1.22.08

Using script to integrate custom code

Date published: 2022-09-01
Date modified: 2022-09-01

CLOUDERA

https://docs.cloudera.com/

Legal Notice

Contents

Using script to integrate custom code

Learn how to use scripting to integrate custom code into MiNiFi C++ agent. You can do it either by using the ExecuteScript processor or custom Python processors.

Note: To use this feature, you need MiNiFi C++ Agent 1.22.08 or later versions.

Scripting allows you to integrate custom code into MiNiFi C++ using Python. You can use either of the following two ways to achieve it:

- ExecuteScript processor

  The ExecuteScript processor runs an external stateless script on each processor run, allowing simpler integration.
- Python processors

  Python processors are loaded from external files, and they keep running a function, while retaining the interpreter state, which makes it ideal for tasks that need to keep state between runs, or need some initialization logic to be run only once.

Initial setup

To use scripting, you need to install the required Python version on all target systems.

For Linux, Python 3.6 is required. This is available on CentOS 7. If you are downloading MiNiFi C++ for Linux, you can find a file called nifi-minifi-cpp-...-extra-extensions-centos-bXX.tar.gz. In this file, you can find the minifi-script-extensions.so file. Copy the minifi-script-extensions.so file to the extensions/ directory, so that the MiNiFi C++ agent can load it on startup.

On Linux, there is an additional workaround required to make scripting work if you are using version 1.22.08. This step is unnecessary if you are using 1.22.10 or later. You need to patch the MiNiFi binary to link to Python:

```
patchelf --add-needed libpython3.6m.so MINIFI_HOME/bin/minifi
```

For Windows, the requirements are Python 3.10 and the 64 bit version of the agent. The Python extension is already part of the normal MiNiFi C++ MSI installer, but not enabled by default. You need to enable it during installation if you want to use it.

Using the ExecuteScript processor

When using the ExecuteScript processor, you need to place a Python script on the agents' filesystems (see the asset push feature of EFM), and point the ExecuteScript processor to use that script. On each execution, the Python script is evaluated, and its onTrigger function is called to receive any incoming flow files, and produce the output.

An example script that reverses the content of flow files:

```
#!/usr/bin/env python
import codecs
import time

class ReadCallback:
    def process(self, input_stream):
        self.content = codecs.getreader('utf-8')(input_stream).read()
        return len(self.content)


class WriteReverseStringCallback:
    def __init__(self, content):
        self.content = content
    def process(self, output_stream):
        reversed_content = self.content[::-1]
        output_stream.write(reversed_content.encode('utf-8'))
        return len(reversed_content)


def onTrigger(context, session):
    flow_file = session.get()
    if flow_file is not None:
        read_callback = ReadCallback()
        session.read(flow_file, read_callback)
        session.write(flow_file, WriteReverseStringCallback(read_callback
.content))
        flow_file.addAttribute('python_timestamp', str(int(time.time())))
        session.transfer(flow_file, REL_SUCCESS)
```

Using a Python processor

The workflow of a Python processor is as follows: at startup, the MiNiFi C++ agent reads the Python script directory specified in the minifi.properties file as the value of the nifi.python.processor.dir property (by default, this is MINIFI_HOME/minifi-python). The agent scans the directory for compatible scripts, and automatically registers them for use.

These Python files are evaluated at startup, their onSchedule function is invoked before starting a flow, and their onTr igger function is invoked regularly as the processor is scheduled.

For more details, see the example provided here:

```
#!/usr/bin/env python
def describe(processor):
    processor.setDescription("Adds an attribute to your flow files")

def onInitialize(processor):
    processor.setSupportsDynamicProperties()

def onTrigger(context, session):
    flow_file = session.get()
    if flow_file is not None:
        flow_file.addAttribute("Python attribute", "attributevalue")
        session.transfer(flow_file, REL_SUCCESS)
```

To add or update processors, place them in the Python script directory, and restart the agent.