Technical Report

# Snap Creator Framework Plug-ins — Programming Guide

Keith Tenzer, NetApp
December 2010 | TR-3890

TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# 1 INTRODUCTION

This technical report provides an overview of NetApp® Snap Creator and Snap Creator Framework plug-ins, and describes the procedures to create, install, and configure Snap Creator Framework plug-ins.

# 2 SNAP CREATOR OVERVIEW

Snap Creator provides a central framework that integrates NetApp Snapshot® technology with applications that do not have SnapManager® products in a way that is seamless to our customers. Normally, this requires a customized script to interface with the application and the NetApp storage system. These customized scripts are written over and over every day. Snap Creator saves time and provides our customers with an excellent, highly reliable solution.

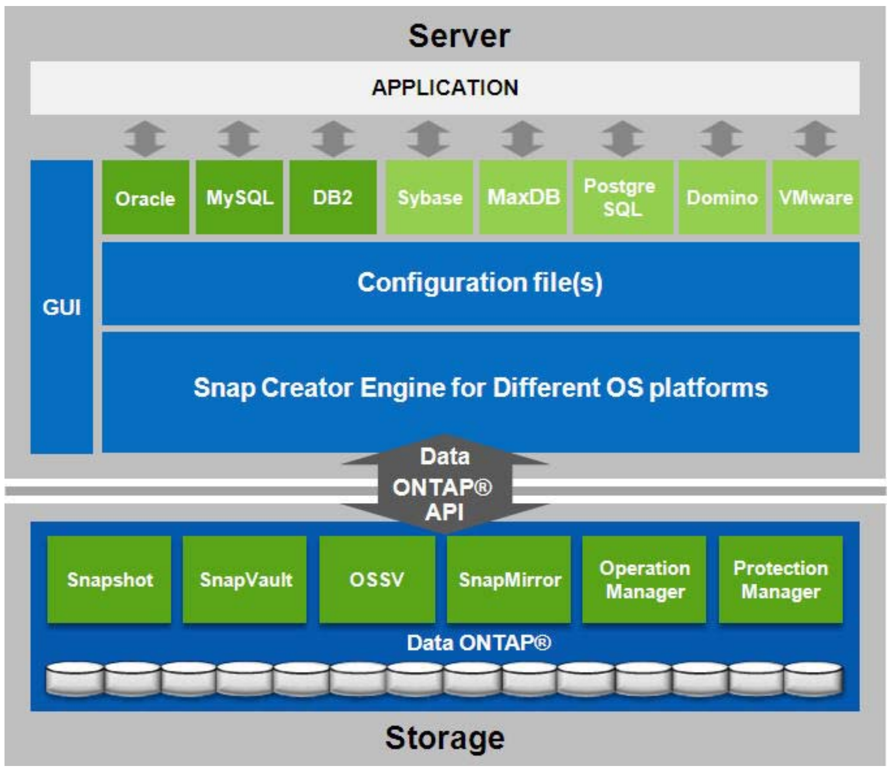
Because integration for most applications is unique and challenging, most products support only a few select applications. In contrast, Snap Creator provides application integration through modules or plug-ins that enables it to support any application anywhere. Snap Creator offers a framework in which you can integrate application consistency (backup) scripts or use the built-in Snap Creator modules.

Currently there are three supported application modules available for Snap Creator, with more on the way. The three plug-in modules are Oracle®, DB2, and MySQL®. Snap Creator handles communication with NetApp storage and performs various tasks such as policy-based Snapshot management (via API or SnapDrive®), an optional LUN or volume clone, seamless integration with SnapMirror® or SnapVault®, and integration with Operations Manager or Protection Manager. Snap Creator is by no means a replacement for our SnapManager and SnapDrive products; in fact, it integrates with both products.

## 2.1 SNAP CREATOR ARCHITECTURE

Figure 1 shows the Snap Creator architecture.

**Figure 1) Snap Creator architecture.**



## 2.2 SNAP CREATOR FEATURES

Snap Creator contains the following features:

- Brower-based GUI on NetApp Web Framework (NWF)
- GUI Backup Scheduler
- Integration with SnapDrive
- Integration with storage controller SnapVault scheduler
- Integration with the following NetApp technologies through ZAPI: Snapshot, SnapVault, SnapMirror, LUN cloning, volume cloning, and IGROUP mapping
- Supported integration modules for Oracle, DB2, and MySQL
- SnapCreator Framework supports plug-ins for Lotus Notes, PostgreSQL, MaxDB, Sybase®, VMware®, SME, and SMSQL
- Integration with any application or database that runs in an open systems environment (you can write the application backup script or plug-in module if one does not exist)
- Integration with NetBackup™, CommVault, or any backup software with CLI commands
- Ability to configure multiple Snapshot or SnapVault policies with different retentions
- Both Snapshot and SnapVault policies are managed from Snap Creator
- Ability to create and manage Snapshot copies and SnapVault retentions across multiple volumes and storage controllers
- Support for volume and qtree SnapMirror
- Support for FAS and vFiler® units
- Support for consistency groups (consistent Snapshot copies across multiple volumes or appliances)
- Support for 1–to-1, fan-in, and fan-out SnapMirror or SnapVault relationships
- Support for cascading SnapMirror Snapshot->SnapVault->SnapMirror
- A dynamic configuration file in which users can pass their own user-defined values
- Granular error logging and the ability to send error messages via e-mail or other tools
- Integration with Operations Manager (the ability to create events in Operations Manager)
- Integration with Protection Manager (Snap Creator backups can be registered in Protection Manager)
- Integration with Open Systems SnapVault (OSSV)
- Support for password encryption so clear-text passwords are not saved in the configuration file
- Support for both volume, file, and SnapVault restore but without direct application integration
- Agent for central backup management
- Global configuration files

Snap Creator **does not** do the following:

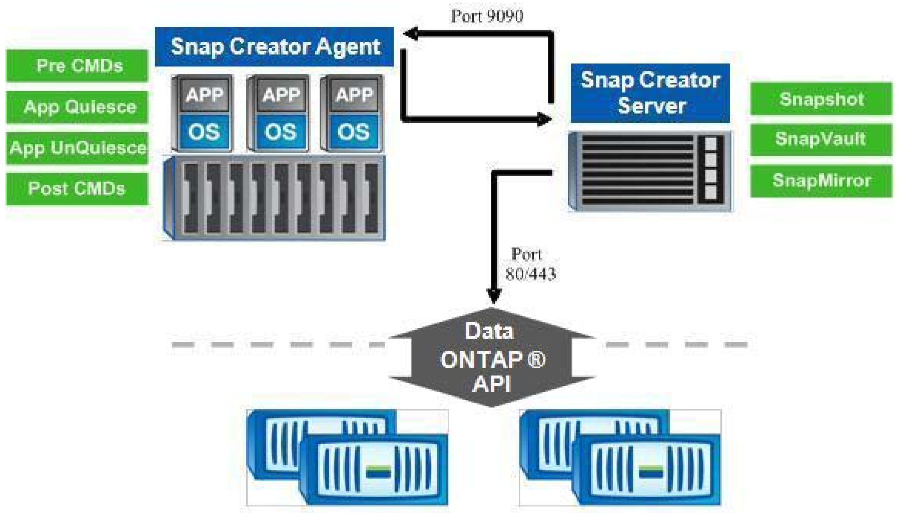
- Snap Creator is not a replacement for SnapManager products.
- Snap Creator does not directly handle the mounting of cloned LUNs; mount commands or SnapDrive is required for that.
- Snap Creator does not create or manage SnapVault and SnapMirror relationships.

