Develop a plug-in for your application

SnapCenter Software 4.6

NetApp
February 21, 2022

Table of Contents

Develop a plug-in for your application

Overview

The SnapCenter Server enables you to deploy and manage your applications as plug-ins to SnapCenter. Applications of your choice can be plugged into the SnapCenter Server for data protection and management capabilities.

SnapCenter enables you to develop custom plug-ins using different programming languages. You can develop a custom plug-in using Perl, Java, BATCH, or other Scripting languages.

To use custom plug-ins in SnapCenter, you must perform the following tasks:

- Create a plug-in for your application using the instructions in this guide
- Create a description file
- Export the custom plug-in to install it on the SnapCenter host
- Upload the plug-in zip file into SnapCenter Server

Generic plug-in handling in all API calls

For every API call, use the following information:

- Plug-in parameters
- Exit codes
- Log error messages
- Data consistency

Use Plug-in parameters

A set of parameters are passed to the plug-in as part of every API call made. The following table lists the specific information for the parameters.

Parameter	Purpose
ACTION	Determines the workflow name. For example, discover, backup, fileOrVolRestore or cloneVolAndLun
RESOURCES	Lists resources to be protected. A resource is identified by UID and Type. The list is presented to the plug-in in the following format: “<UID>,<TYPE>;<UID>,<TYPE>”. For example, “Instance1,Instance;Instance2\\DB1,Database”
APP_NAME	Determines which plug-in is being used. For example, DB2, MYSQL. SnapCenter Server has built-in support for the listed applications. This parameter is case sensitive.

Parameter	Purpose
APP_IGNORE_ERROR	(Y or N) This causes SnapCenter to exit or not exit when an application error is encountered. This is useful when you are backing up multiple databases and do not want a single failure to stop the backup operation.
<RESOURCE_NAME>__APP_INSTANCE_USERNAME	SnapCenter credential is set for the resource.
<RESOURCE_NAME>_APP_INSTANCE_PASSWORD	SnapCenter credential is set for the resource.
<RESOURCE_NAME>_<CUSTOM_PARAM>	Every Resource level custom key value is available to plug-ins prefixed with “<RESOURCE_NAME>_”. For example, if a custom key is “MASTER_SLAVE” for a resource named “MySQLDB”, then it will be available as MySQLDB_MASTER_SLAVE

Use exit codes

The plug-in returns the status of the operation back to the host by means of exit codes. Each code has a specific meaning and the plug-in uses the right exit code to indicate the same.

The following table depicts error codes and their meaning.

Exit code	Purpose
0	Successful operation.
99	Requested operation is not supported or implemented.
100	Failed operation, skip unquiesce, and exit. Unquiesce is by default.
101	Failed operation, continue with backup operation.
other	Failed operation, run unquiesce, and exit.

Log error messages

The error messages are passed from the plug-in to the SnapCenter Server. The message includes the message, log level, and time stamp.

The following table lists levels and their purposes.

Parameter	Purpose
INFO	informational message
WARN	warning message
ERROR	error message
DEBUG	debug message
TRACE	trace message

Preserve data consistency

Custom plug-ins preserve data between operations of the same workflow execution. For example, a plug-in can store data at the end of quiesce, which can be used during unquiesce operation.

The data to be preserved is set as part of result object by plug-in. It follows a specific format and is described in detail under each style of plug-in development.

PERL-based development

You must follow certain conventions while developing the plug-in using PERL.

- Contents must be readable
- Must implement mandatory operations setENV, quiesce, and unquiesce
- Must use a specific syntax to pass results back to the agent
- The contents should be saved as <PLUGIN_NAME>.pm file

Available operations are

- setENV
- version
- quiesce
- unquiesce
- clone_pre, clone_post
- restore_pre, restore
- cleanup

General plug-in handling

Using results object

Every custom plug-in operation must define the results object. This object sends messages, exit code, stdout, and stderr back to the host agent.

