Intermediate Ansible

Cristian Sîrbu - cristian.sirbu@networktocode.com - @cmsirbu

Agenda (at a glance)

- Agenda, Introductions, Housekeeping
- **Module 1:** Jinja Templating and Parsing in Ansible
- **Module 2:** Extending Ansible Through Its Plugin System
  - Focus: Custom Filters, Modules
- **Module 3:** Packaging and Consuming Custom Code
  - Ansible Content Collections
  - Inventory Plugins (File, Script, Nautobot)

Agenda - Module 1

- **Jinja Templating and Parsing in Ansible**
  - Syntax Refresher + Whitespace management
  - Extracting data with filters
  - Parsing unstructured data from network devices
- **Lab 01** - Working with Jinja Templates
- **Lab 01.5** - Practicing Jinja Whitespace Management (Bonus)
- **Lab 02** - Exploring and Testing Jinja Filters
- **Lab 03** - Parsing Show Commands with Ansible

Agenda - Module 2

- **Understanding The Ansible Plugin System**
- **Part 1** - Writing Custom Filters
  - Changing data to fit specific use-cases
  - Using 3rd party Python modules
- **Lab 04** - Expanding an Abbreviated Interface Name
- **Lab 05** - Creating the Expand Range Filter
- **Lab 06** - Implementing a Filter to Display Formatted Tables
- **Lab 07** - Implementing a Filter to Compare Intent with Operational Data

Agenda - Module 2

- **Part 2** - Writing Custom Modules
  - Building modules from scratch
  - Implementing good practices: idempotency & check mode
- **Lab 08** - Custom Module for Querying the IOS-XE REST API
- **Lab 09** - Custom Module for Making Changes to the IOS-XE REST API
- **Lab 10** - Making the VRF Module Idempotent and Adding Support for Check Mode

Agenda - Module 3

- **Packaging and Consuming Custom Code**
  - The Ansible Content Collection System
  - Installing and using 3rd party collections
  - Working with dynamic inventories through plugins
- **Lab 11** - Managing and Using Ansible Content Collections
- **Lab 12** - Packaging Custom Code with Ansible Content Collections
- **Lab 13** - Working with Multiple Inventory Sources in Ansible
- **Lab 14** - Using Nautobot as Inventory Source for Ansible

Housekeeping

- **Course Materials**
  - **You MUST be logged in to your GitHub account to view**
  - https://github.com/networktocode/intermediate-ansible

- Lectures and Demos - **Timing**
- **Breaks**
- Feedback / **Q&A**

Instructor Introduction - Cristian Sîrbu

- Trainer and consultant with 17+ years of networking industry experience in a wide range of roles.
- Diverse technical background - coding, system administration, mobile telecoms, enterprise, operations, design.
- Developed and updated many of the courses and workshops in the current NTC training portfolio.
- CCIE #43453
- Cisco Certified DevNet Professional & DevNet 500
- Based in Dublin, Ireland.

Introductions

- **Name**
- **Job / Role**
- **Relevant experience -** Programming, Ansible, Automation Tools
- **Any personal goals for this training**

Module 1

Using Jinja Templating and Text Parsing in Ansible

Why Templating?

- Automates generation of documents containing dynamic data
  - Web pages
  - Reporting (Audits, Inventories, etc.)
  - Configuration files
- Separation between data and text of the document
  - Template designer can focus on the content
  - Data can come from multiple sources
- Hierarchy and inheritance allow for logical groupings of templates

Jinja Overview

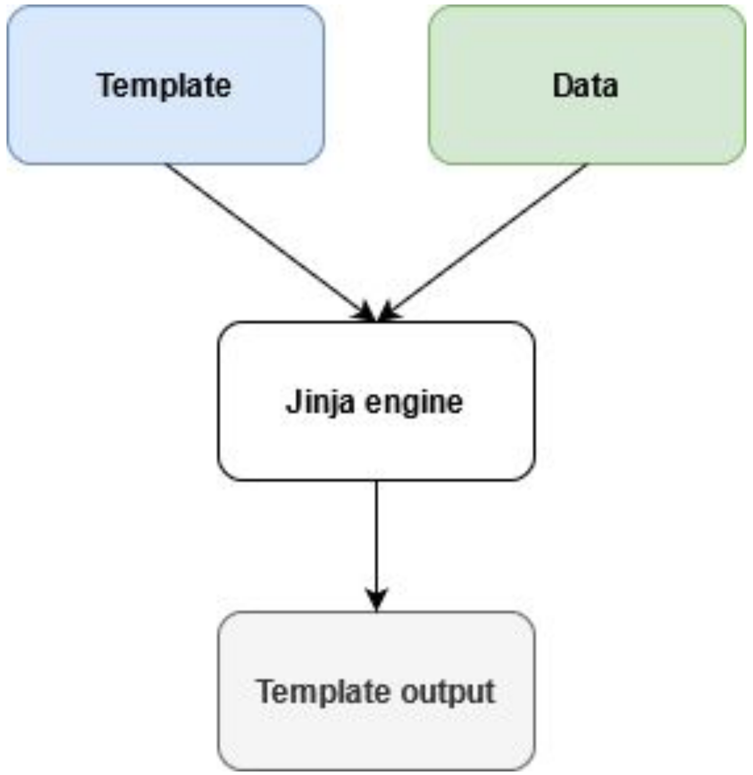
- Templating language written in Python
- Widely used in the Python ecosystem
- Many web and automation frameworks use Jinja as their rendering engine, e.g.:
  - Ansible, Salt
  - Django, Flask
- Comes with rich feature set
- Text based documents, like device configs, can be templated with Jinja

Jinja Features

- Loops and conditionals
- Rich set of built-in filters and tests
- Macros (function-like blocks of code)
- Template inheritance
- Python-like syntax and access to some Python methods
- Configurable syntax