## 3 SNAP CREATOR AGENT

The Snap Creator agent provides the ability to run backups centrally, meaning all Snap Creator configurations can be stored on the same system and all backup jobs can be scheduled from the same host.

Figure 2 depicts the Snap Creator agent architecture.

Figure 2) Snap Creator agent.



To quiesce the application, Snap Creator uses its agent, which runs as a daemon. The default port is 9090, but any port can be chosen. SOAP over the HTTP protocol is used for communication. All application modules are already built into the agent, so all applications supported by Snap Creator are also supported by its agent. In addition to the application modules, all PRE commands, POST commands, and APP commands can be executed remotely through the agent. This provides the ability to mount file systems or perform extra application processing. The agent has an access file under `/path/so/scAgent_v<#>/config/agent.conf` where certain commands can be allowed. By default, everything is denied, which means only built-in modules can execute commands through the agent. PRE or POST scripting commands or scripts need to be allowed in the `agent.conf`.

## 4 SNAP CREATOR FRAMEWORK PLUG-INS

The Snap Creator Framework plug-ins allows integration of any application with Snap Creator and thus with NetApp storage. This enables the creation of reusable plug-ins that provide application integration. One has always been able to integrate scripts or commands within the Snap Creator Framework; however, that does not provide a seamless solution and allow reusability.

Snap Creator Framework plug-ins currently support Perl as the programming language, but plans are in place to add Java™ and possibly other language support. The Perl programming language was chosen first because it is well known and used as a scripting language.

All created plug-ins will run as source code and do not require compilation. Each plug-in has optional and required methods that need to be defined. In addition, objects used with the Snap Creator Framework itself are accessible. The Snap Creator Framework plug-in consists of a constructor, objects, and methods.

Some plug-ins such as Oracle, DB2, and MySQL are already compiled into Snap Creator while other community plug-ins run external and are distributed as source code. There is, from a code perspective, no difference between the source plug-ins and those that are compiled.

## 4.1 SNAP CREATOR FRAMEWORK WORKFLOWS

The Snap Creator Framework provides several generic workflows. Within these workflows are several placeholders where calls can be made to a plug-in (if one is provided). The main workflows that involve the Snap Creator Framework plug-ins are as follows: `snap`, `clone_vol`, `clone_lun`, `restore`, `quiesce`, `unquiesce`, `discover`, and `scdump`. All of these workflows are accessed using the `--action <workflow>` CLI parameter.

**snap** – This is the standard backup workflow. If a plug-in is used, the `quiesce` and `unquiesce` calls will be sent to the defined plug-in. The `quiesce` and `unquiesce` calls are required for any plug-in. The other methods are optional.

**clone_vol** – This is a workflow for cloning a volume using FlexVol®. If a plug-in is used, `quiesce`, `unquiesce`, `clone_pre`, and `clone_post` calls will be sent to the defined plug-in. The `quiesce` and `unquiesce` calls are required; `clone_pre` and `clone_post` are optional though recommended. Since a Snapshot copy is the basis for a clone, Snap Creator will first create a consistent Snapshot copy of the application using the `quiesce`/`unquiesce` calls. Next, `clone_pre` calls (if defined) will be sent to the plug-in. Anything that must be done within the application prior to cloning should be handled in the `clone_pre` method. After `clone_pre` calls, Snap Creator will clone the volume and map or export the cloned volume to a host. Once the clone has been created and mapped, a call will be sent to the plug-in to handle `clone_post` (if defined). Anything that should be done to complete the clone process from an application perspective should be handled in the `clone_post` method. File system mounting or unmounting is not handled directly in the Snap Creator Framework plug-in. That can be done within the plug-in `clone_pre` and `clone_post` methods or by using the built-in `CLONE_CREATE_CMDS`. There are two command sets: `PRE_CLONE_CREATE_CMD<##>` and `POST_CLONE_CREATE_CMD<##>`. Anything the user adds to these commands will be executed on the clone target. Simple mount commands or even SnapDrive can be used here for mounting or unmounting.

**clone_lun** – This is a workflow for cloning a LUN using LUN clone. NetApp strongly recommends using `clone_vol` because FlexClone® has certain advantages. A LUN clone or the `clone_lun` workflow should only be used if no FlexClone license exists or if the LUN clone is temporary, meaning it is deleted between backups. Unlike FlexClone, Snap Creator will support only one LUN clone. Multiple LUN clones are not supported. Otherwise, everything that applies to `clone_vol` workflow also applies here.

**restore** – This is a workflow for performing a restore. There are two ways you can handle the restore: as plug-in driven or Framework driven.

- Plug-in driven: In this method, the plug-in is responsible for doing everything including restore of data using the snapObj or direct ONTAP API calls. Snap Creator will simply send a restore call to the plug-in and the plug-in is responsible for everything else. An example of this can be seen in the VMware (VIBE.pm) plug-in. If this method for restore is desired, then the restore method in the plug-in must be defined and the following parameter in the config file must be set `APP_DEFINED_RESTORE=Y`.
- Framework driven: In this method, Snap Creator provides a user interface (CLI only) that guides the user through the restore of data. The restore interface supports Snapshot copy restore (primary storage) and SnapVault restore (secondary storage). Before the restore interface is triggered, a `restore_pre` call is sent to the plug-in (if defined). After the restore is complete (the user enters “c” to continue or “n” for next) in the restore interface, a `restore_post` call is sent to the plug-in (if defined). If this method for restore is desired, then the `restore_pre` and `restore_post` methods in the plug-in should be defined. The following parameter in the config file must be set: `APP_DEFINED_RESTORE=N`.

**quiesce –** This is a workflow for quiescing an application. Snap Creator will only send the `quiesce` call to the plug-in. To use this workflow the `quiesce` method must be defined in the plug-in. The purpose of this workflow is to support multiple or federated application backups. Since the config file can only support one agent, to quiesce multiple applications on different hosts a separate quiesce operation and config are needed for each. You can create a main config and use the `APP_QUIESCE_CMDS` to call Snap Creator to serially quiesce applications located on different hosts.

**unquiesce –** This is a workflow for unquiescing an application. Snap Creator will only send the unquiesce call to the plug-in. To use this workflow the unquiesce method must be defined in the plug-in. The purpose of this workflow is to support multiple or federated application backups. Since the config file can only support one agent, to unquiesce multiple applications on different hosts a separate unquiesce operation and config are needed for each. You can create a main config and use the `APP_UNQUIESCE_CMDS` to call Snap Creator to serially unquiesce applications located on different hosts.

