Command Parsing Template Syntax

Command parsing templates parse the result from a command to a node, and extract variables from that output which are used in Scenarios.

The scenario command is parse_cmd.

A tool to assist in testing the command parser can be found at Parsing test

When referred to output in the examples, it is the same output you'll see when using the parsing tester.

Text parsing syntax

The text parsing syntax is described below, followed by the table parsing syntax.

Each of the capabilities is described using examples.

Capabilities

- <variable> only parses single words
- <variable:> parses until it encounters a double space, tab or the end of line
- <variable:test> parses until it encounters the word “test”, surrounded by whitespace, or the end of line otherwise.
- <variable:,> parses until it encounters a single character, which doesn't have to be surrounded by whitespaces. (in this case the 'comma')
- <variable*> will put all text in a single variable
- [header] + all of the above
- %keys + all of the above
- indentation, dealing with multiple levels of indentation.
- |*| ignoring anything else on the line.

Variable extraction

A few examples on how to use this. Below is the output given for a Cisco Node with the command show version.

Let's parse this output and extract the following information:

```
* software version
* hardware
* Base mac address
* serial number
```

```
SW1#show version
Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 15.0(2)SE4,
RELEASE SOFTWARE (fc1)
```

```
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2013 by Cisco Systems, Inc.
Compiled Wed 26-Jun-13 02:49 by mnguyen

ROM: Bootstrap program is C2960 boot loader
BOOTLDR: C2960 Boot Loader (C2960-HBOOT-M) Version 12.2(25r)FX, RELEASE
SOFTWARE (fc4)

Switch uptime is 39 minutes
System returned to ROM by power-on
System image file is "flash:c2960-lanbasek9-mz.150-2.SE4.bin"

This product contains cryptographic features and is subject to United
....

cisco WS-C2960-24TT-L (PowerPC405) processor (revision B0) with 65536K bytes
of memory.
Processor board ID FOC1010X104
Last reset from power-on
1 Virtual Ethernet interface
24 FastEthernet interfaces
2 Gigabit Ethernet interfaces
The password-recovery mechanism is enabled.

64K bytes of flash-simulated non-volatile configuration memory.
Base ethernet MAC Address : 00:17:59:A7:51:80
Motherboard assembly number : 73-10390-03
Power supply part number : 341-0097-02
Motherboard serial number : FOC10093R12
Power supply serial number : AZS1007032H
Model revision number : B0
Motherboard revision number : B0
Model number : WS-C2960-24TT-L
System serial number : FOC1010X104
Top Assembly Part Number : 800-27221-02
Top Assembly Revision Number : A0
Version ID : V02
CLEI Code Number : COM3L00BRA
Hardware Board Revision Number : 0x01


Switch Ports Model SW Version SW Image
------ ----- ----- ---------- ----------
* 1 26 WS-C2960-24TT-L 15.0(2)SE4 C2960-LANBASEK9-M


Configuration register is 0xF
```

The parsing template will look like:

```
Cisco IOS Software, <tmp1> Software (<software>), Version <version:,>,
RELEASE SOFTWARE (fc1)
cisco <hardware> (<tmp2>) processor |*|
Base ethernet MAC Address : <basemac>
Motherboard serial number : <serial>
```

Normal text is an exact match on the command output. Some variables are temporary, which we will not use, but can be variable depending on the hardware, like <tmp1> and <tmp2>. The version is matched up till the comma and note that the command is also presented in the 'exact' match text as well.

The output of the parsing is:

```
<basemac>: 00:17:59:A7:51:80
<hardware>: WS-C2960-24TT-L
<serial>: FOC10093R12
<software>: C2960-LANBASEK9-M
<tmp1>: C2960
<tmp2>: PowerPC405
<version>: 15.0(2)SE4
```

If you would want to 'catch' a single word on a line, you could just put a single variable in the parsing template:

```
<word>
```