Jinja Components

- Three elements needed to render template
- Input template - written in Jinja
- Data used by the template (json, yaml, etc.)
- Rendering environment:
  - Ansible template module
  - Python jinja2 package
  - Django, Flask, etc. built-in engine



Jinja Syntax

- Default Jinja delimiters
  - `{{ ... }}` - evaluate and print expressions
  - `{% ... %}` - used for statements e.g. for loops, if conditionals
  - `{# ... #}` - comments, not included in template output

Jinja Variables

- Dictionary with variables passed to templates by application
- Attributes can be accessed using dot (`.`) or subscript (`[]`) syntax
    - Recommendation is to use subscript `[]` syntax
    - `vlan.vid` and `vlan['vid']` do the same thing
    - Can't use dot syntax with special characters (-, ., /, etc.)
- Undefined variables don't raise errors by default
    - Some frameworks, like Ansible, default to raising error
- New variables can be defined using `{% set var_name = ... %}` syntax

Jinja Variable Substitution

Basic variable:

```
{{ hostname }}
```

Output:

```
NYCR01
```

List item:

```
{{ vlans[2] }}
```

Output:

```
50
```

Dictionary key:

```
{{ interfaces['Loopback0']['ip'] }}
```

Output:

```
10.0.0.2
```

Data Variables (inputs):

```
---
hostname: NYCR01

vlans:
  - 10
  - 20
  - 50
interfaces:
  Loopback0:
    ip: 10.0.0.2
    mask: 32
```

Jinja Operators

- Math operators: `+, -, /, //, %, * , **`
- Comparison operators: `==, !=, >, >=, <, <=`
- Logic operators: `and, or, not, ()`
- For concatenating strings use `~` (tilde) operator instead of `+`
- Operator `in` performs containment test

Jinja Operators - In operator

Use in operator to check if item exists in a collection

```
{% if 'Management1' in interfaces %}
Management interface found
{% endif %}

{% if 850 not in vlans %}
Missing Printing vlan
{% endif %}
```

```
Management interface found

Missing Printing vlan
```

```
---
interfaces:
  Management1:
    description: Mgmt interface
  Ethernet1:
    description: leaf01-eth51
  Ethernet2:
    description: tap switch

vlans:
  - 560
  - 700
  - 1250
```

Jinja Operators - String concatenation

- Operator `+` follows Python rules
  - Type error will be raised if arguments are incompatible
- Use `~` operator when building strings
  - With `~` Jinja will cast arguments to string whenever possible

```
{{ 'Vlan' ~ 34 }}
```

```
Interface Vlan34
```

```
{{ 'Vlan' + 34 }}
```

```
    Interface {{ 'Vlan' + 34 }}
TypeError: can only concatenate str (not "int") to str
```

Jinja Loops

- One loop type, `for` loop
- Works with lists, dicts, and other collections
    - Use `items()` to get dict key value pairs
- Changes made to vars are lost outside of the loop
- Special loop vars and namespaces help in getting around it

```
{% for ... %}
...
{% endfor %}
```

Jinja Loops - Lists

```
{% for vlan in vlans %}
vlan {{ vlan['vid'] }}
 name {{ vlan['name'] }}
{% endfor %}

{% for server in name_servers %}
ip name-server {{ server }}
{% endfor %}
```

```
vlan 10
 name data
vlan 20
 name voice

ip name-server 8.8.8.8
ip name-server 8.8.4.4
```

```
---
vlans:
  - vid: 10
    name: data
  - vid: 20
    name: voice

name_servers:
  - 8.8.8.8
  - 8.8.4.4
```

Jinja Loops - Dictionaries

```
{% for name in interfaces %}
interface {{ name }}
 description {{ interfaces[name]['description'] }}
 ip address description {{ interfaces[name]['ip_address'] }}
{% endfor %}
```

```
interface Ethernet1
 description uplink to core
 ip address description 10.5.0.1/30
interface Vlan10
 description Data VLAN
 ip address description 10.4.0.1/25
interface Vlan20
 description Voice VLAN
 ip address description 10.4.0.129/25
```

```
---
interfaces:
  Ethernet1:
    description: uplink to core
    ip_address: 10.5.0.1/30
  Vlan10:
    description: Data VLAN
    ip_address: 10.4.0.1/25
  Vlan20:
    description: Voice VLAN
    ip_address: 10.4.0.129/25
```

Jinja Conditionals

- Jinja uses `{% if %} {% endif %}` for conditional checks.
- For branching out use `{% elif %}` and `{% else %}`

String comparison

```
{% if routing_prot == 'bgp' %}
router bgp {{ bgp['as_number'] }}
  ...
{% elif routing_prot == 'ospf' %}
router ospf {{ ospf['process_id'] }}
  ...
{% else %}
ip route 0.0.0.0/0 {{ def_route_next_hop }}
{% endif %}
```

Number comparison

```
{% if ios_ver >= 15.2 %}
IOS >= 15.2, enabling new features
{% else %}
IOS < 15.2, using legacy features
{% endfor %}
```

>>> network .toCode()

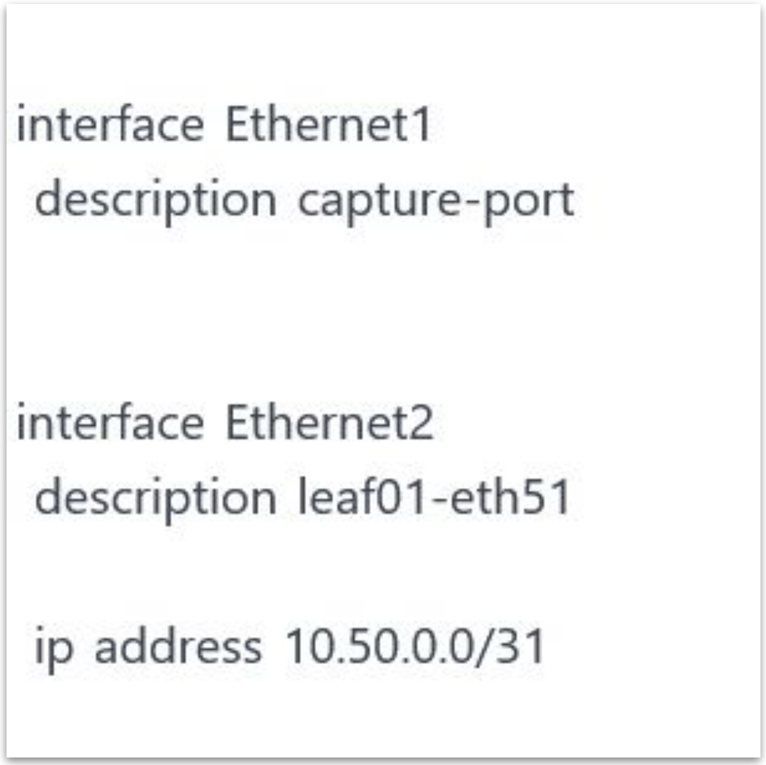
QUIZ TIME!