**discover –** This is a workflow for performing discovery of an application. Snap Creator will only send the discover call to the plug-in. To use this workflow, the discover method must be defined in the plug-in. Discover allows you to detect and check the application storage layout and that you are backing up the correct data. Discover has the ability to dynamically change config parameters on the fly and even make those changes persistent in the config file. To perform auto discovery, the parameter `APP_AUTO_DISCOVERY=Y` must be set in the config file. To make any parameter changes persistent the parameter `APP_CONF_PERSISTENCE=Y` must be set.

**scdump –** This is a workflow for dumping support information for NGS. Snap Creator will gather all logs and configs associated with a profile and zip them. Additionally, a `scdump.txt` file will be created under the `logs/<profile>`directory. The `scdump.txt` will also be included in the zip file. The `scdump.txt` file contains information about Snap Creator, storage controller, SnapDrive (if used), the operating system, and the application. When performing scdump, Snap Creator will send a scdump call to the plug-in (if defined). The plug-in is then responsible for returning SnapDrive, operating system, and application information.

### 4.2 SNAPCREATOR FRAMEWORK PLUG-IN STRUCTURE

The SnapCreator Framework plug-in has a defined structure. All classes, libraries, and global settings required by the plug-in must be loaded at the beginning of the plug-in before any methods are defined. Snap Creator Framework plug-in consists of three important entities: the constructor, methods, and objects. The constructor should always be the same and can be copied from the example. Methods are the calls that the Framework sends to the plug-in. This is where most of the time will be spent in plug-in development. Objects are reusable code from the Framework itself. Using Snap Creator objects is not required, but it will save time and provide consistency. Any object that is used in the Snap Creator Framework is available to the plug-in.

# 5 CONSTRUCTOR

The purpose of the constructor is to instantiate a plug-in as an object. The Snap Creator Framework plug-in defines an application object called `appObj`. This object assumes the properties of your plug-in. For example, if your plug-in is for Oracle, then `appObj` would inherit properties of Oracle, thus giving Snap Creator for that instance Oracle integration capabilities.

Snap Creator determines what plug-in to use by the parameter `APP_NAME`, which is set in the config file.

For every plug-in, the constructor must be created as follows:

```
sub new {
 my $invocant = shift;
 my $class = ref($invocant) || $invocant;
   my $self = {
       @_
   };
   bless ($self, $class);
   return $self;
}
```

# 6 OBJECTS

An object is simply a reusable piece of functionality defined within the Snap Creator Framework. Objects are then exposed to the plug-ins, meaning that all of that code can be reused with little or no effort. This makes the plug-in Framework very powerful. An example of the most commonly used object is the message object, or `msgObj`. This allows one to handle error logging in the plug-in by using the same mechanism as Snap Creator. The use of objects is optional but strongly recommended because it saves a lot of time and provides more seamless integration with the Snap Creator Framework.

## 6.1 MESSAGE OBJECT

The message object is responsible for handling logging. Messages are collected and sent to Snap Creator, where they are displayed and saved in the log file. The message object needs to be instantiated for it to be usable. This is the same for all objects. To do so you would add the following to the beginning of your plug-in:

```
use Snap Creator::Event qw ( INFO ERROR WARN DEBUG COMMENT ASUP CMD DUMP );
```

```
my $msgObj = new Snap Creator::Event();
```

This method has a different function for each type of message possibility. The functions and examples are as follows:

- **info** – Creates an informational message:

  ```
  my @message_a = ();
  $msgObj->collect(\@message_a, INFO, "this is a info collect message");
  [Thu Aug 12 13:30:21 2010] INFO: this is a info message
  ```

- **warn** – Creates a warning message:

  ```
  my @message_a = ();
  $msgObj->collect(\@message_a, WARN, "this is a warn collect message");
  [Thu Aug 12 13:30:21 2010] WARN: this is a warn message
  ```

- **error** – Creates an error message:

  ```
  my @message_a = ();
  $msgObj->collect(\@message_a, ERROR, "this is a error collect message");
  [Thu Aug 12 13:30:21 2010] ERROR: this is a error message
  ```

- **debug** – Creates a debug message:

  ```
  my @message_a = ();
  $msgObj->collect(\@message_a, DEBUG, "this is a debug collect message");
  [Thu Aug 12 13:30:21 2010] DEBUG: this is a debug message
  ```

At the end of your method, you can send all collected messages back to Snap Creator by doing the following:

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

```
return $result;
```

## 6.2 OS OBJECT

The OS object allows basic OS integration. It can return the OS version, create temp files, and execute commands. This is a critical piece of any plug-in and provides standard ways for integrating with the operating system. To use the OS object it needs to be instantiated. This is the same for all objects. To do so you would add the following to the beginning of your plug-in:

```
use Snap Creator::Util::OS;
```

```
my $osObj = new Snap Creator::Util::OS();
```

Once the OS object has been instantiated you can use it. This object has several different methods or handlers. The methods and examples are as follows:

| OS Object | Description |
|---|---|
| isWindows | Returns true if running under Windows®; otherwise undef is returned |
| isUnix | Returns true if running under UNIX®; otherwise undef is returned |
| getWindowsVersion | Returns generic OS information for Windows |
| getUnixVersion | Returns generic OS information for UNIX |
| getSnapDriveVersion | Returns version of SnapDrive |
| getUid | Returns the UID for the user if running under UNIX; this method only supports UNIX |
| createTmpFile(@lines) | Creates a temp file based on an array of lines passed in; this method supports both UNIX and Windows |
| execute($cmd) | Executes the command passed in via a variable; the results are returned in the following hash: `%result_h = (` `exit_code => (),` `stdout => (),` `stderr => ()` `);` |

### OS OBJECT BASIC EXAMPLES

```
my $isWindows=$osObj->isWindows();
my $isUnix=$osObj->isUnix();
my $getWindowsVersion=$osObj->getWindowsVersion();
my $getUnixVersion=$osObj->getUnixVersion();
my $getSnapDriveVersion=$osObj->getSnapDriveVersion();
my $uid=$osObj->getUid();
```

### OS OBJECT CREATE TEMP FILE

To use the createTmpFile method, simply pass the commands to be executed. A temp file named `$cmd_file` will be created for encapsulating these commands. An example of the format of this file is as follows:

```
$cmd_file, $content) = $osObj->createTmpFile(("connect / as sysdba;", "show
parameter CLUSTER_DATABASE;", "exit;"));
```

Where:

`$cmd_file` is the location and name of the temp file created

`$content` is the information inside the temp file

The above example shows three sqlplus statements that are added to the temp file. For certain utilities like sqlplus it is much easier to add all the statements you want into a temp file first and then execute everything together.

