Plugins in opConfig

Introduction

opConfig 3.1.1 introduces a new capability: plugins (written in perl) can now be used to collect or transform configuration data. This document describes this feature.

General Requirements

All plugins have to be written in perl (they may run other programs, but the general scaffolding must be perl).

Plugins are only considered if they're valid perl, and if they fulfil the requirements for a 'Perl Module':

- Each plugin must be stored in the directory `/usr/local/omk/conf/config_plugins/`,
- The file must be named `X.pm`,
- The file must contain a matching `package X;` line, and its package name must not clash with any opConfig or NMIS components.
- The file must end with a line containing "`1;`"
- The plugin must provide at least one of the supported functions: `collect_configuration` or `process_configuration`.
- The plugin **MUST NOT** use Exporter to export anything from its namespace.
- The plugin **MUST NOT** use alarm().
- Running `perl -cw YourPlugin.pm` must report no syntax errors.

It's strongly recommended that the plugin have a version declaration right after the package line, e.g. `our $VERSION = "1.2.3";`

The plugin *may* load extra Perl modules with 'use', but it must not use any package-level global variables. All its variables and any objects that it might create must have local scope.

If opConfig encounters invalid plugins it will ignore these and log a message about the problem.

The global configuration option '`opconfig_plugin_timeout`' (default: 20 seconds) sets the maximum execution time for any opConfig plugin function.

The configuration option '`opconfig_raise_alert_events`' (default: true) controls whether opConfig sends any alert events to NMIS.

Collecting device configuration data with a plugin

To collect device data using a plugin you need a plugin that provides the necessary collection functionality, and you need to configure one or more commands to be delegated to that plugin.

How to delegate collection to a plugin

To delegate data collection the configuration for the command in question has to have the property "`use_collection_plugin`" set to the name of the desired plugin package (ie. "`X`", **not** "`X.pm`").

This can be done for an individual command, or for all commands belonging to the command set in question (if given in the '`scheduling_info`' section). The specification for an individual command overrides any setting in `scheduling_info`.

Data collection can be delegated to exactly one plugin. If collection is delegated, then opConfig does not connect to the node in any way! Instead the plugin has to do anything and everything that is required to procure configuration data for opConfig.

Here is a minimal example command set file (for `conf/command_sets.d/`) that demonstrates how a command is delegated:

```
%hash = (
    'collect_with_plugin' => {
        'os_info' => {
            'os' => '/some special device/',
        },
        # ...omitted scheduling info etc.
        'commands' => [
            {
                'command' => "external command",
                use_collection_plugin => "SpecialDeviceHelper",
            },
# ...
        ],
    }
);
```

What is expected of a plugin for configuration data collection

A plugin can be used for collection if it offers the function `collect_configuration`.

This `collect_configuration` function will be invoked with the following named arguments:

- `node` (the node name),
- `node_info` (a hash reference pointing to the node's configuration record, ie. the structure that you can edit using the Edit Nodes section in the GUI),
- `credential_set (`a hash reference pointing to the selected credential set, ie. the structure that you configure using the Edit Credential Sets section in the GUI),
- `command` (a hash reference pointing to the command in question, basically what the command set contains for this one command)
- `logger` (an OMK::Log object instance)
- `opconfig` (an OMK::opConfig object instance).

The function must not modify any of the arguments that are passed to it.

The function must return a hash reference with the following supported keys:

- `success` (0 or 1),
- `error` (containing an error message that opConfig should handle),
- `ignore` (0 or 1),
- `configuration_data` (text or other data; i.e. the configuration data that opConfig is to associate with this node and command)

if ignore is 1, then the command is ignored altogether and opConfig does not save anything.
if success is 1, then and only then the returned configuration_data is processed and stored by opConfig.
The error response property is ignored if success is 1, otherwise the error message is logged and reported.

