

## Using Agent Parameters in a Dataflow

Date published: 2020-06-22

Date modified: 2020-09-15



# Legal Notice

© Cloudera Inc. 2024. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

# Contents

<b>Overview of CEM and Agent Parameters.....</b>	<b>4</b>
<b>Scenario: Tracking Numerous Agents.....</b>	<b>4</b>
<b>Scenario: Log Collection Aggregation.....</b>	<b>8</b>

## Overview of CEM and Agent Parameters

Cloudera Edge Management (CEM) is an edge management solution for IoT and streaming use cases. You can use CEM to manage, control, monitor thousands of MiNiFi agents deployed at the edge.

CEM helps customers to work on TechOps or IoT type use cases. The challenges with IoT are around edge management like the inability to control and manage edge agents, difficulty in collecting real-time data from edge devices directly and most importantly, the trouble with updating specific sets of agents with edge applications. CEM addresses those challenges. CEM paves the way for transformative solutions across IoT initiatives across multiple industry verticals.

By using CEM you can manage numerous use cases. Here is a list of examples:

- Monitoring oil rig data
- Trucking system
- Laptop tracking
- Monitoring oil refinery data

In this section we will describe two scenarios where you can use parameters.

In the first scenario, we will describe how to create a single flow, parameterize numerous agent IDs, and track data.

In the second scenario, we will describe how to create a single flow, parameterize the log location, and collect data when log location changes.

## Scenario: Tracking Numerous Agents

You are a supervisor in an oil company that is engaged in exploration, development, and production of crude oil. Your company has deployed numerous oil rigs on site X and wants you to monitor data coming out for those oil rigs.

Earlier, you needed to create one dataflow for each and every oil rig. So, you created numerous dataflows to monitor numerous oil rigs.

### Solution

Parameters can solve this challenge of creating numerous dataflow for tracking purposes. With the parameter concept introduced in CEM, you can create a single dataflow and use this to monitor data from all the oil rigs.

1. You use your tooling to create agent specific parameter contexts for the agent IDs of the newly added servers.
2. With these items created, you then publish your flow and make the updates available to agents. EFM, when agents heartbeat in with a specific context, are given an update to a flow with the new V2 agent IDs provided. Those agents that have not had a specific context made will use the default, version 1 agent IDs.

### Initial setup

You have a class with numerous MiNiFi agents deployed on all the oil rigs and running. You have built a dataflow inside this class, and published this dataflow to all the MiNiFi agents deployed on the oil rigs. The dataflow gathers logs from the desired location, for example `/opt/myapp/logs`, on the machines and then performs the associated logic to only get the content which is WARNING or ERROR. You are managing and monitoring the warnings and errors that are collected at the EFM server for every heartbeat from the oil rigs.

### Actual steps

1. To have the agents running the same flow assigned, assign them all to the same class, design your flow, establish your processors and controller services. In this example, the class is `RigClass`.

- 2. Open the controller service you want to parameterize and select the blue arrow next to the property of interest.

### Configuration

#### Settings

SERVICE NAME \*

MyService

#### Properties

ADD PROPERTY

Property	Value
rig_id	<div><div></div>Empty string set<div></div></div>

- Specify the name as `rig_id` for the parameter and an optional default value (here, listed as default) with optional description. Once populated, select Add.

## Create Parameter

NAME \*

`rig_id`

VALUE

`default`

☐ Set empty string

Sensitive Value

☐ Yes ☒ No

DESCRIPTION

`Unique identifier for individual rigs`

- Select Apply on the Controller Service pane to complete this configuration.

This establishes a parameter context for the flow. You can provide agent-specific values to override the default values of `rig_id`. Alternatively, you can choose to deploy the flow with default values.

- Given that there are potentially a large number of instances we are addressing, the design choice was made to make this available programmatically without any specific UX at this juncture. You can issue these commands from the command line, a REST client in your program of choice, or through the Swagger UI (<http://<efm address:port>/efm/swagger/ui.html>). This example shows command line curl and Swagger approaches for addressing these items. If you manually assigned IDs to your agents, through `nifi.c2.agent.identifier` in `bootstrap.conf`, compile your list of those entities. If you have not, we can collect the auto-generated identifiers.

- For collecting the auto-generated IDs, use the REST API.

Through Swagger, use the following endpoint: <http://localhost/efm/swagger/ui.html#/Agents/getAgents>

By clicking Try it out, you can execute. Look through the results and extract the agent identifiers, such as:

```
{
  "identifier": "myagentidentifier1",
  "agentClass": "RigClass",
  "agentManifestId": "b472989e-705a-3125-a2dd-7f26bdd908b9",
```

```

    "status": {
      "uptime": 1572535845540
    },
    "state": "ONLINE",
    "firstSeen": 1572535764479,
    "lastSeen": 1572535845662
  }