Jinja Whitespace Control

- Language blocks, `{{ }}`, `{% %}`, removed from final output
- Whitespaces surrounding the blocks are NOT removed by default
- Formatting of output is often not what was expected
- Indentation in Jinja templates makes things worse
- Manual control of whitespaces is possible but is unintuitive
- Options exist to change default settings to something more sane

Jinja Whitespace Control - Example

Unexpected whitespaces in the output



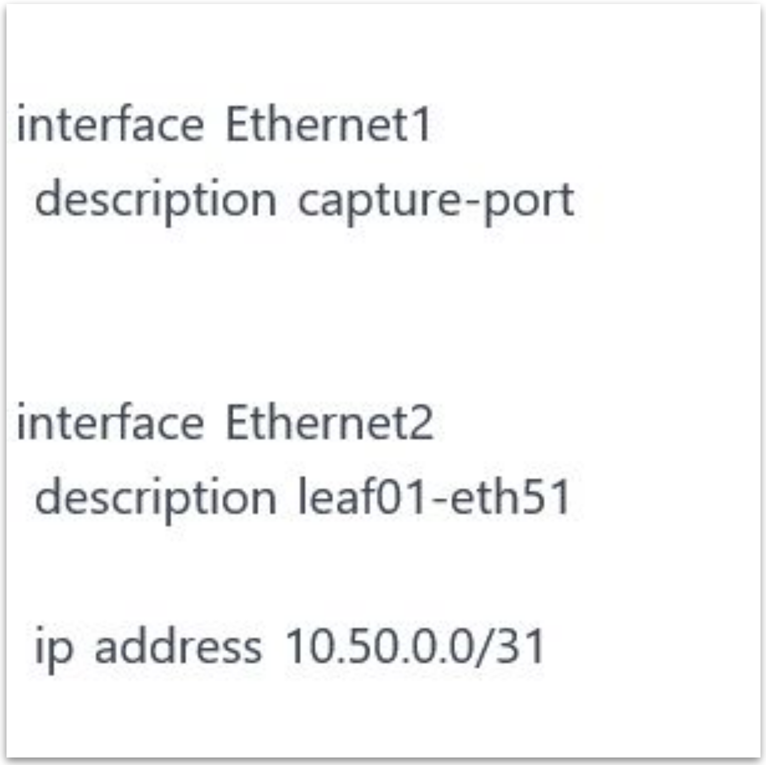
```

interface Ethernet1
 description capture-port


interface Ethernet2
 description leaf01-eth51

 ip address 10.50.0.0/31
```

```
{% for iname, idata in interfaces.items() %}
interface {{ iname }}
 description {{ idata['description'] }}
  {% if idata['ipv4_address'] is defined %}
 ip address {{ idata['ipv4_address'] }}
  {% endif %}
{% endfor %}
```

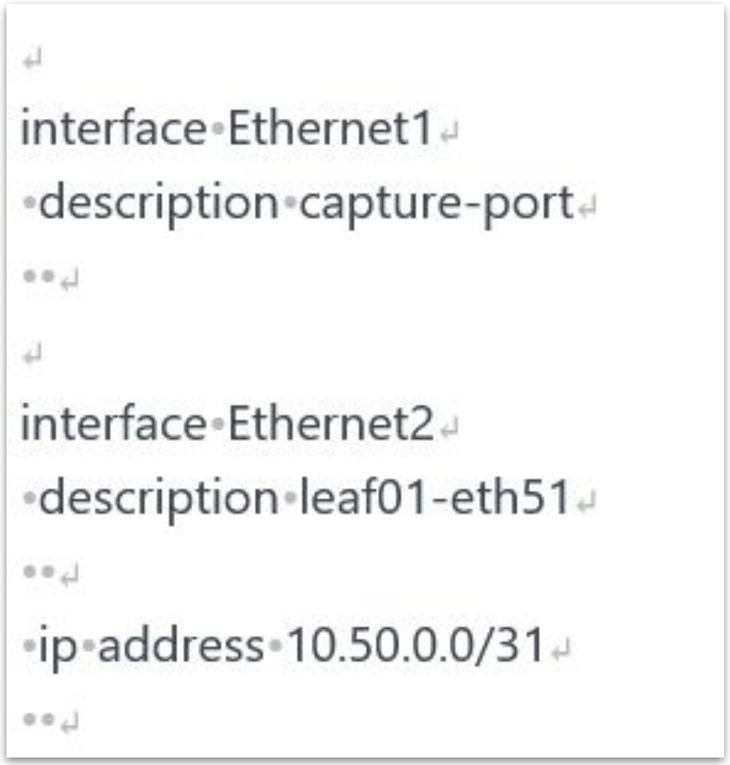
```
---
interfaces:
  Ethernet1:
    description: capture-port
  Ethernet2:
    description: leaf01-eth51
    ipv4_address: 10.50.0.0/31
```