Example Collection Plugin

Here is a minimal collection plugin which uses an external program and reports its output as configuration data back to opConfig:

```
sub collect_configuration
{
    my (%args) = @_;

    my ($node, $node_info, $command, $credential_set, $logger, $opconfig)
            = @args{qw(node node_info command credential_set logger opconfig)};
    $logger->info("Plugin ".__PACKAGE__." about to collect data for $node, command $command->{command}");
    # maybe we need to shell out to some program?
    open(P, "-|", "/usr/local/bin/complicated_operation", $command->{command}, $node_info->{host})
            or return { error => "failed to start complicated_operation: $!" };
    my $goodies = <P>;
    close P;
    return { error => "complicated_operation failed: $?" } if ($?);
    return { success => 1, configuration_data => $goodies };
}
1;
```

How to filter or transform configuration data with a plugin

To transform, filter, analyse or otherwise process configuration data further, you need one or more plugins that provide the desired processing functionality and the command in question needs to be configured so that those plugins are invoked.

How to activate processing plugins

One or more processing plugins can be applied to a command's configuration data. The selected plugins will be invoked in a pipeline sequence, i.e. each plugin is passed the results of the previous plugin for further work. At the end of this pipeline opConfig continues with the processing of the final results.

To apply a plugin the command in question has to set the property "`use_processing_plugins`" to the list of the desired plugin package names (ie. "`X`", not "`X.pm`").

This can be done for an individual command, or for all commands belonging to a command set (i.e. if given in the '`scheduling_info`' section). The specification for an individual command overrides the setting in `scheduling_info`.

Here is a minimal command set example that configures the command "show running configuration" for plugin processing:

```
%hash = (
    'process_with_plugin' => {
        # ...omitted os selector, scheduling info etc.
        'commands' => [
            {
                'command' => "show running configuration",
                'use_processing_plugins' => [ "FilterPlugin", "TranslatorPlugin" ],
            },
# ...
        ],
    }
);
```

What is expected of a processing plugin

A plugin can be used for configuration data processing if it offers the function `process_configuration`. This function is expected to transform a newly collected command output (or configuration

data) before opConfig handles change detection and storage.

The `process_configuration` function will be called with the following named argments:

- `node` (the node name),
- `node_info` (a hash reference pointing to the node's configuration record),
- `command` (a hash reference pointing to the command in question),
- `configuration_data` (text, configuration data from opConfig's data collection or the previous pipeline stage),
- `derived_info` (hash reference pointing to plugin-extracted derived information),
- `alerts` (alerts to be raised/closed in NMIS, as reported by prior pipeline stages),
- `conditions` (detected conditions or exceptions that opConfig should display the state of),
- `logger` (an OMK::Log object instance),
- `opconfig` (an OMK::opConfig object instance).

The function must return a hash reference with the following supported keys:

- `success` (0 or 1),
- `error` (error message that opConfig should handle),
- `configuration_data` (optional: if present that's the modified/filtered data to replace the original input),
- `derived_info` (optional; if present that's information that the plugin has derived from the raw input data),
- `alerts` (optional; if present hashref of alerts to raise/close via NMIS),
- `conditions` (optional; if present a hashref of conditions/events to pass on to opConfig.

If and only if success is 1 will configuration_data, derived_info, alerts and conditions be processed by opConfig.

If a plugin signals an error, then the error message is logged and reported first, then opConfig continues with other plugins in the pipeline.
Note that any data that was returned by the failed plugin is ignored!

If configuration_data is returned, then it replaces the original input and is passed to the next pipeline stage. Otherwise the original input is used.

If derived info, alerts or conditions are returned, then they're merged with any already existing information. In other words, each processing plugin can only add to these, not overwrite what a previous plugin has reported.

How to filter or transform configuration data

Your plugin must provide a `process_configuration` function that works on the given `configuration_data` argument and returns a modified version of that.
No other responses are required, but of course your plugin *may* also choose to report conditions, raise alerts or produce derived information at the same time.
Note that for single-line filtering no plugin is required, because opConfig command sets already allow you to specify single-line regular expressions for ignoring irrelevant lines.

