_ref`	String	yes	
`method`	String	yes	
`interval`	Integer	yes if `method` is 'get'	
`log_level`	String		'off'
`log_name`	String		<block_name>
`options`	Object		

For more specific information on these fields, see the high level description of generic CDP blocks above.

Example CDP In Block:

```
"example_cdpin_mqtt_source_block": {
    "type": "cdpin",
    "config": {
        "transport_addr": "mqtt-tcp://predix-edge-broker",
        "node_ref": "predix_historian",
```

```
        "method": "sub",
        "log_name": "mqtt_source_block",
        "log_level": "err"
    }
}
```

CDP Out

Type
`'cdpout'`

This block is used to send data to any transport on the CDP. Its configuration fields are as follows:

Field	Type	Required	Default
`transport_addr`	String	yes	
`node_ref`	String	yes	
`method`	String	yes	
`log_level`	String		‘off‘
`log_name`	String		<block_name>
`options`	Object		

For more specific information on these fields see the high level description of generic CDP blocks above.

Example CDP Out Block:

```
"example_cdpout_mqtt_sink_block": {
    "type": "cdpout",
    "config": {
        "transport_addr": "mqtt-tcp://predix-edge-broker",
        "node_ref": "data/pressure",
        "method": "pub",
        "log_name": "mqtt_sink",
        "log_level": "err",
        "options": {
            "qos": 1
        }
    }
}
```

CDP Out Queue

Type
`'cdpoutqueue'`

This block is used to send data to any transport on the CDP. It is identical to the CDP Out block above, except that this block will also queue messages on a disk-backed queue (store-and-forward) if it is unable to deliver them. Its configuration fields are as follows:

Field	Type	Required	Default
`transport_addr`	String	yes	
`node_ref`	String	yes	
`method`	String	yes	
`log_level`	String		‘off‘
`log_name`	String		<block_name>
`options`	Object		
`directory`	String	yes	
`max_cache_size`	Integer		90
`max_cache_size_units`	String		‘%‘

Example CDP Out Queue Block:

```
"example_cdpoutqueue_timeseries_sink_block": {
    "type": "cdpoutqueue",
    "config": {
        "transport_addr": "pxtss://timeseries-instance.run.aws-usw02-
pr.ice.predix.io/v1/stream/messages",
        "node_ref": "doesn't_matter_for_timeseries",
        "method": "set",
        "log_name": "timeseries_sink",
        "log_level": "err",
        "directory": "/data/container_name/timeseries_sink/dbq/",
        "max_cache_size": 100,
        "max_cache_size_units": "MB",
        "options": {
            "token_file": "/edge-agent/access_token",
            "predix_zone_id": "01234567-8987-6543-2101-234567898765",
            "proxy_url": "https://proxy.proxy.proxy.com:8080"
        }
    }
}
```

If these blocks are used to communicate to an MQTT broker (and ‘mqtt-tcp’ is specified in the ‘transport_addr’ field), the “options” field can contain the following optional sub-fields:

- ‘qos’: This field can be set to the desired “quality of service” for any MQTT message transfers. This field’s value can be 0, 1, or 2. These values correspond to “at most once”, “at least once”, and “exactly once” message delivery respectively.

- ‘clientid’: This field can be set to the desired client ID for the block’s connection to the MQTT broker. The client ID helps the MQTT broker to identify the block. NOTE: If either the ‘client_id’ is not set, or the ‘qos’ is 0, the MQTT “clean_session” flag will be set when connecting to the broker. Otherwise, the broker will persist this block’s connection information and subscriptions.
- ‘username’: This field can be set if the MQTT broker being connected to requires username/password authentication.
- ‘password’: This field can be set if the MQTT broker being connected to requires username/password authentication.

EGD Protocol Adapters

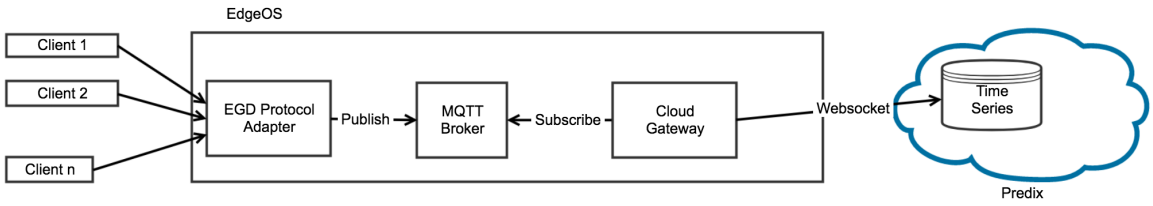
EGD Protocol Adapters Overview

Protocol Adapters - EGD

Ethernet Global Data (EGD) is a protocol that supports the ability to share information between controllers (nodes) in a networked environment. EGD allows one controller, referred to as the producer of the data, to simultaneously send information to any number of peer controllers (consumers) at a fixed periodic rate. A network based on this capability will support a large number of controllers, each of which is capable of both producing and consuming information. Thus, EGD allows data to be shared globally between controllers. In addition, EGD supports a set of commands for accessing data and protocol information on EGD nodes. EGD also defines a mechanism for sharing configuration information among nodes.

The following diagram shows a simple use case where the data from an external device flows through the EGD protocol adapter and is finally sent to the Predix cloud.

Figure: Protocol Adapter - EGD



Where Do I Get the EGD Protocol Translator Application?

The EGD Protocol Translator App and sample configuration file in the table below are stored in Artifactory. Use the following information to ensure you can access the files.

For GE Employees

To access Artifactory downloads, those using a GE email address must first be logged into Artifactory.

Note: If you attempt to download from Artifactory without first logging into Artifactory, you will be asked to **Sign in**, which will not work.

1. Go to Artifactory.
2. Click the **Log In** button.
3. Click the **SAML SSO** icon.
4. Use your SSO to log in.
5. You can then return to the documentation link to access Artifactory.

For Predix Users

To access Artifactory links in the Predix Edge documentation, you must first create an account on predix.io. Your predix.io account sign in credentials will be used to access Artifactory.

When you click an Artifactory link, enter your predix.io username (email address) and password in Artifactory's **Sign In** dialog.

Table 5. EGD Protocol Adapter Downloads

Adapter App	Sample Config
EGD (AMD64/Intel64)	EGD Read/Write Protocol Adapter EGD Dynamic Binding Protocol Adapter

Note: Both the EGD Read/Write and EGD Dynamic Binding protocol adapters are contained in the above images.

To configure and use the EGD Read/Write Protocol Adapter, see EGD Read/Write Protocol Adapter *(page 23)*.

To configure and use the EGD Dynamic Binding Protocol Adapter, see EGD Dynamic Binding Protocol Adapter *(page 33)*.

The latest version of the EGD adapter is `protocol-adapter-egd:amd64-latest`.

Protocol Benchmarking - EGD

The numbers in the tables below represent the ideal throughput for the data pump use case (data traveling from a protocol adapter to the Predix Edge MQTT Broker to the Cloud Gateway to Time Series).

The tests were executed in a controlled environment with one adapter running at a time, under ideal network conditions with a local time series simulator. The rate was deemed successful if there was less than a 1 percent loss rate over the course of a multi-hour test. Based on the testing, data loss rates dramatically increase as tag counts pass these thresholds.

The tested VMs were configured as follows (with a 2GHz vCPU):

Table 6. Tested VM Configurations

VM	Processors	RAM (GB)	Disk Space (GB)
VM (small)	1	1	5
VM (medium)	2	4	20
VM (large)	4	8	20

Table 7. EGD Dynamic

Environment	Period (seconds)	Acceptable Tag Rate
VM (small)	1	4950
	10	21000
VM (medium)	1	4875
	10	23625
VM (large)	1	4875
	10	23625
Predix Edge Gateway 3002	1	5100
	10	18150

The number of tags per adapter does not scale with the device's performance characteristics. It is recommended to add new adapters to support higher tag counts along with corresponding physical hardware to support the additional applications.

EGD Read/Write Protocol Adapter

Overview of Capabilities

The EGD protocol adapter enables data transfer from any controller that supports EGD protocol to the Predix cloud. It supports both Read and Write. It is important to note that while the EGD protocol

supports several classes of devices, the Predix EdgeOS EGD Protocol adapter supports only Class 1 devices and associated data messages.

Deployment and Configuration Resources

There are three possible EGD configuration types: EGD, EGD Flat, and EGD Sink Flat. The following fields are common to all three configuration types: `transport_addr`, `log_level` and `log_name`, and `options`.

`transport_addr`

The transport_addr field determines the location of the EGD endpoint the block will communicate with. Its prefix must be `'egd://'`

`log_level` and `log_name`

For details about the `log_level` and `log_name` fields, see the Generic CDP Blocks *(page 15)* section of the Protocol Adapters *(page 8)* documentation page.

`options`

The `options` field contains any other miscellaneous configuration options necessary for the desired EGD communication.

EGD

Table 8.

Type
`'egd'`

This block is used to source raw data from any number of EGD exchanges. Its configuration fields are as follows:

Table 9.

Field	Type	Required	Default
transport_addr	String	yes	
subs	Array	yes	
log_level	String		'off'
log_name	String		
options	Object		

`subs`

The `subs` field determines which EGD exchanges to subscribe to.

Example EGD Config Block

```
"egd_source_block": {
    "type": "egd",
    "config": {
        "transport_addr": "egd://localhost",
        "subs": [
            "4026531967/150/0/0",
            "5126541937/151/2/1"
        ],
        "log_level": "info",
        "log_name": "egd_source",
        "options": {
            "foo": "bar"
        }
    }
}
```

EGD Flat

Table 10.

Type
'egdflat'

This block is used to source raw data from any number of EGD exchanges and transform the data into flat JSON format. Its configuration fields are as follows:

Table 11.

Field	Type	Required	Default
transport_addr	String	yes	
data_map	Object	yes	
log_level	String		'off'
log_name	String		
options	Object		

`data_map`

The `data_map` field determines which EGD exchanges to subscribe to and how to convert their data to flat JSON format. The `data_map` follows the following format:

```
"data_map": {
    "<EGD Node Reference>": [
```

```
        {<Data Map Object>},
        {<Data Map Object>}
    ],
    "<EGD Node Reference>": [
        {<Data Map Object>},
        {<Data Map Object>}
    ]
}
```

Where the objects shown above have the following fields:

Table 12.

Field	Type	Required
alias	String	yes
bit_offset	Integer	yes
type	String	yes

The `alias` field of the data_map determines the name of the requested value that should mean something to the target application. Examples are “temperature”, “pressure”, etc.

The `bit_offset` field determines what bit offset into the EGD page to start retrieving the value from. The EGD producer has a described data layout that can be retrieved from a controller or workstation via the EGD HTTP configuration API. Refer to the EGD protocol docs to learn more about the Production Data Unit format.

The `type` field of the data_map defines the type of data to pull from the EGD page. This value will determine how many bytes after the `bit_offset` are accessed and how to combine the bytes to form the requested type. Its possible values are as follows:

- 'int8'
- 'uint8'
- 'int16'
- 'uint16'
- 'int32'
- 'uint32'
- 'int64'
- 'uint64'
- 'float'
- 'double'
- 'string'
- 'datetime'
- 'time'

Example Output Data (Flat JSON)

```
{
```

```
    "timestamp": "1504739531776",
    "data": {
        "temperature": {
            "val": 15,
            "type": "int16"
        },
        "pressure": {
            "val": 16.2,
            "type": "float"
        }
    }
}
```

Example EGD Flat Config Block

```
"egd_source_block": {
    "type": "egdflat",
    "config": {
        "transport_addr": "egd://localhost",
        "data_map": {
            "4026531967/150/0/0": [
                {
                    "alias": "sine01",
                    "bit_offset": 1056,
                    "type": "double"
                }
            ],
            "5126541937/151/2/1": [
                {
                    "alias": "valve_on",
                    "bit_offset": 8,
                    "type": "int8"
                },
                {
                    "alias": "valve_status",
                    "bit_offset": 16,
                    "type": "string"
                }
            ]
        },
        "log_level": "info",
        "log_name": "egd_source_flat",
        "options": {
            "foo": "bar"
        }
    }
}
```

EGD Sink Flat

Table 13.

Type
'egdsinkflat'

This block is used to send EGD data to multiple EGD exchanges. It translates input in the flat JSON format into raw EGD payloads. Its configuration fields are as follows:

Table 14.

Field	Type	Required	Default
transport_addr	String	yes	
data_map	Object	yes	
log_level	String		'off'
log_name	String		
options	Object		

`data_map`

The `data_map` field determines how to convert input data from flat JSON format to raw EGD payloads and which EGD exchanges to publish those payloads to. The `data_map` follows the following format:

```
"data_map": {
    "<EGD Node Reference>": [
        {<Data Map Object>},
        {<Data Map Object>}
    ],
    "<EGD Node Reference>": [
        {<Data Map Object>},
        {<Data Map Object>}
    ]
}
```

Where the objects shown have the following fields:

Table 15.

Field	Type	Required
alias	String	yes
bit_offset	Integer	yes
type	String	yes
max_length	Integer	yes if type is 'string'

The `alias` field of the `data_map` determines the name of the requested value in the input flat JSON. Examples are “temperature”, “pressure”, etc.

The `bit_offset` field determines what bit offset into the EGD page to start writing the value to. The EGD producer has a described data layout that can be retrieved from a controller or workstation via the EGD HTTP configuration API. Refer to the EGD protocol docs to learn more about the Production Data Unit format.

The `type` field of the `data_map` defines the type of data to write to the EGD page. This value will determine how many bytes after the bit_offset are written. Its possible values are as follows:

- 'int8'
- 'uint8'
- 'int16'
- 'uint16'
- 'int32'
- 'uint32'
- 'int64'
- 'uint64'
- 'float'
- 'double'
- 'string'
- 'datetime'
- 'time'

The `max_length` field specifies the max number of characters a string can contain. It is only required if the `type` field is 'string'.

Example Input Data (Flat JSON)

```
{
    "timestamp": "1504739531776",
    "data": {
        "temperature": {
            "val": 15,
            "type": "int16"
        },
        "pressure": {
            "val": 16.2,
            "type": "float"
        }
    }
}
```

Example EGD Sink Flat Config Block

```
"egd_sink_block": {
    "type": "egdsinkflat",
    "config": {
        "transport_addr": "egd://localhost",
        "data_map": {
```

```
            "4026531967/150/0/0": [
                {
                    "alias": "sine01",
                    "bit_offset": 1056,
                    "type": "double"
                }
            ],
            "5126541937/151/2/1": [
                {
                    "alias": "valve_on",
                    "bit_offset": 8,
                    "type": "int8"
                },
                {
                    "alias": "valve_status",
                    "bit_offset": 16,
                    "type": "string",
                    "max_length": 10
                }
            ]
        },
        "log_level": "info",
        "log_name": "egd_sink_flat",
        "options": {
            "foo": "bar"
        }
    }
}
```

Example configuration to send commands from MQTT to EGD

The following example may be useful for application authors who want to send EGD signals from MQTT.

`config.json`

```
{
   "blocks": {
       "mqtt_source": {
           "type": "cdpin",
           "config": {
               "transport_addr": "mqtt-tcp://predix-edge-broker",
               "node_ref": "opcua_source_test",
               "method": "sub",
               "log_level": "debug",
               "log_name": "opcua_mqtt_sink"
           }
       },
       "egd_sink_block": {
           "type": "egdsinkflat",
           "config": {
```

```
                "transport_addr": "egd://localhost",
                "data_map": {
                    "4026531967/150/0/0": [
                        {
                            "alias": "sine01",
                            "bit_offset": 1056,
                            "type": "double"
                        }
                    ],
                    "5126541937/151/2/1": [
                        {
                            "alias": "valve_on",
                            "bit_offset": 8,
                            "type": "int8"
                        },
                        {
                            "alias": "valve_status",
                            "bit_offset": 16,
                            "type": "string",
                            "max_length": 10
                        }
                    ]
            },
            "log_level": "info",
            "log_name": "egd_sink_flat",
            "options" {
                "foo": "bar"
            }
    }
}

    },
    "mappings": {
        "mqtt_source:output": "egd_sink_block:input"
    }
}
```

Sample Files

`docker-compose.yml`

```
Version:"3.2"

services:
  egd:
    image: "dtr.predix.io/predix-edge/protocol-adapter-egd:amd64-1.1.0"
    environment:
      config: "/config/config-egd.json"
    deploy:
      restart_policy:
        condition: on-failure
        delay: 5s
```

```
        max_attempts: 5
        window: 30s
    ports:
      - target: 18246
        published: 18246
        protocol: udp
        mode: host
    networks:
      - predix-edge-broker_net

networks:
  predix-edge-broker_net:
    external: true
```

config.json

```
{
  "blocks": {
    "egd": {
      "type": "egdflat",
      "config": {
        "transport_addr": "egd://<IP address>",
        "log_level":"err",
        "data_map": {
          "4026531967/150/2/1": [
            {
              "alias": "sine01",
              "bit_offset": 1056,
              "type": "double"
            }
          ]
        }
      }
    },

    "flat_to_timeseries": {
      "type": "flattotimeseries",
      "config": {
        "attributes": {
          "machine_type": "egd"
        }
      }
    },
    "mqtt_eventhub": {
      "type": "cdpout",
      "config": {
        "transport_addr": "mqtt-tcp://predix-edge-broker",
        "node_ref": "eventhub_data/egd_data",
        "method": "pub",
        "log_level": "err",
        "log_name": "mqtt_eventhub"
      }
    }
```

```
  },

  "mappings": {
    "egd:output": "flat_to_timeseries:input",
    "flat_to_timeseries:output":"mqtt_eventhub:input"
  }
}
```

EGD Dynamic Binding Protocol Adapter

EGD Dynamic Binding Protocol Adapter

EGD Configuration

EGD configuration details need to be distributed to a large audience. EGD configurations support XML format for extensibility and flexibility. The configuration description supports the inherent hierarchy of nodes, exchanges, variables and variable attributes of an EGD producer.

The EGD protocol uses HTTP 1.1 over TCP/IP as the transport mechanism for configuration messages. The profile of the HTTP protocol required by this specification requires servers that are at least conditionally compliant with HTTP 1.1 to be used. Although implementations are not required to support the configuration port (7937), it is encouraged.

Overview of Capabilities

The EGD dynamic binding protocol adapter enables data transfer from any controller that supports EGD protocol to MQTT or Predix cloud. Currently only Class 2 Dynamic Read is supported. The adapter is able to determine when the configuration of the EGD producer it is listening on has changed and adapt in real time by querying for a new configuration via HTTP 1.1 over TCP/IP REST calls. Events that would trigger a configuration include moving a variable from one EGD exchange to another, causing the signature and timestamp of the exchange to update.

Data Format

Once data has been sourced from an EGD producer exchange via UDP Data Production Packet, the binary format of the desired EGD variables are extracted and transformed into a flat json format.

Example:

```
{
    "timestamp": "1504739531776",
    "data": {
        "temperature": {
            "val": 15,
            "type": "int16"
        },
```

```
        "pressure": {
            "val": 16.2,
            "type": "float"
        }
    }
}
```

Configuration Resources

The EGD dynamic binding protocol adapter supports the following configuration options:

Type	Name
type	egddynamicflat

Name	Type	Required
log_level	String	No
log_name	String	No
transport_addr	String	Yes
subscriptions	Array	Yes

Within each subscription, the following configuration options are supported:

Name	Type	Required
port	Integer	No
producer_id	Integer	Yes
config_url	String	Yes
variables	Array	No

Example:

```
{
            "type": "egddynamicflat",
            "config": {
                "log_level": "debug",
                "log_name": "egddynamicflat",
                "transport_addr": "egd://127.0.0.1",
                "subscriptions": [
                    {
                        "port": 18246,
                        "producer_id": 1234,
                        "config_url": "http://192.168.1.8:8080",
                        "variables": [
                            {
                                "name": "01_BOOL",
```

```
                                "alias": "EGD.01_BOOL"
                            }
                        ]
                    },
                    {
                        "port": 18246,
                        "producer_id": 1235,
                        "config_url": "http://192.168.1.8:8080",
                        "variables": [
                            {
                                "name": "02_BOOL",
                                "alias": "EGD.02_BOOL"
                            }
                        ]
                    }
                ]
            }
        }
```

Log Level

Configures the logger for the EGD client used internally to the block, can be any of the following (case insensitive):

- **off** (default)
- **critical**
- **err**
- **warn**
- **info**
- **debug**

Log Name

Used in the log file to associate statements with the block. If omitted, the block name is used for the log name.

Transport Address

The host address where EGD messages are expected to be received. It is recommended to keep this field as "127.0.0.1" unless multicast is used.

Subscriptions

An array that contains JSON objects that describe a particular producer and its associated configuration server. Within each subscription, the following configuration options are supported:

- **Port**: The port on which EGD messages are expected to be received. The default is port 18246.
- **Producer ID**: Producer ID of the EGD producer whose values you want to listen for.

- **Config URL**: URL of the EGD configuration server to request new producer configurations from.
- **Variables**: An array containing objects with the names of variables you want to read from the EGD producer and optional aliases.

Note: If `variables` is not included in config, all variables from the producer will be returned to the user.

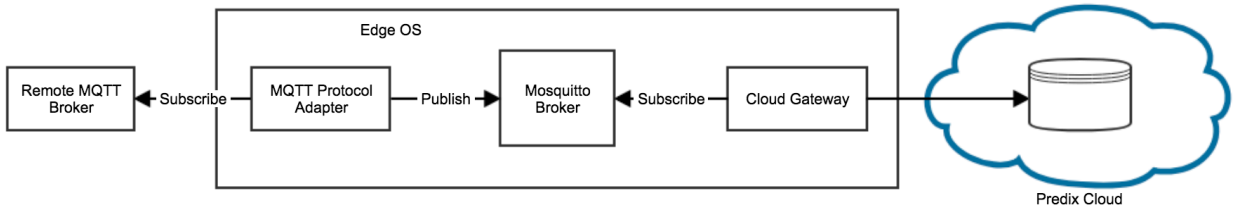
MQTT Protocol Adapter

Protocol Adapters - MQTT

The MQTT Protocol Adapter container enables the transfer of data from any MQTT broker to another using the CDP library. This includes data transfer from a remote broker to the local Predix Edge Broker, from the local broker to remote, remote to remote, or local to local.

The following diagram shows a simple use case where the data from an external device flows through the MQTT protocol adapter and is finally sent to the Predix cloud.

Figure: Protocol Adapter - MQTT



Where Do I Get the MQTT Protocol Translator Application?

The MQTT Protocol Translator App and sample configuration file in the table below are stored in Artifactory. Use the following information to ensure you can access the files.

For GE Employees

To access Artifactory downloads, those using a GE email address must first be logged into Artifactory.

Note: If you attempt to download from Artifactory without first logging into Artifactory, you will be asked to **Sign in**, which will not work.