Headers

Multiple blocks with the same text and variables. Here is how to break them up in sections:

```
# This template parses command results like this:
#  Current Boot Variables:
#
#
#  kickstart variable = bootflash:/n6000-uk9-kickstart.7.0.7.N1.1.bin
#  system variable = bootflash:/n6000-uk9.7.0.7.N1.1.bin
#  Boot POAP Disabled
#
#  Boot Variables on next reload:
#
#
#  kickstart variable = bootflash:/n6000-uk9-kickstart.7.0.7.N1.1.bin
#  system variable = bootflash:/n6000-uk9.7.0.7.N1.1.bin
#  Boot POAP Disabled
#
# To differentiate between current and future boot variables, headers are
specified.
# Headers are denoted between []-brackets and their contents have to match a
line,
```

```
# including all its special characters.
#
# In a scenario, you can now access the current boot system variable as
follows
# (note that special characters ., :, [, ], @, %, <, > are not needed to
avoid confusion with scenario syntax):
#  <Current Boot Variables.current_boot_system%boot_variable>
[Current Boot Variables:]
kickstart variable = <current_boot_kickstart>
system variable = <current_boot_system>
Boot POAP <current_boot_poap_enabled>

[Boot Variables on next reload:]
kickstart variable = <next_boot_kickstart>
system variable = <next_boot_system>
Boot POAP <next_boot_poap_enabled>
```

With the output:

```
[Current Boot Variables]: | <current_boot_kickstart>: bootflash:/n6000-uk9-
kickstart.7.0.7.N1.1.bin | <current_boot_poap_enabled>: Disabled |
<current_boot_system>: bootflash:/n6000-uk9.7.0.7.N1.1.bin

[Boot Variables on next reload]: | <next_boot_kickstart>: bootflash:/n6000-
uk9-kickstart.7.0.7.N1.1.bin | <next_boot_poap_enabled>: Disabled |
<next_boot_system>: bootflash:/n6000-uk9.7.0.7.N1.1.bin
```

In this way, you can pick out the difference between the current boot variables, and the ones on next reload. The line between the squared brackets is a header. When the config parser parses a config, it looks out for blocks whose first line starts with this header. It only looks at the first line of this block, and headers can be made more specific. We allow a lot of special characters for example: spaces, colons, even newlines. All these need to be explicitly included.

Keys

Whenever you have multiple entries, like the example below has with interfaces, you will want to able to loop over this data and extract the necessary. You'll have to assign a variable as 'key, so you can extract variables based on that specific key. See below:

```
# Show cdp all returns an output like this:
#  Ethernet1/1 is up
#      CDP is operationally disabled as interface is in fex-fabric  mode
#      Refresh time is 60 seconds
#      Hold time is 180 seconds
#  Ethernet1/2 is up
#      CDP is operationally disabled as interface is in fex-fabric  mode
#      Refresh time is 60 seconds
#      Hold time is 180 seconds
#  Ethernet1/3 is up
```

```
#      CDP is operationally disabled as interface is in fex-fabric  mode
#      Refresh time is 60 seconds
#      Hold time is 180 seconds
#  Ethernet1/4 is up
#      CDP is operationally disabled as interface is in fex-fabric  mode
#      Refresh time is 60 seconds
#      Hold time is 180 seconds
#  Ethernet1/5 is up
#      CDP enabled on interface
#      Refresh time is 60 seconds
#      Hold time is 180 seconds
#  Ethernet1/6 is down
#      CDP enabled on interface
#      Refresh time is 60 seconds
#      Hold time is 180 seconds
#  et cetera...
#
# To differentiate between interfaces we specify a key. If you do this, this
will override any []-header
# in your results. In a scenario you can now call the result as follows:
<Ethernet1/1.if_status%port_variable>.
#
# The amount of spaces in the indentation in this pattern don't matter,
# as long as there is any kind of indentation.
<%interface> is <if_status>
    CDP <if_cdp> on interface
    Refresh time is <if_refresh_time> seconds
    Hold time is <if_hold_time> seconds
```