Example Filtering Plugin

This example plugin removes PEM encoded private key material from the command output and replaces all of these keys with "<key omitted>".

```
package FilterKey;
our $VERSION = "0.0.0";
use strict;
# replace any PEM private key info in the data with '<key omitted>'
sub process_configuration
{
    my (%args) = @_;
    my ($node, $node_info, $command, $configuration_data,
            $derived, $alerts, $conditions, $logger, $opconfig)
            = @args{qw(node node_info command configuration_data
                derived_info alerts conditions logger opconfig)};
    my $filtered = $configuration_data;
    $filtered
            =~ s/-----BEGIN \S+ PRIVATE KEY-----.+?-----END \S+ PRIVATE KEY-----/\n<key omitted>\n/sg;
    return { success => 1,
             configuration_data => $filtered };
}
1;
```

How to raise alerts and open (or close) NMIS events with a plugin

To programmatically raise or close alerts with a plugin, your plugin must provide a `process_configuration` function that returns a list of alerts to raise or close in the `alerts` return value.
Your plugin *may* also perform filtering or condition reporting at the same time.

The `alerts` response structure must be a hashref if it is present at all.
In the simplest case, the key is the alert/event name, and the value is 0 or 1. In this case opConfig raises (value 1) or closes (value 0) an event in NMIS with the given event name and for the current node.

As a more precise alternative, the value may be a hash with the following supported keys:

- `event`: event name (default: the alerts key is used)
- `upevent`: event name for closing the event (default: NMIS derives that from the event name)
- `element`: what element the event is associated with, optional
- `node`: the name of the node the event should be associated with (default: current node)
- `level`: must be one of Normal, Warning, Minor, Major Critical or Fatal.
  Normal closes the alert/event, all others raise the event. If not present, the default level "Major" is used.
- `details`: extra information to attach to the event, optional

Example Alerting Plugin

This example plugin shows the different supported formats for alert handling:

```
package RaiseAlerts;
our $VERSION = "0.0.0";
use strict;
sub process_configuration
{
    my (%args) = @_;
    my ($node, $node_info, $command, $configuration_data,
            $derived, $alerts, $conditions, $logger, $opconfig)
            = @args{qw(node node_info command configuration_data
derived_info alerts conditions logger opconfig)};
    # ...insert logic that looks at context of command, configuration_data, node etc
    # to determine what alerts to raise

    return { success => 1,
             alerts => {
               "Node Unhappy" => 1, # raise event
               "Node Unreachable" => 0, # clear event
               elsewhere => { event => "Node Not Cooperating",
                              node => "some_other_node",
                              details => "this alert applies to another node",
                              level => "Minor", },
             },
    };
}
1;
```

How to use a plugin to report conditions to opConfig

A plugin can report named conditions to opConfig, which will be stored and displayed with the command in question. In opConfig a condition has a name and a ternary logic value (true, false, unknown).
Like before the plugin *may* also perform filtering or alert raising operations at the same time.

To report conditions the plugin must provide a `process_configuration` function that returns a `conditions` return value which must be a (two levels deep) hashref if present.

The outer key is not displayed by opConfig at this time but separates conditions signalled by different plugins, so we recommend that each plugin picks a unique outer key.
The data structure behind that outer key must be a hashref again.

The inner hashref's key must be he name of the condition that is to be signalled, and the value must be one of 0, 1 or undef.
0 means a bad state for this condition, 1 means a good state for the condition, undef signals that the state of this condition is not known.

Example Plugin

The following small example plugin reports some conditions, using its plugin package name as the separating key:

```
package ReportConditions;
our $VERSION = "0.0.0";
use strict;
sub process_configuration
{
    my (%args) = @_;
    my ($node, $node_info, $command, $configuration_data,
            $derived, $alerts, $conditions, $logger, $opconfig)
            = @args{qw(node node_info command configuration_data
derived_info alerts conditions logger opconfig)};
    # ...insert logic that looks at context of command, configuration_data, node etc
    # to determine what conditions and states these imply

    return { success => 1,
             conditions => {
               "ReportConditions" => {
                  "node is lacking something" => 0,
                  "command implies something is in good shape" => 1,
                  "something unexpectedly couldn't be analysed" => undef,
               },
             },
           };
}
1;
```

How to use a plugin to prepare derived information (or knowledge) for opConfig

A plugin may report extracted 'knowledge', information that it derived from the configuration data, to opConfig for display and storage.
To do so, the plugin must have a `process_configuration` function that returns a `derived_info` response.