Jinja Whitespace Control - Example

Same output with whitespaces made visible



```

interface Ethernet1
 description capture-port


interface Ethernet2
 description leaf01-eth51

 ip address 10.50.0.0/31

```



```
↵
interface·Ethernet1↵
·description·capture-port↵
··↵
↵
interface·Ethernet2↵
·description·leaf01-eth51↵
··↵
·ip·address·10.50.0.0/31↵
··↵
```

Jinja Whitespaces - How to find them

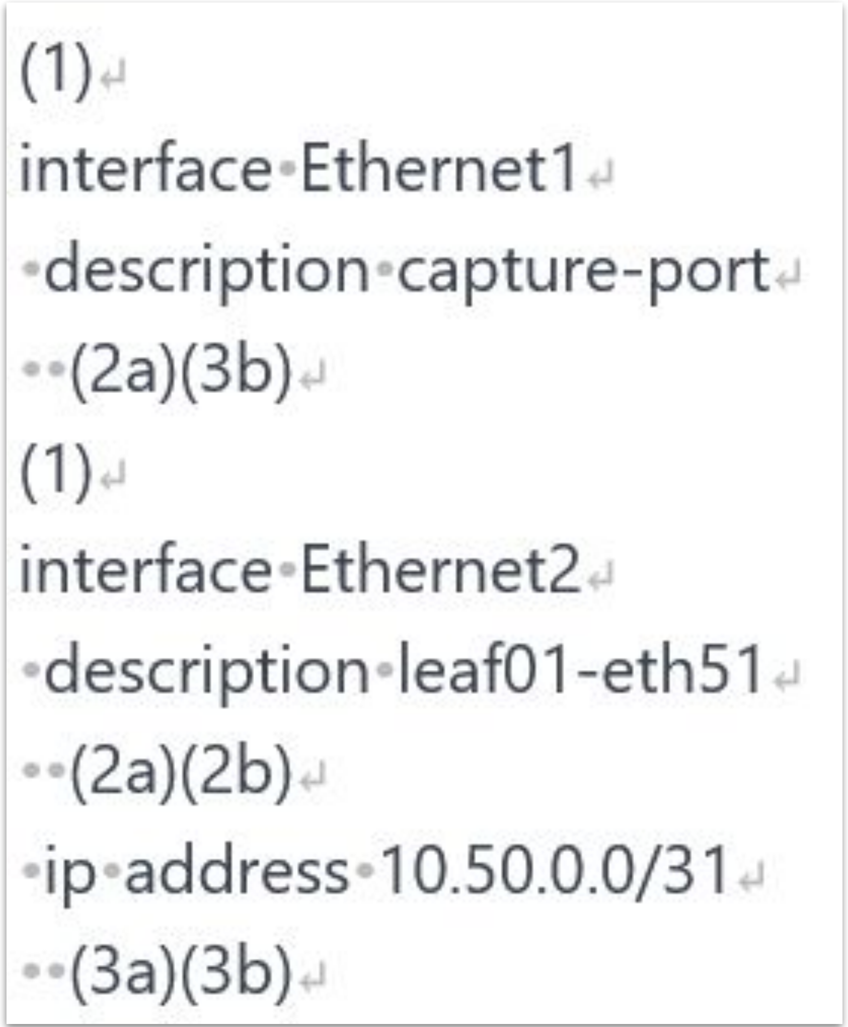
```
{% for iname, idata in interfaces.items() %}(1)
interface {{ iname }}
 description {{ idata['description'] }}
  (2a){% if idata['ipv4_address'] is defined %}(2b)
 ip address {{ idata['ipv4_address'] }}
  (3a){% endif %}(3b)
{% endfor %}
```

```
(1)
interface Ethernet1
 description capture-port
  (2a)(3b)
(1)
interface Ethernet2
 description leaf01-eth51
  (2a)(2b)
 ip address 10.50.0.0/31
  (3a)(3b)
```

Jinja Whitespaces - How to find them

```
{% for iname, idata in interfaces.items() %}(1)
interface {{ iname }}
 description {{ idata['description'] }}
  (2a){% if idata['ipv4_address'] is defined %}(2b)
 ip address {{ idata['ipv4_address'] }}
  (3a){% endif %}(3b)
{% endfor %}
```

```
(1)
interface Ethernet1
 description capture-port
  (2a)(3b)
(1)
interface Ethernet2
 description leaf01-eth51
  (2a)(2b)
 ip address 10.50.0.0/31
  (3a)(3b)
```



```
(1)↵
interface·Ethernet1↵
·description·capture-port↵
··(2a)(3b)↵
(1)↵
interface·Ethernet2↵
·description·leaf01-eth51↵
··(2a)(2b)↵
·ip·address·10.50.0.0/31↵
··(3a)(3b)↵
```

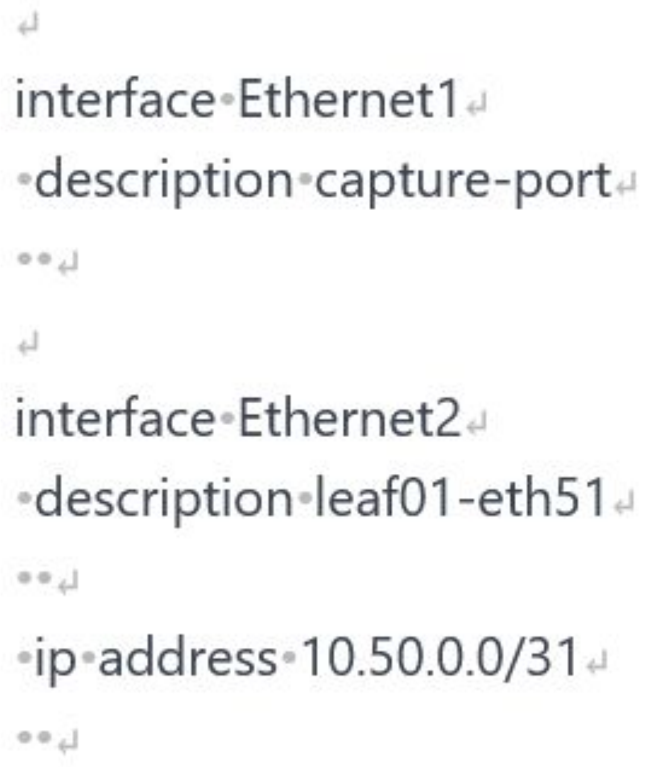
Jinja Whitespace - Manual control

- Minus sign `-` strips whitespaces before, or after, the block
    - `{%- ... %}` - strip whitespaces preceding the block
    - `{% ... -%}` - strip whitespaces following the block
- No space between minus sign and the block character
- ALL preceding/following whitespaces are removed, not just the ones on the same line as the block
- Very difficult to get right