Output:

```
[Ethernet1/1]: | <if_hold_time>: 180 | <if_refresh_time>: 60 | <if_status>:
up | <interface>: Ethernet1/1
[Ethernet1/2]: | <if_hold_time>: 180 | <if_refresh_time>: 60 | <if_status>:
up | <interface>: Ethernet1/2
[Ethernet1/3]: | <if_hold_time>: 180 | <if_refresh_time>: 60 | <if_status>:
up | <interface>: Ethernet1/3
[Ethernet1/4]: | <if_hold_time>: 180 | <if_refresh_time>: 60 | <if_status>:
up | <interface>: Ethernet1/4
[Ethernet1/5]: | <if_cdp>: enabled | <if_hold_time>: 180 |
<if_refresh_time>: 60 | <if_status>: up | <interface>: Ethernet1/5
[Ethernet1/6]: | <if_cdp>: enabled | <if_hold_time>: 180 |
<if_refresh_time>: 60 | <if_status>: down | <interface>: Ethernet1/6
```

Indentation

Multiple hierarchical indentation also will be parsed. The templates need to follow the exact same indentation pattern and that will suffice.

```
#
# Ethernet1/48 is up
#  Dedicated Interface
#   Belongs to Po1
#   Hardware: 1000/10000 Ethernet, address: 002a.6ac4.b457 (bia
002a.6ac4.b457)
#   Description: 2e int portchannel1 -trunk-
#   MTU 1500 bytes, BW 10000000 Kbit, DLY 10 usec
#   reliability 255/255, txload 1/255, rxload 1/255
#   Encapsulation ARPA
#   Port mode is FabricPath
#   full-duplex, 10 Gb/s, media type is 10G
#   Beacon is turned off
#   Input flow-control is off, output flow-control is off
#   Rate mode is dedicated
#   Switchport monitor is off
#   EtherType is 0x8100
#   Last link flapped 3week(s) 6day(s)
#   Last clearing of "show interface" counters never
#   6 interface resets
#   30 seconds input rate 6944 bits/sec, 6 packets/sec
#   30 seconds output rate 120 bits/sec, 0 packets/sec
#   Load-Interval #2: 5 minute (300 seconds)
#     input rate 6.42 Kbps, 6 pps; output rate 160 bps, 0 pps
#   RX
#     15 unicast packets  26389404 multicast packets  1779 broadcast packets
#     26391200 input packets  2915102855 bytes
#     0 jumbo packets  0 storm suppression bytes
#     0 runts  0 giants  2 CRC  0 no buffer
#     2 input error  0 short frame  0 overrun   0 underrun  0 ignored
#     0 watchdog  0 bad etype drop  0 bad proto drop  0 if down drop
#     0 input with dribble  0 input discard
#     0 Rx pause
#   TX
#     20825 unicast packets  1015782 multicast packets  35 broadcast packets
#     1036642 output packets  249577269 bytes
#     0 jumbo packets
#     0 output error  0 collision  0 deferred  0 late collision
#     0 lost carrier  0 no carrier  0 babble 0 output discard
#     0 Tx pause

<%interface> is <if_status>
 Dedicated Interface
  Belongs to <if_port_channel>
  Hardware: <if_hardware:>
  Description: <if_description:>
  MTU <if_mtu:>
  Port mode is <if_port_mode>
  RX
    <if_unicast_packets> unicast packets  <if_multicast_packets> multicast
```

```
packets  <if_broadcast_packets> broadcast packets
  TX
    <if_unicast_packets> unicast packets  <if_multicast_packets> multicast
packets  <if_broadcast_packets> broadcast packets
```

Table parsing syntax

Capabilites

- [headers], can consist of multiple lines and must include all characters to form a perfect match
- |<lines>| for flexible tables
    - %keys, to be able to identify unique entries
    - <variable> only parses single words, can be used for capturing indentation
    - <variable:> parses until 2 consecutive whitespaces or the end of line
    - <variable:anchor> parses line up to and including anchor. This can also be any character, aside from newlines or <>-carats
    - // ignoring the line.
- ^<lines> for fixed tables
    - Any character matching % is appended to the key
    - Any character matching 1 is appended to the first variable (excluding the key)
    - Any character matching 2 is appended to the second variable
    - ... et cetera
    - Any character matching a is appended to the tenth variable
    - Amy character matching b is appended to the eleventh variable
    - ... et cetera
    - Any character matching . is ignored