Results object:

```
my $result = {
```

```
      exit_code => 0,
      stdout => "",
      stderr => "",
};
```

Returning the results object:

```
return $result;
```

Preserving data consistency

It is possible to preserve data between operations (except cleanup) as part of same workflow execution. This is done using key-value pairs. The key-value pairs of data are set as part of result object and are retained and available in the subsequent operations of same workflow.

The following code sample sets the data to be preserved:

```
my $result = {
  exit_code => 0,
  stdout => "",
  stderr => "",
};
  $result->{env}->{'key1'} = 'value1';
  $result->{env}->{'key2'} = 'value2';
  ….
  return $result
```

The above code sets two key-value pairs, which are available as input in the subsequent operation. The two key-value pairs are accessible using the following code:

```
sub setENV {
    my ($self, $config) = @_;
    my $first_value = $config->{'key1'};
    my $second_value = $config->{'key2'};
    …
}
```

```
=== Logging error messages
```

Each operation can send messages back to the host agent, which displays and stores the content. A message contains the message level, a timestamp, and a message text. Multiline messages are supported.

```
Load the SnapCreator::Event Class:
my $msgObj = new SnapCreator::Event();
my @message_a = ();
```

Use the msgObj to capture a message by using the collect method.

```
$msgObj->collect(\@message_a, INFO, "My INFO Message");
$msgObj->collect(\@message_a, WARN, "My WARN Message");
$msgObj->collect(\@message_a, ERROR, "My ERROR Message");
$msgObj->collect(\@message_a, DEBUG, "My DEBUG Message");
$msgObj->collect(\@message_a, TRACE, "My TRACE Message");
```

Apply messages to the results object:

```
$result->{message} = \@message_a;
```

Using plug-in stubs

Custom plug-ins must expose plug-in stubs. These are methods that the SnapCenter Server calls, based on a workflow.

Plug-in Stub	Optional/Required	Purpose
setENV	required	This stub sets the environment and the configuration object. Any environment parsing or handling should be done here. Each time a stub is called, the setENV stub is called just before. It is only required for PERL-style plug-ins.
Version	Optional	This stub is used to get application version.

Plug-in Stub	Optional/Required	Purpose
Discover	Optional	This stub is used to discover application objects like instance or database hosted on the agent or host. The plug-in is expected to return discovered application objects in specific format as part of the response. This stub is only used in case the application is integrated with SnapDrive for Unix. Linux file system (Linux Flavors) is supported. AIX/Solaris (Unix Flavors) are not supported.
discovery_complete	Optional	This stub is used to discover application objects like instance or database hosted on the agent or host. The plug-in is expected to return discovered application objects in specific format as part of the response. This stub is only used in case the application is integrated with SnapDrive for Unix. Linux file system (Linux flavors) is supported. AIX and Solaris (Unix flavors) are not supported.
Quiesce	required	This stub is responsible for performing a quiesce, which means placing application into a state where you can create a Snapshot copy. This is called before Snapshot copy operation. The metadata of application to be retained should be set as part of response, which shall be returned during subsequent clone or restore operations on corresponding storage Snapshot copy in the form of configuration parameters.

Plug-in Stub	Optional/Required	Purpose
Unquiesce	required	This stub is responsible for performing a unquiesce, which means placing application into a normal state. This is called after you create a Snapshot copy.
clone_pre	optional	This stub is responsible for performing preclone tasks. This assumes you are using the built-in SnapCenter Server cloning interface and is triggered when performing clone operation.
clone_post	optional	This stub is responsible for performing post clone tasks. This assumes you are using the built-in SnapCenter Server cloning interface and is triggered only when performing clone operation.
restore_pre	optional	This stub is responsible for performing prerestore tasks. This assumes you are using the built-in SnapCenter Server restore interface and is triggered while performing restore operation.
Restore	optional	This stub is responsible for performing application restore tasks. This assumes you are using the built-in SnapCenter Server restore interface and is only triggered when performing restore operation.