1. Go to Artifactory.
2. Click the **Log In** button.

3. Click the **SAML SSO** icon.
4. Use your SSO to log in.
5. You can then return to the documentation link to access Artifactory.

For Predix Users

To access Artifactory links in the Predix Edge documentation, you must first create an account on predix.io. Your predix.io account sign in credentials will be used to access Artifactory.

When you click an Artifactory link, enter your predix.io username (email address) and password in Artifactory's **Sign In** dialog.

Adaptor App	Config
MQTT (AMD64/Intel64)	Sample Config

The latest version of the MQTT adapter is `protocol-adapter-mqtt-amd64:latest`.

Overview of Capabilities

Currently supported:

- Subscribe and publish to a topic on any MQTT broker endpoint.

Limitations:

- Can subscribe to only a single topic in each block in the configuration file. If multiple topics are desired, multiple blocks must be included in the configuration.

Subscribe to a Topic

When an application is subscribed to a topic on an MQTT broker, it will receive any data that is published to that topic on that broker. In order to subscribe to a topic on an MQTT broker endpoint using the MQTT Protocol Adapter, you must define a block in the configuration file of type `cdpin`. The following block configuration example will subscribe to the topic `input_data` on the broker located at `broker.ip.com:1883`.

```
"mqtt_subscriber": {
    "type": "cdpin",
    "config": {
        "transport_addr": "mqtt-tcp://broker.ip.com:1883",
        "method": "sub",
        "node_ref": "input_data",
        "log_level": "info",
```

```
        "log_name": "mqtt_subscribe_test"
    }
}
```

Publish to a Topic

When an application publishes to a topic on an MQTT broker, any subscribers of that topic will receive the published data. In order to publish to a topic on an MQTT broker endpoint using the MQTT Protocol Adapter, you must define a block in the configuration file of type `cdpout`. The following block configuration example will publish data to the topic output_data on the broker located at `broker.ip.com:1883`.

```
"mqtt_publisher": {
    "type": "cdpout",
    "config": {
        "transport_addr": "mqtt-tcp://broker.ip.com:1883",
        "method": "pub",
        "node_ref": "output_data",
        "log_level": "info",
        "log_name": "mqtt_publish_test"
    }
}
```

Authentication

Username and password authentication are supported. No authentication is also supported.

To remove authentication, remove any authentication fields (`username`, `password`) from the options object.

Username/Password

To use username/password authentication, the `username` and `password` fields need to be in the `options` object of your configuration.

```
"options": {
      "username": "<UserName>",
      "password": "<Password>"
  }
```

Configuration Properties for MQTT Protocol Adapter

The following are the configuration properties for the blocks named `mqtt_source` and `mqtt_sink` in the configuration below which have the types `cdpin` and `cdpout` respectively. Read the Generic CDP Blocks *(page 15)* section of the Protocol Adapters *(page 8)* documentation page for more information on these blocks and the fields within them.

Table 16.

Property	Type	Required	Default Value
transport_addr	String	yes	mqtt-tcp://predix-edge-broker
node_ref	String	yes	
method	String	yes	pub
log_name	String	no	
log_level	String	no	
options	Object	no	

Sample Files

`docker-compose.yml`

```
version: "3.0"

services:
  protocol_translator_mqtt:
    image: "protocol-adapter-mqtt-amd64:latest"
    environment:
      config: "/config/config.json"
    healthcheck:
      timeout: 5s
      retries: 3
      interval: 5s
    networks:
      - predix-edge-broker_net

networks:
  predix-edge-broker_net:
    external: true
```

config.json

The sample configuration below receives data from the MQTT broker located at `remote.hostname.or.ip.com` on port 1883 via the block named `mqtt_source`. This block is of type `cdpin`, the generic input CDP block. It then forwards that data directly to the local Predix Edge Broker which has the hostname `predix-edge-broker`.

```
{
    "blocks": {
        "mqtt_source": {
            "type": "cdpin",
            "config": {
                "transport_addr": "mqtt-tcp://
remote.hostname.or.ip.com:1883",
                "method": "sub",
                "node_ref": "data/temperature",
                "log_level": "info",
                "log_name": "mqtt_source_test"
            }
        },
        "mqtt_sink": {
            "type": "cdpout",
            "config": {
                "transport_addr": "mqtt-tcp://predix-edge-broker",
                "method": "pub",
                "node_ref": "remote-device0/data/temperature",
                "log_level": "info",
                "log_name": "mqtt_sink_test",
                "options": {
                    "qos": 2
                }
            }
        }
    },
    "mappings": {
        "mqtt_source:output": "mqtt_sink:input",
    }
}
```

Modbus Protocol Adapter

Where Do I Get the Modbus Protocol Translator Application?

The Modbus Protocol Translator App and sample configuration file in the table below are stored in Artifactory. Use the following information to ensure you can access the files.

For GE Employees

To access Artifactory downloads, those using a GE email address must first be logged into Artifactory.

Note: If you attempt to download from Artifactory without first logging into Artifactory, you will be asked to **Sign in**, which will not work.

1. Go to Artifactory.
2. Click the **Log In** button.
3. Click the **SAML SSO** icon.
4. Use your SSO to log in.
5. You can then return to the documentation link to access Artifactory.

For Predix Users

To access Artifactory links in the Predix Edge documentation, you must first create an account on predix.io. Your predix.io account sign in credentials will be used to access Artifactory.

When you click an Artifactory link, enter your predix.io username (email address) and password in Artifactory's **Sign In** dialog.

Adapter App	Config
Modbus (AMD64/Intel64)	Sample Config

The latest version of the OPC-UA adapter is `protocol-adapter-modbus-amd64:latest`.

Protocol Benchmarking - Modbus

The numbers in the tables below represent the ideal throughput for the data pump use case (data traveling from a protocol adapter to the Predix Edge MQTT Broker to the Cloud Gateway to Time Series).

The tests were executed in a controlled environment with one adapter running at a time, under ideal network conditions with a local time series simulator. The rate was deemed successful if there was less than a 1 percent loss rate over the course of a multi-hour test. Based on the testing, data loss rates dramatically increase as tag counts pass these thresholds.

The tested VMs were configured as follows (with a 2GHz vCPU):

Table 17. Tested VM Configurations

VM	Processors	RAM (GB)	Disk Space (GB)
VM (small)	1	1	5
VM (medium)	2	4	20
VM (large)	4	8	20

Table 18. Modbus-Poll

Environment	Period (seconds)	Acceptable Tag Rate
VM (small)	1	1350
	10	9225
VM (medium)	1	1200
	10	9525
VM (large)	1	1425
	10	11850
Predix Edge Gateway 3002	1	675
	10	4650

The number of tags per adapter does not scale with the device's performance characteristics. It is recommended to add new adapters to support higher tag counts along with corresponding physical hardware to support the additional applications.

Overview of Capabilities

Modbus communication is via a client–server technique, in which only one device (the client) can initiate transactions (called 'queries'). The other devices (the servers) respond by supplying the requested data to the client, or by taking the action requested in the query. Predix EdgeOS with the Modbus protocol adapter functions as a client device. Typical servers include programmable controllers, RTUs, DCS, I/O systems, data concentrators, flow computers and various instrumentation. The client can address individual servers, or can initiate a broadcast message to all servers. Servers return a message (called a 'response') to queries that are addressed to them individually.

The Modbus protocol establishes the format for the client's query by placing into it the device (or broadcast) address, a function code defining the requested action, any data to be sent, and an error–checking field. The server's response message is also constructed using Modbus protocol. It contains fields confirming the action taken, any data to be returned, and an error–checking field. If an error occurred in receipt of the message, or if the server is unable to perform the requested action, the server will construct an error message and send it as its response.

Currently Supported:

- Read/write: Supports reads and writes. Pseudo-subscription through polling.
- TCP and RTU (serial) communication.

Limitations:

- Writing to coils is not supported.
- Writing arrays to a series of adjacent registers is not supported.

- The Modbus adaptor does not support Modbus 20 or Modbus 21 commands.

Read

Supported function codes include:

- 01 Read Coil Status (0x register)
- 02 Read Input Status (1x register)
- 03 Read Holding Registers (4x register)
- 04 Read Input Registers (3x register)

Note: Function Code 20 Read General Reference (6x register) is not supported.

Write

Supported function codes include:

- 06 Preset Single Register
- 16 Preset Multiple Registers

Note: The following function codes are not supported:

- 21 Write General Reference (6x register)
- 05 and 15 Coil (0x register) Writes

TCP Communication

A dedicated header is used on TCP/IP to identify the Modbus Application Data Unit. It is called the MBAP header (ModBus Application Protocol header). This header provides some differences compared to the Modbus RTU application data unit used on serial line:

- The Modbus ‘server address’ field usually used on Modbus Serial Line is replaced by a single byte ‘Unit Identifier’ within the MBAP Header. The ‘Unit Identifier’ is used to communicate via devices such as bridges, routers and gateways that use a single IP address to support multiple independent Modbus end units.
- All Modbus requests and responses are designed in such a way that the recipient can verify that a message is finished. For function codes where the Modbus PDU has a fixed length, the function code alone is sufficient. For function codes carrying a variable amount of data in the request or response, the data field includes a byte count.
- When Modbus is carried over TCP, additional length information is carried in the MBAP header to allow the recipient to recognize message boundaries even if the message has been split into multiple packets for transmission. The existence of explicit and implicit length rules, and use

of a CRC-32 error check code (on Ethernet) results in an infinitesimal chance of undetected corruption to a request or response message.

RTU (Serial) Communication

The Modbus protocol defines a simple Protocol Data Unit (PDU) independent of the underlying communication layers. The mapping of MODBUS protocol on specific buses or networks can introduce some additional fields on the Application Data Unit (ADU). For Modbus Serial, the Address field only contains the slave address. The valid slave nodes addresses are in the range of 0 – 247 decimal. The individual slave devices are assigned addresses in the range of 1 – 247. A master addresses a slave by placing the slave address in the address field of the message. When the slave returns its response, it places its own address in the response address field to let the master know which slave is responding.

The function code indicates to the server what kind of action to perform. The function code can be followed by a data field that contains request and response parameters.

Configuration Properties for Modbus Protocol Adapter

Modbus specific configurations are stored in the `modbus_source` block, and there are three types: Modbus *(page 45)*, Modbus Flat *(page 46)*, and Modbus Sink Flat *(page 49)*.

The following fields are common amongst all three block types: `transport_addr`, `log_level` and `log_name`, and options.

`transport_addr`

The `transport_addr` field determines the location of the Modbus endpoint the block will communicate with. Its prefix can be any of the following:

- 'modbus-tcp'
- 'modbus-rtu'
- 'modbus-ascii'
- 'modbus-rtu-tcp'
- 'modbus-ascii-tcp'

`log_level` and `log_name`

For details about the `log_level` and `log_name` fields, see the Generic CDP Blocks *(page 15)* section of the Protocol Adapters *(page 8)* documentation page.

`options`

The options field contains any other miscellaneous configuration options necessary for the desired Modbus communication.

Modbus

Table 19.

Type
'modbus'

This block is used to source raw data from a single range of modbus registers or coils.

In addition to the configuration fields found in all three Modbus types, this configuration type has the `node_ref` and interval fields.

Table 20.

Field	Type	Required	Default
transport_addr	String	yes	
node_ref	String	yes	
interval	Integer	yes	
log_level	String		'off'
log_name	String		
options	Object		

`node_ref`

The node_ref field defines what registers to retrieve data from on the Modbus endpoint and how. It is composed of three fields:

`/Opcode/FirstRegisterToRead/NumberOfRegistersToRead`

The opcode can be any of the following:

Table 21.

Possible opcode values	Meaning
hreg	GET from holding registers
ireg	GET from input registers
bit	GET from coils

`interval`

The `interval` field determines the interval (in milliseconds) at which the block will poll its endpoint for data.

For example, if this is set to 1000, the block will attempt to get data every second.

Example Config Block

```
"modbus_source": {
    "type": "modbus",
    "config": {
        "transport_addr": "modbus-tcp://localhost:502",
        "node_ref": "/hreg/0/1",
        "interval": 1000,
        "log_level": "info",
        "log_name": "modbus_source_block",
        "options": {
            "slave_id": 0
        }
    }
}
```

Modbus Flat

Table 22.

Type
'modbusflat'

This block is used to source raw data from multiple ranges of modbus registers and transform the data into flat JSON format.

In addition to the configuration fields found in all three Modbus types, the Modbus Flat has the `data_map` and `interval` configuration fields.

Table 23.

Field	Type	Required	Default
transport_addr	String	yes	
data_map	Array	yes	
interval	Integer	yes	
log_level	String		'off'
log_name	String		
options	Object		

`data_map`

The `data_map` field defines what registers to retrieve data from on the Modbus endpoint and how to convert that data to flat JSON format. The `data_map` is an array of objects of the following structure:

Table 24.

Field	Type	Required
alias	String	yes
reg_type	String	yes
address	Integer	yes
type	String	yes if reg_type is 'holding' or 'input'
bit_index	Integer	yes if type is 'bool'
count	Integer	yes if type is 'string' or reg_type is 'coil'

The `alias` field of the `data_map` determines the name of the requested value that should mean something to the target application. Examples are “temperature” and “pressure”.

The `reg_type` field determines what register type to retrieve data from.

Possible values are:

- 'holding'
- 'input'
- 'coil'

The `address` determines the starting address of the data point. This should be any number that is a valid Modbus register.

The type determines the type of data to pull from the Modbus endpoint. This value will determine how many registers are accessed and how to combine the registers to form the requested type.

Possible values are:

- 'bool'
- 'int8'
- 'uint8'
- 'int16'
- 'uint16'
- 'int32'
- 'uint32'
- 'int64'
- 'uint64'

- 'float'
- 'double'
- 'string'

The `bit_index` is only used when type is `'bool'`. This specified which bit the boolean value is stored in.

The `count` is only used when `type` is `'string'` or `reg_type` is `'coil'`. This specifies the number of registers or bytes to read in order to form a string or byte array.

`interval`

The interval field determines the interval (in milliseconds) at which the block will poll its endpoint for data.

For example, if this is set to 1000, the block will attempt to get data every second.

Example Output Data (Flat JSON)

```
{
    "timestamp": "1504739531776",
    "data": {
        "temperature": {
            "val": 15,
            "type": "int16"
        },
        "pressure": {
            "val": 16.2,
            "type": "float"
        }
    }
}
```

Example Config Block

```
"modbus_source": {
    "type": "modbusflat",
    "config": {
        "transport_addr": "modbus-tcp://localhost:1502",
        "interval": 1000,
        "data_map": [
            {
                "alias": "valve_on",
                "reg_type": "holding",
                "address": 15,
                "type": "bool",
                "bit_index": 8
            },
            {
                "alias": "valve_status",
                "reg_type": "holding",
```

```
                "address": 16,
                "type": "string",
                "count": 15
            }
        ],
        "log_level": "debug",
        "log_name": "modbus_source_flat",
        "options": {
            "slave_id": 0
        }
    }
}
```

Modbus Sink Flat

Table 25.

Type
'modbussinkflat'

This block is used to take flat JSON formatted data and write it to any number of modbus registers on a single Modbus server.

In addition to the configuration fields found in all three Modbus types, the Modbus Sink Flat type has the `data_map`, `default_byte_order`, `first_16_bit_low`, and `first_32_bit_low` fields.

Table 26.

Field	Type	Required	Default
transport_addr	String	yes	
data_map	Array	yes	
default_byte_order	Boolean		true
first_16_bit_low	Boolean		true
first_32_bit_low	Boolean		true
log_level	String		'off'
log_name	String		
options	Object		

`data_map`

The `data_map` field defines what registers to retrieve data from on the Modbus endpoint and how to convert that data to flat JSON format. The `data_map` is an array of objects of the following structure:

Table 27.

Field	Type	Required
alias	String	yes
reg_type	String	yes
address	Integer	yes
type	String	yes if reg_type is 'holding' or 'input'
bit_index	Integer	yes if type is 'bool'
count	Integer	yes if type is 'string' or reg_type is 'coil'

The `alias` field of the `data_map` determines the name of the requested value that should mean something to the target application. Examples are “temperature” and “pressure”.

The `reg_type` field determines what register type to retrieve data from.

Possible values are:

- 'holding'
- 'input'
- 'coil'

The `address` determines the starting address of the data point. This should be any number that is a valid Modbus register.

The type determines the type of data to pull from the Modbus endpoint. This value will determine how many registers are accessed and how to combine the registers to form the requested type.

Possible values are:

- 'bool'
- 'int8'
- 'uint8'
- 'int16'
- 'uint16'
- 'int32'
- 'uint32'
- 'int64'
- 'uint64'
- 'float'
- 'double'
- 'string'

The `bit_index` is used only when type is `'bool'`. This specifies which bit the boolean value is stored in.

The `count` is used only when type is `'string'` or `reg_type` is `'coil'`. This specifies the number of registers or bytes to read in order to form a string or byte array.

`default_byte_order`

The byte order used by the Modbus adapter can be changed from the default Modbus byte ordering (big endian) to Intel byte ordering (little endian) by using this option. By default it is the normal setting for Modbus compatible devices. If the device uses Intel byte ordering, setting this option to false will enable the Modbus driver to properly read Intel formatted data.

`first_16_bit_low`

Two consecutive registers' addresses in a Modbus device are used for 32-bit data types, like Integer and Float. It can be specified whether the adapter should assume the first 16 bits is the low or the high word of the 32-bit value. The default, first word low, follows the convention of the Modicon Modsoft programming software. This is also applicable to the two 32-bit data in 64-bit data types, like Long and Double.

`first_32_bit_low`

Four consecutive registers' addresses in a Modbus device are used for 64-bit data types, like Long and Double. It can be specified whether the driver should assume the first 32 bits is the low or the high double word of the 64-bit value. The default, first 32 bits low, follows the default convention of 32-bit data types.

Note: The interaction of multi-register values (e.g., 32-bit integers, 64-bit doubles, long strings) with these settings is a little unintuitive. This is an illustration of the behavior on a 64-bit unsigned integer 0x8877665544332211 for a few configurations.

Table 28.

default_byte_order	first_16_bit_low	first_32_bit_low	transmitted as
true	true	true	[11][22][33][44][55][66][77][88]
false	true	true	[22][11][44][33][66][55][88][77]
true	false	true	[33][44][11][22][77][88][55][66]
true	true	false	[55][66][77][88][11][22][33][44]
false	true	false	[66][55][88][77][22][11][44][33]

You can think of these settings as only applying within "chunks" of the next largest size, i.e., `default_byte_order` controls byte orders within two-byte registers. `first_16_bit_low` controls byte orders within four-byte "double registers", and `first_32_bit_low` controls byte orders within

8-byte “quad registers”. The National Instruments publication “The Modbus Protocol In-Depth” contains another discussion of byte order in Modbus.

Example Config Block

```
"modbus_output": {
  "type": "modbussinkflat",
  "config": {
    "transport_addr": "modbus-rtu://<PATH_TO_DEVICE_FILE>",
    "log_level": "debug",
    "interval": 1000,
    "first_16_bit_low": false,
    "data_map": [
      {
        "aliases": [
          "modbus_byte"
        ],
        "address": 0,
        "type": "int8"
      },
      {
        "aliases": [
          "modbus_string"
        ],
        "address": 4,
        "type": "string",
        "count": 10
      }
    ]
  }
}
```

Sample Files

`docker-compose.yml`

```
version: "3.0"

services:
  modbus:
    image: "dtr.predix.io/predix-edge/protocol-adapter-modbus:amd64-1.1.0"
    environment:
      config: "/config/config-modbus.json"
    deploy:
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 5
        window: 30s
    networks:
      - predix-edge-broker_net
```

```
networks:
  predix-edge-broker_net:
    external: true
```

config.json

This sample configuration file as written will:

- read from a Modbus TCP connection at the configured host and port, polling every 1000 milliseconds, assuming the Modbus source is configured for reverse byte order within two-byte words.
- write to a Modbus RTU connection mounted as a file into the container, with the order of two-byte words inside 32-byte chunks reversed.
- write the data to the MQTT broker on the topic `out`.
- read data from the MQTT broker on the topic `in`.

```
{
  "blocks": {
    "modbus_input": {
      "type": "modbusflat",
      "config": {
        "transport_addr": "modbus-tcp://
<MODBUS_TCP_HOST>:<MODBUS_TCP_PORT>",
        "log_level": "debug",
        "interval": 1000,
        "default_byte_order": false,
        "data_map": [
          {
            "alias": "modbus_byte",
            "reg_type": "input",
            "address": 0,
            "type": "int8"
          },
          {
            "alias": "modbus_bool",
            "reg_type": "holding",
            "address": 4,
            "type": "bool",
            "bit_offset": 9
          }
        ]
      }
    },
    "flat_to_timeseries": {
      "type": "flattotimeseries",
      "config": {
        "attributes": {
          "machine_type": "modbus"
        }
```

```
      }
    },
    "mqtt_sink": {
      "type": "cdpoutqueue",
      "config": {
        "transport_addr": "mqtt-tcp://<MQTT_HOST>:<MQTT_PORT>",
        "node_ref": "predix_historian",
        "method": "pub",
        "log_level": "debug",
        "directory": "/mqtt_store",
        "max_cache_size_units": "%",
        "max_cache_size": 90
      }
    },
    "modbus_output": {
      "type": "modbussinkflat",
      "config": {
        "transport_addr": "modbus-rtu://<PATH_TO_DEVICE_FILE>",
        "log_level": "debug",
        "interval": 1000,
        "first_16_bit_low": false,
        "data_map": [
          {
            "aliases": [
              "modbus_byte"
            ],
            "address": 0,
            "type": "int8"
          },
          {
            "aliases": [
              "modbus_string"
            ],
            "address": 4,
            "type": "string",
            "count": 10
          }
        ]
      }
    },
    "mqtt_source": {
      "type": "cdpin",
      "config": {
        "transport_addr": "mqtt-tcp://<MQTT_HOST>:<MQTT_PORT>",
        "node_ref": "in",
        "method": "sub",
        "log_level": "debug"
      }
    }
  },
  "mappings": {
    "modbus_input:output": "flat_to_timeseries:input",
    "flat_to_timeseries:output": "mqtt_sink:input",
```

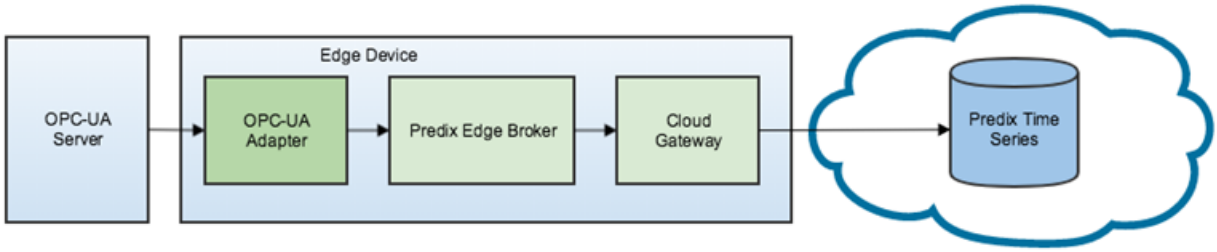
```
    "mqtt_output:output": "modbus_output:input"
  }
}
```

OPC UA Protocol Adapter

Protocol Adapters - OPC-UA

OPC Unified Architecture (OPC-UA) is an industrial communication protocol with robust security features and a complex information architecture that asset maintainers can leverage to model and store their data.