Jinja Whitespace - Manual control

- Initial render, no stripping

```
{% for iname, idata in interfaces.items() %}
interface {{ iname }}
 description {{ idata['description'] }}
  {% if idata['ipv4_address'] is defined %}
 ip address {{ idata['ipv4_address'] }}
  {% endif %}
{% endfor %}
```



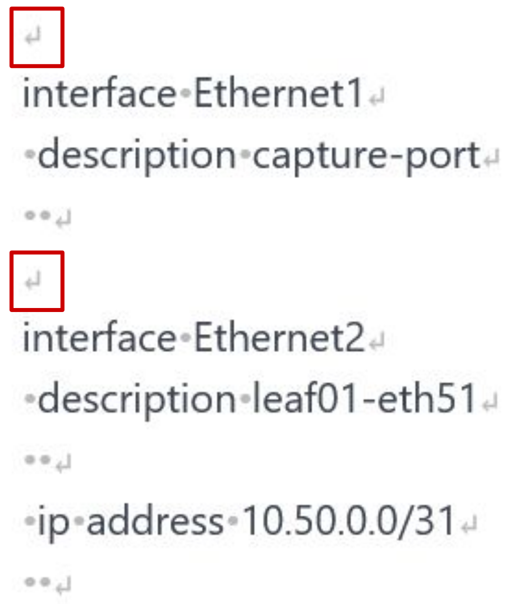
```
↵
interface·Ethernet1↵
·description·capture-port↵
··↵
↵
interface·Ethernet2↵
·description·leaf01-eth51↵
··↵
·ip·address·10.50.0.0/31↵
··↵
```

Jinja Whitespace - Manual control

- Manual stripping in `for` loop

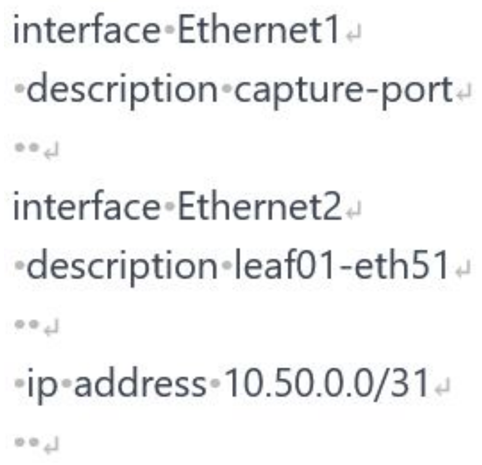
```
{% for iname, idata in interfaces.items() -%}
interface {{ iname }}
 description {{ idata['description'] }}
  {% if idata['ipv4_address'] is defined %}
 ip address {{ idata['ipv4_address'] }}
  {% endif %}
{% endfor %}
```

Before



```
↵
interface·Ethernet1↵
·description·capture-port↵
··↵
↵
interface·Ethernet2↵
·description·leaf01-eth51↵
··↵
·ip·address·10.50.0.0/31↵
··↵
```

After



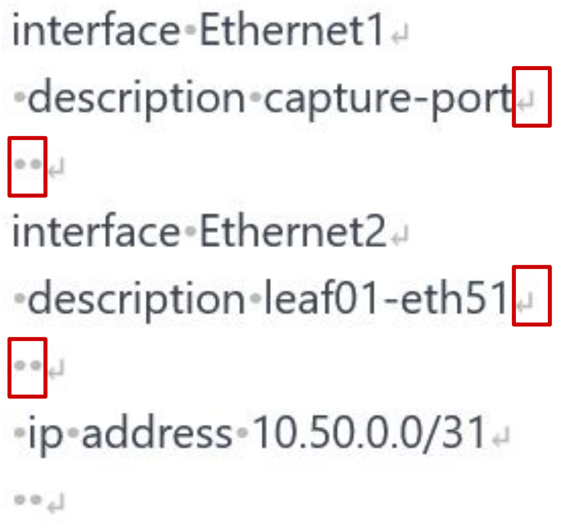
```
interface·Ethernet1↵
·description·capture-port↵
··↵
interface·Ethernet2↵
·description·leaf01-eth51↵
··↵
·ip·address·10.50.0.0/31↵
··↵
```

Jinja Whitespace - Manual control

- Manual stripping in `if` block

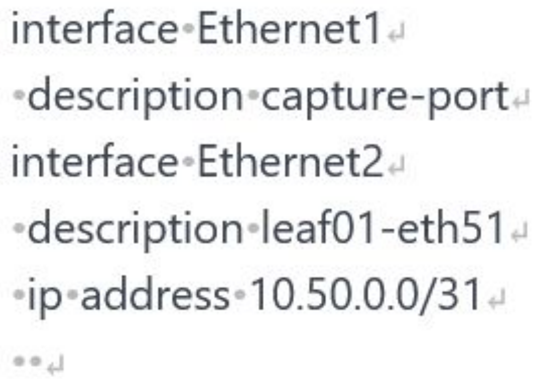
```
{% for iname, idata in interfaces.items() -%}
interface {{ iname }}
 description {{ idata['description'] }}
  {%- if idata['ipv4_address'] is defined %}
 ip address {{ idata['ipv4_address'] }}
  {% endif %}
{% endfor %}
```