Plug-in Stub	Optional/Required	Purpose
Cleanup	optional	This stub is responsible for performing cleanup after backup, restore, or clone operations. Cleanup can be during normal workflow execution or in the event of a workflow failure. You can infer the workflow name under which cleanup is called by referring to configuration parameter ACTION, which can be backup, cloneVolAndLun, or fileOrVolRestore. The configuration parameter ERROR_MESSAGE indicates if there was any error while executing the workflow. If ERROR_MESSAGE is defined and NOT NULL, then cleanup is called during workflow failure execution.
app_version	Optional	This stub is used by SnapCenter to get application version detail managed by the plug-in.

Plug-in package information

Every plug-in must have following information:

```
package MOCK;
our @ISA = qw(SnapCreator::Mod);
=head1 NAME
MOCK - class which represents a MOCK module.
=cut
=head1 DESCRIPTION
MOCK implements methods which only log requests.
=cut
use strict;
use warnings;
use diagnostics;
use SnapCreator::Util::Generic qw ( trim isEmpty );
use SnapCreator::Util::OS qw ( isWindows isUnix getUid
createTmpFile );
use SnapCreator::Event qw ( INFO ERROR WARN DEBUG COMMENT ASUP
CMD DUMP );
my $msgObj = new SnapCreator::Event();
my %config_h = ();
```

Operations

You can code various operations like setENV, Version, Quiesce, and Unquiesce, which are supported by the custom plug-ins.

setENV operation

The setENV operation is required for plug-ins created using PERL. You can set the ENV and can easily access plug-in parameters.

```
sub setENV {
    my ($self, $obj) = @_;
    %config_h = %{$obj};
    my $result = {
      exit_code => 0,
      stdout => "",
      stderr => "",
    };
    return $result;
}
```

Version operation

The version operation returns the application version information.

```
sub version {
  my $version_result = {
    major => 1,
    minor => 2,
    patch => 1,
    build => 0
  };
  my @message_a = ();
  $msgObj->collect(\@message_a, INFO, "VOLUMES
$config_h{'VOLUMES'}");
  $msgObj->collect(\@message_a, INFO,
"$config_h{'APP_NAME'}::quiesce");
  $version_result->{message} = \@message_a;
  return $version_result;
}
```

Quiesce operations

Quiesce operation performs application quiesce operation on resources listed in the RESOURCES parameter.

```
sub quiesce {
  my $result = {
      exit_code => 0,
      stdout => "",
      stderr => "",
  };
  my @message_a = ();
  $msgObj->collect(\@message_a, INFO, "VOLUMES
$config_h{'VOLUMES'}");
  $msgObj->collect(\@message_a, INFO,
"$config_h{'APP_NAME'}::quiesce");
  $result->{message} = \@message_a;
  return $result;
}
```

Unquiesce operation

Unquiesce operation is required to unquiesce the application. The list of resources is available in the RESOURCES parameter.

```
sub unquiesce {
  my $result = {
      exit_code => 0,
      stdout => "",
      stderr => "",
  };
  my @message_a = ();
  $msgObj->collect(\@message_a, INFO, "VOLUMES
$config_h{'VOLUMES'}");
  $msgObj->collect(\@message_a, INFO,
"$config_h{'APP_NAME'}::unquiesce");
  $result->{message} = \@message_a;
  return $result;
}
```

NATIVE style

SnapCenter supports non-PERL programming or scripting languages to create plug-ins. This is known as NATIVE style programming, which can be script or BATCH file.