The OPC-UA protocol adapter is a Predix Edge application that allows you to communicate with your OPC-UA server. The following diagram shows a simple use case where data from an OPC-UA server is retrieved by the OPC-UA adapter and finally send to the Predix cloud.
Figure: OPC-UA Sample Use Case



Where Do I Get the OPC-UA Protocol Adapter Application?

The OPC-UA Protocol Adapter App and sample configuration file in the table below are stored in Artifactory. Use the following information to ensure you can access the files.

For GE Employees

To access Artifactory downloads, those using a GE email address must first be logged into Artifactory.

Note: If you attempt to download from Artifactory without first logging into Artifactory, you will be asked to **Sign in**, which will not work.

1. Go to Artifactory.
2. Click the **Log In** button.
3. Click the **SAML SSO** icon.
4. Use your SSO to log in.

5. You can then return to the documentation link to access Artifactory.

For Predix Users

To access Artifactory links in the Predix Edge documentation, you must first create an account on predix.io. Your predix.io account sign in credentials will be used to access Artifactory.

When you click an Artifactory link, enter your predix.io username (email address) and password in Artifactory's **Sign In** dialog.

Adapter App	Config
OPC-UA (AMD64/Intel64)	Sample Config

The latest version of the OPC-UA adapter is `protocol-adapter-opcua-amd64:latest.`

Protocol Benchmarking - OPC-UA

The numbers in the tables below represent the ideal throughput for the data pump use case (data traveling from a protocol adapter to the Predix Edge MQTT Broker to the Cloud Gateway to Time Series).

The tests were executed in a controlled environment with one adapter running at a time, under ideal network conditions with a local time series simulator. The rate was deemed successful if there was less than a 1 percent loss rate over the course of a multi-hour test. Based on the testing, data loss rates dramatically increase as tag counts pass these thresholds.

The tested VMs were configured as follows (with a 2GHz vCPU):

Table 29. Tested VM Configurations

VM	Processors	RAM (GB)	Disk Space (GB)
VM (small)	1	1	5
VM (medium)	2	4	20
VM (large)	4	8	20

Table 30. OPC-UA Poll

Environment	Period (seconds)	Acceptable Tag Rate
VM (small)	1	4200
	10	47407
VM (medium)	1	4500

Environment	Period (seconds)	Acceptable Tag Rate
	10	52983
VM (large)	1	5100
	10	53555
Predix Edge Gateway 3002	1	4500
	10	31275

The number of tags per adapter does not scale with the device's performance characteristics. It is recommended to add new adapters to support higher tag counts along with corresponding physical hardware to support the additional applications.

Overview of Capabilities

Currently supported:

- Read/write: Supports write operations. Supports polling and subscription-style read operations.
- Events: Supports reception of base event attributes.
- Communication encryption.
- Authentication:
  - username/password
  - certificates

Details of Capabilities

In both reading and writing (GET and SET), tags are addressed using OPC-UA XML Syntax:

- String NodeID form: `"ns=4;s=Foo.Bar.Baz"`
- IntegerNodeID form: `"ns=34;i=11902"`

If ns is omitted, namespace 0 is assumed: `"s=Foo.Bar.Baz"`

The value will be returned as a JSON string of the form:

```
{"val":<data>, "type":<typename>}
```

Only numeric and the STRING and DATE_TIME types are returned currently. Other data types are not supported and the value will be "UNKNOWN".

The supported types are:

- float

- double
- bool
- byte
- sbyte
- int16
- uint16
- int32
- uint32
- int64
- uint64
- string
- datetime

To be compatible with JSON style SET node refs (described below) the GET method will accept node refs starting with “/json?”. So the two node refs `/json?ns=4;i=1900` and `ns=4;i=1900` are equivalent.

The supported types in flat JSON and their corresponding versions in the OPC-UA specification are as follows:

Table 31. Supported Flat JSON Types and OPC-UA Equivalents

Flat JSON	OPC-UA
bool	boolean
int8	sbyte
int16	int16
int32	int32
int64	int64
uint8	byte
uint16	uint16
uint32	uint32
uint64	uint64
float	float
double	double
string	string
datetime	datetime

Read

OPC-UA supports both polling and subscription-style reads.

Polling (OPC-UA Poll Flat)

OPC-UA polling will read data from an OPC-UA server at a user specified frequency. Information about configuring the interval is found in the Deployment and Configuration Resources *(page 69)*.

Subscription (OPC-UA Sub Flat)

OPC-UA polling will read data from a specific topic on an OPC-UA server. Information about configuration is found in the Deployment and Configuration Resources *(page 68)*.

Write

SET supports setting simple primitive types. There are two ways to set values, using a simple binary structure or using JSON.

Events

SET supports setting simple primitive types. There are two ways to set values, using a simple binary structure or using JSON.

Communication Encryption

Certificate encrypted connections will only be active if both `encryption_cert_path` and `encryption_private_key_path` are specified in the `options` configuration object and are nonempty. The `encryption_cert_password` option can be omitted if the private key at `encryption_private_key_path` is not password-protected.

To use communication encryption, the `application_uri`, `security_mode` and `pki_root_path` must also be specified in the `options` configuration object.

Create an Encryption Certificate on a Unix-type System

To create an encryption certificate using a Unix-like system, the following command can be used:

```
openssl req \
    -new \
    -newkey rsa:2048 \
    -days 3650 \
    -keyout encryption.key \
    -subj "/C=DE/ST=/O=Organization/
CN=urn:ge.edge.research.com:GEPredix:OPCUAClient" \
    -reqexts SAN \
```

```
    -extensions SAN \
    -config <(cat /etc/ssl/openssl.cnf <(printf
 "[SAN]\nsubjectAltName=URI:urn:ge.edge.research.com:GEPredix:OPCUAClient"))
 \
    -x509 \
    -outform DER \
    -out encryption.der
```

Create an Encryption Certificate on a Windows System

1. Download OpenSSL for your Windows device (32-bit or 64-bit).
2. Navigate to where you downloaded the OpenSSL and install using the executable.
3. We first need to build the Encryption key, which uses the default `openssl.cfg` file, but we need to modify it for Predix Edge by copying it to a new config called `openssl_predix_edge.cfg` and modifying this new file. From the folder in which you want to generate your encryption certificate, open a command prompt as Administrator and run:

   ```
   copy C:\OpenSSL-Win64\bin\openssl.cfg C:\OpenSSL-Win64\bin
   \openssl_predix_edge.cfg && ^
       echo
    [SAN]\nsubjectAltName=URI:urn:ge.edge.research.com:GEPredix:OPCUAClient
    >> C:\OpenSSL-Win64\bin\openssl_predix_edge.cfg && ^
       C:\OpenSSL-Win64\bin\openssl.exe req ^
       -new ^
       -newkey rsa:2048 ^
       -days 3650 ^
       -keyout encryption.key ^
       -subj "/C=DE/ST=MA/O=Organization/
   CN=urn:ge.edge.research.com:GEPredix:OPCUAClient" ^
       -reqexts SAN ^
       -extensions SAN ^
       -config "C:\OpenSSL-Win64\bin\openssl_predix_edge.cfg" ^
       -x509 ^
       -outform DER ^
       -out encryption.der
   ```

4. Your `encryption.der` and `encryption.key` files are now in the folder and need to be used in both the Predix Edge device and the OPC-UA server.

application_uri

The application URI will be in the form `urn:URI:COMPANY:APPLICATION` where you replace the URI, COMPANY, and APPLICATION fields with your URI, company and application name (see openssl command above). This is created when creating your encryption certificate in the subj parameter of the openssl command above.

security_mode

The following are the options for the security_mode field:

- 'NONE'
- 'BASIC128RSA15_SIGN'
- 'BASIC128RSA15_SIGN_ENCRYPT'
- 'BASIC256_SIGN'
- 'BASIC256_SIGN_ENCRYPT'
- 'BASIC256SHA256_SIGN'
- 'BASIC256SHA256_SIGN_ENCRYPT'

This should match what is set on the OPC-UA server.

pki_root_path

The `pki_root_path` is where the client PKI structure is created. Default `pki_root_path`:

```
pki/
### client
#   ### revoked
#   ### trusted
### issuer
    ### certs
    ### revoked
```

The default `pki_root_path` is `/tmp/pki` but can be changed to a custom location where you'd like to store your encryption certificates.

This encryption certificate must be trusted by the server in order to form a secure connection.

Authentication

Username/password and certificate authentication are supported. Anonymous/no authentication is also supported.

Username/password authentication can only be used when certificate authentication is inactive, i.e., if all of `username`, `user_cert_path`, and `user_private_key_path` are specified, the OPC-UA transport will assume that the user wants to use certificate authentication and ignore the username option.

Certificate user authentication will only be active if both `user_cert_path` and `user_private_key_path` are specified and nonempty. The `user_cert_password` option can be omitted if the private key at `user_private_key_path` is not password-protected.

Anonymous/no authentication is also possible. To remove authentication, remove any authentication fields (`username`, `password`, `user_cert_path`, `user_cert_password`, `user_private_key_path`) from the options object.

Username/Password

To use username/password authentication, the `username` and `password` fields need to be in the `options` object of your configuration.

```
"options": {
      "username": "<UserName>",
      "password": "<Password>"
  }
```

Certificates

To use certificate authentication, the `user_cert_path`, `user_private_key_path` and `user_cert_password` fields need to be active in the `options` object of your configuration, or the `client_cert_path` and `client_private_key_path` fields need to be in that object.

```
"options":{
              "user_cert_path": "/config/client.der",
              "user_private_key_path":"/config/client.key",
              "user_cert_password":"<your password>"
           }
```

Note: The `user_cert_password` is the password used when you created your authentication certificate.

To generate an authentication certificate using a Windows machine:

1. Download OpenSSL for your Windows device (32-bit or 64-bit).
2. Navigate to where you downloaded OpenSSL and install using the executable.
3. From the folder you want to generate your authentication certificate, open a command prompt and run:

   ```
   > set RANDFILE=C:\<your folder structure>\.rnd
   > set OPENSSL_CONF=C:\OpenSSL-Win64\bin\openssl.cfg
   > C:\OpenSSL-Win64\bin\openssl.exe
   ```

4. You should now be at an **OpenSSL**> prompt, where you should run:

   ```
   req -newkey rsa:2048 -days 3650 -keyout client.key -x509 -outform DER -
   out client.der
   ```

To generate an authentication certificate on a Unix-like system, use the following command.

```
openssl req -newkey rsa:2048 -days 3650 -keyout client.key -x509 -outform
 DER -out client.der
```

Note: If you are running a simulator, be sure to copy these authentication certificates to your OPC-UA simulator.

Alarm Acknowledgement

To acknowledge an activated alarm a Set or Pub with a JSON payload of type string must be performed on the alarm object node Id. The alarm node reference must be prefixed with `/acknowledge?` followed by the standard OPC-UA identifier `ns=<idx>;[s|i]=<OPC ID>`. The string value in the payload represents the comment to be added to the acknowledgement.

```
 { "type": "string", "val": "my comment" }
```

`SET /acknowledge?ns=2;MyLevel.Alarm`

This will change `ns=2;s=MyLevel.Alarm/0:AckedState` to "Acknowledged" and `ns=2;s=MyLevel.Alarm/0:Comment` to "my comment".

To be notified of an alarm becoming active or inactive simply perform a Get or Sub on the Active State variable of the alarm object.

Example: The prosys opcua simulation server (https://www.prosysopc.com/products/opc-ua-simulation-server/) has an alarm object `ns=2;s=MyLevel.Alarm.`

Using the OPC-UA Protocol Adapter you can subscribe to the tag `ns=2;s=MyLevel.Alarm/0:ActiveState` to be notified when the alarm has been activated.

```
"data_map": [
    {
      "alias": "example_alarm",
      "id": "ns=2;s=MyLevel.Alarm/0:ActiveState"
    }
]
```

If simulation is running, an alarm will activate/deactivate approximately every 30 seconds.

Configuration Properties for OPC-UA Protocol Adapter

OPC-UA configurations are stored in the `opcua` block, and there are four different types:

- OPC-UA Sub Flat *(page 68)*
- OPC-UA Poll Flat *(page 69)*
- OPC-UA Sink Flat *(page 71)*

- OPC-UA Event *(page 59)*

The following fields are common among all four block types: `transport_addr`, `data_map`, `log_level`, `log_name`, `trace_level` and `options`.

`transport_addr`

The transport_addr field determines the location of the OPC-UA endpoint the block will communicate with. Its prefix can be any of the following:

Possible prefixes
'opc-tcp'

`data_map`

An array of objects. Each object has a type key and an alias key. The top-level key is the address of the tag to be written on the OPC-UA server, in OPC-UA XML Notation.

- The type value is the data type to be written. It corresponds to one of the types described in Key Concept - Flat JSON format.
- The alias value is an array of strings. If the string "X" is in alias, then any JSON in flat JSON format received where data.X is a key will have data.X.val written to the OPC-UA server.

Note: The `data_map` field's structure is different for the OPC-UA Sub/Poll Flat and the OPC-UA Sink Flat blocks. To see the structure for each block, see that block's section below.

`log_level` and `log_name`

For details about the log_level and log_name fields, see the Generic CDP Blocks *(page 15)* section of the Protocol Adapters *(page 8)* documentation page.

`trace_level`

The `trace_level` enables logging for the underlying client that connects to the OPC-UA device. It should be set once for the adapter and will affect every OPC-UA block (poll, subscription, etc.) in the configuration file. When used multiple times, the first `trace_level` encountered will be used. The default value is `none`.

Possible Values
`content`, `debug`, `info`, `warning`, `error`, `none`

`options`

Field	Type	Required	Default	Description
username	String	yes if using username/password authentication		Username for OPC-UA username/password authentication
password	String	yes		Password for private key in `client_private_key_path`when using certificate authentication or `username`when using username/password authentication
client_private_key_path	String			Path to the private key for OPC-UA certificate authentication. This private key should be kept secret and not moved to any other server. Interchangeable with `user_private_key_path`
client_cert_path	String			Path to the public certificate for OPC-UA certificate authentication. The file at this path should be added to your OPC-UA server's trust store. Interchangeable with `user_cert_path`.
user_cert_path	String			Path to the public certificate this client uses for user authentication. Only DER encoded files are currently supported. The certificate should match with the private key specified by user_private_key_path. Interchangeable with `client_cert_path.`
user_private_key_path	String			Path to the private key this client uses for user certificate. Only PEM encoded private keys are supported. Interchangeable with `client_private_key_path`

Field	Type	Required	Default	Description
user_cert_password	String			Password to the private key this client uses for user certificate. Only PEM encoded private keys are supported.
encryption_cert_path	String			Path to the public certificate this client uses for encrypted connection. Only DER encoded files are currently supported. The certificate should match with the private key specified by `encryption_private_key_path`.
encryption_private_key_path	String			Path to the private key this client uses for encrypted connection. Only PEM encoded private keys are supported.
encryption_cert_password	String			Password to the private key this client uses for encrypted connection. Only PEM encoded private keys are supported.
application_uri	String			The application URI for this client. Must match the URI in the client's encryption certificate.
pki_root_path	String			The path where the client PKI structure is created if encryption connection is used. Only applies if encryption certificate is used.
security_mode	String			Specifies the security policy to use for encrypted connection. Only applies if encryption certificate is used.

Field	Type	Required	Default	Description
session_timeout	Integer		1200000 (20 minutes)	Indicates how long, in milliseconds, the server will retain a connection without receiving a message from the client.
connect_timeout	Integer		5000	The length of time, in milliseconds, a client will wait for a reply to a connect request.
watchdog_interval	Integer		5000	The number of milliseconds between watchdog checks.
watchdog_timeout	Integer		5000	The maximum amount of time, in milliseconds, a client will wait for a response to a watchdog request. After the first failure, the watchdog timeout doubles; after a second failure, the connection ends.
publishing_interval	Integer		1000	Controls how often, in milliseconds, the client will check for available data (to which the client is subscribed). **Note:** Used only by blocks that support subscription.
lifetime_count	Integer		1200	Controls the lifetime, in milliseconds, of the subscription (as opposed to the lifetime of the session). The lifetime of the subscription is set to **publishing interval** * **lifetime count**. The default is 1200*1000 (20 minutes). **Note:** Used only by blocks that support subscription.

Field	Type	Required	Default	Description
max_batch_size	Integer		200	The maximum number of tags that can be subscribed to in a single network request. For example, an adapter with 1000 tags and a max_batch_size of 200 would send five network requests. **Note:** Used only by blocks that support subscription.
max_nodes_per_sub	Integer		800	The maximum number of subscription tags that can be associated with a single subscription. Once this limit is reached, a new subscription is created. **Note:** Valid only for OPC-UA adapters that subscribe to tags (otherwise the option is ignored).

OPC-UA Sub Flat

Table 32.

Type
'opcuasubflat'

The OPC-UA Sub Flat adapter block is used to subscribe to a number of OPC-UA Variable nodes in order to receive any data as it changes, and convert that data to flat JSON format.

The following table details the fields of this block's `config` object. For details on what any common fields mean, see Configuration Properties for OPC-UA Protocol Adapter *(page 63)*.

Table 33.

Field	Type	Required	Default
transport_addr	String	yes	

Field	Type	Required	Default
data_map	Array	yes	
log_level	String		'off'
log_name	String		
options	Object		
report_bad_quality	Boolean		false

The `data_map` field is an array that contains objects with two fields: `alias` and `id`.

- The `alias` field gives that node an alias to be used in the flat JSON output data. This alias becomes the Time Series Tag if the data is converted to Time Series Format using the Flat to Time Series conversion block.
- The `id` field determines the Node ID of the node on the OPC-UA server.

`report_bad_quality`

When set to 'true', the `report_bad_quality` field is used to send bad quality data about the subscribed tags to the timeseries when connectivity to the OPC UA server is lost. The value of the tags would be set to 'NULL' and the data is sent exactly once to the timeseries. When connectivity is restored, normal data is sent again. If this field is set to 'false', then no data would be sent to timeseries when connectivity to the OPC UA server is lost.

Example Config Block

```
"opcua": {
      "type": "opcuasubflat",
      "config": {
          "transport_addr": "opc-tcp://<OPCUA_HOST>:<OPCUA_PORT>",
          "log_level": "debug",
          "report_bad_quality": false,
          "data_map": [
              {
                "alias": "Integration.App.Device1",
                "id": "ns=5;s=Counter1"
              }
          ]
      }
}
```

OPC-UA Poll Flat

Table 34.

Type
'opcuapollflat'

OPC-UA Poll flat also requires an additional field: `interval`.

The OPC-UA Poll Flat adapter block is used to poll a number of OPC-UA Variable nodes to retrieve their data at a specified polling interval and convert that data to flat JSON or Predix Timeseries format.

The following table details the fields of this block's `config` object. For details on what any common fields mean, see Configuration Properties for OPC-UA Protocol Adapter *(page 63)*.

Table 35.

Field	Type	Required	Default
transport_addr	String	yes	
data_map	Array	yes	
interval	Integer	yes	
log_level	String		'off'
log_name	String		
options	Object		
report_bad_quality	Boolean		false
source_timestamp	Boolean		true
output_format	String		'flat_json'

The `data_map` field is an array that contains objects with two fields: `alias` and `id`.

- The `alias` field gives that node an alias to be used in the flat JSON output data. This alias becomes the Time Series Tag if the data is converted to Time Series Format using the Flat to Time Series conversion block.
- The `id` field determines the Node ID of the node on the OPC-UA server.

`interval`

The `interval` field determines the interval (in milliseconds) at which the block will poll its endpoint for data. For example, if this is set to 1000, the block will attempt to get data every second.

`report_bad_quality`

When set to 'true', the `report_bad_quality` field is used to send bad quality data about the polled tags to the timeseries when connectivity to the OPC UA server is lost. The value of the tags would be set to 'NULL' and the data is sent to the timeseries at every polling interval. When connectivity

is restored, normal data is sent again. If this field is set to 'false', then no data would be sent to the timeseries when connectivity to the OPC UA server is lost.

`source_timestamp`

When set to 'true', the `source_timestamp` field uses the timestamp of the data from the OPC UA server. If it is set to 'false', the timestamp from the Predix EDGE device is used. The caveat here is that `report_bad_quality` takes precedence. When `report_bad_quality` is set to 'true', the `source_timestamp` field is internally forced to 'false', regardless of the value set by the user. The value of the `source_timestamp` field set by the user is honoured only when `report_bad_quality` is set to 'false'.

`output_format`

The `output_format` field takes a string parameter and can be either `flat_json` or `time_series`.

Example Config Block

```
"opcua_input": {
    "type": "opcuapollflat",
    "config": {
        "transport_addr": "opc-tcp://<OPCUA_HOST>:<OPCUA_PORT>",
        "log_level": "debug",
        "interval": 1000,
        "report_bad_quality": false,
        "source_timestamp": false,
        "output_format": "flat_json",
        "options": {
            "username": "user",
            "password": "pass",
            "client_cert_path": "/config/client.der",
            "client_private_key_path": "/config/keys/client.key"
        },
        "data_map": [
            {
                "alias": "SampleValue",
                "id": "ns=1;s=SampleValue"
            }
        ]
    }
}
```

OPC-UA Sink Flat

Table 36.

Type
'opcuasinkflat'

The OPC-UA Sink Flat adapter block is used to write data to a number of OPC-UA Variable nodes. It expects input data to be in the flat JSON format.

The following table details the fields of this block's `config` object. For details on what any common fields mean, see Configuration Properties for OPC-UA Protocol Adapter *(page 63)*.

Table 37.

Field	Type	Required	Default
transport_addr	String	yes	
data_map	Array	yes	
log_level	String		'off'
log_name	String		
options	Object		

The `data_map` field is an object of the form:

```
{
    "node_id_1": {
        "type": "<type>",
        "aliases": ["alias1.1", "alias1.2"]
    },
    "node_id_2": {
        "type": "<type>",
        "aliases": ["alias2"]
    }
}
```

The `node_id_*` field names should correspond to the Node IDs of the OPC-UA nodes and should be in [OPC-UA XML notation] format.

The value of the `type` field should be one of the data types supported by flat JSON (listed in Details of Capabilities *(page 57)*), except for the `datetime` type.

The value of the `aliases` field is an array of keys into the `data` field of the input flat JSON. This field is how the block knows which datapoints in the input data are to be sent to which OPC-UA node.