### OS OBJECT EXECUTE COMMAND

The createTmpFile and execute command methods work closely together. Of course a command can be executed without creating a temp file, but when creating a temp file you would execute it using this method. In this example, `$sqlplus` is the command to be executed. In this case, we are executing sqlplus and passing it to our temp file, which contains the SQL commands to be executed.

```
my $sqlplus="sqlplus /nolog \@$cmd_file";

my $result = $osObj->execute($sqlplus);
```

To check the results of our **`$sqlplus`** command, do the following:

```
if ($result->{exit_code} != 0) {
$msgObj->collect(\@message_a, ERROR, "Oracle SQL*Plus command [$sqlplus] failed with
return code $result->{exit_code}");
push (@message_a, @{$result->{message}}) if exists ($result->{message});
 } else {
$msgObj->collect(\@message_a, INFO, "Oracle SQL*Plus command [$sqlplus] completed
successfully");
}
```

## 6.3 CONFIG OBJECT

The config object allows Snap Creator to read/write the config file and encrypt/decrypt passwords. The main use for plug-ins is to decrypt passwords. It may be necessary to store encrypted passwords for your plug-in. To use the config object it needs to be instantiated. This is the same for all objects. To do so you would add the following to the beginning of your plug-in:

```
use Snap Creator::Util::Config;

my $configObj = new Snap Creator::Util::Config();
```

Once the config object has been instantiated you can use it. This object has several different methods or handlers. The read/set functions you can get access to through other means (see the discover method). The password encrypt/decrypt functions and examples are as follows:

**pwdEncrypt**—Encrypts a clear-text password

```
my $encrypted_password=$configObj->pwdEncrypt(“clear_password”);
```

**pwdDencrypt**— Decrypts a clear-text password

```
my $decrypted_password=$configObj->pwdDecrypt(“encrypted_password”);
```

## 6.4 CONNECTION OBJECT

The connection object creates a connection to a NetApp storage controller, vFiler (direct or through tunnel), or DFM server. To create a connection, a username/password, API major/minor number, port, and protocol are required. Once the connection is saved within the object it can be reused. The connect object it needs to be instantiated for it to be usable. This is the same for all objects. To do so you would add the following to the beginning of your plug-in:

```
my $connectObj = new Snap Creator::Connect();
```

Once the connect object has been instantiated you can use it. This object has several different methods or handlers. The methods and examples are as follows:

- **getApiVer**(`$filer, $user, $pwd`)

  Returns two values: the `apiMajorVersion` and `apiMinorVersion`. Both `apiMajorVersion` and `apiMinorVersion` are required to create a `filer`, `vfiler`, or `dfm` object.

  `$filer` is the storage system name

  `$user` is the user name

  `$pwd` is the password

  ```
  my ($apiMajorVersion,$apiMinorVersion)=$connectObj->getApiVer($filer,$user,$pwd);
  ```

- **filer**(`$filer, $apiMajorVersion, $apiMinorVersion, $user, $pwd`)

  Returns a hashref of the filer connect object. The connection object maintains persistency so the connection is only done once per filer. This object is then used to execute ZAPI commands.

  `$filer` is the storage system name

  `$apiMajorVersion` is the API major number

  `$user` is the user name

  `$pwd` is the password

  ```
  my $connectRef=$connectObj-
  >filer($filer,$apiMajorVersion,$apiMinorVersion,$user,$pwd);
  ```

- **vfiler**(`$filer, $vfiler, $apiMajorVersion, $apiMinorVersion, $user, $pwd`)

  Returns a hashref of the `vfiler` connect object. The connection object maintains persistency so the connection is only done once per `vfiler`. This object is then used to execute ZAPI commands.

  `$filer` is the storage system name

  `$vfiler` is the vFiler name for tunnel

  `$apiMajorVersion` is the API major number

  `$apiMinorVersion` is the API minor number

  `$user` is the user name

  `$pwd` is the password

  ```
  my $connectRef=$connectObj-
  >vfiler($filer,$vfiler,$apiMajorVersion,$apiMinorVersion,$user,$pwd);
  ```

- **dfm**(`$filer, $vfiler, $apiMajorVersion, $apiMinorVersion, $user, $pwd`)

Returns a hashref of the dfm connect object. The connection object maintains persistency so the connection is only done once per DFM server. This object is then used to execute ZAPI commands.

`$dfmServer` is the DMF server name

`$apiMajorVersion` is the API major number

`$apiMinorVersion` is the API minor number

`$user` is the user name

`$pwd` is the password

**`my $connectRef=$connectObj->dfm($dfmServer, $apiMajorVersion, $apiMinorVersion, $user,$pwd);`**

### CONNECT OBJECT CONNECT TO STORAGE SYSTEM

Before connecting to the storage system, obtain the API version and use this information to establish a connection to the storage system. Save this connection as `$ntapConnect,` which can be layer passed to other objects such as the `$snapObj` to perform actions on the storage system.

```
my ($apiMajorVersion,$apiMinorVersion)=();
eval {
($apiMajorVersion,$apiMinorVersion)=$connectObj->getApiVer($filer,$user,$pwd);
  };
 # Error Logging
 if ($@) {
       die “Couldn’t get api version: $@\n”;
        }
```

Once we have the AP version we can establish our connection as follows:

```
my $connectRef=();
 eval {
   $ntapConnect=$connectObj-
>filer($filer,$apiMajorVersion,$apiMinorVersion,$user,$pwd);
       };
# Error Logging
 if ($@) {
       die “Couldn’t get api version: $@\n”;
        }
```

**Note**: Executing our method or handler within an eval allows us to trap an error condition in the downstream class.

## 6.5 SNAP OBJECT

The snap object allows all Snapshot operations that Snap Creator supports. This includes Snapshot restore, consistency group, and more. The main purpose of this object in the case of plug-ins is to restore. If a plug-in uses the restore method it is required to perform its own Snapshot restore, in which case this object is very useful. It may also be desirable to turn off Snapshot copy creation in Snap Creator and handle that within the plug-in. To use the snap object it needs to be instantiated. This is the same for all objects. To do so you would add the following to the beginning of your plug-in:

**`my $snapObj = new Snap Creator::Snap();`**

Once the snap object has been instantiated you can use it. This method has several different functions. The functions and examples are as follows:

- **cgstart**(`$filer`, `$ntapConnect`, `$newSnapshotName`, `$consistency_group_a`)

The `cgstart` method I/O fences all volumes for a specified storage controller. CG Snapshot is a two-step process for each storage controller; first use cgstart and then cgcommit.

This method returns the CG ID, required to do the cgcommit.

```
my $cgID=$snapObj-
>cgstart($filer,$ntapConnect,$newSnapshotName,\@consistency_group_a);
```

Where:

`$filer` is the storage controller name

`$ntapConnect` is the connection object

`$newSnapshotName` is the name of the Snapshot copy to be created

`$consistency_group_a` is the array reference of the volumes that you want to back up for a given storage system

- **cgcommit**(`$filer`, `$ntapConnect`, `$newSnapshotName`, `$cgID`)

  The cgcommit method creates a Snapshot copy of all volumes and removes the I/O fence. CG Snapshot copy is a two-step process for each storage controller; first perform cgstart and then cgcommit.