Before



```
interface·Ethernet1↵
·description·capture-port↵
··↵
interface·Ethernet2↵
·description·leaf01-eth51↵
··↵
·ip·address·10.50.0.0/31↵
··↵
```

After



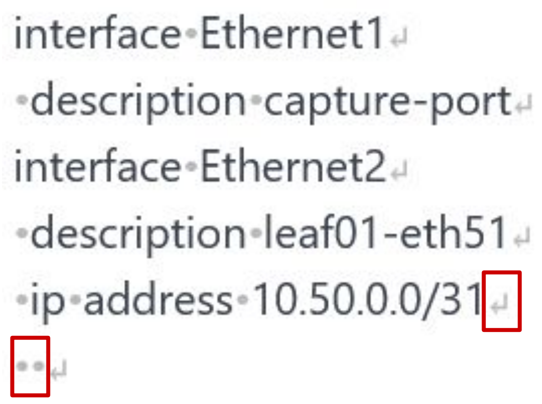
```
interface·Ethernet1↵
·description·capture-port↵
interface·Ethernet2↵
·description·leaf01-eth51↵
·ip·address·10.50.0.0/31↵
··↵
```

Jinja Whitespace - Manual control

- Manual stripping in `endif` block

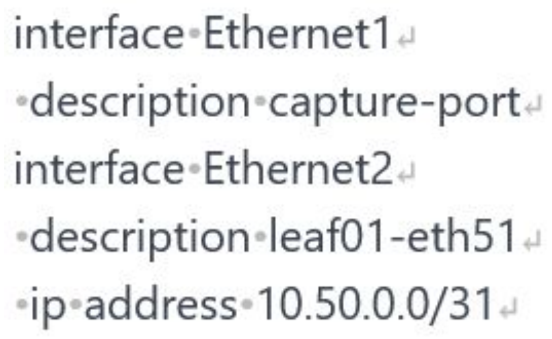
```
{% for iname, idata in interfaces.items() -%}
interface {{ iname }}
 description {{ idata['description'] }}
  {%- if idata['ipv4_address'] is defined %}
 ip address {{ idata['ipv4_address'] }}
  {%- endif %}
{% endfor %}
```

Before



```
interface·Ethernet1↵
·description·capture-port↵
interface·Ethernet2↵
·description·leaf01-eth51↵
·ip·address·10.50.0.0/31↵
··↵
```

After



```
interface·Ethernet1↵
·description·capture-port↵
interface·Ethernet2↵
·description·leaf01-eth51↵
·ip·address·10.50.0.0/31↵
```

Jinja Whitespace - Indenting blocks

- Indentation makes template logic easier to read and understand
- It's common to follow Python's way of indentation in Jinja
  - This amplifies whitespace rendering problems
- Instead move indentation to within the Jinja blocks
- Long blocks can span multiple lines
  - New lines within multiple-lines block won't be rendered

Jinja Whitespace - Indenting blocks

- Indentation moved to the inside of blocks
    - one or more spaces per indentation level

Python-style indentation

```
{% for vlan in vlans %}
 {% if vlan['status'] == 'active' %}
vlan {{ vlan['vid'] }}
  {% if vlan['vid'] < 100 %}
 name {{ vlan['name'] ~ '_local' }}
  {% else %}
 name {{ vlan['name'] }}
  {% endif %}
 {% endif %}
{% endfor %}
```

Jinja-recommended indentation

```
{% for vlan in vlans %}
{%  if vlan['status'] == 'active' %}
vlan {{ vlan['vid'] }}
{%   if vlan['vid'] < 100 %}
 name {{ vlan['name'] ~ '_local' }}
{%   else %}
 name {{ vlan['name'] }}
{%   endif %}
{%  endif %}
{% endfor %}
```

Jinja Whitespace - Global options

- **trim_blocks** - automatically removes the 1st new line after tag
- **lstrip_blocks** - strips same line spaces and tabs preceding the block
- Python `jinja2` default is **trim_blocks=False, lstrip_blocks=False**
- Ansible `template` default is **trim_blocks=True, lstrip_blocks=False**
- Recommendation is to enable both **trim_blocks and lstrip_blocks**

Jinja Whitespace - Python

- Trimming and stripping configured with `Environment` object
- `lstrip_blocks` and `trim_blocks` parameters control the behavior

```
j2_env = Environment(
    loader=FileSystemLoader("./templates"),
    lstrip_blocks=True,
    trim_blocks=True
)
```

Jinja Whitespace - Ansible

- Stripping per template
  - Add `#jinja2: lstrip_blocks: True` to the top of the template
- Trimming and stripping at the `template` module level
  - Use `lstrip_blocks` and `trim_blocks` parameters

```
    - name: "TASK 1: RENDER TEMPLATE"
      template:
        src: "base_config.j2"
        dest: "base_cfg.txt"
        lstrip_blocks: true
        trim_blocks: false
```

>>> network .toCode()

LAB TIME!

Labs: 01 and 01.5

Jinja Filters - Overview

- Filters are used to transform the data
- Use pipe `|` operator to apply filter to a variable
- Filters are essentially Python functions
- Jinja comes with a lot of prebuilt filters
- Filters can be chained - apply filter to results returned by another filter
- Frameworks using Jinja, like Ansible, provide many more
- You can create custom filters

Jinja Filters - Default

`default` - return predefined value if variable is not defined

```
{% for interface in interfaces %}
interface {{ interface['name'] }}
 mtu {{ interface['mtu'] | default(1500) }}
{% endfor %}
```

```
interface Ethernet1
 mtu 9124
interface Ethernet2
 mtu 1500
interface Ethernet3
 mtu 1500
```

```
---
interfaces:
  - name: Ethernet1
    mtu: 9124
  - name: Ethernet2
  - name: Ethernet3
```