Example Config Block

```
"opcua_sink": {
    "type": "opcuasinkflat",
```

```
    "config": {
        "transport_addr": "opc-tcp://<OPCUA_HOST>:<OPCUA_PORT>",
        "log_level": "debug",
        "data_map": {
            "ns=5;Counter1": {
                "type": "int32",
                "aliases": [
                    "Integration.App.Device1.RASP1"
                ]
            }
        }
    }
}
```

OPC-UA Events

Table 38.

Type
'opcuaevent'

The OPC-UA Event adapter block is used to subscribe to an OPC-UA event node to receive base event notifications and convert them into JSON format. The following table details the fields of this block's config object. For details on what any common field means, see Configuration Properties of OPC-UA Protocol Adapter *(page 63)*.

Table 39.

Field	Type	Required	Default
transport_addr	String	Yes	
event_node	Array	Yes	
event_attributes	Array	Yes	
log_level	String	No	'off'
log_name	String	No	<block_name>
interval [milliseconds]	Integer	No	1000
options	String	No	

The `event_node` field is an array with one single entry. The entry contains two fields: `alias` and `id`:

- The `alias` field assigns that node an alias to be used in the flat JSON output data. The alias is prepended to the event attribute name and becomes the Time Series Tag if the data is converted to Time Series Format using the Flat to Time Series conversion block.

- The `id` field specifies the Node ID of the node on the OPC-UA server. The standard node ID for events defined by OPC is “ns=0;i=2253”.

The OPC-UA defined base event attributes are: "SourceNode", "SourceName", "Time", "EventId", "EventType", "LocalTime", "Message", "ReceiveTime", "Severity".

Note that the event attributes are case sensitive. Undefined names or duplicates are silently discarded.

The `interval` defines the retransmission interval for subscription requests from the client after connection to the OPC-UA server has been established.

Example Config Block

```
"opcua": {
      "type": "opcuaevent",
      "config": {
          "transport_addr": "opc-tcp://<OPCUA_HOST>:<OPCUA_PORT>",
          "log_level": "debug",
          "event_node": [
              {
                "alias": "Server-Prosys",
                "id": "ns=0;i=2253"
              }
          ],
          "event_attributes": ["SourceNode", "SourceName", "Time",
                               "EventId", "EventType", "LocalTime",
                               "Message", "ReceiveTime", "Severity"]
      }
}
```

The following is an example JSON output event (line formatted for simple display) from alias “Server-Prosys” and attributes “Message” and “Severity”:

```
{"data": {
     "Server-Prosys.Message":{"type":"string","val":"Level exceeded"},
     "Server-Prosys.Severity":{"type":"uint16","val":500}
    },
 "timestamp":1551250457139}
}
```

Sample Files

docker-compose.yml

```
version: "3.0"

services:
```

```
  opcua:
    image: "protocol-adapter-opcua-amd64:latest"
    environment:
      config: "/config/config-opcua.json"
    healthcheck:
      timeout: 5s
      test: exit 0
      retries: 3
      interval: 5s
    networks:
      - predix-edge-broker_net
    deploy:
      restart_policy:
        condition: on-failure

networks:
  predix-edge-broker_net:
    external: true
```

config.json

This sample config file will:

- Log in using certificate authentication and simply ignore the username parameter. The password parameter will be used as the passphrase to the private key at private_key/client.key.
- Use subscriptions to read data from the OPC-UA server.
- Write the data to the MQTT broker on the topic out.
- Read data from the MQTT broker on the topic in.

```
{
  "blocks": {
    "opcua_input": {
      "type": "opcuasubflat",
      "config": {
          "transport_addr": "opc-tcp://<OPCUA_HOST>:<OPCUA_PORT>",
          "log_level": "debug",
          "options": {
            "username": "user",
            "password": "pass",
            "client_cert_path": "/config/client.der",
            "client_private_key_path": "/config/keys/client.key"
          },
          "data_map": [
              {
                "alias": "SampleValue",
                "id": "ns=1;s=SampleValue"
              }
          ]
      }
    },
```

```
    "flat_to_timeseries": {
      "type": "flattotimeseries",
      "config": {
        "attributes": {
          "machine_type": "opcua"
        }
      }
    },
    "mqtt_sink": {
      "type": "cdpoutqueue",
      "config": {
        "transport_addr": "mqtt-tcp://<MQTT_HOST>",
        "node_ref": "out",
        "method": "pub",
        "log_level": "debug",
        "log_name": "opcua_mqtt_sink",
        "directory": "/mqtt_store",
        "max_cache_size_units": "%",
        "max_cache_size": 90
      }
    },
    "mqtt_source": {
      "type": "cdpin",
      "config": {
        "transport_addr": "mqtt-tcp://<MQTT_HOST>:<MQTT_PORT>",
        "node_ref": "in",
        "method": "sub",
        "log_name": "gateway_mqtt_source",
        "log_level": "debug"
      }
    },
    "opcua_output": {
      "type": "opcuasinkflat",
      "config": {
        "transport_addr": "opc-tcp://<OPCUA_SOURCE>:<OPCUA_PORT>",
        "log_level": "debug",
        "options": {
          "username": "user",
          "password": "pass",
          "client_cert_path": "/config/client.der",
          "client_private_key_path": "/config/keys/client.key"
        },
        "data_map": {
          "ns=1;s=SampleValue": {
            "type": "int8",
            "aliases": ["SampleValue"]
          }
        }
      }
    }
  }
 },
  "mappings": {
```

```
    "opcua_input:output": "flat_to_timeseries:input",
    "flat_to_timeseries:output": "mqtt_sink:input",
    "mqtt_source:output": "opcua_output:input"
 }
}
```

Command Handler Block

The OPC-UA Command Handler adapter block is capable of receiving requests from a topic on the Predix Edge Broker, sending those requests to an OPC-UA server, and returning the results from those requests back to a topic on the Predix Edge Broker.

This block's "type" field is "opcuacommandhandler", and an example configuration file that includes the block is shown later in this document.

The following diagrams display the steps of operation for a single request to the OPC-UA Command Handler block in the OPC-UA Protocol Adapter.
Figure: OPC-UA Command Handler Operation Steps

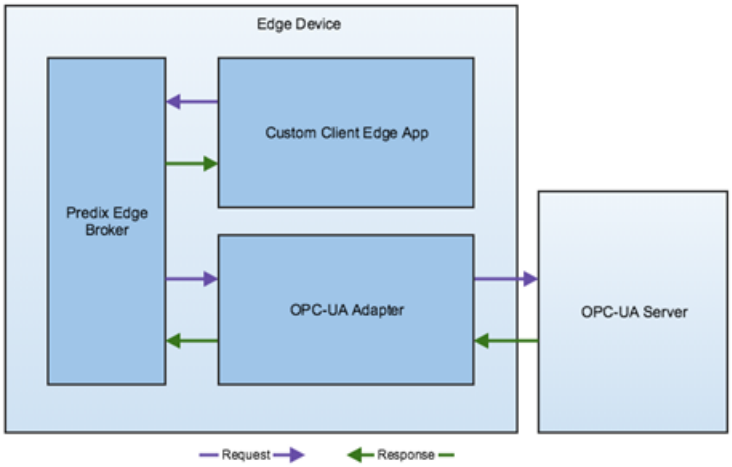
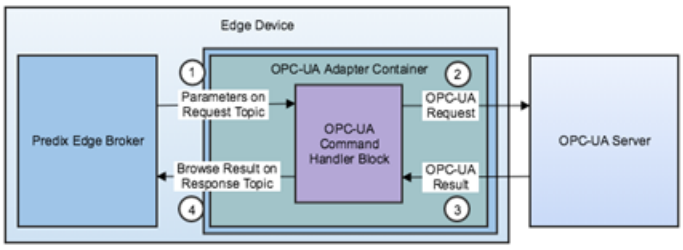


Figure: OPC-UA Command Handler Operation Steps



First, a request is sent to the Predix Edge Broker on the request topic to which the OPC-UA Command Handler block is subscribed. In the above diagram, the sender of this request is the

“Custom Client Edge App”. The request is then translated to the OPC-UA-specific request that is sent to the OPC-UA server. The result is processed and published to the Predix Edge Broker on the corresponding response topic. Any application subscribed to the response topic will then receive the result of the request. The response topic is based off of the request topic and is described in more detail below.

Request and Response Topics

Request and response topics are of the following format respectively:

- `/edgeAgent/predix-edge-opc-ua-browser/<request_id>/request`
- `/edgeAgent/predix-edge-opc-ua-browser/<request_id>/response`

Where `<request_id>` is a unique identifier for the request that is set by whichever application is sending the request. The application that sends the request should first subscribe to the response topic with the same `<request_id>` in order to ensure it receives the response successfully.

Base Input Request

The following is the format of a JSON input request to the OPC-UA Command Handler block.

```
{
 "command": "<command type>",
 "params": {
  <command-specific parameters>
 }
}
```

- `Command` is a string that determines the type of request the block will execute.
- `Params` is an object that contains specific arguments for the given command.

Example Configuration

The OPC-UA Command Handler block’s `type` field is `opcuacommandhandler`. An example of its configuration is:

```
{
 "blocks": {
  "opcua-command-handler": {
   "type": "opcuacommandhandler",
   "config": {
    "broker_address": "mqtt-tcp://predix-edge-broker",
    "log_name": "opcua-handler",
    "log_level": "debug"
   }
  }
 },
 "mappings": {}
```

```
}
```

The `config` object for a block of type `opcuacommandhandler` contains the following fields.

Field	Type	Required	Default
log_name	String	No	<block name>
log_level	String	No	'off'
broker_address	String	No	'mqtt-tcp://predix-edge-broker'

The `log_name` and `log_level` fields are consistent with other blocks and configure how log messages will be printed The `broker_address` field specifies the address of the MQTT broker from which it will receive requests.

Note: As no fields of the `config` object are required, the `config` field itself is not required. However, the block and its `type` field still must be included in the configuration file.

OPC-UA Browse Requests

The OPC-UA specification allows you to browse the child nodes of a given starting node on an OPC-UA server. The OPC-UA Command Handler adapter block allows you to do this using the BrowseServer command.

Browse Input Request

The following is an example JSON input request.

```
{
 "command": "BrowseServer",
 "params": {
  "endpoint": "opc-tcp://your-opcua-server"
  "nodeid": "ns=0;i=84",
  "depth": 1
 }
}
```

- `command` is a string that determines what type of request the block will execute. In this example, `BrowseServer` corresponds to a browse request.
- `params` is an object that contains the arguments for the command.
- `endpoint` is a string that determines which server to connect to for the request.
- `nodeid` is a string that specifies at which OPC-UA node to start the browse.
- `depth` is an integer that specifies how many levels deep to browse nodes. If the request's `depth` field is 2, the result will contain the starting node, its children, and its children's children.

Browse Output Response

The following is an example JSON output response.

```
{
 "nodeClass": "Object",
 "identifier": "i=84",
 "displayName": "Root",
 "children": [
  {
   "nodeClass": "Object",
   "identifier": "i=85",
   "displayName": "Objects",
   "children": []
  },
  {
   "nodeClass": "Object",
   "identifier": "i=86",
   "displayName": "Types",
   "children": []
  },
  {
   "nodeClass": "Object",
   "identifier": "i=87",
   "displayName": "Views",
   "children": []
  }
 ]
}
```

For each node, the result will contain four fields: “

- `nodeClass`: corresponds to the OPC-UA spec’s `NodeClass` attribute. This essentially specifies the type of the node.
- `identifier`: corresponds to the Node ID
- `displayName`: provides a name for the node to be displayed in a more readable format.
- `children`: a list of the child nodes of the current node. This list can be empty either if the node has no children, or if the `depth` field in the input request has been reached.

OPC-UA Node Attributes Requests

All nodes in an OPC-UA server contain some list of attributes, and this list varies with the type of node – i.e., the `NodeClass` of the node. The OPC-UA Command Handler adapter block allows you to query for the values of this list of attributes using the GetNodeAttributes command.

Node Attributes Input Request

The following is an example JSON input request.

```
{
 "command": "GetNodeAttributes",
```

```
 "params": {
  "endpoint": "opc-tcp://your-opcua-server"
  "nodeid": "ns=2;s=Counter1",
 }
}
```

- `command` is a string that determines what type of request the block will execute. In this example, `GetNodeAttributes` corresponds to a node attributes request.
- `params` is an object that contains the arguments for the command.
- `endpoint` is a string that determines which server to connect to for the request.
- `nodeid` is a string that specifies from which OPC-UA node to retrieve the attributes.

Node Attributes Output Response

The following is an example JSON output response.

```
{
 "NodeId": "ns=2;s=Counter1",
 "NodeClass": "Variable",
 "BrowseName": "Counter1",
 "DisplayName": "Counter1",
 "WriteMask": 0,
 "UserWriteMask": 0,
 "Value": 42,
 "DataType": "i=6",
 "ValueRank": -1,
 "AccessLevel": 3,
 "UserAccessLevel": 3,
 "MinimumSamplingInterval": 0,
 "Historizing": false
}
```

The fields of the response will differ depending on the type of node being queried and the attributes set on that node. A list of possible attributes an OPC-UA node can have is available here. The names of the fields in the response will directly correspond to the names of the attributes in that link.

OSI PI Protocol Adapters

Where Do I Get the OSI-Pi Protocol Translator Application?

The OSI-PI Protocol Translator App and sample configuration file in the table below are stored in Artifactory. Use the following information to ensure you can access the files.

For GE Employees

To access Artifactory downloads, those using a GE email address must first be logged into Artifactory.

Note: If you attempt to download from Artifactory without first logging into Artifactory, you will be asked to **Sign in**, which will not work.

1. Go to Artifactory.
2. Click the **Log In** button.
3. Click the **SAML SSO** icon.
4. Use your SSO to log in.
5. You can then return to the documentation link to access Artifactory.

For Predix Users

To access Artifactory links in the Predix Edge documentation, you must first create an account on predix.io. Your predix.io account sign in credentials will be used to access Artifactory.

When you click an Artifactory link, enter your predix.io username (email address) and password in Artifactory's **Sign In** dialog.

Adapter App	Config
OSI-Pi (AMD64/Intel64)	Sample Config

The latest version of the OSI-PI adapter is `protocol-adapter-osipi:amd64-latest`.

Protocol Benchmarking - OSI-Pi

The numbers in the tables below represent the ideal throughput for the data pump use case (data traveling from a protocol adapter to the Predix Edge MQTT Broker to the Cloud Gateway to Time Series).

The tests were executed in a controlled environment with one adapter running at a time, under ideal network conditions with a local time series simulator. The rate was deemed successful if there was less than a 1 percent loss rate over the course of a multi-hour test. Based on the testing, data loss rates dramatically increase as tag counts pass these thresholds.

The tested VMs were configured as follows (with a 2GHz vCPU):

Table 40. Tested VM Configurations

VM	Processors	RAM (GB)	Disk Space (GB)
VM (small)	1	1	5
VM (medium)	2	4	20

VM	Processors	RAM (GB)	Disk Space (GB)
VM (large)	4	8	20

Table 41. OSI-Pi Poll

Environment	Period (seconds)	Acceptable Tag Rate
VM (small)	1	150
	10	2000*
VM (medium)	1	225
	10	2000*
VM (large)	1	225
	10	2000*
Predix Edge Gateway 3002	1	525
	10	2000*

* This value represents the limit to which testing was conducted, rather than a limit that indicates increased tag counts would result in a 1 percent loss rate.

The number of tags per adapter does not scale with the device's performance characteristics. It is recommended to add new adapters to support higher tag counts along with corresponding physical hardware to support the additional applications.

Overview of Capabilities

Currently supported:

- Read. Supports subscription-style polling.
- Authentication (username/password).

Details of Capabilities

The PI Web API is a RESTful interface to the PI system. It gives client applications access to their data over HTTPS.

GE Digital Engineering has verified that the OSI-PI Protocol Adapter will function with PI Data Archive version 3.4.395.64 or later AND PI AF 2014R2 2.6.x or later. Please note that if you are running PI Data Archive 3.4.395.64 or later, having the updated PI AF 2014R2 is required.

Read

The PI Web API returns a JSON from the API using the web_id configured in the data_map configuration field *(page 84)*. The JSON is a string of data that is then parsed to get the value and timestamp.

Authentication

The API is over an HTTPS protocol. There is an option to set up username/password authentication to allow acces to the PI Web API, and there is an option to set up user accounts so certain users have access only to certain data.

Configuration Details

There are three different OSI-PI configuration types:

- `native`: Produces output in OSI-PI JSON format
- `time_series`: Produces output in Predix Time Series JSON format
- `flat_json`: Produces flat JSON output

All three configuration types have the same fields in their config:

Table 42.

Field	Type	Required	Default
transport_addr	String	Yes	
data_map	Array	Yes	
log_level	String	No	'off'
interval_ms	Int	No	1000
username	String	No	
password	String	No	
proxy_url	String	No	$https_proxy
validate_certs	Bool	No	True
output_format	String	No	flat_json

`transport_addr`

The `transport_addr` field determines the location of the OPC-UA endpoint the block will communicate with. this can be any valid web URL.

`data_map`

The `data_map` field defines the registers to retrieve data from on the Modbus endpoint and how to convert that data to flat JSON format. The `data_map` is an array of objects of the following structure:

Table 43.

Field	Type	Required
alias	String	yes
webid	String	yes
piPointName	String	yes

The `alias` field of the `data_map` determines the name of the requested value that should mean something to the target application. Examples are “temperature” and “pressure”.

The `webid` field of the `data_map` is the id of the data tag you are trying to read in the server.

The `piPointName` field of the `data_map` is the human readable name of the data tag you are trying to read in the server.

Either the `webid` or the `piPointName` must be specified in the configuration along with the `alias`. If both `webid` and the `piPointName` are specified, the `piPointName` is ignored.

`log_level` and `log_name`

For details about the `log_level` and `log_name` fields, see the Generic CDP Blocks *(page 15)* section of the Protocol Adapters *(page 8)* documentation page.

`interval_ms`

The `interval_ms` field determines the interval (in milliseconds) at which the block will poll its endpoint for data. The default is 1000.

`username`

The `username` field is the username for the PiWebApi endpoint. The default is none.

`password`

The `password` field is the username for the PiWebApi endpoint. The default is none.

`proxy_url`

The `proxy_url` field determines the proxy address used to connect to the PiWebApi endpoint. The default is the environment variable `$https_proxy`.

`validate_certs`

The `validate_certs` field determines whether the adapter will validate the certificates of the PiWebApi endpoint. Use this field if your PiWebApi does not have a valid ceritificate. The default is an empty string.

`output_format`

The `output_format` field determines the output format of the data retrieved from the PiWebApi.

- `flat_json` will return the data in flat JSON format.
- `time_series` will return the data in Predix Time Series format.
- `native` will return the data in the native PiWebApi JSON format.

Sample Files

docker-compose.yml

```
version: "3.0"

services:
  opcua:
    image: "protocol-adapter-osipi:amd64-latest"
    environment:
      config: "/config/config-osipi.json"
    healthcheck:
      timeout: 5s
      test: exit 0
      retries: 3
      interval: 5s
    networks:
      - predix-edge-broker_net
    deploy:
      restart_policy:
        condition: on-failure

networks:
```

```
  predix-edge-broker_net:
    external: true
```

config.json

```
{
 "blocks": {
  "osipi_input": {
   "type": "osipipollingsource",
   "config": {
    "transport_addr": "osipi-https://<OSIPI server address>",
    "log_level": "debug",
    "data_map": [
     {
      "alias": "tag1",
      "webid":
 "F1DPscOFnmu2m0yOXeU7eqsGfQWQEAAAU0pDMURQUFQwM1xPUEMgVUEuT1BDLVVBIFNJTVVMQVRPUi4yLlNJTV
     },
     {
      "alias": "tag2",
      "piPointName": "\\\\test_osipiserver\\test_tagname_2"
     },
     {
      "alias": "tag3",
      "webid":
 "F1DPscOFnmu2m0yOXeU7eqsGfQWgEAAAU0pDMURQUFQwM1xPUEMgVUEuT1BDLVVBIFNJTVVMQVRPUi4yLlNJTV
      "piPointName": "\\\\test_osipiserver\\test_tagname_3"
     },
     {
      "alias": "tag4",
      "piPointName": "test_tagname_4"
     },
     {
      "alias": "tag5",
      "webid":
 "F1DPscOFnmu2m0yOXeU7eqsGfQ4gAAAAU0pDMURQUFQwM1xPUEMgVUEuT1BDLVVBIFNJTVVMQVRPUi4yLlNJTV
     }
    ],
    "password": "PEASPassword",
               "username": "PEASTeam",
               "proxy_url": "$https_proxy",
               "validate_certs": false,
               "interval_ms": 1000,
               "output_format": "flat_json"
   }
  },
  "flat_to_timeseries": {
   "type": "flattotimeseries",
   "config": {
    "log_level": "debug"
   }
  },
```

```
  "mqtt_sink": {
   "type": "cdpout",
   "config": {
    "transport_addr": "mqtt-tcp://predix-edge-broker",
    "node_ref": "osipi_data",
    "method": "pub",
    "log_level": "debug"
}
  }
 },
 "mappings": {
  "osipi_input:output": "flat_to_timeseries:input",
  "flat_to_timeseries:output": "mqtt_sink:input"
 }
}
```

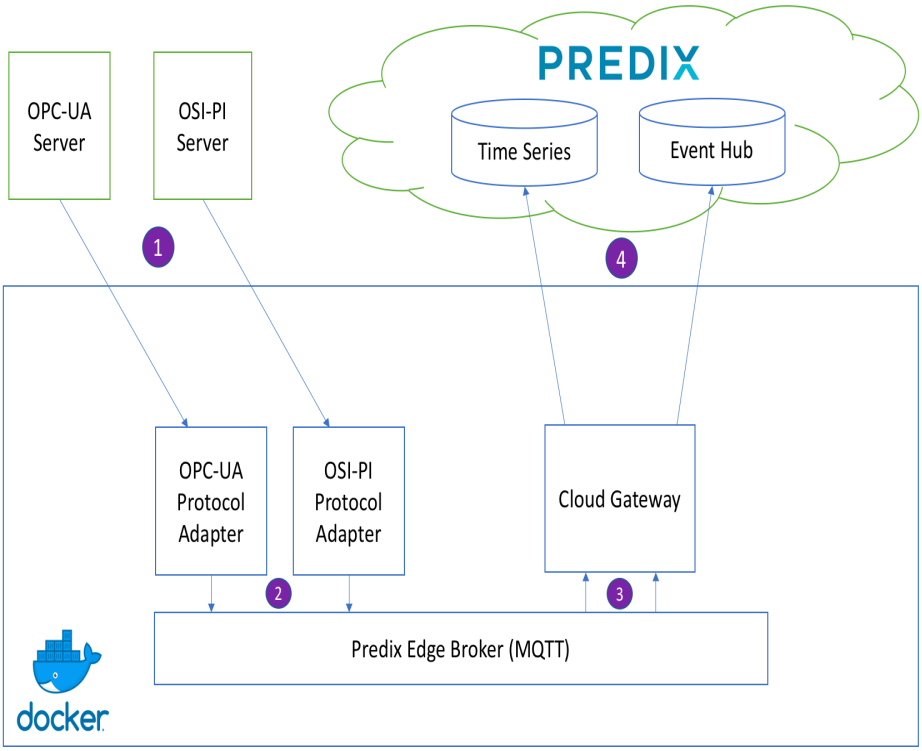
Predix Edge Cloud Gateways

About Predix Edge Cloud Gateway

The Cloud Gateway Edge App provides functionality to publish both to Time Series as well as Event Hub from one container. It also features the ability to publish to multiple Time Series or Event Hub instances simultaneously, and it provides detailed logging among other features.