Flexible table

All table parsing templates start with a header, followed by a table syntax. Multiple headers and table syntaxes can be provided, multiple headers can precede a table syntax, but only one table syntax belongs to one header. An example:

```
#  ------------------------------------------------------------------------
-------
#  Ethernet      VLAN    Type Mode   Status  Reason                   Speed
Port
#  Interface
Ch #
#  ------------------------------------------------------------------------
-------
#  Eth1/1        1       eth  fabric up      none
10G(D) 100
#  Eth1/2        1       eth  fabric up      none
10G(D) 100
#  Eth1/3        1       eth  fabric up      none
10G(D) 101
#  Eth1/4        1       eth  fabric up      none
```

```
10G(D) 101
#  Eth1/5        602     eth  access up      none
1000(D) 1000
#  Eth1/6        999     eth  trunk  down    SFP not inserted
10G(D) 1008
#  et cetera
#
# There can be multiple tables, and each table can have headers of multiple
lines.
# All these lines in a header can become one pattern header to recognize
which table
# syntax is applicable.
#
# A table can be parsed as follows: its syntax is enclosed in between |-
pipes.
# What follows is a list of variables. These will parse the table result,
separated
# by spaces.
# <%variable> parses the key in this entry. So getting the vlan in a
scenario, you would need e.g. <Eth1/1.vlan%interfaces>
# <variable> is a regular variable, no spaces.
# <variable:> is a variable containing spaces. The parser only jumps to the
next variable in line after encountering two consecutive spaces.
#
[-------------------------------------------------------------------------
-----
Ethernet      VLAN    Type Mode   Status  Reason                   Speed
Port
Interface
Ch #
-------------------------------------------------------------------------
----]
|<%interface> <vlan> <type> <mode> <status> <reason:> <speed>
<port_channel>|

[-------------------------------------------------------------------------
-----
Port-channel VLAN    Type Mode   Status  Reason                   Speed
Protocol
Interface
-------------------------------------------------------------------------
----]
|<%interface> <vlan> <type> <mode> <status> <reason:> <speed> <protocol>|

[-------------------------------------------------------------------------
-----
Port   VRF          Status IP Address                              Speed
MTU
-------------------------------------------------------------------------
----]
```

```
|<%interface> <vrf> <status> <address> <speed> <mtu>|

[-------------------------------------------------------------------------
----
Interface Secondary VLAN(Type)                    Status Reason
-------------------------------------------------------------------------
---]
|<%interface> <secondary_vlan> <status> <reason>|

[-------------------------------------------------------------------------
-----
Interface     Status     Description
-------------------------------------------------------------------------
----]
|<%interface> <status> <description>|
```

Note how the # in the header in this case is not a comment, but part of the header. Headers have to match exactly, and if a table syntax has two headers, either one can match.

Flexible table with indentation and ignoring lines

With this, most tables can be parsed, even some with a weird layout. The problem with this table is that it has some rows that are just there for filler and aren't meant to be parsed. These can be ignored.

```
# Some tables have an unusual layout, and need a few workarounds to be
parsed correctly
# For example the command show port-channel traffic returns something like
this:
#  ChanId      Port Rx-Ucst Tx-Ucst Rx-Mcst Tx-Mcst Rx-Bcst Tx-Bcst
#  ------ --------- ------- ------- ------- ------- ------- -------
#       1   Eth1/47  95.82%  99.76%  71.85%  97.78%  50.29%  98.07%
#       1   Eth1/48   4.17%   0.23%  28.14%   2.21%  49.70%   1.92%
#  ------ --------- ------- ------- ------- ------- ------- -------
#      11    Eth2/1    0.0%    0.0%    0.0%    0.0%    0.0%    0.0%
#  ------ --------- ------- ------- ------- ------- ------- -------
#    1002  Eth100/1/4    0.0%    0.0%    0.0%    0.0%    0.0%    0.0%
#    1002  Eth101/1/4    0.0%    0.0%    0.0%    0.0%    0.0%    0.0%
#
# Any table row which starts indented can be parsed correctly by labelling
the
# first variable as a garbage-one (for example <x>), and the rest of the
line will parse correctly.
#
# Also to avoid parsing the dashed lines, you can put two slashes before it
in the pattern.
# This will tell the command parser to simply ignore that line if it runs
into it.
```

```
[ChanId      Port Rx-Ucst Tx-Ucst Rx-Mcst Tx-Mcst Rx-Bcst Tx-Bcst
------ --------- ------- ------- ------- ------- ------- -------]
|<x> <%port_channel> <port> <rx_ucst> <tx_ucst> <rx_mcst> <tx_mcst>
<rx_bcst> <tx_bcst>|
//------ --------- ------- ------- ------- ------- ------- -------
```