The `derived_info` response structure must be a hashref if it is present, and can have any depth.
The outer key is used to separate information reported by different plugins (or different kinds of information); a suitably unique key needs to be chosen by the plugin author.

The following keys have special meaning and should not be used for other purposes in your inner datastructure: `type, tag, value, title, labels, rows`.

All derived_info data is stored by opConfig, but at this time the opConfig gui will only display derived info entries whose inner structure conforms to the following layout:

- property 'type' must be set to 'table',
- property 'labels' must be an array whose elements are displayed as column headings (setting a label to undef causes this column to be skipped in the display),
- property 'rows' must be an array of arrays whose elements are to be shown as table.
- property 'title' may be set but is optional.

Example Plugin

The following example plugin produces two tabular instances of derived information, which will be displayed by the opConfig GUI, and further example data which will be stored but not displayed.

```
package DerivedInfo;
our $VERSION = "0.0.0";
use strict;
sub process_configuration
{
    my (%args) = @_;
    my ($node, $node_info, $command, $configuration_data,
            $derived, $alerts, $conditions, $logger, $opconfig)
            = @args{qw(node node_info command configuration_data
              derived_info alerts conditions logger opconfig)};
    # ...insert logic that derives knowledge from the context

    return { success => 1,
             derived_info => {

               some_reportable_thing =>
               {
                 type => "table",
                 title => "Reported by ".__PACKAGE__,
                 labels => [ 'one col', 'another col', ],
                 rows => [ [ 'maybe key', 'maybe value', ],
                           [ 'maybe key2', 'maybe value', ],
                           [ 'last row', 'last value', ],
                         ],
               },
               my_filtered_table => {
                 type => 'table',
                 labels => [ 'first col', 'second', undef, 'and at long last' ],
                 rows => [
                           [ 'note that', 'there is', 'a gap', 'something that you were not meant to see' ],
                           [ 1, 2, 3, 4, ],
                         ],
               },

               ours_but_not_displayed =>
               {
                 doesnt_display => "whatever",
                 stored =>  [ 4, 7, 29 ],
                 deep_is_ok => { other => { thing => [ 1..10 ],  } },
               },
             },
    };
}
1;
```

Plugin Input Argument Structures

node_info

contains the same data that you get when running ./`bin/opnode_admin.pl act=export node=XYZ`, ie. the node's configuration, connection information and os information.

credential_set

holds the credential set details that was configured for this node.
The credential set structure carries the same properties as the credential set editing gui screen, with the following keys:

- username (always present),
- password (may be empty if ssh_key is given)
- password_privileged (may be empty if always_privileged is 1)
- ssh_key (may be empty or holds an ssh private key in the typical openssh format, e.g. "-----BEGIN RSA PRIVATE KEY-----...keymaterial....")
- setname and description
- always_privileged (0 or 1, if 1 password_privileged is ignored)

command

contains the command in question, plus some meta data.

This is structured similar to the command set definition, but with some extended or expanded properties:

- all settings that are inheritable from per-set scheduling_info are expanded,
- the property command_set holds the name of the command set this came from.

logger

refers to an OMK::Log instance, a subclass of Mojo::Log.

you can use the following methods to log information, in order of severity: fatal error warn info debug (and debug2 to debug9).

your information is logged if the global verbosity level is at least as verbose as your chosen method. opConfig's default level is 'info'. e.g. $logger->warn ("this is pretty bad")

will very likely end up in the opConfig log, while $logger->debug5("nuisance") will almost certainly be suppressed.

opconfig

refers to an OMK::opConfig instance, which can be used to schedule command execution, retrieve other command revisions and the like.

please consult the opConfig API documentation for details.