Jinja Filters - Dictsort

`dictsort` - sorts dictionary by key or value. Sort by key is the default.

```
{% for pl_name, pl_lines in prefix_lists | dictsort %}
ip prefix list {{ pl_name }}
{%  for line in pl_lines %}
 {{ line }}
{%  endfor %}
{% endfor %}
```

```
ip prefix list pl-cogent-out
 permit 10.0.0.0/23
ip prefix list pl-ld-out
 permit 10.0.0.0/24
ip prefix list pl-zayo-out
 permit 10.0.1.0/24
```

```
---
prefix_lists:
  pl-ld-out:
     - permit 10.0.0.0/24
  pl-zayo-out:
     - permit 10.0.1.0/24
  pl-cogent-out:
     - permit 10.0.0.0/23
```

Jinja Filters - Join

`join` - concatenate elements in the collection and return resulting string. Separator is empty by default.

```
{% for port in ports if port['allowed_vlans'] is defined %}
Interface {{ port['name'] }}
Allowed vlans: {{ port['allowed_vlans'] | join(', ') }}
----------
{% endfor %}
```

```
Interface PortChannel51
Allowed vlans: 1, 105, 200
----------
Interface PortChannel57
Allowed vlans: 1, 105, 126
----------
```

```
---
ports:
  - name: Ethernet1
  - name: PortChannel51
    allowed_vlans:
      - 1
      - 105
      - 200
  - name: PortChannel57
    allowed_vlans:
      - 1
      - 105
      - 126
```

Jinja Filters - Map

`map` - apply filter to all objects or lookup an attribute

```
Remote AS numbers:

{{ bgp_neighbors | map(attribute='remote_as') | list }}
```

```
Remote AS numbers:

['65180', '65182', '65184', '65004', '64514']
```

```
---
bgp_neighbors:
  - neighbor: "175.135.172.146"
    remote_as: "65180"
  - neighbor: "42.116.171.166"
    remote_as: "65182"
  - neighbor: "71.163.23.191"
    remote_as: "65184"
  - neighbor: "87.90.154.72"
    remote_as: "65004"
  - neighbor: "13.167.37.139"
    remote_as: "64514"
```

Jinja Filters - Map

`map` - apply filter to all objects or lookup an attribute

```
Normalized device role names:

{{ device_roles| map('lower') | list }}
```

```
Normalized device role names:

['switch-access', 'switch-dist', 'switch-core', 'router-edge', 'firewall']
```

```
---
device_roles:
  - Switch-Access
  - Switch-Dist
  - Switch-core
  - router-Edge
  - Firewall
```

Jinja Filters - Selectattr

`selectattr` - return objects whose selected attribute passes given test

```
{% for vlan in vlans | selectattr('status', 'eq', 'active') %}
vlan {{ vlan['vid'] }}
 name {{ vlan['name'] }}
{% endfor %}
```

```
vlan 10
 name Data
vlan 20
 name Voice
```

```
---
vlans:
  - vid: 10
    name: Data
    status: active
  - vid: 20
    name: Voice
    status: active
  - vid: 50
    name: Printing
    status: planned
  - vid: 60
    name: wap
    status: offline
```

Jinja Filters - Rejectattr

`rejectattr` - filter out objects whose selected attribute passes given test

```
{% for vlan in vlans | rejectattr('role', 'eq', 'server') %}
{%  if loop.first %}
==== Non-server VLANs ====
{%  endif %}
- VID: {{ vlan['vid'] }}; Name: {{ vlan['name'] }}
{% endfor %}
```

```

==== Non-server VLANs ====
- VID: 10; Name: Data
- VID: 20; Name: Voice
```

```
---
vlans:
  - vid: 10
    name: Data
    role: office--data
  - vid: 20
    name: Voice
    role: office--data
  - vid: 1200
    name: SRV_APP
    role: server
  - vid: 1210
    name: SRV_DB
    role: server
```

Parsing Command Output

- CLI interactions result in semi-structured text meant for human consumption.
- Computers work with structured data:
  - In-memory objects (programming language specific).
  - Serializable formats for storage and transmission (JSON, XML, YAML etc.).
- For legacy devices that only provide unstructured text output and have no API, you have some options:
  - Use Regular Expressions (RegEx) to search for useful data in a blob of text.
  - Leverage libraries like pyATS/Genie, TTP, TextFSM that do the parsing for you (with limitations!)
- Screen scraping is a fragile temporary solution to a never ending problem.

Text Parsing Libraries

- TextFSM with ntc-templates
    - TextFSM is a "Python module which implements a template based state machine for parsing semi-formatted text" created by Google (opensource).
    - It takes two inputs, the text input and a template (a series of RegExs to help extract useful data).
    - **ntc-templates** is an NTC managed, community built repository of TextFSM templates for various network device CLI commands.
- Cisco pyATS (Automation Test System) and the Genie Library
    - Full testing suite with a parser library built-in (main interest here).
    - The Genie library provides a LOT of parsers for commands on (mostly) Cisco products.
- Template Text Parser (TTP)
    - If you hate RegExs but love doing something like reverse-Jinja templating.
- Ansible has integrations with these libraries to make life easier.

TextFSM with ntc-templates

```
> show cdp neighbors

Capability Codes: R - Router, T - Trans Bridge, B- Source Route Bridge
            S - Switch, H - Host, I - IGMP, r - Repeater, P - Phone

Device ID         Local Intrfce Holdtme    Capability   Platform   Port ID
R1-PBX            Gig 1/0/10    144         R S I       2811       Fas 0/0
TS-1              Gig 1/0/39    122         R           2611       Eth 0/1
```

```
[
   {
       "neighbor": "R1-PBX",
       "local_interface": "Gig 1/0/10",
       "capability": "R S I",
       "platform": "2811",
       "neighbor_interface": "Fas 0/0"
   },
   {
       "neighbor": "TS-1",
       "local_interface": "Gig 1/0/39",
       "capability": "R",
       "platform": "2611",
       "neighbor_interface": "Eth 0/1"
   }
]
```

>>> network .toCode()

LAB TIME!