  `$filer` is the storage controller name

  `$ntapConnect` is the connection object

  `$cgID` is the CG ID that is returned from the cgstart method

  ```
  $snapObj->cgcommit ($cgFiler,$ntapConnect,$newSnapshotName,$cgID);
  ```

- **snap** (`$filer`, `$ntapConnect`, `$volume`, `$newSnapshotName`)

  The snap method will creates a Snapshot copy of a given filer:volume.

  `$filer` is the storage controller name

  `$ntapConnect` is the connection object

  `$volume` is the volume of which a Snapshot copy is to be created

  `$newSnapshotName` is the name of the Snapshot copy to be created

  ```
  $snapObj->snap($filer,$ntapConnect,$volume,$newSnapshotName);
  ```

- **svsnap**(`$filer`, `$ntapConnect`, `$volume`, `$sched`)

  The svsnap method creates a Snapshot copy compatible with the built-in SnapVault filer scheduler.

  `$volume` is the volume of which a Snapshot copy is to be created

  `$snapName` is the current Snapshot copy name

  `$newSnapName` is the name to which your'd like to rename your Snapshot copy

  `$sched` is the name of the SnapVault filer schedule, i.e., nightly

  ```
  $snapObj->svsnap ($filer,$ntapConnect,$volume,$ENV{'SNAP_TYPE'});
  ```

- **svunsched**`($filer, $ntapConnect, $volume)`

  The svunsched method will remove all SnapVault schedules for a volume.

`$volume` is the volume of which a Snapshot copy is to be created

**`$snapObj->svunsched($filer,$ntapConnect,$volume);`**

- **svsched**(`$filer, $ntapConnect, $volume, $sched, $retention`)

  The svsched method will set the Data ONTAP SnapVault scheduler to the Snap Creator retention policy.

  `$volume` is the volume of which a Snapshot copy is to be created

  `$snapName` is the current Snapshot copy name

  `$newSnapName` is the name to which your'd like to rename your Snapshot copy

  `$sched` is the name of the SnapVault filer schedule, i.e., nightly

  `$retention` is the retention for SnapVault scheduler

  ```
  $snapObj->svsched($filer,$ntapConnect,$volume,$ENV{'SNAP_TYPE'},$ntapRetentions_h{$ENV{'SNAP_TYPE'}});
  ```

- **rename**(`$filer, $ntapConnect, $volume, $snapName, $newSnapName`)

  This method renames a Snapshot copy for a given filer:volume.

  `$filer` is the storage controller name

  `$ntapConnect` is the connection object

  `$volume` is the volume of which a Snapshot copy is to be created

  `$snapName` is the current Snapshot copy name

  `$newSnapName` is the name to which your'd like to rename your Snapshot copy

  ```
  $snapObj->rename($filer,$ntapConnect,$volume,$snapInfo_h{$filer}{$volume}{$key},$newSnapName);
  ```

- **delete**(`$filer, $ntapConnect, $volume, $snapName`)

  The delete method will delete a Snapshot copy for a given filer:volume.

  `$filer` is the storage controller name

  `$ntapConnect` is the connection object

  `$volume` is the volume of which a Snapshot copy is to be created

  `$snapName` is the current Snapshot copy name

  ```
  $snapObj->delete($filer,$ntapConnect,$volume,$snapInfo_h{$filer}{$volume}{$key});
  ```

- **list**(`$filer, $ntapConnect, $volume`)

  The list method will list Snapshot copies for a given filer:volume. It returns an array of hashes with each hash being a Snapshot copy.

  `$filer` is the storage controller name

  `$ntapConnect` is the connection object

`$volume` is the volume of which a Snapshot copy is to be created

```
my $result=$snapObj->list($filer,$ntapConnect,$volume);
foreach my $key (@{$result}) {
my $snapName=$key->child_get_string("name");
my $accessTime=$key->child_get_string("access-time");
}
```

- **volRestore**`($filer, $ntapConnect, $volume, $snapName)`

  The volRestore method will restore a Snapshot copy for a given filer:volume.

  `$filer` is the storage controller name

  `$ntapConnect` is the connection object

  `$volume` is the volume of which a Snapshot copy is to be created

  `$snapName` is the name of Snapshot copy to be restored

  ```
  $snapObj->volRestore($filer,$ntapConnect,$volume,$snapName);
  ```

- **fileRestore**`($filer, $ntapConnect, $snapName, $lunPath, $newPath)`

  The fileRestore method will restore a LUN in a given filer:volume.

  `$filer` is the storage controller name

  `$ntapConnect` is the connection object

  `$snapName` is the name of Snapshot copy to be restored

  `$lunPath` is the path to the LUN we want to restore `/vol/volume/lun`

  `$newPath` is the path to a new LUN in case we do an alternate path restore; optional

  ```
  $snapObj->fileRestore($filer,$ntapConnect,$snapName,$orgPath);
  ```

# 7 METHODS

Plug-ins expose certain methods to Snap Creator. A method is simply a function that a plug-in provides to Snap Creator, for example, to quiesce. Snap Creator handles everything generically, so when using quiesce within our backup workflow or any other plug-in method Snap Creator will execute that code as specified by the plug-in. This is the heart of the integration. Some methods are required and others are optional. If the required methods are not defined, Snap Creator throws an exception.

Snap Creator uses the optional methods, if they are defined within the plug-in; otherwise that functionality is skipped. From a backup workflow perspective, think of methods as stubs—things that may or may not be implemented. As stubs are generic they can be anything; for example, Oracle quiesce.

## 7.1 SETENV (REQUIRED)

The `setENV` method is required and allows the Snap Creator Framework to pass the plug-in to the information stored in the config file. Snap Creator does not use a database but rather stores information in a config file. The file is simply a bunch of key/value pairs, for example: <key>=<value>. All key/value pairs are read and stored in the `%config_h` hash. The `%config_h` hash is sent to the plug-in through the `setENV` method. The plug-in can then read anything in the hash, meaning anything stored in the Snap Creator config file. This is very handy because the plug-in, in order to be dynamic, requires information like database name, user, etc. The `setENV` method is a special method that does not require returning a result like other methods. Its only purpose is to pass information stored in the config file to the plug-in. This occurs whenever a method is called. The `setENV` method should be implemented as follows:

```
sub setENV {
    my ($self, $obj) = @_;
    %config_h = ();
    %config_h = %{$obj};
     ### Parse plug-in specific parameters ###
}
```

**Note:** Do not make changes to this method.

## 7.2 QUIESCE (REQUIRED)

The `quiesce` method is required and defines how to quiesce or begin backup mode for the given application plug-in. Below is an example that simply prints `<Plug-in Name>::quiesce`.

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

This functionality is triggered as follows:

- **Standalone CLI**
  ```
  ./Snap Creator --profile <profile> --action quiesce <Optional Arguments>
  ```
- **Snap Creator backup workflow**
  ```
  ./Snap Creator --profile <Profile> --action snap --policy <Policy> <Optional
  Arguments>
  ```

## 7.3 UNQUIESCE (REQUIRED)

The unquiesce method is required and defines how to unquiesce or end backup mode for the given application plug-in. Below is an example that simply prints `<Plug-in Name>::unquiesce`.