```

Through the command line, use curl and the jq tool to extract these, such as highlighted in this script (get\_agents\_for\_class.sh):

```

#!/bin/bash -e

# Specify the base location of the EFM server
efm_address='http://localhost'
efm_api_base_url="${efm_address}/efm/api"

[ -z "$1" ] && { echo "No class name supplied"; exit; }

query_class=$1

# Select all agent identifiers for the provided query_class and
class_agents=$(curl -s -X GET "${efm_api_base_url}/agents" -H "accept:
application/json" |
    jq -r --arg class "$query_class" \
        '.[]
        | select(.agentClass == $class)
        | .identifier')

echo "${class_agents}"

```

This gives a listing of all identifiers:

```

./get_agents_for_class.sh RigClass
3c7b311f-c9c0-4f5f-bb03-c10be82beab6
Myagentidentifier1

```

- You would then like to assign values for each instance and formulate an update to change these items. As you were thinking about scaling, you decided to opt for programmatic changes in lieu of manually editing each item. Again, you can address this in two manners, either through Swagger or command line.

Through Swagger, use the following endpoint: <http://localhost/efm/swagger/ui.html#/Agents/createAgentParameters>

By clicking Try it out, you can generate the payload and hit execute. In the case of multiple overrides, additional JSON objects to this payload would be provided.

More likely however, you will have a repository/store of information you would like to impart on each asset. Doing this programmatically via CLI (or other tooling) is the most common path we have seen and could be accomplished through an interaction as shown in (create\_params\_for\_agent.sh):

```

#!/bin/sh -e

efm_address='http://localhost'
efm_api_base_url="${efm_address}/efm/api"

[ -z "$1" ] && { echo "No agent ID specified"; exit; }
agent_id=$1
agent_key=$2
agent_value=$3

# Update the agent i

```

```
echo "Updating Agent ID: ${agent_id} parameter context to override va
lue for ${agent_key} to ${agent_value}"
curl -s -X DELETE "${efm_api_base_url}/agents/${agent_id}/parameters"
curl -s -X POST "${efm_api_base_url}/agents/${agent_id}/parameters" \
-H "accept: application/json" \
-H "Content-Type: application/json" \
-d "[{
  \"name\": \"${agent_id}\",
  \"sensitive\": false,
  \"description\": \"Agent overridden value\",
  \"value\": \"${agent_value}\"
}]"
```

By iterating over the identifiers we found before with their associated values.

```
./create_params_for_agent.sh myagentidentifier1 rig_id rigid1
Updating Agent ID: myagentidentifier1 parameter context to override va
lue for rig_id to rigid1
{"id":"cfd553d8-fc2c-491d-af1b-9554a59c90d2","name":"Agent specific con
text for agent myagentidentifier1","parameters":[{"name":"myagentidentif
ier1","sensitive":false,"description":"Agent overridden value","value":"
rigid1"}]}{"id":"038d0129-1150-4016-ab99-328ffbf09437","name":"Agent spe
cific context for agent myagentidentifier1","parameters":[{"name":"myage
ntidentifier1","sensitive":false,"description":"Agent overridden value",
"value":"rigid1"}]}
```

**6.** With everything configured, go back to the flow and publish.

Next time agents heartbeat in, the agents receive the updated flow with their associated overrides based on their identifiers. Do note that these parameter overrides are immutable, so if you want to adjust, add, or remove values for an agent, it will need to first be deleted for the agent before creating it.

## Scenario: Log Collection Aggregation

You are a supervisor in a car rental company that is engaged in renting cars and trucks. Your company has put numerous cars and trucks on rent and you monitor logs coming out for those vehicles.

Eventually you have more servers doing this same processing and they are provisioned with a new version of myapp and the log location has changed. Earlier, for example, your logs resided in `/opt/myapp/logs`. Now, for example, your logs reside in `/opt/myappv2/logs`. Due to policy on making Configuration Management changes on existing servers, you have the same content and flow but have different needs on configuration for this source data.

This caters to a log collection aggregation use case. In this scenario, agents may be deployed on a large number of servers that are all performing this log collection process.

CEM has a standard way to filter and transform these logs to filter out only the events of interest.

Earlier, you needed to create one dataflow for each and every vehicle. So, you created numerous dataflows to monitor numerous vehicles. If the log path changed in this situation, then you updated each and every dataflow to update the path.

### Solution

Parameters can solve this challenge. With the parameter concept introduced in CEM, you can create a single dataflow to monitor data from all the vehicles and parameterize the log path in the flow. So, if the log path changes, you just need to update it in the dataflow once.

1. You first update your LogCollection flow for your class to parameterize the GetFile location to `log.location` and specify the default value of `/opt/myapp/logs`.



2. With these items created, you then publish your flow and make the update available to agents. EFM, when agents heartbeat in with a specific context, are given an update to a flow with the new v2 location provided. Those agents that have not had a specific context made will use the default, version 1 location of the logs.

### Initial setup

You have a class called LogCollection with numerous MiNiFi agents deployed on all the vehicles and running. You have built a dataflow inside this class and published this dataflow to all the MiNiFi agents deployed on the vehicles. The dataflow gathers logs from the desired location, for example `/opt/myapp/logs`, on the machines and then performs the associated logic to only get the content which is `WARNING` or `ERROR`. You are managing and monitoring the warnings and errors that are collected at the EFM server for every heartbeat from the vehicles.