The NATIVE-style plug-ins must follow certain conventions given below:

The plug-in must be executable

- For Unix systems, the user who runs the agent must have execute privileges on the plug-in

- For Windows systems, PowerShell plug-ins must have the suffix .ps1, other windows scripts must have either .cmd or .bat suffix and must be executable by the user
- The plug-ins must react to command-line argument like "-quiesce", "-unquiesce"
- The plug-ins must return exit code 99 incase an operation or function is not implemented
- The plug-ins must use a specific syntax to pass results back to the server

General plug-in handling

Logging error messages

Each operation can send messages back to the server, which displays and stores the content. A message contains the message level, a timestamp, and a message text. Multiline messages are supported.

Format:

```
SC_MSG#<level>#<timestamp>#<message>
SC_MESSAGE#<level>#<timestamp>#<message>
```

Using plug-in stubs

SnapCenter plug-ins must implement plug-in stubs. These are methods that the SnapCenter Server calls based on a specific workflow.

Plug-in Stub	Optional/Required	Purpose
quiesce	required	This stub is responsible for performing a quiesce. It places the application into a state where we can create a Snapshot copy. This is called before storage Snapshot copy operation.
unquiesce	required	This stub is responsible for performing a unquiesce. It places the application in a normal state. This is called after storage Snapshot copy operation.
clone_pre	optional	This stub is responsible for performing pre clone tasks. This assumes that you are using the built-in SnapCenter cloning interface and also is only triggered while performing action "clone_vol or clone_lun".

Plug-in Stub	Optional/Required	Purpose
clone_post	Optional	This stub is responsible for performing post clone tasks. This assumes you are using the built-in SnapCenter cloning interface and also is only triggered while performing "clone_vol or clone_lun" operations.
restore_pre	Optional	This stub is responsible for performing pre restore tasks. This assumes you are using the built-in SnapCenter restore interface and is only triggered while performing restore operation.
restore	optional	This stub is responsible for performing all restore actions. This assumes you are not using built-in restore interface. It is triggered while performing restore operation.

Examples

Windows PowerShell

Check if the script can be executed on your system. If you cannot execute the script, set Set-ExecutionPolicy bypass for the script and retry the operation.

```
if ($args.length -ne 1) {
  write-warning "You must specify a method";
  break;
}
function log ($level, $message) {
  $d = get-date
  echo "SC_MSG#$level#$d#$message"
}
function quiesce {
  $app_name = (get-item env:APP_NAME).value
  log "INFO" "Quiescing application using script $app_name";
  log "INFO" "Quiescing application finished successfully"
}
function unquiesce {
  $app_name = (get-item env:APP_NAME).value
  log "INFO" "Unquiescing application using script $app_name";
  log "INFO" "Unquiescing application finished successfully"
}
  switch ($args[0]) {
    "-quiesce" {
    quiesce;
}
"-unquiesce" {
    unquiesce;
}
default {
    write-error "Function $args[0] is not implemented";
    exit 99;
  }
}
exit 0;
```

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

Plug-in characteristic	Java plug-in
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.
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).

Method	Required	Description	Called when and by whom?
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.

Custom plug-in in SnapCenter

Custom plug-in in SnapCenter

The custom plug-in created using Java, PERL, or NATIVE style can be installed on the host using SnapCenter Server to enable data protection of your application. You must have exported the plug-in to install it on the SnapCenter host using the procedure provided in this tutorial.

Creating a plug-in description file

For every plug-in created, you must have a description file. The description file describes the details of the plug-in. The name of the file must be Plugin_descriptor.xml.

Using plug-in descriptor file attributes and its significance

Attribute	Description
Name	Name of the plug-in. Alpha numeric characters are allowed. For example, DB2, MYSQL, MongoDB For plug-ins created in NATIVE style, ensure that you do not provide the extension of the file. For example, if the plug-in name is MongoDB.sh, specify the name as MongoDB.
Version	Plug-in version. Can include both major and minor version. For example, 1.0, 1.1, 2.0, 2.1
DisplayName	The plug-in name to be displayed in SnapCenter Server. If multiple versions of the same plug-in are written, ensure that the display name is the same across all versions.
PluginType	Language used to create the plug-in. Supported values are Perl, Java and Native. Native plug-in type includes Unix/Linux shell scripts, Windows scripts, Python or any other scripting language.
OSName	The host OS name where the plug-in is installed. Valid values are Windows and Linux. It is possible for a single plug-in to be available for deployment on multiple OS types, like PERL type plug-in.
OSVersion	The host OS version where plug-in is installed.
ResourceName	Name of resource type that the plug-in can support. For example, database, instance, collections.

