Java style

SnapCenter Software

Nirupama Sriram
June 14, 2021

Table of Contents

Java style

A Java custom plug-in interacts directly with an application like database, instance and so on.

Limitations

There are certain limitations that you should be aware of while developing a plug-in using Java programing language.

Plug-in characteristic	Java plug-in
Complexity	Low to Medium
Memory footprint	Up to 10-20 MB
Dependencies on other libraries	Libraries for application communication
Number of threads	1
Thread runtime	Less than an hour

Reason for Java limitations

The goal of the SnapCenter Agent is to ensure continuous, safe, and robust application integration. By supporting Java plug-ins, it is possible for plug-ins to introduce memory leaks and other unwanted issues. Those issues are hard to tackle, especially when the goal is to keep things simple to use. If a plug-in's complexity is not too complex, it is much less likely that the developers would have introduced the errors. The danger of Java plug-in is that they are running in the same JVM as the SnapCenter Agent itself. When the plug-in crashes or leaks memory, it may also impact the Agent negatively.

Supported methods

Method	Required	Description	Called when and by whom?
Version	Yes	Needs to return the version of the plug-in.	By the SnapCenter Server or agent to request the version of the plug-in.

Method	Required	Description	Called when and by whom?
Quiesce	Yes	Needs to perform a quiesce on the application. In most cases, this means putting the application into a state where the SnapCenter Server can create a backup (for example, a Snapshot copy).	Before the SnapCenter Server creates a Snapshot(s) copy or performs a backup in general.
Unquiesce	Yes	Needs to perform an unquiesce on the application. In most cases, this means putting the application back into a normal operation state.	After the SnapCenter Server has created a Snapshot copy or has performed a backup in general.
Cleanup	No	Responsible for cleaning up anything that the plug-in needs to clean up.	When a workflow on the SnapCenter Server finish (successfully or with a failure).
clonePre	No	Should perform actions that need to happen before a clone operation is performed.	When a user triggers a "cloneVol" or "cloneLun" action and uses the built-in cloning wizard (GUI/CLI).
clonePost	No	Should perform actions that need to happen after a clone operation was performed.	When a user triggers a "cloneVol" or "cloneLun" action and uses the built-in cloning wizard (GUI/CLI).
restorePre	No	Should perform actions that need to happen before the restore operation is called.	When a user triggers a restore operation.
Restore	No	Responsible for performing a restore/recovery of application.	When a user triggers a restore operation.
appVersion	No	To retrieve application version managed by the plug-in.	As part of ASUP data collection in every workflow like Backup/Restore/Clone.

Tutorial

This section describes how to create a custom plug-in using the Java programming language.

Setting up eclipse

1. Create a new Java Project "TutorialPlugin" in Eclipse
2. Click **Finish**
3. Right click the **new project → Properties → Java Build Path → Libraries → Add External JARs**
4. Navigate to the ../lib/ folder of host Agent and select jars scAgent-5.0-core.jar and common-5.0.jar
5. Select the project and right click the **src folder → New → Package** and create a new package with the name com.netapp.snapcreator.agent.plugin.TutorialPlugin
6. Right-click on the new package and select New → Java Class.
   a. Enter name as TutorialPlugin.
   b. Click the superclass browse button and search for "*AbstractPlugin". Only one result should show up:

      ```
      "AbstractPlugin - com.netapp.snapcreator.agent.nextgen.plugin".
      ```

   c. Click **Finish**.
   d. Java class:

```
package com.netapp.snapcreator.agent.plugin.TutorialPlugin;
import
com.netapp.snapcreator.agent.nextgen.common.result.Describe
Result;
import
com.netapp.snapcreator.agent.nextgen.common.result.Result;
import
com.netapp.snapcreator.agent.nextgen.common.result.VersionR
esult;
import
com.netapp.snapcreator.agent.nextgen.context.Context;
import
com.netapp.snapcreator.agent.nextgen.plugin.AbstractPlugin;
public class TutorialPlugin extends AbstractPlugin {
  @Override
  public DescribeResult describe(Context context) {
    // TODO Auto-generated method stub
    return null;
  }
  @Override
  public Result quiesce(Context context) {
    // TODO Auto-generated method stub
    return null;
  }
  @Override
  public Result unquiesce(Context context) {
    // TODO Auto-generated method stub
    return null;
  }
  @Override
  public VersionResult version() {
    // TODO Auto-generated method stub
    return null;
  }
}
```

Implementing the required methods

Quiesce, unquiesce, and version are mandatory methods that each custom Java plug-in must implement.