Fixed table

These kind of tables have a fixed starting point for each column. With could have 1 or multiple whitespaces between them, depending on the information stored. For these type of tables the following syntax is used:

```
# This example gives a fixed header with a dynamic value for the total
number
# of entries, this can be ignored by only using the single line header of
the table
#
#  Flags: * - Adjacencies learnt on non-active FHRP router
#         + - Adjacencies synced via CFSoE
#         # - Adjacencies Throttled for Glean
#         D - Static Adjacencies attached to down interface
#
#  IP ARP Table for context default
#  Total number of entries: 3
#  Address         Age       MAC Address     Interface
#  192.168.60.1    00:17:01  0050.56c0.0002  Ethernet2/1
#  192.168.60.7    00:09:34  000c.29e0.6768  Ethernet2/1
#  192.168.60.50   00:04:40  5000.0003.0000  Ethernet2/1
#
#
#
#

[Address         Age       MAC Address     Interface]
|<%address> <age> <mac> <int>|
```

```
# Some tables cannot be parsed based on separation by spaces, for example:
#  ------------------------------------------------------------------------
-------
#  Port          Name               Status    Vlan      Duplex  Speed   Type
#  ------------------------------------------------------------------------
-------
#  Eth1/1        1e int portchannel connected 1         full    10G
10Gbase-SR
#  Eth1/2        2e int portchannel connected 1         full    10G
10Gbase-SR
#  Eth1/3        1e int portchannel connected 1         full    10G
10Gbase-SR
#  Eth1/4        2e int portchannel connected 1         full    10G
```

```
10Gbase-SR
#  Eth1/5        Member of Po1000,  connected 602       full    1000
SFP-1000BAS
#  Eth1/6        Member of Po1008,  sfpAbsent trunk     full    10G     --
#  et cetera
#
#
# Headers can be of multiple lines. They can also directly follow each
other.
# What this means is that the same rules should apply to both of them when a
match is found.
#

[-------------------------------------------------------------------------
-----
Port          Name               Status    Vlan      Duplex  Speed   Type
-------------------------------------------------------------------------
----]
^<%interface> <if_name> <if_status> <if_vlan> <if_duplex> <if_speed>
<if_type>
%%%%%%%%%%%%.11111111111111111.222222222.333333333.4444444.5555555.6666666
66666
```

Command Scope

Our command parser can handle a lot of different syntaxes, but it won't be able to parse everything. For example, if you want to parse the running configuration, we redirect you to the config parser. There are a number of other patterns that the command parser will NOT be able to parse:

- Any table whose rows consist out of multiple lines. For example:

```
VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
---
1    default                          active    Po1007, Po1008, Po1009,
Po1010
                                                Eth1/6, Eth1/7, Eth1/8,
Eth1/9
                                                Eth1/10, Eth1/11, Eth1/12
                                                Eth1/13, Eth1/14, Eth1/15
```

- Any command that returns a very large string of text, combining any regular parseable text together with tables. It will either be just parsing tables, or text, not both.
    - You will probably want to split or filter that using filters on the command you're using on the node itself.

From:
https://wiki.netyce.com/ - **Technical documentation**

Permanent link:
https://wiki.netyce.com/doku.php?id=guides:user:command_parsing_templates

Last update: **2024/07/03 12:31**