The Cloud Gateway is your single solution to get data from the Edge to the Cloud. This low footprint Edge App can subscribe to multiple topics on a MQTT Broker and forward that data to both Predix Time Series and Predix Event Hub simultaneously. The client ID field is passed directly to the MQTT client, and must be unique across all applications connecting to the same broker. For more information, refer to the Mosquitto MQTT broker documentation.

The following diagram shows a simple use case where data from multiple external data sources is forwarded to the Predix Edge Broker via Protocol Adapters *(page 8)* and then to Time Series and Event Hub using the single Cloud Gateway.



Where Do I Get It?

The Cloud Gateway Edge App and sample configuration in the table below are stored in Artifactory. Use the following information to ensure you can access the files.

For GE Employees

To access Artifactory downloads, those using a GE email address must first be logged into Artifactory.

Note: If you attempt to download from Artifactory without first logging into Artifactory, you will be asked to **Sign in**, which will not work.

1. Go to Artifactory.
2. Click the **Log In** button.
3. Click the **SAML SSO** icon.
4. Use your SSO to log in.
5. You can then return to the documentation link to access Artifactory.

For Predix Users

To access Artifactory links in the Predix Edge documentation, you must first create an account on predix.io. Your predix.io account sign in credentials will be used to access Artifactory.

When you click an Artifactory link, enter your predix.io username (email address) and password in Artifactory's **Sign In** dialog.

Application	Config
Pre-Packaged Cloud Gateway (AMD64/Intel64)	Sample Config

Overview of Capabilities

Currently Supported

- Receive data from multiple configured MQTT topics.
- Publish data to:
  - Predix Time Series service (one or more instances).
  - Predix Event Hub service (one or more instances).
  - Both services simultaneously.
- Persist any data that fails to publish in the Time Series Publisher and re-transmit in the background.
- Store (on disk) and forward all data in the Event Hub Publisher.
- Predix UAA Authentication with both the Time Series and Event Hub services.
- Coordination of disk usage between blocks.
- Time Series data validation.
- Detailed logging.

Limitations

- The Cloud Gateway does not transform input data to Predix Time Series format. If the Time Series Publisher is used, it will discard any input data not formatted correctly for Time Series ingestion.
- Any Event Hub topics the Cloud Gateway is configured to publish to must be created in your Predix Event Hub instance *(page)* in advance. The Event Hub Publisher does not have the capability to create Event Hub topics on the fly due to potential security risks.
- There is no guarantee of the order in which publish requests will be reflected in their respective cloud endpoints.

Time Series Publisher Capabilities

The Time Series Publisher block can subscribe to multiple MQTT topics, send data to Predix Time Series, and it features detailed logging and input data validation.

See The Blocks Section *(page 94)* for an explanation of what a "block" is in the Cloud Gateway.

See Time Series Publisher Block Config *(page 95)* for an explanation of how to configure the Time Series Publisher block.

Time Series Publishing

The Cloud Gateway can send data to any Predix Time Series instance that the device has ingestion permissions for. The Cloud Gateway will automatically authenticate with the Time Series instance it is configured to communicate with as long as its access token has the appropriate Predix Time Series scopes *(page)*.

It is also capable of publishing to multiple Predix Time Series instances simultaneously if it has valid permissions for each.

The Time Series Publisher block batches data up before it sends to Time Series to minimize the number of network requests required. You can also configure the maximum interval at which that these batches will be sent.

Note: Any data received from MQTT subscriptions must already match the Predix Time Series data format shown in the Data Ingestion Request example *(page)*. If data is not properly formatted, the Cloud Gateway will throw that data out.

Time Series MQTT Subscriptions

This block is capable of subscribing to multiple MQTT topics rather than just one at a time. The data from all topics specified in a single Time Series Publisher block will all be forwarded to that block's configured Predix Time Series endpoint.

Note: The Time Series Publisher block does NOT currently support wildcard MQTT subscriptions (e.g., `data/#`). Any wildcard subscriptions will not be properly forwarded to Predix Time Series.

This block also supports an optional, configurable quality of service and client ID for its MQTT subscriptions.

Note: One big difference between this block and its `timeseriessink` predecessor in the old Time Series Cloud Gateway is that this block directly subscribes to MQTT topics. The `cdpin` block should NOT be used in this Cloud Gateway to create MQTT subscriptions.

Note: Any data received from MQTT subscriptions must already match the Predix Time Series data format shown in the Data Ingestion Request example *(page)*. If data is not properly formatted, the Cloud Gateway will throw that data out.

Time Series Data Storage

The Time Series block stores data on disk only if the publish to Predix Time Series fails or data comes in too quickly to store in memory.

`Store on Failure` uses in-memory queueing and multi-threaded publishing to increase the throughput of the application in situations where data is accumulating faster than the `max batch interval` timer. Data is only stored persistently if the publish request to Time Series fails or if input data is received much more quickly than it can be sent.

For information on the `policy` field in the Cloud Gateway configuration file, see Time Series Publisher Block Config *(page 95)*.

Event Hub Publisher Capabilities

The Event Hub Publisher block can subscribe to multiple MQTT topics, store your data on disk before you forward it to the cloud, send data to Predix Event Hub, and features detailed logging.

See The Blocks Section *(page 94)* for an explanation of what a `block` is in the Cloud Gateway.

See Event Hub Publisher Block Config *(page 98)* for an explanation of how to configure the Event Hub Publisher block.

Event Hub Publishing

The Cloud Gateway can send data to any Predix Event Hub instance that the device has publish permissions for. The Cloud Gateway will automatically authenticate with the Event Hub instance it is configured to communicate with as long as its access token has the appropriate scopes. It will also format publish requests automatically using gRPC, so there is no need to pre-format data before sending it to the Event Hub Publisher block.

It is also capable of publishing to multiple Predix Event Hub instances simultaneously if it has valid permissions for each.

The Event Hub Publisher block batches data up before it sends to Predix Event Hub to minimize the number of network requests required. You can also configure the maximum interval at which that these batches will be sent.

Event Hub MQTT Subscriptions

This block is capable of subscribing to multiple MQTT topics rather than just one root topic at a time. The data from all topics specified in a single Event Hub Publisher block will all be forwarded to that block's configured Predix Event Hub endpoint. The specific topics that data will be published to are specified in the topic map in the configuration file.

This block also supports an optional, configurable quality of service and client ID for its MQTT subscriptions.

Note: The Event Hub Publisher block does NOT currently support wildcard MQTT subscriptions (e.g., `eventhub_data/#`). Any wildcard subscriptions will not be properly forwarded to Predix Event Hub.

Note: The way the Event Hub Publisher block handles MQTT subscriptions and what Event Hub topics to publish to is significantly different from the way its predecessor, the Event Hub Cloud Gateway, handled them. Instead of configuring a root MQTT topic to subscribe to in the Cloud Gateway and forcing the adapters to publish to a subtopic of that root topic to determine what Event Hub topic to publish to, the new Event Hub Publisher block specifies a mapping between input MQTT topics and output Event Hub topics in its configuration file. This allows the adapters or any other data sources to be blissfully unaware of what Event Hub topics their data will eventually be published to. The `cdpin` block also should NOT be used in this Cloud Gateway to create MQTT subscriptions.

Event Hub Store and Forward

The Event Hub Publisher block only has one option for store and forward functionality. As soon as a batch of input data is filled (to the maximum Event Hub publish request size) or the maximum batch interval (in the configuration file) has been reached, data will be stored in one of many disk-backed queues. The data is then popped off of these queues once it has been successfully published to Predix Event Hub. In the event of power loss, if some data has not been sent yet, it will still be available on disk, and the Cloud Gateway will pick up where it left off.

How Do I Deploy It?

Refer to the Packaging and Deployment *(page)* for instructions on how to deploy an Edge App (e.g., the pre-packaged Cloud Gateway above).

If you wish to modify the `docker-compose.yml` file, refer to Sample Files *(page 102)*, or in the pre-packaged Cloud Gateway tarball as a starting point and repackage the image with your new file as explained in Packaging and Deployment *(page)*.

How Do I Configure It?

The Cloud Gateway requires a single configuration file. A sample configuration file can be found in the Sample Files *(page 102)*. The name of this configuration file can be changed as long as its name matches the `config` environment variable in the `docker-compose.yml` file.

If you configured one of the Protocol Adapters, the format of the Cloud Gateway's configuration file should feel familiar, but with a few key changes. This configuration file is a JSON file that contains one main `blocks` section with each block's config section within it.

```
{
    "blocks": {
        ...
    }
}
```

See The Blocks Section *(page 94)* for an explanation of what a "block" is in the Cloud Gateway.

Note: The `mappings` section required for the Protocol Adapters is NOT used in the Cloud Gateway's configuration.

The Blocks Section

The `blocks` section is used to initialize the blocks that will be used by the Cloud Gateway. Think of each as a `block` of functionality. There are two main types of blocks in the Cloud Gateway. One that can send data to Predix Time Series service, and one that can send data to Predix Event Hub service.

Every block must have a `type` and `config` field in the configuration file. The `type` field defines what type of block is to be instantiated. The `config` section defines the configuration fields for that block and will look different depending on the `type` of the block. This `config` section is passed to the block when it is instantiated.

In the following example, two blocks are defined; one named `block1` of type `timeseries`, and another named `block2` of type `eventhub`.

```
{
    "blocks": {
        "block1": {
            "type": "timeseries",
            "config": {
```

```
                ...
            }
        },
        "block2": {
            "type": "eventhub",
            "config": {
                ...
            }
        },
        ...
    }
}
```

Time Series Publisher Block Config

The Time Series Publisher block can be instantiated by using the block type `timeseries`.

See The Blocks Section *(page 94)* for an explanation of what a `block` is in the Cloud Gateway.

The configuration fields (in the `config` portion of the block configuration) for the `timeseries` block are as follows:

Table 44. Configuration Fields

Field	Type	Required	Default
log_name	String	no	<block name>
log_level	String	no	'off'
mqtt/transport_addr	String	no	mqtt-tcp://predix-edge-broker
mqtt/qos	Integer	no	0
mqtt/client_id	String	no	
mqtt/topics	Array of Strings	yes	
policy	String	no	store_on_failure
store_forward	Object	yes	
store_forward/ max_store_percent	Integer	no	10
store_forward/ max_batch_interval	Integer	no	1000
timeseries/compress	Boolean	no	true
timeseries/transport_addr	String	yes	
timeseries/predix_zone_id	String	yes	
timeseries/token_file	String/Object	no	/edge-agent/access_token

Field	Type	Required	Default
timeseries/proxy_url	String	no	

The following is a sample block config for the `timeseries` block. This should be placed in the `blocks` section of the overall configuration file.

The block below is configured to subscribe to the MQTT topic `input_data` on the Predix Edge Broker and ingest all data received from that topic to a Predix Time Series instance with the Predix Zone ID `xxx-xxx` at the URL `wss://dummy_url.run.aws-usw02-pr.ice.predix.io/v1/stream/messages`.

The block's name in this example is `time_series_sender`, but it can be any string you wish to use.

```
"time_series_sender": {
    "type": "timeseries",
    "config": {
        "log_name": "time_series_block",
        "log_level": "debug",
        "mqtt": {
            "transport_addr": "mqtt-tcp://predix-edge-broker",
            "qos": 2,
            "client_id": "time_series_mqtt_client0",
            "topics": [
                "input_data"
            ]
        },
        "store_forward": {
            "max_store_percent": 30,
            "max_batch_interval": 2000,
            "policy": "store_on_failure"
        },
        "timeseries": {
            "transport_addr": "wss://dummy_url.run.aws-usw02-
pr.ice.predix.io/v1/stream/messages",
            "predix_zone_id": "xxx-xxx",
            "token_file": "/edge-agent/access_token",
            "proxy_url": "$http_proxy"
        }
    }
}
```

log_level and log_name

For details on the `log_level` and `log_name` fields of the Time Series Publisher block's `config` section, see Common Block Config Fields *(page 101)*.

mqtt

For details on fields within the `mqtt` portion of the Time Series Publisher block's `config` section, see Common Block Config Fields *(page 101)*.

store_forward/max_store_percent and store_forward/max_batch_interval

For details on the `store_forward/max_store_percent` and `store_forward/max_batch_interval` fields of the Time Series Publisher block's `config` section, see Common Block Config Fields *(page 101)*.

timeseries/compress

The `timeseries/compress` tag is an optional boolean (true/false) that defaults to 'true'. When enabled, the Time Series service will receive JSON payloads compressed (GZIP) by the cloud gateway. The size limit for the actual JSON payload is 512 KB regardless of the ingestion request format. For compressed payloads, this means the decompressed payload cannot exceed 512 KB.

timeseries/transport_addr

The `transport_addr` field within the `timeseries` section should be set to the URI of whatever Predix Time Series instance you wish to publish data to.

timeseries/predix_zone_id

The `predix_zone_id` field within the `timeseries` section should be set to the Predix Zone ID of whatever Predix Time Series instance you wish to publish data to.

timeseries/token_file

The `token_file` field within the `timeseries` section should usually be set to the path to the file on your Predix Edge device that holds your UAA token for authentication with your Predix Time Series instance.

To use a separate UAA other than the one utilized by Edge Manager you may provide a JSON object with the following keys:

- `uaa_url` - The URL of the host where the UAA service is running. The UAA service provides the access token that is subsequently used to push to the Timeseries server. The path `oauth/token` is appended and the resulting URL is used to request the access token.
- `client_id` - The client ID associated with an account that has access to the desired Timeseries zone.
- `client_secret` - The secret associated with the client_id.
- `proxy_url` - The proxy requried to access the UAA URL.

timeseries/proxy_url

The `proxy_url` field within the `timeseries` section should be set to the URL of whatever proxy you want to use (if any) to connect to whatever Predix Time Series instance you wish to publish data to. This field can be omitted or set to an empty string if no proxy is desired.

Event Hub Publisher Block Config

The Event Hub Publisher block can be instantiated by using the block type `eventhub`.

See The Blocks Section *(page 94)* for an explanation of what a `block` is in the Cloud Gateway.

Note: The Event Hub block's configuration has changed significantly from its earlier iterations.

The configuration fields (in the `config` portion of the block configuration) for the `eventhub` block are as follows:

Table 45. Configuration Fields

Field	Type	Required	Default
log_name	String	no	<block name>
log_level	String	no	'off'
mqtt/transport_addr	String	yes	
mqtt/qos	Integer	no	0
mqtt/client_id	String	no	
mqtt/topics	Array of Strings	yes	
store_forward	Object	yes	
store_forward/ max_store_percent	Integer	no	10
store_forward/ max_batch_interval	Integer	no	1000
eventhub/transport_addr	String	yes	
eventhub/predix_zone_id	String	yes	
eventhub/token_file	String	yes	
eventhub/topic_map	Array of Objects	yes	

The following is a sample block config for the `eventhub` block. This should be placed in the `blocks` section of the overall configuration file.

The block below is configured to subscribe to the MQTT topic `input_data` on the Predix Edge Broker and publish all data received from that topic to the Event Hub topic `output_data` on a Predix Time Series instance with the Predix Zone ID `xxx-xxx` at the URL `event-hub-aws-usw02.data-services.predix.io:443`.

The block's name in this example is `event_hub_sender`, but it can be any string you wish to use.

```
"event_hub_sender": {
    "type": "eventhub",
    "config": {
        "log_name": "eventhub_block",
        "log_level": "debug",
        "mqtt": {
            "transport_addr": "mqtt-tcp://predix-edge-broker",
            "qos": 1,
            "client_id": "event_hub_mqtt_client0",
            "topics": [
                "input_data"
            ]
        },
        "store_forward": {
            "max_store_percent": 30,
            "max_batch_interval": 2000
        },
        "eventhub": {
            "transport_addr": "event-hub-aws-usw02.data-
services.predix.io:443",
            "predix_zone_id": "xxx-xxx",
            "token_file": "/edge-agent/access_token",
            "topic_map": [
                {
                    "eventhub_topic": "output_data",
                    "mqtt_topics": [
                        "input_data"
                    ]
                }
            ]
        }
    }
}
```

log_level and log_name

For details on the `log_level` and `log_name` fields of the Event Hub Publisher block's `config` section, see Common Block Config Fields *(page 101)*.

mqtt

For details on fields within the `mqtt` portion of the Event Hub Publisher block's `config` section, see Common Block Config Fields *(page 101)*.

store_forward/max_store_percent and store_forward/max_batch_interval

For details on the `store_forward/max_store_percent` and `store_forward/max_batch_interval` fields of the Event Hub Publisher block's `config` section, see Common Block Config Fields *(page 101)*.

eventhub/transport_addr

The `transport_addr` field within the `eventhub` section should be set to the URI of whatever Predix Event Hub instance you wish to publish data to.

eventhub/predix_zone_id

The `predix_zone_id` field within the `eventhub` section should be set to the Predix Zone ID of whatever Predix Event Hub instance you wish to publish data to.

eventhub/token_file

The `token_file` field within the `eventhub` section should be set to the path to the file on your Predix Edge device that holds your UAA token for authentication with your Predix Event Hub instance.

eventhub/topic_map

The `proxy_url` field within the `eventhub` section specifies how data will be forwarded from input MQTT topics to output Event Hub topics. This section is an array of objects with the following fields:

Field	Type	Required
eventhub_topic	String	yes
mqtt_topics	Array of Strings	yes

Data received from subscriptions to the topics in the `mqtt_topics` field of one object will be published to the Event Hub topic in the `eventhub_topic` field of that same object.

Note: If an MQTT topic is not specified anywhere in the topic map, it will NOT be subscribed to even if it is in the `topics` list in the `mqtt` section of the block's `config` section.

Note: The Event Hub block does not currently support forwarding data from one MQTT topic to multiple Predix Event Hub topics.

Common Block Config Fields

See The Blocks Section *(page 94)* for an explanation of what a `block` is in the Cloud Gateway.

log_level

The `log_level` field determines which level of logs to output. If the field is not set to one of the following values, the block will not log anything. The values below are listed in order from most to least verbose:

- `debug`
- `info`
- `warn`
- `err`
- `critical`

log_name

The `log_name` field defines a name to identify the block's logs. This is typically prepended to the log output and can be any string you wish to set it to. If unset, it defaults to the block's name.

mqtt/transport_addr

The `transport_addr` field within the `mqtt` section should be set to the URI of the MQTT broker you wish to receive data from.

Note: This field is not required for the Time Series Publisher block (as it defaults to "mqtt-tcp://predix-edge-broker"), but it IS currently required for the Event Hub Publisher block.

Note: Supported URI prefixes for the Time Series Publisher block include `mqtt-tcp`, `mqtt`, and `tcp`. However, the Event Hub Publisher block supports only `mqtt-tcp` as the URI prefix for this field.

mqtt/qos

The `qos` field within the `mqtt` section can be set to the desired "quality of service" for the block's MQTT subscriptions.

This field's value can be 0, 1, or 2. These values correspond to "at most once", "at least once", and "exactly once" message delivery from the MQTT broker to the block.

mqtt/client_id

The `cliend_id` field within the `mqtt` section can be set to the desired client ID for the block's MQTT subscriptions.

This client ID helps the MQTT broker to identify the block. If the Cloud Gateway is restarted, and during that restart, data is published to topics the block was subscribed to, the broker will be able to deliver that data to the block after the Cloud Gateway comes back up as long as it uses the same client ID.

mqtt/topics

The `topics` field within the `mqtt` section should be set to an array of strings denoting the topics that the block should subscribe to on the MQTT broker specified by the `transport_addr` field.

store_forward/max_store_percent

The `max_store_percent` field within the `store_forward` section should be set to the max percent of disk space that the block's store and forward functionality is allowed to use.

Note: This value may be reduced (proportionally to other blocks) at runtime if the total `max_store_percent` set by all of the blocks is too large.

store_forward/max_batch_interval

The `max_batch_interval` field within the `store_forward` section should be set to the desired maximum interval (in milliseconds) between batch publish requests.

If input data is not received quickly enough to fill the maximum batch size for the respective block, the current batch will be completed at this interval (regardless of its size) and stored or sent according to the type of block and store forward functionality.

Sample Files

docker-compose.yml

The following sample file determines how to deploy the Cloud Gateway Edge App.

Note: The `config` environment variable must specify the file path to the configuration file inside the Docker container that will be deployed. If the file name does not match the configuration file applied to the Edge App, the Cloud Gateway will be unable to find it.

```
version: "3"
```

```
services:
  cloud-gateway:
    image: "dtr.predix.io/predix-edge/cloud-gateway:amd64-1.1.0"
    environment:
      config: "/config/config-cloud-gateway.json"
    env_file:
      - /etc/environment
    deploy:
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 5
        window: 30s
    networks:
      - predix-edge-broker_net

networks:
  predix-edge-broker_net:
    external: true
```

config.json

The following sample configuration file can be used to configure the Cloud Gateway to send data to both Predix Time Series and Predix Event Hub.