### Actual steps

1. Open the flow for the LogCollection agent class.
2. Design the flow.
3. Find the one or more properties or configurations that may need unique values on a per agent basis and create parameters. Keep track of name which will be used as an ID in our REST API calls. You will reference this as parameter name moving forward.

The screenshot shows a 'Create Parameter' dialog box overlaid on a configuration interface. The dialog box contains the following fields and options:

- NAME \***: Input Delimiter
- VALUE**: (Empty text box)
- ☐ Set empty string
- SENSITIVE VALUE**: ☐ Yes ☒ No
- DESCRIPTION**: (Empty text box)
- CANCEL** and **ADD** buttons at the bottom.

The background interface shows a table with a 'Value' column and a list of parameters:

Value
#{log.location}
No value set
TailFileState
No value set
Single file

4. Switch over to Swagger to perform manual steps for update (this could be automated through Configuration Management tooling like Salt, Puppet, and Ansible).

5. Confirm the agent identifier(s) you wish to augment (this is driven by the value provided by users for `nifi.c2.agent.t.identifier` or automatically generated) by specifying a custom value for the one or more parameter name values created above.

Through Swagger, use the following endpoint: `http://localhost/efm/swagger/ui.html#/Agents/getAgents`

```
curl -X GET "http://localhost/efm/api/agents" -H "accept: application/json"
```

Sample response:

```
[
  {
    "identifier": "test_agent_1",
    "agentClass": "default",
    "agentManifestId": "39344613-9b36-41e7-9416-e6b755d038c9",
    "flowId": "7650ef4e-5258-11ea-ba2c-0242ac120002",
    "deviceId": "15831645727656044827",
    "status": {
      "uptime": 1434101,
      "repositories": {
        "flowfile": {
          "size": 0
        },
        "provenance": {
          "size": 0
        }
      },
      "components": {
        "FlowController": {
          "running": true
        }
      }
    },
    "state": "MISSING",
    "firstSeen": 1582035123611,
    "lastSeen": 1582036507014
  },
  {
    "identifier": "test_agent_2",
    "agentClass": "default",
    "agentManifestId": "39344613-9b36-41e7-9416-e6b755d038c9",
    "flowId": "c4a1627c-5259-11ea-be90-0242ac120009",
    "deviceId": "12645594159739466366",
    "status": {
      "uptime": 888065,
      "repositories": {
        "flowfile": {
          "size": 0
        },
        "provenance": {
          "size": 0
        }
      },
      "components": {
        "FlowController": {
          "running": true
        }
      }
    },
    "state": "MISSING",
    "firstSeen": 1582035621624,
    "lastSeen": 1582036507139
  }
]
```

```

    },
    {
      "identifier": "test_agent_3",
      "agentClass": "default",
      "agentManifestId": "39344613-9b36-41e7-9416-e6b755d038c9",
      "flowId": "c4a1922e-5259-11ea-ba73-0242ac120008",
      "deviceId": "11588253182801567996",
      "status": {
        "uptime": 889065,
        "repositories": {
          "flowfile": {
            "size": 0
          },
          "provenance": {
            "size": 0
          }
        },
        "components": {
          "FlowController": {
            "running": true
          }
        }
      },
      "state": "MISSING",
      "firstSeen": 1582035621721,
      "lastSeen": 1582036508237
    }
  ]

```

**6. Create agent specific parameter contexts for those properties you wish to override.**

Through Swagger, use the following endpoint: <http://localhost/efm/swagger/ui.html#/Agents/createAgentParameters>

```
curl -X POST "http://localhost/efm/api/agents/<AGENT ID>/parameters" -H
"accept: application/json" -H "Content-Type: application/json"
```

The body that gets posted contains one or more key value names for the designated parameter name values.

Sample body (one parameter for an agent):

```
[
  {
    "name": "parameter name 1",
    "sensitive": false,
    "description": "Agent parameter name override ",
    "value": "parameter value"
  }
]
```

Sample body (multiple parameters for an agent):

```
[
  {
    "name": "parameter name 1",
    "sensitive": false,
    "description": "Agent parameter name override ",
    "value": "parameter value 1"
  },
  {
    "name": "parameter name 2",
    "sensitive": false,
    "description": "Agent parameter name override ",
    "value": "parameter value 2"
  }
  ...
]
```

```
]
```

7. Optionally, confirm the creation of the agent parameter contexts by using Swagger or curl.

Through Swagger, use the following endpoint: <http://localhost/efm/swagger/ui.html#/Agents/getAgentParameters>

```
curl -X GET "http://localhost/efm/api/agents/<AGENT ID>/parameters" -H "
accept: application/json"
```

Sample response:

```
[
  {
    "name": "parameter name 1",
    "sensitive": false,
    "description": "Agent parameter name override ",
    "value": "parameter value"
  }
]
```

8. In the CEM UI, push publish to deploy flow to agents with associated parameter contexts.