Attribute	Description
Parent	In case, the ResourceName is hierarchically dependent on another Resource type, then Parent determines the parent ResourceType. For instance, DB2 plug-in, the ResourceName “Database” has a parent “Instance”.
RequireFileSystemPlugin	Yes or No. Determines if the recovery tab is displayed in the restore wizard.
ResourceRequiresAuthentication	Yes or No. Determines if the resources, which are auto discovered or have not been auto discovered need credentials to perform the data protection operations after discovering the storage.
RequireFileSystemClone	Yes or No. Determines if the plug-in requires FileSystem plug-in integration for clone workflow.

An example of the Plugin_descriptor.xml file for custom plug-in DB2 is as follows:

```
<Plugin>
<SMSServer></SMSServer>
<Name>DB2</Name>
<Version>1.0</Version>
<PluginType>Perl</PluginType>
<DisplayName>Custom DB2 Plugin</DisplayName>
<SupportedOS>
<OS>
<OSName>windows</OSName>
<OSVersion>2012</OSVersion>
</OS>
<OS>
<OSName>Linux</OSName>
<OSVersion>7</OSVersion>
</OS>
</SupportedOS>
<ResourceTypes>
<ResourceType>
<ResourceName>Database</ResourceName>
<Parent>Instance</Parent>
</ResourceType>
<ResourceType>
<ResourceName>Instance</ResourceName>
</ResourceType>
</ResourceTypes>
<RequireFileSystemPlugin>no</RequireFileSystemPlugin>
<ResourceRequiresAuthentication>yes</ResourceRequiresAuthentication>
<SupportsApplicationRecovery>yes</SupportsApplicationRecovery>
</Plugin>
```

Creating a ZIP file

After a plug-in is developed and a descriptor file is created, you must add the plug-in files and the Plugin_descriptor.xml file to a folder and zip it.

You must consider the following before creating a ZIP file:

- The script name must be same as the plug-in name.
- For PERL plug-in, the ZIP folder must contain a folder with the script file and the descriptor file must be outside this folder. The folder name must be the same as the plug-in name.
- For plug-ins other than the PERL plug-in, the ZIP folder must contain the descriptor and the script files.
- The OS version must be a number.

Examples:

- DB2 plug-in: add DB2.pm and Plugin_descriptor.xml file to “DB2.zip”.

- Plug-in developed using Java: add jar files, dependent jar files, and Plugin_descriptor.xml file to a folder and zip it.

Uploading the plug-in ZIP file

You must upload the plug-in ZIP file to SnapCenter Server so that the plug-in is available for deployment on the desired host.

You can upload the plug-in using the UI or cmdlets.

UI:

- Upload the plug-in ZIP file as part of **Add** or **Modify Host** workflow wizard
- Click **"Select to upload custom plug-in"**

PowerShell:

- Upload-SmPluginPackage cmdlet

  For example, PS> Upload-SmPluginPackage -AbsolutePath c:\DB2_1.zip

  For detailed information about PowerShell cmdlets, use the SnapCenter cmdlet help or see the cmdlet reference information.

SnapCenter Software Cmdlet Reference Guide.

Deploying the custom plug-ins

The uploaded custom plug-in is now available for deployment on the desired host as part of the **Add** and **Modify Host** workflow. You can have multiple version of plug-ins uploaded to the SnapCenter Server and you can select the desired version to deploy on a specific host.

For more information on how to upload the plug-in see, Add hosts and install plug-in packages on remote hosts