In this example, data received from the Predix Edge Broker on the MQTT topic `timeseries_data` will be ingested into the Time Series instance with Zone ID `xxx-xxx-xxx`. Data received from the MQTT topics `eventhub_data/osipi_data`, `eventhub_data/opc_ua_data`, `eventhub_data/modbus_data`, and `eventhub_data/egd_data` will be published to the Event Hub topic `topic` in the Event Hub instance with Zone ID `yyy-yyy-yyy`.

```
{
    "blocks": {
        "time_series_sender": {
            "type": "timeseries",
            "config": {
                "log_name": "time_series_sender",
                "log_level": "debug",
                "mqtt": {
                    "transport_addr": "mqtt-tcp://predix-edge-broker",
                    "qos": 2,
                    "client_id": "time_series_sender_mqtt_client",
                    "topics": [
                        "timeseries_data"
                    ]
                },
                "store_forward": {
                    "policy": "store_on_failure",
                    "max_store_percent": 15,
                    "max_batch_interval": 1000
```

```
                },
                "timeseries": {
                    "transport_addr": "wss://gateway-predix-data-
services.run.aws-usw02-pr.ice.predix.io/v1/stream/messages",
                    "predix_zone_id": "xxx-xxx-xxx",
                    "token_file": "/edge-agent/access_token",
                    "proxy_url": "$http_proxy"
                }
            }
        },
        "event_hub_sender": {
            "type": "eventhub",
            "config": {
                "log_name": "event_hub_sender",
                "log_level": "debug",
                "mqtt": {
                    "transport_addr": "mqtt-tcp://predix-edge-broker",
                    "qos": 1,
                    "topics": [
                        "eventhub_data/osipi_data",
                        "eventhub_data/opc_ua_data",
                        "eventhub_data/modbus_data",
                        "eventhub_data/egd_data"
                    ]
                },
                "store_forward": {
                    "max_store_percent": 60,
                    "max_batch_interval": 1000
                },
                "eventhub": {
                    "transport_addr" : "event-hub-aws-usw02.data-
services.predix.io:443",
                    "predix_zone_id" : "yyy-yyy-yyy",
                    "token_file": "/edge-agent/access_token",
                    "topic_map": [
                        {
                            "eventhub_topic": "topic",
                            "mqtt_topics": [
                                "eventhub_data/osipi_data",
                                "eventhub_data/opc_ua_data",
                                "eventhub_data/modbus_data",
                                "eventhub_data/egd_data"
                            ]
                        }
                    ]
                }
            }
        }
    }
}
```

Predix Edge Deadband Application

Introduction

The Deadband App provides the ability to manage Edge sites to have the deadband enabled for the respective tags to filter the amount of data pushed and realize savings for the data and its cost associated with Predix Time Series.

Protocol Benchmarking

The Predix Edge Deadband App sits between the protocol adapter and the cloud gateway (data travels from a protocol adapter to the Predix Edge MQTT Broker to the Deadband App to Cloud Gateway to Time Series). The throughput is highly dependent on the protocol adapter acting as the source. Throughput details for each adapter are available in the adapter's corresponding section.

The effect of the Deadband app on throughput was calculated by adding the Deadband app and configuring the tags in such a way that all data is passes by the Deadband app and nothing is filtered. The tags configured in the Deadband app are the same as the maximum throughput of each adapter without the Deadband app. In such a case, a maximum of 25% degradation is observed in the throughput. However, this should be offset by a decrease in the number of tags sent due to deadband application in any real application.

Where Do I Get It?

The Predix Edge Deadband App and sample configuration in the table below are stored in Artifactory. Use the following information to ensure you can access the files.

For GE Employees

To access Artifactory downloads, those using a GE email address must first be logged into Artifactory.

Note: If you attempt to download from Artifactory without first logging into Artifactory, you will be asked to **Sign in**, which will not work.

1. Go to Artifactory.
2. Click the **Log In** button.
3. Click the **SAML SSO** icon.

4. Use your SSO to log in.
5. You can then return to the documentation link to access Artifactory.

For Predix Users

To access Artifactory links in the Predix Edge documentation, you must first create an account on predix.io. Your predix.io account sign in credentials will be used to access Artifactory.

When you click an Artifactory link, enter your predix.io username (email address) and password in Artifactory's **Sign In** dialog.

Application	Config
Predix Edge Deadband App	Sample Config

Overview of Capabilities

Currently supported capabilities:

- Absolute and Percent value deadband
- Deadband timeout
- Configurable subscribe and publish topics on Predix Edge MQTT broker

Details of Capabilities

The Deadband app is mainly used for filtering data. It subscribes to the value of the **listenTopic** key and applies deadband on the data as per the configuration. It then sends the data after the deadband has been applied to the **publishTopic**. The deadband configuration is on a per tag basis. The deadband type can be specified as either **absolute** or **percent**. The value can be any numeric value in the case of absolute deadband and should be less than 100 in the case of percent deadband. The **timeout** is specified in milliseconds.

For every tag received, the following deadband logic is applied.

Absolute Deadband

If the absolute difference between the current value of the tag and the last sent value of the tag is greater than the deadband value, the current value of the tag is sent. The current value and current time are stored as last sent value and last sent time respectively for use in the next comparison cycle. If the absolute difference is less than the deadband value, the current time is compared against the

last sent time and if the difference is greater than the timeout, the current value of the tag is sent. Otherwise it is filtered.

Percent Deadband

If the absolute difference between the current value of the tag and the last sent value of the tag is greater than the deadband value percentage of last sent value, the current value of the tag is sent. The current value and current time are stored as last sent value and last sent time respectively for use in the next comparison cycle. If the absolute difference is less than the deadband value percentage of last sent value, the current time is compared against the last sent time and if the difference is greater than the timeout, the current value of the tag is sent. Otherwise it is filtered.

Configuration Details

The parameters in the configuration file are shown below:

Table 46.

Field	Type	Required	Default
listenTopic	String	Yes	
publishTopic	String	Yes	
tags	Object	Yes	
log_level	String	No	off
clientid	String	Yes	
qos	Integer	No	

`listenTopic`

The `listenTopic` field determines the MQTT topic on which this app listens for incoming data.

`publishTopic`

The `publishTopic` field determines the MQTT topic on which this app publishes the filtered data.

`tags`

The `tags` field defines the variables on which deadband is to be applied. The tags is an object of the following structure.

Table 47.

Field	Type	Required
<variable name 1>	String	Yes
<variable name 2>	String	No
...	String	No

The fields <variable name 1>, <variable name 2>, etc. represent individual variable names for which the values are objects that have the deadband details below.

Table 48.

Field	Type	Required
type	String	Yes
value	Any numeric	Yes
timeoutInMS	Integer	Yes

- `type` determines the type of deadband; possible values are absolute and percent

- `value` is the deadband value to be used
- `timeoutInMS` is the timeout value (in milliseconds) to be used during deadband calculation

`log_level`

For details about the `log_level` and `log_name` fields, see the Generic CDP Blocks *(page 15)* section of the Protocol Adapters *(page 8)* documentation page.

`clientid`

The `clientid` is used to specify the name of the client in the MQTT client options. It can be any unique string.

`qos`

The `qos` field is used to specify the qos (quality of service) to the MQTT client during publishing.

Sample Files

docker-compose.yml

```
version: "3.1"

services:
```

```
  edge-app:
    image: "predix-edge-deadband-app:amd64-latest"
    environment:
      config: "/config/config-deadband.json"
    networks:
      - predix-edge-broker_net
    deploy:
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 5
        window: 30s

networks:
  predix-edge-broker_net:
    external: true
```

config.json

```
{
 "tags": {
  "variable name 1": {"type": "absolute", "value": 2.5, "timeoutInMS":
 100000 },
  "variable name 2": {"type": "percent", "value": 10, "timeoutInMS":
 500000 }
 },
 "listenTopic":"timeseries_data",
 "publishTopic":"timeseries_data_deadband",
 "clientid":"predix-edge-deadbanding",
 "qos":1,
 "log_level": "warn"
}
```

Custom Applications

Building an Application

Building an Application

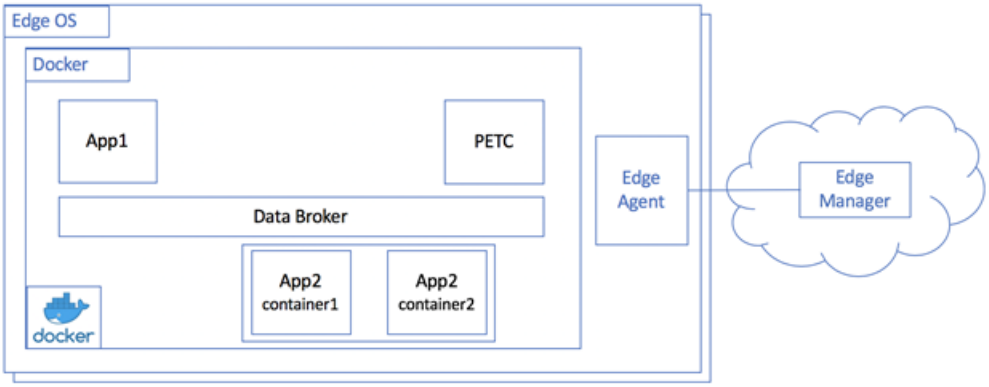
Predix Edge Manager enables remote deployment of multi-container applications and configurations to many devices, while the Predix Edge Technician Console (PETC) can deploy applications and configurations to a single Predix Edge device. The following explains how to design application deployments and create custom applications.

Application Architecture

A Predix Edge system consists of five major components:

- **Predix Edge OS** - The foundational, Yocto-based Linux that has been hardened and purpose-built for industrial Edge applications.
- **Predix Edge Agent** – Runs as a native process on the OS, communicates with Edge Manager and PETC, manages application deployment and the application lifecycle (starting, stopping, configuring applications). Edge applications do not directly interact with Edge Agent.
- **Docker Stack** – Runs applications launched via the Edge Agent.
- **Predix Edge Technician Console (PETC)** – Web UI to manage device enrollment with Edge Manager, application lifecycle and log retrieval; is integrated with Edge Agent.
- **Predix Edge Data Broker** – An MQTT service with Publish/Subscribe features provided as an Edge Application itself. It facilitates communication between single-container or multi-container Edge Apps.

Figure: Application Architecture

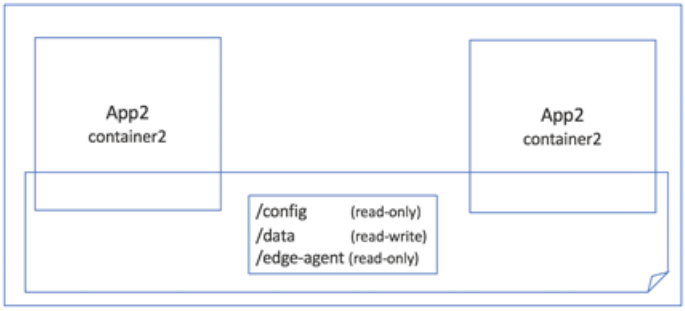


Predix Edge creates a common file system context when Edge Manager or PETC are used to deploy applications. Edge Agent will make the following directories available to the applications running inside Predix Edge's Docker Stack subsystem:

- The `/config` directory is read-only. When you use Edge Manager or PETC to deploy a zipped configuration file to a given application, Predix Edge Agent deploys and extracts that file into the selected application's configuration directory.
- The `/data` directory is read-write and available to all containers shared within a common application context.
- The `/edge-agent` directory is where the secure token is placed for communicating to Predix services, usually in the Predix cloud.
- The `/shared` directory is read-only and provides an `mqtt_config.json` file with app-specific details for communiciating privately over the Data Broker.

Note: The `/config`, `/data`, `/shared` and `/edge-agent` folders are reserved and will be automatically mounted to a container deployed to Predix Edge via PETC or Edge Manager. These folders will be mounted over any existing folder(s) in a container image with the same name.

Figure: Application Containers



Only containers in the same application can access the same `/config` or `/data` volume mount.

The `mqtt_config.json` file provides the following information:

```
{
  "requestTopic": "edgeAgent/app_<deployment-id>/request/#",
  "statusTopic": "edgeAgent/app_<deployment-id>/response/status",
  "username": " app_<deployment-id>",
  "passwordFile": "/var/run/secrets/p_mqtt_secret",
  "host": "predix-edge-broker",
  "port": "1883"
}
```

These details enable one app's containers to communicate privately with another over the Data Broker because they share common connection details.

Application Containers Can Communicate Over a Docker Network

In order for applications to publish or subscribe to messages to the Data Broker, they need to participate in the same Docker network as the broker container. The name of this network is `predix-edge-broker_net` and its usage is described in the next section.

Docker Networks

Predix Edge provides a common predix-edge-broker_net Docker network that all containers can connect to. Additionally, containers within an application can share separate Docker networks privately. For example, one can create a business logic container that would privately interact with a Postgres network on a separate network. That business logic container could also be exposed to predix-edge-broker_net to pull data from the Predix Edge Broker. This design would provide some measure of network segmentation.

Hostnames

Containers see other containers as hosts, as if they were separate computers with a unique hostname. The docker-compose format allows a given app to join the common `predix-edge-broker_net` network (or private networks) and will put those containers on the same network. These containers can then refer to other containers by their hostname. For example, HTTPS APIs are available to other containers.

You can add this entry to the docker-compose.yml for each container:

```
hostname: “my-container-hostname”
```

It is possible to create more elaborate network scenarios where you can expose some Hosts or Ports externally and some internally.

Ports

Application architecture should consider ports exposed to other containers or to external systems in the design.

- Port 443 is already used by PETC to expose itself outside of the device. Applications with HTTPS URLs should use another port.
- Edge Data Broker exposes port 1883 internally to the `predix-edge-broker_net` network.
- Configure external ports in the `docker-compose.yml` service definitions to avoid port conflicts.
- Work with your IT or Network administrator to ensure access to ports or through firewalls is properly configured for your application.
- All ports are available on all network interfaces except for 443 which is restricted to the LAN interface on the Predix Edge Gateway.

Docker Compose Structure

This example `docker-compose-local.yml` file below shows a one-container application that has mounted the `/config` and `/data` directories relative to the current folder on the computer the application is being launched from.

Note: As mentioned above, the `/data` and `/config` directories are created on behalf of the app when your app is deployed to Predix Edge. This `.local` configuration is purely for writing and testing apps outside of Predix Edge in Linux or on a Mac.

Network settings allow the container to access the Edge Broker to publish or subscribe to MQTT messages.

Finally, it is exposing the internal port 1880 as 1880 externally.

```
version:
“3.0”
 #This file combines all the edge services and our services so that it can
 be deployed as a unit
 services:
   my-container:
     image: "myorg/my-container-name:1.0.0"
     volumes:
       - ./config:/config
       - ./data:/data
     networks:
       - predix-edge-broker_net
     ports:
       - 1880:1880
 networks:
   predix-edge-broker_net:
     external: true
```

Packaging an Application

Introduction

Predix Edge facilitates the uploading, versioning and remote deployment of applications to Edge devices. The following covers packaging requirements for both Predix-provided applications and custom applications.

Containers

Containers vs. Predix Edge Apps

Containers are running instances of Docker images. The word container is often used when it is actually referring to an image.

Predix Edge applications are one or more Docker images that are “tar'd” and compressed with a docker-compose.yml file.

Packaging a Container

In a multi-container application, each container has a lifecycle and version. Containers are often released to a Docker repository (e.g., https://dockerhub.com).

Predix Edge Containers

Predix Edge containers are stored in Artifactory. Use the following information to ensure you can access the containers.

For GE Employees

To access Artifactory downloads, those using a GE email address must first be logged into Artifactory.

Note: If you attempt to download from Artifactory without first logging into Artifactory, you will be asked to **Sign in**, which will not work.

1. Go to Artifactory.
2. Click the **Log In** button.
3. Click the **SAML SSO** icon.
4. Use your SSO to log in.
5. You can then return to the documentation link to access Artifactory.

For Predix Users

To access Artifactory links in the Predix Edge documentation, you must first create an account on predix.io. Your predix.io account sign in credentials will be used to access Artifactory.

When you click an Artifactory link, enter your predix.io username (email address) and password in Artifactory's **Sign In** dialog.

Use this link to browse to Predix Edge containers on Artifactory.

Typical Container Directory Structure

The following is a typical directory structure for a container. In this case, the example (https://github.com/PredixDev/predix-edge-sample-scaler-nodejs) has some node.js JavaScript code:

- config
- data
- Dockerfile
- package.json
- sample
    - sample.html
    - sample.js
    - icons
        - sample.png
- README.md
- LICENSE

Table 49.

config	The properties files for the container are located here. Edge Manager pushes the application and then the configurations separately to the Edge device. This directory is mounted for you when you deploy your application to Predix Edge.
data	A location for volatile data needed for the application. This directory is mounted for you when you deploy your application to Predix Edge.
Dockerfile	The Docker file contains instructions when running '**docker build**' to create the Docker image. A best practice for a Docker file is to add LABEL entries within it. For example: • LABEL maintainer="Predix Builder Relations" • LABEL hub="https://hub.docker.com" • LABEL org="https://hub.docker.com/u/predixadoption" • LABEL version="1.0.19" • LABEL support="https://forum.predix.io" • LABEL license="https://github.com/PredixDev/predix-docker-samples/blob/master/LICENSE.md"
LICENSE	Every GitHub source code repository and Docker image should have a LICENSE file or link. Predix Sample code containers typically link to https://github.com/PredixDev/predix-docker-samples/blob/master/LICENSE.md, which allows you to freely use the container or source code (at your own risk). Product Containers provided by Predix Edge Engineering are subject to a product license, are supported by Predix Support and may require a paid contract. They should be marked as such using LABELs in the Docker file.

Note: Containers and Sample code shared among or with GE Industrial businesses, even if shared on internal servers, should have a LICENSE.

Publicly contributed containers and source code may contain any open source license, such as the MIT license.

Container Naming Conventions

To share your source code or containers, please use predix-edge-contrib as a prefix to the name to make it clear that the container or source code is not maintained by Predix or Predix Edge.

Each Docker Hub or GitHub repository should contain a README file, written in Markdown, that indicates how to load, run, deploy and use the repository.

Applications

Predix Edge Applications

Predix Edge Applications consist of a single tar file containing one or more Docker images (saved as tar files) and a docker-compose.yml.

One App or Many Apps?

Each Predix Edge application can have one or multiple containers. Each application may communicate with other applications via the Predix Edge Data Broker, as long as the containers are on the same Docker network as the one set in docker-compose.yml.

Why might you separate an application into multiple Predix Edge applications?

You might use pre-defined Predix Edge applications as components of your larger application.

Alternatively, the needs of your release cycle, performance considerations, or scheduling considerations might require that you divide your application into several applications.

Use cases that require distinct containers to share a common /data folder should bundle their containers into one application. For example, one container might pull data off of the Data Broker, run some analysis, and store analyzed data in the /data directory. Another container might provide a web server that enables users to browse the data and download those files locally or view them inside a web browser.

Application Packaging

Application tar files should be uploaded to Predix Edge Manager and stored. Once in Edge Manager, they can be deployed to many remotely managed Predix Edge devices. Application tar files may also be uploaded to a single device using the Predix Edge Technician Console (PETC).

The Edge Manager UI allows you to upload application tar files. Also, an API is available for DevOps. The Predix Edge Reference App provides a tutorial and script that shows how to call this API.

Best practices include using a consistent naming pattern and adding the version to the application file name.

Docker (docker-compose.yml files)

The Predix Edge Reference App provides some advice about the development lifecycle and the use of several different docker-compose files.

- docker-compose-local.yml - Use when testing locally on your VM, Windows, or OSX.
- docker-compose-edge-broker.yml - Use a separate .yml file for the Predix Edge Data Broker because Predix Edge OS is shipped with the broker already running inside, so it won't need to be packaged.
- docker-compose.yml - This is the default name and is required when packaging the Predix Edge application.

Application Size Limitations

Edge Manager *(page)* and PETC *(page)* enforce application size limitations. Before starting the development process, ensure your tar files sizes, container sizes and config sizes do not exceed these limits.

Application Signing

Before they can be deployed to a Predix Edge production OS, applications must first be signed. For more information, see Application Signing.

Application Signing

Application Signing

To release an edge app in the production environment, you must first have the application package cryptographically signed.

By default, production Predix Edge enabled devices will reject any application that has not been signed by GE Digital. Updated applications must be re-signed prior to release.

GE Digital will sign any application that meets internal validation criteria designed to protect Predix Edge devices.

Note: Apps used with Predix Edge developer builds do not need to be signed.

Obtain a GE Signature For Your Application

To have your application signed, please e-mail edge-app-signing@ge.com with the following information:

- Application name.
- Application version.
- Author name.
- Development Point of Contact E-Mail Address.

- Development Point of Contact Telephone Number.
- Attachment: Application tarball (or a link to download the application tarball).

If sending an application to GE Digital for cryptographic signing is not possible, see Self-Sign Your Application *(page 118)*.

Validation and application signing is performed in Vancouver, BC, from 9 a.m. to 5 p.m. (Pacific Standard Time). Requests will be processed within one business day.

Responses will contain either a link to download the signed copy of your application, or a list of changes required to meet GE Digital's internal validation criteria.

If you receive a link to download the signed copy of your application, no further action is required. The tarball provided at the link can be distributed to customers and deployed on production Predix Edge devices.

Self-Sign Your Application

If it is not possible for you to send your application to GE Digital for cryptographic signing, you can obtain your own keys to self-sign the application.

Note: Self-signing production applications is not recommended for the following reasons:

- All third-party signed keys are allowed when **allow-third-party-apps** is enabled. No mechanism exists to limit signed application verification to any particular third-party key. This allows Company A applications signed with a Company A key to pass verification in Company B's environment, if Company B enables **allow-third-party-apps**.
- Third-party applications are not subject to the same security verification and rigor that current GE Digital-signed Edge applications undergo when submitted to the current GE Digital application signing process.

To obtain your own signing keys, download the signing utility from https://github.build.ge.com/EdgeSecurity/edge-app-tools and run it as follows:

```
git clone https://github.build.ge.com/EdgeSecurity/edge-app-tools.git
cd edge-app-tools
chmod a+x ./signing-util
signing_util gen_key -n my_dev_key
```

This will create two files:

- `my_dev_key`
- `my_dev_key.pub`

It is imperative that my_dev_key be kept secret and not be shared outside your organization. The file `my_dev_key.pub` should be sent to edge-app-signing@ge.com. The signing utility can be used to generate the requisite information to be sent via e-mail:

```
signing_util key_info -n my_dev_key
To request a signature for this key, please e-mail the following
 information to edge-app-signing@ge.com

Key Owner: PLEASE INSERT YOUR NAME AND EMAIL ADDRESS HERE
Key Name: my_dev_key
Key Hash:
 165a6e50b0c733aa9314fa154fe5f06f95342aad95c284ce7ccc90123f32a310218fb6d3349d347be5bc6a04

-----BEGIN PUBLIC KEY-----
MHYwEAYHKoZIzj0CAQYFK4EEACIDYgAE5evAX2M4xplIhv0jKtnP8miqC3qxGzYC
WxO1BpEA/PDfhxrexbVE6en2+u5jfUTIUfx46b0qTCeBqd6XhyTFEzPH64ti3AfG
F/RgBI0qMLAuX8tJcEjki4PF1rzkyesW
-----END PUBLIC KEY-----
```

Requests to edge-app-signing@ge.com will be processed between 9 a.m. and 5 p.m. (Pacific Standard Time) and may take up to five business days to complete.

In response to your request, you will receive an e-mail with an attachment called `my_dev_key.pub.sig`. This signature file is required for signing applications and must be saved in the same location as `my_dev_key.pub`.

Now that you have obtained your own keys, you can sign your application.

1. Write your application's `docker-compose.yml` file.
2. Use validation utility: `edge-app-compose -i docker-compose.yml`
3. Review the results and fix any validation errors.
4. Sign it via: `signing-util sign_app -p my_app.tgz -x my_app_signed.tgz -s /home/j/my_key -t`
5. Distribute `my_app_signed.tgz` to your customers.
6. Inform your customers that they must set ‘allow-third-party-apps’ to ‘on’ via edge-manager in order to deploy their application.