The following is a version method to return the version of the plug-in.

```
@Override
public VersionResult version() {
    VersionResult versionResult = VersionResult.builder()
                                                .withMajor(1)
                                                .withMinor(0)
                                                .withPatch(0)
                                                .withBuild(0)
                                                .build();
    return versionResult;
}
```

```
Below is the implementation of quiesce and unquiesce method. These will be
interacting with   the application, which is being protected by SnapCenter
Server. As this is just a tutorial, the
application part is not explained, and the focus is more on the
functionality that SnapCenter   Agent provides the following to the plug-
in developers:
```

```
@Override
  public Result quiesce(Context context) {
    final Logger logger = context.getLogger();
    /*
      * TODO: Add application interaction here
    */
```

```
logger.error("Something bad happened.");
logger.info("Successfully handled application");
```

```
    Result result = Result.builder()
                    .withExitCode(0)
                    .withMessages(logger.getMessages())
                    .build();
    return result;
}
```

The method gets passed in a Context object. This contains multiple helpers, for example a Logger and a Context Store, and also the information about the current operation (workflow-ID, job-ID). We can get the logger by calling final Logger logger = context.getLogger();. The logger object provides similar methods known from other logging frameworks, for example, logback. In the result object, you can also specify the exit code. In this example, zero is returned, since there was no issue. Other exit codes can map to different failure scenarios.

Using result object

The Result object contains the following parameters:

Parameter	Default	Description
Config	Empty config	This parameter can be used to send config parameters back to the server. It can be parameters that the plug-in wants to update. Whether this change is actually reflected in the config on the SnapCenter Server is dependent on the APP_CONF_PERSISTENCY=Y or N parameter in the config.
exitCode	0	Indicates the status of the operation. A "0" means the operation was executed successfully. Other values indicate errors or warnings.
Stdout	Empty List	This can be used to transmit stdout messages back to the SnapCenter Server.
Stderr	Empty List	This can be used to transmit stderr messages back to the SnapCenter Server.
Messages	Empty List	This list contains all the messages that a plug-in wants to return to the server. The SnapCenter Server displays those messages in the CLI or GUI.

The SnapCenter Agent provides Builders (Builder Pattern) for all its result types. This makes using them very straightforward:

```
Result result = Result.builder()
                    .withExitCode(0)
                    .withStdout(stdout)
                    .withStderr(stderr)
                    .withConfig(config)
                    .withMessages(logger.getMessages())
                    .build()
```

For example, set exit code to 0, set lists for Stdout and Stderr, set config parameters and also append the log

messages that will be sent back to the server. If you do not need all the parameters, send only the ones that are needed. As each parameter has a default value, if you remove .withExitCode(0) from the code below, the result is unaffected:

```
Result result = Result.builder()
                      .withExitCode(0)
                      .withMessages(logger.getMessages())
                      .build();
```

VersionResult

The VersionResult informs the SnapCenter Server the plug-in version. As it also inherits from Result, it contains the config, exitCode, stdout, stderr, and messages parameters.

Parameter	Default	Description
Major	0	Major version field of the plug-in.
Minor	0	Minor version field of the plug-in.
Patch	0	Patch version field of the plug-in.
Build	0	Build version field of the plug-in.

For example:

```
VersionResult result = VersionResult.builder()
                                    .withMajor(1)
                                    .withMinor(0)
                                    .withPatch(0)
                                    .withBuild(0)
                                    .build();
```

Using the Context Object

The context object provides the following methods:

Context method	Purpose
String getWorkflowId();	Returns the workflow id that is being used by the SnapCenter Server for the current workflow.
Config getConfig();	Returns the config that is being send from the SnapCenter Server to the Agent.

Workflow-ID

The workflow-ID is the id that the SnapCenter Server uses to refer to a specific running workflow.

Config

This object contains (most) of the parameters that a user can set in the config on the SnapCenter Server. However, due to security reasons, some of those parameters may get filtered on the server side. Following is an example on how to access to the Config and retrieve a parameter:

```
final Config config = context.getConfig();
String myParameter =
config.getParameter("PLUGIN_MANDATORY_PARAMETER");
```

""// myParameter" now contains the parameter read from the config on the SnapCenter Server If a config parameter key doesn't exist, it will return an empty String ("").

Exporting the plug-in

You must export the plug-in to install it on the SnapCenter host.

In Eclipse perform the following tasks:

1. Right click on the base package of the plug-in (in our example com.netapp.snapcreator.agent.plugin.TutorialPlugin).
2. Select **Export → Java → Jar File**
3. Click **Next**.
4. In the following window, specify the destination jar file path: tutorial_plugin.jar The plug-in's base class is named TutorialPlugin.class, the plug-in must be added to a folder with the same name.

If your plug-in depends on additional libraries, you can create the following folder: lib/

You can add jar files, on which the plug-in is dependent (for example, a database driver). When SnapCenter loads the plug-in, it automatically associates all the jar files in this folder with it and adds them to the classpath.