Cloudera Edge Management 1.4.1

Managing Agent Manifest Resolution

Date published: 2019-04-15
Date modified: 2022-07-21

https://docs.cloudera.com/

Legal Notice

Contents

Introduction to agent manifest resolution strategy

The manifest of an agent is its full and detailed list of capabilities. A manifest includes all extension components, including processors and controller services, and how they are configured.

When an agent first communicates to EFM through the C2 Protocol (heartbeats), the EFM server asks the agent to describe itself, and the agent supplies its manifest to the server.

An agent class is a group of agents. Currently, agent classes are defined by a unique property in each agent, and agents with matching class names are put in the same agent class in the EFM server.

An agent manifest is necessary in order to design a flow for an agent class, but because an agent class can contain many agents, the EFM server must decide which agent manifest to use for the entire agent class when loading the flow for that class in the flow designer. The logic that decides this is referred to as the *Agent Manifest Resolution Strategy*.

EFM supports multiple strategies to resolve agent manifest for a class at run time:

Strategy	Description
First In	This is the default manifest resolution strategy and binds an agent class to the first manifest reported for it.
Last In	This strategy updates the associated manifest to the most recently reported by any agent associated with the specified class.
Static	This strategy allows mapping of an agent class to a specific manifest ID.

The strategy is configurable at the application level and applies to all agent classes.

In efm.properties, set the global agent manifest resolution strategy:

```
# This property does not exist, so you will have to add it anywhere in efm.p
roperties
# The default is 'First In'
efm.manifest.strategy=Last In
```

Static mappings act as overrides to the global manifest resolution strategy. If a static mapping of agent class to manifest ID is configured, then the application uses the static manifest strategy for that class and ignore the globally configured strategy. Classes without a static mapping falls back to the global stratey (i.e., First In or Last In).

Static mappings are created through the REST API (or Swagger UI) and stored in the EFM database.

Here is an example of setting a mapping for a class to manifest using curl:

```
# Get all agent manifests
curl -X GET "http://localhost:10090/efm/api/agent-manifests" -H "accept: a
pplication/json"

# Make a note of the manifest identifer to use in the mapping
# Create a mapping
curl -X POST "http://localhost:10090/efm/api/agent-class-manifest-config" -H
 "accept: application/json" --data { "agentClassName": "MyAgentClass", "agen
tManifestId": "27165b44-c54a-4504-8d47-7e3bec421a00" }

# Later, update to map the agent class to a new manifest
curl -X PUT "http://localhost:10090/efm/api/agent-class-manifest-config" -H
 "accept: application/json" --data { "agentClassName": "MyAgentClass", "agen
tManifestId": "90af998a-f7ff-4422-b8fb-2ed08f273959" }
```

See the Swagger UI Section *Agent class manifest config* for more details.

Note: When importing a flow for an agent class, the application uses static strategy irrespective of the globally configured strategy. In this scenario, if there is an existing mapping of agent class to manifest configured and if the input manifest does not match with the configured one, then an illegal state exception (HTTP 409 Conflict) error is thrown. If there is no existing mapping configured, then a mapping of agent class to the input manifest ID will be persisted.

Showing changes after agent upgrade for existing agent class

After agents are upgraded, the changes in manifest, for example, new processors or changes in processor properties, can be made visible by adding the upgraded agents to a totally new agent class. Everything should work out of the box this way. The main drawback is that you need to re-create the existing flow for this new agent class manually which could be problematic for complex flows.

This will be addressed and made more user friendly in the future, however, in the meantime there are some manual steps that can be followed to update the manifest of an existing flow to reflect the new agent capabilities. But the risk is high here as the compatibility between the already existing flow and the updated manifest can not be guaranteed. It always depends on the flow itself.

Important:

- For an existing agent class, all agents should be updated first and all agents should have the same manifest definition before the manual steps are executed.
- If there are incompatibilities between the old and new manifests, this breaks the designed flow and error appears when a change is attempted to be made in the designer.

Perform the following manual steps for existing flows with updated agents, through REST endpoints; check /swagger for endpoint details:

1. Export the flow definition that needs to be updated and store it locally; later this file needs to be updated.

   ```
   GET /designer/{agentClassName}/flows/export
   ```

2. After agent upgrade determines the new manifest ID belonging to the agent, retrieve information for the agent.

   ```
   GET /agents/{id}
   ```

   You can pick any of the upgraded agents. All agents point to the same shared manifest ID.
3. Download the new updated manifest that should be used

   ```
   GET /agent-manifests/{manifest-id-from-previous-step}
   ```

4. Replace the agentManifest section of the exported flow definition (stored in the first step) with the one retrieved in the previous step.
5. In the exported flow definition for the defined processors the bundle versions need to be replaced to be in sync with the versions in the bundles coming from the new manifest.
6. Import the updated flow definition file.

   ```
   POST /designer/{agentClassName}/flows/import
   ```

   In the flow designer the new or updated processors should be available.
7. Double check the processor properties as property name changes could occur in this case. The renamed or deleted property appears as custom property on the processor.

Note: Use this approach with caution as manually updating definitions is inherently error prone.