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

This functionality is triggered as follows:

- **Standalone CLI**
  ```
  ./Snap Creator --profile <profile> --action unquiesce <Optional Arguments>
  ```
- **Snap Creator backup workflow**
  ```
  ./Snap Creator --profile <Profile> --action snap --policy <Policy> <Optional
  Arguments>
  ```

## 7.4 DISCOVER (OPTIONAL)

The discover method is optional and provides unique functionality. The purpose of this method is to allow dynamic discovery of the NetApp storage. This can be implemented in two way—dynamic discovery and volume validation.

### DYNAMIC DISCOVERY

This implementation will not only detect and determine that the NetApp volumes you are backing up are correct, but also update the information. The `VOLUMES` parameter in the Snap Creator config file can be overwritten and updated through the discover method. In fact, any parameter stored in the config file can be overwritten dynamically in the application plug-in through the discover method.

### VOLUME VALIDATION

This implementation simply lists the volumes that are to be backed up and returns them to Snap Creator. A parameter called `VOLUME_VALIDATION` can be enabled `VOLUME_VALIDATION=DATA`. Snap Creator will throw an exception if the volumes that were discovered do not match what is saved in the config file.

Additionally, using the die function, you can cause Snap Creator to exit if the configuration isn't as required. However, using the `VOLUME_VALIDATION` interface is much cleaner.

Below is an example of how you can update and print the `VOLUMES` parameter:

```
sub discover {
    my $result = {
        exit_code => 0,
        stdout => "",
        stderr => "",
    };
    my %env = ();
    $env{'VOLUMES'}="filer1:vol1";
    $result->{env} = \%env;
    my @message_a = ();
    $msgObj->collect(\@message_a, INFO, "$config_h{'VOLUMES}::discover");
    $result->{message} = \@message_a;
    return $result;
}
```

**Note**: When updating parameters such as `VOLUMES` in Snap Creator, be sure to follow examples for how the information should be stored. Use the `default.conf` for such examples.

This functionality is triggered as follows:

- **Standalone CLI**

  ```
  ./Snap Creator --profile <profile> --action discover <Optional Arguments>
  ```
- **Snap Creator backup workflow**

  ```
  ./Snap Creator --profile <Profile> --action snap --policy <Policy> <Optional
  Arguments>
  ```
- **Snap Creator restore workflow**

  ```
  ./Snap Creator --profile <Profile> --action restore --policy <Policy> <Optional
  Arguments>
  ```

## 7.5 CLONE_PRE (OPTIONAL)

The `clone_pre` method is optional and defines how to prepare an application for cloning. This includes all steps that need to be taken before creating a NetApp volume clone. Below is an example that simply prints `<Plug-in Name>::clone_pre`.

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

This functionality is triggered as follows:

**Snap Creator backup workflow**

```
./Snap Creator --profile <Profile> --action clone_vol --policy <Policy> <Optional
Arguments>
```

## 7.6 CLONE_POST (OPTIONAL)

The `clone_post` method is optional and defines how to complete the application cloning process. This would include all the steps that need to be taken after creating a NetApp volume clone. Below is an example that simply prints `<Plug-in Name>::clone_post`.

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

This functionality is triggered as follows:

**Snap Creator backup workflow**

```
./Snap Creator --profile <Profile> --action clone_vol --policy <Policy> <Optional
Arguments>
```

## 7.7 RESTORE_PRE (OPTIONAL)

The `restore_pre` method is optional and defines the steps to be performed before starting a restore. Snap Creator has a built-in CLI restore interface where you can restore a NetApp volume or LUN. The `restore_pre` method is executed before entering that CLI interface. This allows you to just deal with application-specific things relating to restoring and not any NetApp API calls. Below is an example that simply prints `<Plug-in Name>::restore_pre.x`

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

This functionality is triggered as follows:

**Snap Creator restore workflow**

```
./Snap Creator --profile <Profile> --action restore --policy <Policy> <Optional
Arguments>
```

## 7.8 RESTORE_POST (OPTIONAL)

The `restore_post` method is optional and defines the required steps to take after completing a restore. This would include all the steps that need to be taken after a NetApp Snapshot restore is done. Snap Creator has a built-in CLI restore interface where you can restore a NetApp volume or LUN. The `restore_post` method is executed after leaving that CLI interface. This allows you to just deal with application-specific things relating to restoring and not any NetApp API calls. Below is an example that simply prints `<Plug-in Name>::restore_post`.

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

This functionality is triggered as follows:

**Snap Creator restore workflow**

```
./Snap Creator --profile <Profile> --action restore --policy <Policy> <Optional
Arguments>
```

## 7.9 RESTORE (OPTIONAL)

The restore method allows the application plug-in to take over every aspect of restore, including performing the NetApp Snapshot restore. The difference between `restore_pre` / `restore_post` and the restore method is that the `restore_pre` / `restore_post` is built around the Snap Creator restore CLI. NetApp recommends this method if you want to control every aspect of the restore. Below is an example that simply prints `<Plug-in Name>::restore.`

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

This functionality is triggered as follows:

**Snap Creator restore workflow**

```
./Snap Creator --profile <Profile> --action restore --policy <Policy> <Optional
Arguments>
```