By default, self-signed applications will not work on Predix Edge OS deployments. Customers must configure their Predix Edge device(s) to accept third-party signatures.

Running an Application

Applications can be launched using either Predix Edge Manager or Predix Edge Technician Console (PETC). These programs interact with Predix Edge Agent, which launches the application to the local Docker Stack system running on the Predix Edge device. Predix Edge Agent launches

applications using Docker Stack. A `docker-compose.yml` file defines the number and behavior of containers in the application. It is recommended developers test their docker-compose files to confirm they execute as expected using Docker Stack running locally on a development machine (either Mac or the Predix Dev VM).

To start an application in a Dev environment using Docker Stack:

```
docker stack deploy -c docker-compose-local.yml my-app-name
```

To stop an application:

```
docker stack rm my-app-name
```

When you test locally, your local `docker-compose.yml` file should include volume mounts for `/config` and `/data`. The `docker-compose.yml file` that is ultimately deployed to Predix Edge can either comment these out, or remove them altogether. It is recommended to have two `docker-compose.yml` files as follows:

- `docker-compose.yml` – Predix Edge requires this spelling when uploading an application tar.
- `docker-compose-local.yml` – Use this file when running locally for testing.

Deploying to PETC

Note that the application will launch after the Deploy step. You may need to stop the application and Apply the config, then start the application.

Running applications is a three-step process.

1. Upload the application.
2. Deploy the application.
3. Apply the configuration.

Deploying to Edge Manager

Note that applications will launch after the Deploy step. You may need to stop the application and Apply the config, then start the application.

1. Upload the application and config to the repository.
2. Deploy the application to a group of devices.
3. Apply the configuration.

Troubleshooting

While there are numerous problems that can occur with running multi-container applications, most are easily solvable. The following tips may be helpful when building and running Predix Edge applications.

Local:

- Run Locally – In the initial stages of development most problems are with the new code you write. Run locally and check the log files using the docker logs or docker service logs commands.
- Check Running Status – When running locally ‘docker stack ps <app-name>’ shows the status of each container. ‘docker stack ps <app-name> --no-trunc’ may also provide additional useful information.
- Check Log Files – ‘docker ps’ followed by ‘docker logs <id-here>’.
- Env Vars – Check that the `docker-compose-local.yml` has the required environment variables edefined.
- Volume Mounts - Check that the `docker-compose-local.yml` has the `/config`, `/data` or `/edge-agent` volume mounts.
- Docker Network – Check that the network is set up like the examples provided, with `predix-edge-broker_net`.
- Proxy – Ensure the environment variables are set and that the container knows about them. Add this to the `docker-compose-local.yml`.

  Note: This differs when running in Predix Edge OS.

```
http_proxy: ${http_proxy}
https_proxy: ${https_proxy}
HTTP_PROXY: ${HTTP_PROXY}
HTTPS_PROXY: ${HTTP_PROXY}
no_proxy: ${no_proxy}
```

Common to all deployments:

- Flopping Container – The Docker Stack keeps stopping and restarting a new instance of the container. Usually this is because the source code is unable to start. Check or try the following:
  - Dockerfile – the Dockerfile might be referencing an invalid folder.
  - Configs in `/config` – Ensure the code can find the configs in the `/config` volume mount.
  - Data in `/data` – Ensure the volatile code is writing to `/data`. It is possible that it is writing to a different read-only file system folder, which might cause unexpected behavior the second or third time it is launched.
- Have a look around. Execute these commands to use ssh to get inside the container. Look for permissions problems, launching the app problems, access to `/config` and `/data`.
  - docker ps

    - docker exec -it <id-of-container> /bin/sh
- Launch the app yourself – update the `docker-compose.yml` with an entrypoint override, then execute the commands above to get inside the container and launch the app manually.
    - entrypoint: ["sh", "-c", "sleep 500000"]

PETC and Edge Manager:

- Check Running Status – In the UI visit the Applications/Application page. Check the status for each container, it should be Running.
- Apply Configs – A common mistake is to Deploy the app but not the Configs. Most container logic will fail if missing its configuration.
- Env Vars – Check that `docker-compose.yml`, bundled with the app tar, has the env vars needed.
- Volume mounts – Upload will fail if `docker-compose.yml` has volume mounts.
- Docker Network – Check that the network is set up like the examples provided, with `predix-edge-broker_net`.
- Hostname vs. IP – In `docker-compose.yml` give your container a Hostname since the IP will vary when running in an Edge OS VM or physical instance.
- Proxy – Ensure the container is aware of proxy environment variables. In PETC this is set on each device, usually before enrollment.
    - Add the following to a service that has logic looking for proxy env vars (e.g., HTTPS_PROXY). PETC puts them in `/etc/environment` and `docker-compose.yml` needs to have an entry that loads it.
        - env_file: -/etc/environment
    - b. Some code does not reference env vars for proxy info, ensure the config is set up.

PETC:

Check logs. In the UI:

- Navigate to the **Applications** page.
- Choose **Service Name**.
- Choose **Additional Options**.
- Set **Message Priority** to **Debug**.
- The log viewer only shows the first 20 rows in the date range, so you might need to narrow the range.
- Confirm the time range is accurate.
- Click the **Update Preview** button.
- Download the logs locally to view more details.

Edge Manager:

Upload the logs using a command:

1. Navigate to **Device Manager** > **Devices**.
2. Select the device.

3. Select **Commands**.
4. Click the **Execute Command** button.
5. Select **Get Journal Log**. Click **Next**.
6. Fill out the form. Click **Execute**.
7. Wait up to 30 seconds for the command to execute.
8. Click the **Download** link to view the file.

Accessing Devices

Making I/O Devices Available to Applications

I/O devices on Predix Edge OS are accessed through device files in the `/dev` folder. There are two types of devices: block devices and character devices. Only character devices are currently supported. A device file is exposed to the applications by creating a device configuration file. System builders typically create these configuration files.

1. The system builder needs to create the configuration file `/opt/edge-agent/device-mapping.json`.
2. Edit the file to list the devices to be made available to applications. For each device, add an entry to the device mapping list that specifies:
   - `file`: The device file on the Predix Edge OS that corresponds to the device that will be exposed to the applications.
   - `type`: The device type. As of Predix Edge 2.3.0, `SERIAL` is the only supported device type.
   - `id`: A unique, user-defined device identifier that will be used to map the application to the device.
   - `description`: A user-defined descriptor for the device.

   Sample device mappings file:

```
{
    "devices": [
        {
            "file": "/dev/ttyS0",
            "type": "SERIAL",
            "id": "device1",
            "description": "Weight Scale"
        },
        {
            "file": "/dev/ttyS1",
            "type": "SERIAL",
            "id": "device2",
            "description": "Bar Code Reader"
        },
        {
            "file": "/dev/ttyS2",
            "type": "SERIAL",
```

```
            "id": "device3",
            "description": "Position Encoder"
        }
    ]
}
```

3. Edit the file `/opt/edge-agent/agent-data.json` to add the key `device_mapping` with the value `/opt/edge-agent/device-mapping.json`, as in:

```
"device_mapping": "/opt/edge-agent/device_mapping.json"
```

Requesting Access to an I/O Device on the Host

Follow this procedure to provide an application with access to an I/O device on the host.

1. Edit the application's `manifest.json` file.
2. Add one instance of the `devices` key for each device you want to map to the application. The devices listed must be present in the `/opt/edge-agent/device-mapping.json` file.
3. For each device specify the:
    - `id`: This identifier must match the one specified for the device in the `/opt/edge-agent/device-mapping.json` file.
    - `service`: The application service into which the device will be mounted.
    - `file`: The name of the device file in the service's container. The application will use this file to access the device (using functions such as open, close, ioctl, etc.).
    - `type`: The device type. As of Predix Edge 2.3.0, `SERIAL` is the only supported device type.
    - `description`: A user-defined descriptor for the device.

Sample application manifest:

```
{
  "manifest": {
    "name": "serial-port-readout",
    "capabilities": [
      {
        "name": "serial-port-readout",
        "version": "1.0.0",
        "handler": "unused"
      }
    ],
    "devices": [
      {
        "id": "device3",
        "service": "main",
        "file": "/dev/ttyApp",
        "type": "SERIAL",
        "description": "Input From Infinite Improbability Drive"
      }
    ]
```

```
  }
}
```

The application uses the device `id` to create the mapping between the device and the application. If a device `id` specified in `mappings.json` does not exist in the `/opt/edge-agent/device-mapping.json` file, or if the device is already mapped to another application (regardless of whether that application is running or not), the mapping will fail and the application will not deploy.

An application may not have access to the device files immediately after it starts. It may take up to one minute before the application is granted access. An application should loop for a while and try again if `open()` on the device file fails. (I.e. if the `open("/dev/ttyApp", O_RDONLY)` system call fails.)

When an application using a device is removed, the device is returned to the pool of available devices.

Application Custom Commands

Expose a Capability Via the Application

A capability is something that can be exposed by an application to indicate to the broader system (PETC, Edge Manager, etc.) that it can perform certain actions or that it will behave in certain ways. A capability currently includes an identifier and a version. The existence of a capability may mean that the application supports specific commands, specific packages, and will report status in a given format. An example of this is an application that supports the Predix.Edge.AnalyticsEngine (v 1.0.0) capability. The existence of this capability will enable the analytics UI in Edge Manager and allow Edge Manager to deploy analytic templates and data maps to the application, start, stop, and delete analytic templates that are running the application, and will expect the application to produce status about the deployed analytics that will drive the UI in Edge Manager.

Note: Capabilities within Predix.Edge.* are restricted to capabilities that are known to the Predix Edge ecosystem.

An application manifest file named `manifest.json` contains:

- A list of capabilities, each containing:
  - Name string.
  - Version string.
  - Optional handler.

The format of an application manifest is:

```
{
  "manifest": {
```

```
    "capabilities": [
      {
        "name": "some_capability_name",
        "version": "some_version",
        "handler": "some_handler"
      },
      {
        "name": "another_capability_name",
        "version": "another_version"
      },
      ...
    ],
    ...
  }
}
```

MQTT Configuration

A configuration file is available to applications that will specify various settings for interacting with the mqtt broker around command execution. This file is located in `/shared/mqtt_config.json` and has the following format:

```
{
 "requestTopic": "edgeAgent/app_{app_name}/request/{request_topic}”,
 "statusTopic": "edgeAgent/app_{app_name}/response/status",
 "host": "predix-edge-broker”,
 "port": "1883”,
 "username": "app_{app_name}",
 "passwordFile": "{passwordFile}"
}
```

Securely Connecting to the Command Topic

The **requestTopic** is used for receiving requests for commands from the system; the **statusTopic** is used for returning status from the application to the rest of the ecosystem. The Predix Edge software stack will provide the application with the request topic as defined in the MQTT configuration section, along with the username and `passwordFile` location. The username and password contained within the `passwordFile` file must be used when connecting to the broker in order to communicate over the request topic. Inside the container, the `passwordFile` contains the plaintext version of the password. This file is secured by a docker secret on the host.

Note: The command topic is application-specific and not accessible from other applications.

The application's docker compose file must be of a version equal to, or greater than, 3.1. If the docker compose file is of version 3.0 or earlier, the password file used to securely connect to the broker will not be present in the application.

Implementing Commands

Commands are communicated to the application via the secure command channel and an application should subscribe to a wildcard under the applications command topic (`<command_topic>/*`). The commands and command responses are json-based, as opposed to protobuf-based as they were in Predix Machine. The majority of the information is the same.

The initial command message will be published to `<command_topic>/<taskID>` where **command_topic** is passed into the application and the **taskID** is a unique identifier for the current command or operation created at dispatch time by the edge stack. The application is expected to publish its response to the **responseTopic**, which is included in the command information passed into the application for the given task. Command response will be in the following format:

```
{
            "status": "SUCCESS or FAILURE or NOTSUPPORTED",
            "status_message": "status message",
            "status_detailed_message": "detailed status message",
            "output": "UTF-8 encoded output from the command"
}
```

Each field is UTF-8 encoding.

Note: Output is not sent to Edge Manager for deployments (**DeployAnalyticTemplate** and **DeployDataMap**).

The **status_detailed_message** will appear in the execution logs column in EdgeManager in the Command History page.

Note: Responses should be published with the retained flag. After reading the response, Edge Agent will issue two blank messages to clear out the completed command and response.

Exposing Status Information

Status is a way for an application to provide information to the broader ecosystem (Edge Manager, PETC, etc.) that is automatically queried rather than triggered via user interaction. This is commonly used to relay information to the cloud for state such as which applications exist and are running, which analytics are running or stopped, or information about connectivity (Wi-Fi signal strength, cell signal strength, etc.). For certain capabilities there is a prescribed format the status requires in order to be consumable by Predix Edge, but custom capabilities can have their own status format that can be retrieved from Edge Manager or via the PETC.

The application provides status back to the Edge Ecosystem via a status topic provided in the `mqtt_config.json` file as the **statusTopic**. Status message payload expected for the

AnalyticEngine is a json object that is a representation of the old protobuf-based status in the status section and the **capabilityId** of **Predix.Edge.AnalyticsEngine** and a version of 1.0.0.

```
[
   {
         "handler": "handler name",
         "capabilityId": " Predix.Edge.AnalyticEngine",
         "capabilityVersion": "1.0.0",
         "status": "…"
    } ...
]
```

Note: Status should be published with the retained flag. Edge Agent will not remove the status message after reading it.

Note: The status message is retrieved and sent to EdgeManager at each synch interval, it is not automatically sent when a new message arrives into the topic, nor is a status message guaranteed to be delivered if the synch interval is not triggered while the status message is the most recent.

Analytics Framework

Introduction

The analytics framework enables application developers to integrate analytic engines into Predix Edge. This includes the ability to lifecycle manage analytic templates and instances running in remote edge devices from Predix Edge Manager.

Application developers need to expose the analytics engine capability (Predix.Edge.AnalyticEngine), which requires implementation of the following content.

- Commands:
  - startAnalyticsTemplate(templateId)
  - stopAnalyticstemplate(templateId)
  - deleteAnalyticsTemplate(templateId)
  - Deploy
    - Analytic template
    - Analytic datamap (component_descriptor) [optional]
    - Status
      - Provide status/state messages as described above *(page 127)*.

Implementing Commands

Refer to Implementing Commands *(page 127)*.

The following are sample commands for each of the required AnalyticEngine commands.

Start Analytic

```
{
            "command": "startAnalyticsTemplate",
            "handler":"Analytics",
            "responseTopic":"edgeAgent/<AppID>/response/
<task_id>",
            "params": {
                        "templateId": "1001"

            }
}
```

Stop Analytic

```
{
            "command": "stopAnalyticsTemplate",
            "handler":"Analytics",
            "responseTopic":"edgeAgent/<AppID>/response/
<task_id>",
            "params": {
                        "templateId": "1001"

            }
}
```

Delete Analytic

```
{
            "command": "deleteAnalyticsTemplate",
            "handler":"Analytics",
            "responseTopic":"edgeAgent/<AppID>/response/
<task_id>",
            "params": {
                        "templateId": "1001"

            }
}
```

Deploy Analytic Template

```

{
            "type": "analytics_template",
            "package": "/shared/downloads/filename",
            "responseTopic":"edgeAgent/<AppID>/response/
<task_id>",
            "handler":"foghornML|forhornCEP|CSense",
            "params": {
                  "name": "test-wx-analytics",
                "description": "",
```

```
                "id": "13454",
                "version": "1.0.0",
                "parentId": ""
                }
}
```

Deploy Analytic Data Map

```
{
            "type": "analytics_data_map",
            "package": "/shared/downloads/filename",
            "responseTopic":"edgeAgent/<AppID>/response/
<task_id>",
            "handler":"foghornML|forhornCEP|CSense",
            "params": {
                 "name": "test-wx-analytics-data-map",
                "description": "",
                "id": "13455",
                "version": "1.0.0",
                "parentId": "13454"
            }
}
```

Sending Status

Refer to Exposing Status Information *(page 127)*.

Status for the analytic engine contains:

```
{
  "component_status_list":
    "status":[
    {
      "id": "{template_id_from_deployment}",
      "state": "EDGE_ANALYTICS_COMPONENT_STATE_UNKNOWN|
EDGE_ANALYTICS_COMPONENT_STATE_ACTIVE|
EDGE_ANALYTICS_COMPONENT_STATE_INACTIVE",
      "state_message": "<string>"
    },
    //..
  ],

  "timestamp": "<timestamp>",
  "attributes": {
    "<string>": {
      "value": "<string>",
      "data_type": "[STRING|BINARY|BOOLEAN|FLOAT|DOUBLE|INT|LONG|
TIMESTAMP]"
    },

    //…
  }
```

```
}
```

The status message format for the Predix.Edge.AnalyticEngine (1.0.0) capability must follow the schema defined below.

```
{

 "$schema": "http://json-schema.org/draft-07/schema#",
 "type": "object",
 "required": [
  "component_status_list",
  "timestamp",
  "attributes"
    ],
    "properties": {
            "component_status_list":
            {
                "$ref": "#/definitions/component_status_list"
            },
            "timestamp":
            {
                "$ref": "#/definitions/timestamp"
            },
            "attributes":
            {
                "$ref": "#/definitions/attributes"
            }
    },
    "definitions": {
        "component_status_list": {
   "$id": "#/definitions/component_status_list",
   "type": "object",
   "required": [
    "status"
   ],
   "properties": {
    "status": {
                    "type":"array",
                    "items":{
                        "$ref":"#/definitions/status_element"
                    }
                }
   }
        },
        "status_element": {
            "type": "object",
            "required": [
                "id",
                "state",
                "state_message"
            ],
            "properties": {
```

```
                "id": {
                    "type": "string",
                    "pattern": "^[0-9]+$"
                },
                "state": {
                    "type": "string",
                    "enum":["EDGE_ANALYTICS_COMPONENT_STATE_ACTIVE",
 "EDGE_ANALYTICS_COMPONENT_STATE_INACTIVE",
 "EDGE_ANALYTICS_COMPONENT_STATE_UNKNOWN"]
                },
                "state_message": {
                    "type": "string",
                    "examples": [
                        "Running"
                    ],
                    "pattern": "^.+$"
                }
            }
        },
  "timestamp": {
   "$id": "#/definitions/timestamp",
   "type": "string",
   "examples": [
    "2018-12-11T17:58:53.171Z"
   ],
   "pattern": "^.+$"
  },
  "attributes": {
   "$id": "#/definitions/attributes",
   "type": "object",
            "^(.+)/([^/]+)$": {
                "$ref":"#/definitions/attributes_element"
            }
        },
        "attributes_element":{
            "$id": "#/definitions/attributes_element",
            "type":"object",
            "required": [
                "value"
            ],
            "properties": {
                "value": {
                    "type": "string"
                },
                "dataType": {
                    "type": "string",
                    "enum":["DATATYPE_STRING", "DATATYPE_BINARY",
 "DATATYPE_BOOLEAN","DATATYPE_FLOAT","DATATYPE_DOUBLE",
 "DATATYPE_INT","DATATYPE_LONG","DATATYPE_TIMESTAMP"]
                }
            }
        }
 }
```

```
}
```

Creating the Application Manifest

The application manifest is a file used to provide metadata to Predix Edge about the application. To enable the analytics framework the application manifest must expose that the application exposes the AnalyticEngine (Predix.Edge.AnalyticEngine) capability.

The sample manifest below includes a sample where the AnalyticEngine capability is exposed. In a production manifest there will also be a files section, which is automatically added by GE when performing application signing and is not the responsibility of the application developer.

```
{
  "manifest": {
    "capabilities": [
      {
        "name": "Predix.Edge.AnalyticEngine",
        "version": "1.0.0"
      },
      ...
    ],
  }
}
```

Analytic Engine Capabilities

The Analytic Engine capability enables the management of analytic instance lifecycles as well as runtime state.

Capability ID: predix.edge.analyticengine

Version: 1.0.0

Commands

startAnalyticsTemplate

Starts the analytic template associated with templateId.

Table 50. startAnalyticsTemplate Parameters

Parameter	Type	Description
templateId	String representing an integer	An identifier representing the analytic to be acted upon

stopAnalyticsTemplate

Stops the analytic template associated with templateId.

Table 51. stopAnalyticsTemplate Parameters

Parameter	Type	Description
templateId	String representing an integer	An identifier representing the analytic to be acted upon

deleteAnalyticsTemplate

Removes the analytic template associated with templateId, the analytic should no longer be reported in the status.

Table 52. deleteAnalyticsTemplate Parameters

Parameter	Type	Description
templateId	String representing an integer	An identifier representing the analytic to be acted upon

Packages

analytics_template

An analytic template that can be used to instantiate an analytic or that can be used in conjunction with a data map to instantiate an analytic.

Table 53. analytics_template Parameters

Parameter	Type	Description
name	String	Name of the analytic template
description	String	Description of the analytic template
id	String representing an integer	ID of the analytic template
version	Straing representing a version (e.g., 1.0.0)	Version of the analytic template

analytics_data_map

An analytic data map that can be used to instantiate an analytic based on a previously deployed analytic template.

Table 54. analytics_data_map Parameters

Parameter	Type	Description
name	String	Name of the analytic template
description	String	Description of the analytic template
id	String representing an integer	ID of the analytic template
version	Straing representing a version (e.g., 1.0.0)	Version of the analytic template

Parameter	Type	Description
parentId	ring representing an integer	ID of the template this data map is associated with

Status

Status message JSON schema:

```
{

        "$schema": "http://json-schema.org/draft-07/schema#",
        "type": "object",
        "required": [
                "component_status_list",
                "timestamp",
                "attributes"
    ],
    "properties": {
            "component_status_list":
            {
                "$ref": "#/definitions/component_status_list"
            },
            "timestamp":
            {
                "$ref": "#/definitions/timestamp"
            },
            "attributes":
            {
                "$ref": "#/definitions/attributes"
            }
    },
    "definitions": {
        "component_status_list": {
                    "$id": "#/definitions/component_status_list",
                    "type": "object",
                    "required": [
                            "status"
                    ],
                    "properties": {
                            "status": {
                    "type":"array",
                    "items":{
                        "$ref":"#/definitions/status_element"
                    }
                }
                    }
        },
        "status_element": {
            "type": "object",
            "required": [
                "id",
```

```
                "state",
                "state_message"
            ],
            "properties": {
                "id": {
                    "type": "string",
                    "pattern": "^[0-9]+$"
                },
                "state": {
                    "type": "string",
                    "enum":["EDGE_ANALYTICS_COMPONENT_STATE_ACTIVE",
 "EDGE_ANALYTICS_COMPONENT_STATE_INACTIVE",
 "EDGE_ANALYTICS_COMPONENT_STATE_UNKNOWN"]
                },
                "state_message": {
                    "type": "string",
                    "examples": [
                        "Running"
                    ],
                    "pattern": "^.+$"
                }
            }
        },
              "timestamp": {
                    "$id": "#/definitions/timestamp",
                    "type": "string",
                    "examples": [
                           "2018-12-11T17:58:53.171Z"
                    ],
                    "pattern": "^.+$"
              },
              "attributes": {
                    "$id": "#/definitions/attributes",
                    "type": "object",
            "^(.+)/([^/]+)$": {
                "$ref":"#/definitions/attributes_element"
            }
        },
        "attributes_element":{
            "$id": "#/definitions/attributes_element",
            "type":"object",
            "required": [
                "value"
            ],
            "properties": {
                "value": {
                    "type": "string"
                },
                "dataType": {
                    "type": "string",
                    "enum":["DATATYPE_STRING", "DATATYPE_BINARY",
 "DATATYPE_BOOLEAN","DATATYPE_FLOAT","DATATYPE_DOUBLE",
 "DATATYPE_INT","DATATYPE_LONG","DATATYPE_TIMESTAMP"]
```

```
                    }
                }
            }
        }
}
```

Logging

Predix Edge Logs

When debugging an application or troubleshooting a device you will likely need to obtain the log files from the Predix Edge device. Logs can be obtained from either Edge Manager *(page 137)* or from the command line *(page 138)* of the device. Obtaining logs from the command line is only available if you are using the Development version of the Predix Edge device that has ssh enabled.

