Table of Contents

Scenario examples

See the sub-menu entries for the examples.

These include command parsing, config parsing, how to work with scenarios, loops and conditions.

Scenario basics

Scenarios control what a job does when it is executed. Jobs are submitted to the scheduler to be launched at the desired time. When launched the scenario contains the various steps to make the desired configuration changes.

The basic scenario has a description line for the job log and is followed with one or more scenario commands that perform high-level actions on the desired node. The scenario commands include actions like `reboot_node` and `cmd_exec`.

Each scenario command uses option flags to give it its arguments like `-n` to give it the nodename to work on. Most commands use some mandatory arguments and some optional arguments. Consult the reference guide on Scenario Commands for details on each command.

As with the templates, scenarios use variables which are enclosed in angled brackets. The most common one is **<node>** which has the selected nodename from the tool that submitted the job.

To catch failed scenario commands the **<error>** variable is used. By checking on on <error> an appropriate action can be taken. By default the scenario continues execution until it reaches the end. If it reaches the end of the scenario (or executes the `end` command), the job is considered 'SUCCESSFUL'. Only if the scenario includes error checking (using `if <error>`) and executes the **`stop`** command will the job be listed as 'ABORTED'. Also see note at bottom on **`stop on-error`**.

Scenarios can be stored for re-use as they are created for specific tasks. These saved scenarios are maintained using the "**Operate - Scenarios**" tool. These saved scenarios can be included in other scenarios and jobs by specifying them as a **`task`**. Multiple tasks can be executed in a scenario and tasks can be nested.

The simplest of scenarios is used in the "**Command job**" tool. This widely used tool uses this scenario text as default:

```
[parameters]

[scenario]
Description <node> Command job...
task := Command_job

end
```

The `[parameter]` section is used to pass your own variables into the `[scenario]` section should it need any. The `[scenario]` section here just has the `description` line and uses the task `command_job`. This stored scenario contains but one command, **`cmd_exec`** and its basic error

handling:

```
Description <node> Command job

Cmd_exec -n <node> -f <node>.cmd <verbose>
if <error>
    Log_action -n <node> -a Command_job -m "Failed executing commands"
    stop
endif

Log_action -n <node> -a Command_job -m "Completed executing commands"
```

Both the job [scenario] and the task have a `description` line. That is good practice since it is the first encountered description line that is used in the job logs.

The **`cmd_exec`** command used here is used to execute the cli commands in the file `<node>.cmd`. The cmd_exec first creates this file from the “Commands” text-box of the tool. Next a session is set up to the <node> and a configuration snapshot is taken. Then the cli commands are executed and the configuration saved. Finally another snapshot is taken. Should one of the cli commands produce an error or time-out, the <error> variable is returned.

All vendor specific handling of protocols, login, errors, backup, file transfers, commits and saves are handled by the NetYCE vendor modules. The `<node>` sets the context for the vendor, addresses, credentials, etc.

Scenarios (and tasks) can be extended to handle complex tasks where variables or a state can be retrieved from a node and then used to generate the correct commands and execute them. By adding more programmable syntaxes like loops changes can be made to various devices in the network if desired.

More on the scenario capabilities can be found in the reference guide on Scenario Syntax.

Stop on-error

The scenarios default behaviour to ignore `<error>` unless testing and handling it, can be modified using **`stop on-error`**. From the point onward where it is included will the scenario abort when it encounters an `<error>` without having to test or handling it.

If you need to revert to explicit error handling include the command **`stop default`**.

More on this can be fount in Scenario Commands - Error handling