## 7.10 SCDUMP (OPTIONAL)

The `scdump` method provides SnapDrive, operating system, and application information to Snap Creator when performing `scdump`. The `scdump` method collects all information for a given profile and zips it together. For community plug-ins, `scdump` is not required; however, you may find it useful for troubleshooting and it certainly aids in support if others are going to use the plug-in. The `scdump` method requires that information sent back to Snap Creator is formatted correctly. This is done when returning the result; `$result->osDump` is used to store OS and SnapDrive information. `$result->dbDump` is used to store application information. Below is an example for Oracle (a built-in supported plug-in), which demonstrates how to collect and send this information back to Snap Creator:

```
sub scdump {
    my $self = shift;
    my $result = {
        exit_code => 0,
        stdout => "",
        stderr => "",
    };
    my @message_a = ();
  my $osDump = {};
   my $dbDump = {};

### Get OS Version ###

$msgObj->collect(\@message_a, DEBUG, "Collecting OS information");

if (defined (Snap Creator::Util::OS->isWindows())) {

    $result=Snap Creator::Util::OS->getWindowsVersion();

    if ($result->{exit_code} != 0) {

    $msgObj->collect(\@message_a, ERROR, "Collection of OS information failed - $@");

    } else {

        $osDump->{os}="$result->{rval}";

        $msgObj->collect(\@message_a, DEBUG, "Collection of OS information completed
successfully");

    }

} elsif (defined Snap Creator::Util::OS->isUnix()) {

   $result=Snap Creator::Util::OS->getUnixVersion();

    if ($result->{exit_code} != 0) {

        $msgObj->collect(\@message_a, ERROR, "Collection of OS information failed -
$@");

    } else {

        $osDump->{os}="$result->{rval}";

        $msgObj->collect(\@message_a, DEBUG, "Collection of OS information completed
successfully");

    }

}
### Get SnapDrive Version ###
$msgObj->collect(\@message_a, DEBUG, "Collecting SnapDrive information");
if ($config_h{'SNAPDRIVE'} =~ m/^y$/i) {
    $result=Snap Creator::Util::OS->getSnapDriveVersion();
    if ($result->{exit_code} != 0) {
        $msgObj->collect(\@message_a, ERROR, "Collection of SnapDrive information
failed - $@");
    } else {
        $osDump->{snapdrive}="$result->{rval}";
        $msgObj->collect(\@message_a, DEBUG, "Collection of SnapDrive information
completed successfully");
    }
} else {
    my $sdVersion = "none";
    $osDump->{snapdrive}="$sdVersion";
    $msgObj->collect(\@message_a, DEBUG, "Collection of SnapDrive information
completed successfully");
}
### Oracle Information ###
foreach my $db (keys %dbConfig_h) {
    $msgObj->collect(\@message_a, DEBUG, "Collecting Oracle information for database
$db");
    my $user=$dbConfig_h{$db}{'user'};
    my $version=$checkOraVersion->($db,$user);
    if ($version->{exit_code} != 0) {
        $msgObj->collect(\@message_a, ERROR, "Collecting Oracle information for
database $db failed");
        push (@message_a, @{$version->{message}}) if exists ($version->{message});
        $result->{exit_code} = 1;
    } else {
        my $dbHashRef = ();
        $dbDump->{$db}=$version->{stdout};
        $msgObj->collect(\@message_a, DEBUG, "Oracle information for database $db
collected successfully");
    }
}
$result->{message} = \@message_a;
my $return = ();
$return->{result} = $result;
$return->{osDump} = $osDump;
$return->{dbDump} = $dbDump;
return $return;
}
```

This functionality is triggered as follows:

**Snap Creator SCDUMP workflow**

```
./Snap Creator --profile <Profile> --action scdump <Optional Arguments>
```

## 8 PLUG-IN INPUT / OUTPUT

A Snap Creator plug-in will receive input and must produce output for each method or handler.

### INPUT

When a plug-in method or handler is called, the plug-in receives as input all parameters set in the config file. They are sent as a hash to the setENV method. Any parsing required by plug-in specific config parameters should be done within the setENV method. Any parameter in the config file will show up as a key/value pair in the `%config_h` hash.

```
sub setENV {
   my ($self, $obj) = @_;
   %config_h = ();
   %config_h = %{$obj};
   ### Parse Config Parameters ###
}
```

### OUTPUT

Whenever a plug-in method or handler is called, it must return the result of the operation. Usually this is not one operation but many, so all messages are collected using the `$msgObj` and, along with that, the exit code (either 0 or 1) is returned using the `%result` hash. The `%result` hash should be defined at the beginning of the method or handler as follows:

```
my @message_a = ();
my $result = {
   exit_code => 0,
   stdout => "",
   stderr => "",
 };
```

Any message that needs to be collected and saved should be done as follows:

```
$msgObj->collect(\@message_a, INFO, "<Enter Message Text Here>");
```

At the end of the method or handler the messages must be bound to the `%result` hash and it must be returned. An example of this is as follows:

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

## 9 SNAP CREATOR CONFIG FILE

Using the Snap Creator config file is critical to plug-in development. It will be necessary to set up parameters in the config file to enable plug-ins to be dynamic. It may be necessary to store information like database or application name/instance, user/password, CLI command paths used, and so on. This information can be saved in the Snap Creator config file that gets passed to the plug-in. All that is required is to add the key/value pair to the config file (key=value) for each parameter. As mentioned above, that information will be sent to the plug-in through the setENV method. Below is an example that assumes that we need the database name and user name for our plug-in to be set dynamically:

In the Snap Creator config file add the following parameters:

```
MY_DATABASE=db1
```

```
MY_USER=user1
```

In the plug-in to access these parameter in the quiesce method for example do the following:

```
$config_h{' MY_DATABASE '}");
$config_h{' MY_USER '}");
Below is an example of how to print those values from the plug-ins quiesce method:
sub quiesce {
    my $result = {
        exit_code => 0,
        stdout => "",
        stderr => "",
    };
    my @message_a = ();
         $msgObj->collect(\@message_a, INFO, "Database Name = $config_h{' MY_DATABASE
'}");
    $msgObj->collect(\@message_a, INFO, "User Name = $config_h{' MY_USER '}");
    $result->{message} = \@message_a;
    return $result;
}
```

# 10 CONFIGURING PLUG-INS

There are a few requirements for Snap Creator to use a plug-in. The plug-in must conform to the plug-in naming convention, located under the plug-in directory, and be set in the APP_NAME parameter.

## PLUG-IN NAMING CONVENTION

A Snap Creator plug-in must conform to the following naming convention:

`<PLUG-IN NAME>.pm`

For example `TEST.pm`

## PLUG-IN DIRECTORY

For Snap Creator to use a plug-in, it must be located in the plug-in directory.
If the plug-in is supposed to run on the Snap Creator Server (scServer), that directory location is as follows:

`/path/to/scServer_v<##>/plug-ins`

If the plug-in is supposed to run on the Snap Creator agent (scAgent), that directory location is as follows:

`/path/to/scAgent_v<##>/plug-ins`

## APP_NAME PARAMETER

The APP_NAME parameter tells Snap Creator which plug-in to load. If no agent is involved, Snap Creator Server simply looks in its plug-in directory for `<APP_NAME>.pm`. If the agent is involved, Snap Creator Server sends a SOAP request to the agent. The agent will then look in its plug-in directory for `<APP_NAME>.pm`. Once the plug-in is found it will be loaded and used for all defined methods. Below is an example for using a plug-in called `TEST.pm`:

```
APP_NAME=TEST
```

**`scServer`:** Loads plug-in from `/path/to/scServer_v<##>/plug-ins/TEST.pm`

**`scAgent`:** Loads plug-in from `/path/to/scAgent_v<##>/plug-ins/TEST.pm`

### CREATING CUSTOM PLUG-IN PARAMETERS

All dynamic data required for a plug-in must be saved in the config file. Information such as application or database name, user, CLI commands, and so on, should all be provided as config file parameters so that the plug-in is reusable and isn't required to be updated to use it. The format for storing data in the config file is as follows:

```
<Parameter>=<Value>
```

For example: `MY_DATABASE=db1:user1`

In the above example, a parameter called `MY_DATABASE` with a value of db1:user1 is stored in the config file. Anything stored in the config file in this format will be sent to the plug-in with any call. This is done using the setENV method. This means that whenever Snap Creator sends a call to the plug-in it in fact sends both the setENV and the call itself. This enables every method to receive all the config file parameters. Parsing config parameters should for this reason occur in the setENV method. Config parameters can be referenced in the setENV method of the plug-in as follows:

```
$config_h{'<Parameter>'}

$config_h{'MY_DATABASE'}
```

Here is an example using the setENV method of how to parse and set the config parameter `MY_DATABASE`. This requires defining the `$db` and `$user` scalars globally, meaning outside any method. This means that in each method we could reference $db, which would be db1, and user, which would be user1. These scalars could automatically be changed by editing the `MY_DATABASE` parameter in the `config` file.

```
sub setENV {
   my ($self, $obj) = @_;
   %config_h = ();
   %config_h = %{$obj};
   ### Parse Parameters ###
   ($db,$user) = split (":",$config_h{'MY_DATABASE'} );
}
```

## 11 DEBUGGING PLUG-INS

There are several tools and techniques for debugging Snap Creator plug-ins. Some are provided by Perl itself while others are built into the Snap Creator Framework.