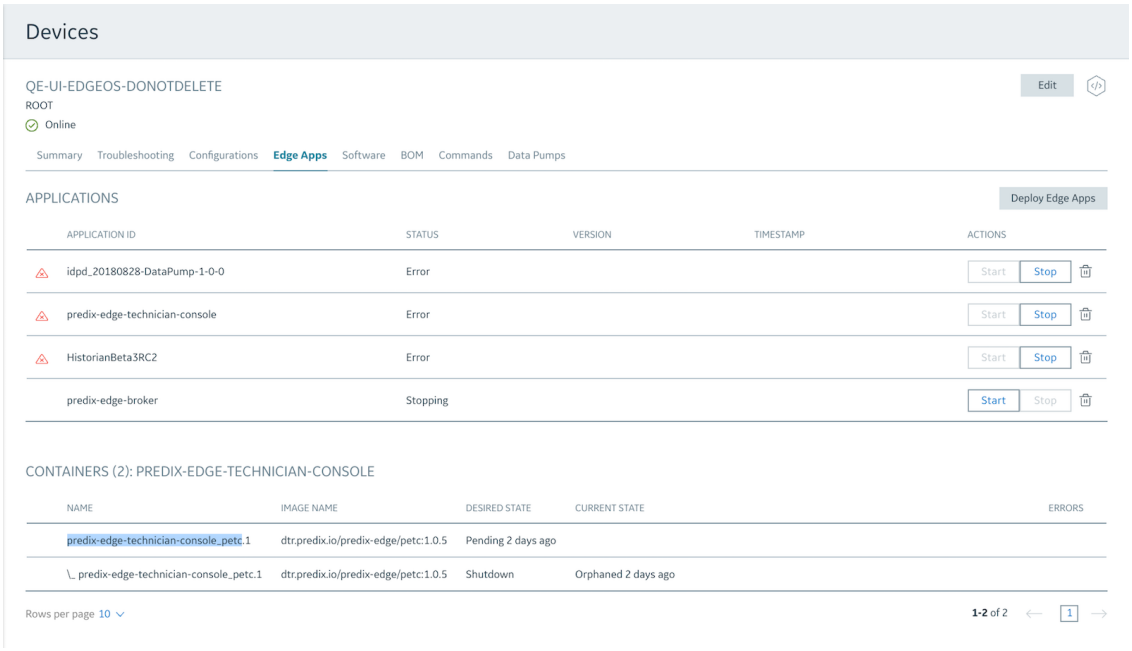
Retrieving Predix Edge Device Logs

You can use Predix Edge Manager to retrieve log files from the edge device when debugging an application or troubleshooting a device.

If you are using the development version of Predix Edge, with ssh enabled, you can also obtain logs from the command line of the device.

1. In the left navigation pane, select  **Device Manager > Devices**.
2. Select the device for which to retrieve logs, then select **Device Operations > Execute Command**.
3. In the **Execute Command** dialog window select **Predix Edge** as the platform, then select the **Get Journal Log** tile, and click **Next**.
4. In the **Execute Command** dialog window, set filters (using boolean values "true" and "false) for the type, quantity, and formatting of the logs to retrieve, and click **Execute**.
   Edge Manager requests the logs from the device.
   Optionally, to filter the logs for a specific application running on the device, you must first obtain the Container Name from the app. To do this:
   a. On the **Edge Apps** tab for the device, select the **Application ID** and copy the name of the container.

Note: The container name may have a `.1` appended at the end of the name. Do not include that in the copied text.



b. In the **Execute Command** dialog box, paste the value you copied in the previous step into the **Show Logs from Specified Application Service** field.

5. In the confirmation dialog window, click **Close**.

6. In the **Device Manager** page, click the link for the device, then click the **Commands** tab. The **Commands History** displays the command history for the device and the status of each command. Once the **Status** displays "Success" you can click the **download** link in the **Output** column to download the log file.

Retrieve Logs From the Command Line

1. Use **ssh** to connect to the IP address of your device. The credentials are root/root.

```
$ ssh root@x.x.x.x
$ password: root
```

2. Use the **docker ps** command to view the Docker containers running on the device.

```
$ docker ps
```

3. From the list of running containers, copy the container name.
4. Use the **journalctl** command with a `CONTAINER_NAME` filter to retrieve the logs.

```
$ journalctl CONTAINER_NAME=your-container-name
```

5. If you would like to follow a real time list of your logs as they are generated, add a `-f` parameter.

```
$ journalctl CONTAINER_NAME=your-container-name -f
```

Predix Edge Applications and Services Release Notes

OPC-UA Protocol Adapter Release Notes 21.7.0

Enhancements

This release contains the following enhancements.

New Configuration Options

Six new configuration options have been added for the OPC-UA blocks:

- session_timeout
- connect_timeout
- watchdog_interval
- watchdog_timeout
- publishing_interval
- lifetime_count

For more information, see Configuration Properties for OPC-UA Protocol Adapter *(page 63)*.

General Improvements

A number of other changes have been made to improve the adapter's synchronization and stability.

OPC-UA Protocol Adapter Release Notes 20.2.0

Enhancements

This release contains the following enhancements:

Report Bad Quality

The `report_bad_quality` feature is now supported in the 'Poll Flat' and 'Sub Flat' blocks. This feature ensures that NULL values are sent out when connectivity to the server is lost.

Source Timestamp

The `source_timestamp` feature is now supported in the 'Poll Flat' block. This feature allows the OPC-UA server timestamp to be used for the tags, as opposed to the Predix Edge device timestamp.

Bug Fixes

This release contains the following bug fixes:

Exception Handling

Resolved an issue where exceptions (such as non-finite floating point values) in the 'EGD Sink Flat', 'Json Splitter', 'JsonTimeseries', 'Simulink Json' and 'Flat to Timeseries' blocks would cause adapters to crash.

Bad Type in OPC-UA Data

Fixed an issue where bad type in OPC-UA data (when using the 'OPC-UA Poll Flat' block with an output format of 'flat_json') caused subsequent values to be null in flat_json. Now, only the value of the specific tag will be set to 'NULL'.

Data Quality

Fixed an issue where the quality of OPC-UA data was set to '3' (Good) even though the value was 'NULL'. Now, the quality will correctly be set to '0' (Bad).

Support for Null Timeseries Datapoints

Added support for null timeseries datapoints in OPC-UA and Flat-to-Timeseries.

Best-effort Data Reporting

Implemented best-effort data reporting so that bad data will not cause good data to be discarded when messages contain a mix of both good and bad data.

OSI-PI Protocol Adapter Release Notes 21.5.0

Enhancements

This release contains the following enhancements:

Fetching Server Parameters

The adapter now fetches required server parameters as needed. This resolves an issue where a failed fetch that was not retried resulted in malformed read requests.

Tags

Resolved an issue where an OSI-PI tag with bad quality was reported as 'uncertain', rather than 'bad'.

General Improvements

A number of other changes have been made to improve the adapter's synchronization and stability.

OSI-PI Protocol Adapter Release Notes 20.2.0

Enhancements

This release contains the following enhancements:

Container Image

The container image distribution has been upgraded from Alpine 3.5 to Alpine 3.10.

Tags

Tags can now be specified using point name in addition to webid.

EGD Dynamic Binding Protocol Adapter Release Notes 21.03.0

New Features

Support for Multiple Producers

The EGD dynamic binding protocol adapter has been extended to support multiple producers, each identified by their producer id and their associated configuration server. Users will need to update their configuration files, as the format has changed in a way that is not backwards compatible. See EGD Dynamic Binding Protocol Adapter *(page 33)*.

Deadband Application Release Notes 20.4.1

Bug Fix

The 20.4.1 version of the Deadband application behaves and functions the same as the 20.4.0 version. The only change is the following:

App Signing

This release contains an app signing fix for Edge Manager deployment.

Deadband Application Release Notes 20.4.0

New Feature

This is a new feature release for Predix Edge Applications and Services.

The Deadband App provides the ability to manage Edge sites to have the deadband enabled for the respective tags to filter the amount of data pushed and realize savings for the data and its cost associated with Predix Time Series. See Deadband Application *(page 105)*.

Cloud Gateway Release Notes 21.07.0

Enhancements

This release contains the following enhancements:

Time Series Token File

There is a new **timeseries/proxy_url** option, which sets a separate proxy server for the UAA. This change allows for the accommodation of architectures where the time series destination is on a separate network from Edge Manager. For additional information and configuration details, see Time Series Publisher Block Config *(page 95)*.

General Improvements

Improvements have been made to the Cloud Gateway's performance and stability.

Cloud Gateway Release Notes 20.12.0

Enhancements

This release contains the following enhancements:

Compressed Format Accepted for Data Ingestion

A new configuration option has been added to the Time Series Publisher Block. The **timeseries/compress** tag is an optional boolean (true/false) that defaults to 'true'. When enabled, the Time Series service will receive JSON payloads compressed (GZIP) by the cloud gateway.

See Time Series Publisher Block Config *(page 95)*.

Time Series Token File

The time series **token_file** configuration option now allows you to use a separate UAA other than the one utilized by Edge Manager. For additional information and configuration details, see Time Series Publisher Block Config *(page 95)*.

Cloud Gateway Release Notes 20.3.0

Enhancements

This release contains the following enhancements:

Performance Improvements

Many code paths that added latency have been removed. For configuration files that use 500 to 1000 tags:

- CPU usage has been reduced up to 30 times
- Memory consumption has been reduced by five times

Ubuntu Support

This version of the cloud gateway has been validated with Edge Agent on Ubuntu 18.04, which is currently in limited availability. For more information on Edge Agent on Ubuntu please contact predix-edge-inquiries@ge.com.

Store Always Capability Removed

In situations where a configuration is set to **store always**, this setting will be ignored and **store-on-failure** will be used instead.

Predix Edge Applications and Services Release Notes 12-19

Bug Fixes

This release contains the following bug fix.

Cloud Gateway Data Transmission

Resolved an issue that would see the cloud gateway stop sending data to the cloud when it encountered bad network conditions, necessitating a restart.

Known Issues

This release has the following known issues:

Duplicate Client IDs when Connecting to MQTT

Applications with no `clientid` configured may encounter an issue with duplicate client IDs when connecting to the MQTT broker resulting in the applications not working as expected. To avoid this issue, configure the `clientid` field either in the PEAS applications or in custom applications when connecting to the broker.

Excessive Logging Depletes Memory

Applications that generate excessive logs can create conditions where Docker (dockerd) consumes a large amount of memory. If memory usage is of concern, configure applications to produce fewer logs.

EGD Failure During Power Cycle

The EGD application may fail to start or deploy if a previous attempt to start/deploy was interrupted by a power cycle. The workaround for this issue is to reboot the device before attempting to start/deploy the application.

OPC UA and Timeseries Output Format

Users of `opcuapollflat` blocks should not use the `flat_to_timeseries` block to generate timeseries output as this will result in the loss of tag-based timestamps because the `flat_to_timeseries` format uses the first timestamp it encounters for the entire tag list and discards the rest.

Instead, the `output_format` field should be set to `time_series`.

Cloud Gateway Performance

When the cloud connection is re-established and there is a large store buffer, cloud gateway performance will be slow for a couple of minutes.

Raw Modbus Data

Raw modbus data passed from an adapter into a raw modbus sink will fail with an error similar to:

- **Couldn't parse {"data":{"tagName": {"val":19}},"timestamp":1562096019014}: std::exception**

Raspberry Pi

Cloud Gateway on Raspberry Pi is known to fail after running for an extended period of time, due to insufficient memory on the Raspberry Pi device.

Note: The Raspberry Pi image remains unchanged from the 2.1.0 release.

Predix Edge Applications and Services Release Notes 2.4.0

Enhancements

This release has the following enhancements.

Cloud Gateway 1.4.0

- Upgraded the container image distribution from Alpine 3.5 to Alpine 3.10
- Significant performance improvements
- Reduced verbosity of logs

Bug Fixes

This release contains the following bug fixes:

Cloud Gateway 1.4.0

- Fixed performance degradation over time when using store-on-failure
- Fixed issues that could cause stability problems
- Fixed issues relating to shutdown

Known Issues

This release has the following known issues:

Duplicate Client IDs when Connecting to MQTT

Applications with no `clientid` configured may encounter an issue with duplicate client IDs when connecting to the MQTT broker resulting in the applications not working as expected. To avoid this issue, configure the `cliendid` field either in the PEAS applications or in custom applications when connecting to the broker.

Excessive Logging Depletes Memory

Applications that generate excessive logs can create conditions where Docker (dockerd) consumes a large amount of memory. If memory usage is of concern, configure applications to produce fewer logs.

EGD Failure During Power Cycle

The EGD application may fail to start or deploy if a previous attempt to start/deploy was interrupted by a power cycle. The workaround for this issue is to reboot the device before attempting to start/deploy the application.

OPC UA and Timeseries Output Format

Users of `opcuapollflat` blocks should not use the `flat_to_timeseries` block to generate timeseries output as this will result in the loss of tag-based timestamps because the `flat_to_timeseries` format uses the first timestamp it encounters for the entire tag list and discards the rest.

Instead, the `output_format` field should be set to `time_series`.

Cloud Gateway Performance

When the cloud connection is re-established and there is a large store buffer, cloud gateway performance will be slow for a couple of minutes.

Raw Modbus Data

Raw modbus data passed from an adapter into a raw modbus sink will fail with an error similar to:

- **Couldn't parse {"data":{"tagName": {"val":19}},"timestamp":1562096019014}: std::exception**

Raspberry Pi

Cloud Gateway on Raspberry Pi is known to fail after running for an extended period of time, due to insufficient memory on the Raspberry Pi device.

Note: The Raspberry Pi image remains unchanged from the 2.1.0 release.

Predix Edge Applications and Services Release Notes 2.3.2

Bug Fixes

This release contains the following bug fixes:

OPC-UA Adapter Polling

1. Resolved an issue where OPC-UA adapter polling would stop processing data in a block once a tag with bad data quality was encountered. This issue exists in Predix Edge releases 2.3 and 2.3.1.

Cloud Gateway

Resolved several resource leaks and stability issues within the Cloud Gateway.

Known Issues

This release has the following known issues:

Raspberry Pi

Cloud Gateway on Raspberry Pi is known to fail after running for an extended period of time, due to insufficient memory on the Raspberry Pi device.

Note: The Raspberry Pi image remains unchanged from the 2.1.0 release.

Cloud Gateway Performance

When the cloud connection is re-established and there is a large store buffer, cloud gateway performance will be slow for a couple of minutes.

Raw Modbus Data

Raw modbus data passed from an adapter into a raw modbus sink will fail with an error similar to:

- **Couldn't parse {"data":{"tagName": {"val":19}},"timestamp":1562096019014}: std::exception**

Predix Edge Applications and Services Release Notes 2.3.0

New Features

This release contains the following new features:

Edge File Agent and Cloud File Gaeway

The ability to remotely manage file send/receive capability between Predix Edge and Cloud is now available. The APM File Gateway stores/retrieves files from your Tenant's Blobstore and places a message on Event Hub. The Edge File Agent allows

any Edge application to send/receive files in the Edge applications/data folder. ETL or other data movement applications will leverage this capability to keep customer data sources in sync, receive events or transfer log files.

OPC-UA Protocol Adapter

The following capabilities have been added to the OPC-UA Protocol Adapter:

- A new block has been introduced for OPC-UA events subscriptions:
  - Subscribe to OPC-UA events, and change the parameters of the subscription, or unsubscribe.
  - Allow for configurable event attributes.

OSI-PI Protocol Adapter

The following capabilities have been added to the OSI-PI Protocol Adapter:

- Increased ability to handle polling from ~200 to ~2000 tags.
- Added extra configuration options:
  - `proxy_url` (string) – The `proxy_url` field determines the proxy address used to connect to the PiWebApi endpoint. It defaults to environment variable `$https_proxy`.
  - `validate_certs` (bool) – The `validate_certs` field determines whether the adapter will validate the certificates of the PiWebApi endpoint. Use this field if your PiWebApi does not have a valid certificate. It defaults to an empty string.
  - `interval_ms` (int) – The `interval_ms` field determines the interval (in milliseconds) at which the block will poll its endpoint for data. The default is 1000.
  - `output_format` (string) – The `output_format` field determines the output format of the data retrieved from the PiWebApi:
    - `flat_json` will return the data in flat JSON format.
    - `time_series` will return the data in Predix Time Series format.
    - `native` will return the data in the native PiWebApi JSON format.

Resolved Issues

The following known issues have been resolved:

Transmitting Frequently Changing Data

Previously, when using OSI-PI to transmit data at a rate higher than 100 tags per second for an extended period of time, the stability of the application was not guaranteed. This issue is now fixed.

IGS Configuration Updates During IGS Connection Failure

Fixed an issue where data would be retrieved indefinitely based on the previous configuration until a valid configuration with reachable IGS server IP and PORTS were applied during IGS OPC-DA configuration updates.

OPC-UA Browsing

When the OPC-UA server was configured with child nodes that referred to ancestor nodes, the OPC-UA Browser did not account for this corner use case. If the functionality was used against a server with nodes configured in this manner, it resulted in an infinite loop. This issue has been resolved.

Known Issues

This release has the following known issue:

Raspberry Pi

Cloud Gateway on Raspberry Pi is known to fail after running for an extended period of time, due to insufficient memory on the Raspberry Pi device.

Predix Edge Applications and Services Release Notes 2.2.0

New Features

This release contains the following new features:

EGD Dynamic Binding

The EGD Protocol Adapter has been enhanced to support Dynamic Binding. It can now automatically determine changes in the configuration of the EGD controller and adapt in real time, adjusting all data tag subscriptions.

IGS Adapter

A new IGS Protocol Adapter is now available. This adapter allows Predix Edge to connect to Kepware's Industrial Gateway Server (IGS). IGS is a third-party product that runs on a Windows Server and can connect to 200+ industrial protocols. IGS then converts the protocol data to OPC-UA to be consumed by other applications. In this initial release, the IGS Adapter can be configured to subscribe to OPC-DA data sources via IGS. The adapter works in tandem with the Predix Edge OPC-UA Protocol Adapter to subscribe to the OPC-DA results and publish them on the Predix Edge Data Broker.

MQTT Protocol Adapter Authentication

The MQTT adapter can now be configured to authenticate to the external MQTT source using a username and password.

OPC-UA Protocol Adapter

The following capabilities have been added to the OPC-UA Protocol Adapter:

- Data tag node browsing, including hierarchy – An application can browse the OPC-UA server and discover all of its available nodes programmatically. Node browsing provides the ability to browse the tree of objects to a configurable depth.
- Read data tags metadata – An application can programmatically interrogate a given node in the tree and return the list of node attributes.

Resolved Issues

The following known issue has been resolved:

MQTT Protocol Adapter on Raspberry Pi

When the MQTT protocol adapter on Raspberry Pi loses its connection/signal with the Broker, the adapter can now come back online/establish a connection by itself without a restart.

Known Issues

This release has the following known issues:

Transmitting Frequently Changing Data

When using OSI-PI to transmit frequently changing data to the Cloud or to an internal application, Edge Engineering guarantees the OPC-UA protocol adapter will transmit up to 100 tags per second. If you are transmitting data at a rate higher than 100 tags per second for an extended period of time, the stability of the application is not guaranteed.

IGS Configuration Updates During IGS Connection Failure

During IGS OPC-DA configuration updates, if IGS is unreachable due to a connection failure or invalid IP/PORT configuration, the IGS adapter will enter into a retry stage in an attempt to reach the unreachable IGS and eventually timeout based on the retry interval specified in the configuration file. However, data will still be retrieved indefinitely based on the previous configuration until a valid configuration with reachable IGS server IP and PORTS are applied.

Raspberry Pi

Cloud Gateway on Raspberry Pi is known to fail after running for an extended priod of time, due to insufficient memory on the Raspberry Pi device.

OPC-UA Browsing

Technically, it is possible for an OPC-UA server to be configured with child nodes that refer to ancestor nodes. The OPC-UA Browser does not account for this corner use case. If the functionality is used against a server with nodes configured in this manner, it may result in an infinite loop.

Predix Edge Applications and Services Release Notes 2.1.0

These are the new features and known and resolved issues for Predix Edge Applications and Services, version 2.1.0.

New Features

This release contains the following new features:

ARM-based Predix Edge Applications

ARM versions of all protocol adapters and cloud gateway components are now available. These will enable Raspberry Pi devices to run these components, providing an accessible, low cost and developer friendly device platform.

Cloud Gateway Enhancements

A new simplified and unified Cloud Gateway component has been implemented with store and forward data buffering capability to transmit data to both Predix Cloud Event Hub and Predix Cloud Time Series.

New Historian Applications

The following new Historian applications have been added:

- Historian Web-admin Console – dashboard for Historian Data Archiver.
- Historian S2S Collector – Data Streamer between Historian servers.

Known Issues

This release has the following known issues:

Raspberry Pi

- Cloud Gateway on Raspberry Pi is known to fail after running for an extended priod of time, due to insufficient memory on the Raspberry Pi device.
- When the MQTT protocol adapter on Raspberry Pi loses its connection/signal with the Broker, the adapter will not come back online/establish a connection by itself and will be in a locked state. Should this occur, restart the MQTT protocol adapter container.

Transmitting Frequently Changing Data

- When using OSI-PI to transmit frequently changing data to the Cloud or to an internal application, Edge Engineering guarantees the OPC-UA protocol adapter will transmit up to 100 tags per second. If you are transmitting data at a rate

higher than 100 tags per second for an extended period of time, the stability of the application is not guaranteed.