### 11.1 PERL SYNTAX

Before checking to see if a plug-in will run, check its syntax using the following command:

```
perl –c plug-ins/<plug-in>/<plug-in>.pm
```

Perl will check the plug-in for any syntax error. The drawback of using this command is that any objects or classes used from the SnapCreator framework will throw errors. This is because Perl itself, objects, and classes are packaged inside the binary and therefore the external Perl interpreter sees them as missing.

NetApp recommends commenting out any classes coming from Snap Creator and also commenting out the lines where they are used.

## 11.2 EXIT CODES

The Snap Creator Framework has different exit codes. Some have to do with the Framework while others have to do with plug-ins. The plug-in must send back the exit code for a method or handler. This will always be either 0 for success or 1 for failure.

### SNAP CREATOR FRAMEWORK EXIT CODES

The Snap Creator Framework has the following exit codes:

**Table 1) Framework exit codes.**

| Exit Code | Description |
|---|---|
| 0 | Exit code 0 means the Snap Creator action was successful. |
| 1 | Exit code 1 means the Snap Creator action failed but the error was within the Framework. |
| 2 | Exit code 2 means the Snap Creator action failed but the error was within the plug-in. |

### PLUG-IN EXIT CODES

The plug-in has the following exit codes:

**Table 2) Plug-in exit codes.**

| Exit Code | Description |
|---|---|
| 0 | Exit code 0 means the plug-in method or handler completed successfully. |
| 1 | Exit code 1 means the plug-in method or handler failed. |
| 99* | Exit code 99 means the plug-in method or handler was not defined. This is not an error condition unless the handler or method is quiesce, unquiesce, or setENV, which are all required. |
| 100* | Exit code 100 means that a watchdog-forced unquiesce occurred and the autounquiesce functionality is disabled for the current operation. The watchdog (if enabled) is a process that gets spawned whenever the Snap Creator agent receives a quiesce request. It waits until a defined timeout and, if no unquiesce request has been received, it forces unquiesce. |

* These are special exit codes used internally within the Snap Creator Framework. A plug-in should always return either 0 or 1 for a given method or handler.

### 11.3 AGENT DEBUG

If using the Snap Creator agent, it is important to run the agent in debug mode. This allows you to see any print statements used for troubleshooting. To start the agent in debug mode, run the following command:

```
./Snap Creator --start-agent <port> --verbose --debug
```

If the plug-in code is changed it is necessary to restart the agent. Config file changes, however, does not require restart of the agent.

### 11.4 SERVER DEBUG

If the agent is used, both debug on the agent side and server side should be enabled. On the server side debug will also show all API calls to the storage controller and the return `stdout` as well as `stderr` for any commands being run. This is very helpful for seeing the return of CLI commands used to perform various actions on the application through the plug-in. To start the server in debug mode, run the following command:

```
./Snap Creator --profile <profile> --action <action> --policy <policy> --
verbose –debug
```

If the agent is not used, debugging will only be done on the server side. Because the server does not run as a daemon, it does not have to be restarted for changes to the plug-in to take effect.

## 12 APPENDICES

### 12.1 EXAMPLE UML REPRESENTATION

**Table 3) UML representation.**

| Plug-in |
|---|
| Attributes |
| Operations |

| Plug-in |
|---|
| Attributes |
| setENV()<br>quiesce()<br>unquiesce()<br>discover()<br>scdump()<br>restore()<br>restore_pre()<br>restore_post()<br>clone_pre<br>clone_post |

## 12.2 EXAMPLE PLUG-IN

The following is an example plug-in called `TEST.pm`. The above method examples were all pulled from this plug-in. It is good to take a look at this to see how everything fits together. In this example, the text in green indicates what is required, and the text in blue indicates methods that can be implemented. The method names (`quiesce`, `unquiesce`, `clone_pre`, `clone_post`, `restore_pre`, `restore_post`, `restore`, and `discover`), however, cannot be changed. The `quiesce` and `unquiesce` methods are also required, so they must be defined; other methods can be removed if they are not to be implemented.

```
#############################################################################
### Copyright NetApp Inc.  All rights reserved
### Author: ktenzer
### Date: 08/01/2010
###
### Name: Snap Creator::MOD::Mock
#############################################################################

#############################################################################
Changes
#############################################################################
### Ver Author      Description
### 1.0 ktenzer     Initial Version
#############################################################################
package TEST;
$| = 1;
our @ISA = qw(Snap Creator::Mod);
use strict;
use warnings;
use diagnostics;
### Load Snap Creator Objects ###
use Snap Creator::Util::Generic qw ( trim isEmpty );
use Snap Creator::Util::OS qw ( isWindows isUnix getUid createTmpFile );
use Snap Creator::Event qw ( INFO ERROR WARN DEBUG COMMENT ASUP CMD DUMP );
my $msgObj = new Snap Creator::Event();
my %config_h = ();
my $result = {
    exit_code => 0,
    stdout => 'Finished successfully',
    stderr => ''
};

sub new {
  my $invocant = shift;
  my $class = ref($invocant) || $invocant;
    my $self = {
        @_
    };
    bless ($self, $class);
    return $self;
}
sub setENV {
    my ($self, $obj) = @_;
    %config_h = %{$obj};
    my @message_a = ();
    $msgObj->collect(\@message_a, INFO, "$config_h{'APP_NAME'}::setENV");
    $result->{message} = \@message_a;
    return $result;
}

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

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

sub clone_pre {
    my $result = {
        exit_code => 0,
        stdout => "",
        stderr => "",
    };

    my @message_a = ();
    $msgObj->collect(\@message_a, INFO, "$config_h{'APP_NAME'}::clone_pre");
    $result->{message} = \@message_a;
     return $result;
}

sub clone_post {
    my $result = {
        exit_code => 0,
        stdout => "",
        stderr => "",
    };

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

sub discover {
    my $result = {
        exit_code => 0,
        stdout => "",
        stderr => "",
    };

    my @message_a = ();
    $msgObj->collect(\@message_a, INFO, "$config_h{'APP_NAME'}::discover");
    $result->{message} = \@message_a;
     return $result;
}
1;
```

## 13 FURTHER QUESTIONS/HELP

If you have questions, comments, or suggestions, contact the Snap Creator Plugins community:

http://communities.netapp.com/community/products_and_solutions/databases_and_enterprise_apps/Snap Creator/plug-ins