Labs: 02-03

Module 2

Extending Ansible Through Its Plugin System

Ansible Plugins - Overview

- A lot of Ansible functionality comes from its Plugins
- **Vars Plugins**
  - Inject data from other sources apart from inventory/playbook/cmdline
  - host_vars, group_vars are loaded via plugin
- **Inventory Plugins**
  - Code that pulls dynamic inventory data from other sources (e.g. Nautobot, APIs, Databases etc.)
- **Callback Plugins**
  - Control how Ansible responds to events. Output coming from the CLI.
  - Make Ansible output in JSON or even send API calls to Slack
- **Other: Action, Cache, Lookup, Strategy, Become, Connection**

Ansible Plugins

Connection Plugins

- How Ansible connects to managed devices
- Most common is SSH via Paramiko or the native SSH client

```
> ansible-doc -t connection -l
docker       Run tasks in docker containers
httpapi      Use httpapi to run command on network appliances
kubectl      Execute tasks in pods running on Kubernetes
local        execute on controller
napalm       Provides persistent connection using NAPALM
netconf      Provides a persistent connection using the netconf protocol
network_cli  Use network_cli to run command on network appliances
paramiko_ssh Run tasks via python ssh (paramiko)
saltstack    Allow ansible to piggyback on salt minions
ssh          connect via ssh client binary
vmware_tools Execute tasks inside a VM via VMware Tools
... and many more!
```

Ansible Plugins

Become Plugins

- Help Ansible use various privilege escalation systems
- As in “become” another user

```
> ansible-doc -t become -l
doas       Do As user
dzdo       Centrify's Direct Authorize
enable     Switch to elevated permissions on a network device
ksu        Kerberos substitute user
machinectl Systemd's machinectl privilege escalation
pbrun      PowerBroker run
pfexec     profile based execution
pmrun      Privilege Manager run
runas      Run As user
sesu       CA Privileged Access Manager
su         Substitute User
sudo       Substitute User DO
```

Filter Plugins

- **Filter Plugins** manipulate data and are a feature of Jinja
  - **Jinja Built-in Filters**
    - **https://jinja.palletsprojects.com/en/3.0.x/templates/#builtin-filters**
  - **Ansible Filters**
    - **https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html**
  - **Your Custom Filters**
- They are an integral part of Ansible's DSL (Domain Specific Language) - i.e. the YAML playbooks and its templating engine (including the **template** module!)
  - https://github.com/ansible/ansible/tree/devel/lib/ansible/plugins/filter

Locating Ansible Plugins

Where does Ansible look for plugins?

- Any folder added to the ANSIBLE_PLUGIN_TYPE_PLUGINS environment variable (it's a colon-separated list like your system PATH)
  - e.g. ANSIBLE_FILTER_PLUGINS
  - Change the plugin search path using a custom **ansible.cfg** file
- Loaded from specific folders found next to the playbook
  - e.g. **filter_plugins,** lookup_plugins, connection_plugins etc.
  - this makes for simple packaging and distribution with your plays

- https://docs.ansible.com/ansible/latest/dev_guide/developing_locally.html

LAB TIME!

Labs: 04-05

LAB TIME!

Labs: 06-07

Ansible Modules

- A **Module** is a reusable, standalone script that Ansible runs (via API, **ansible** or **ansible-playbook**)
  - Ansible provides the runtime framework, the input parameters, and captures the results (or the output)
  - Thousands of built-in modules - either from Redhat or third parties like NetworkToCode, Cisco, Juniper, Amazon, VMware etc.
- If you need functionality that's not available you can write your own
  - Others might find it useful so consider open sourcing it! :)

Ansible Modules

- Where does Ansible look for modules?
  - Any folder added to the ANSIBLE_LIBRARY environment variable (it's a colon-separated list like your system PATH)
  - **(default)** $HOME/.ansible/plugins/modules
  - **(default)** /usr/share/ansible/plugins/modules/
  - Change the module search path using a custom **ansible.cfg** file

```
> ansible --version
ansible 2.9.9
  config file = /etc/ansible/ansible.cfg
  configured module search path = ['/etc/ntc/ansible/library']
  ansible python module location = /usr/local/lib/python3.6/site-packages/ansible
  executable location = /usr/local/bin/ansible
  python version = 3.6.8 (default, Jun 11 2019, 01:16:11) [GCC 6.3.0 20170516]
```

Ansible Modules

- A **Module** is a python script - its filename is the module name!
  - A large part will be documentation… make sure you include some with your module to make it easy to use it.
  - Documentation on writing Module Documentation
- Modules take arguments, which with their options = argument_spec
  - **required**, default, choices, aliases
  - **type**: str (default), list, dict, bool, int, float
- Modules can and *should* support check_mode (aka dry run)
- Ansible expects modules to output JSON
  - `{ 'changed': false, 'failed': true, 'msg': 'host unreachable' }`

LAB TIME!

Labs: 08-10

Module 3

Using Ansible Content Collections to Consume and Package Additional Plugins

Dynamic Inventory Management in Ansible

Ansible Versions & Collections

- Ansible up to 2.9 is "batteries included".
  - You install the "ansible" package and get 3700+ modules.
  - These modules and other plugins are mostly community supported.
  - Difficult to manage fixes, updates, releases for all these moving parts.
- Most plugins are now moved into Content Collections.
  - Ansible Galaxy is the public hub and tool to manage collections.
- Major releases and changes to Ansible packages:
  - Up to (and including) **2.9.x** you installed one package: **ansible**
  - In **2.10** the core is called **ansible-base 2.10.x** and the collections are in **ansible 3.0.x**
  - From **2.11** the core is called **ansible-core 2.11.x** and the collections are in **ansible 4.0.x**

Using Ansible Content Collections

- Introduced in Ansible 2.8, usable in 2.9, getting better in 2.10+
- Collections are a packaging format for Ansible related content
  - Modules and Plugins (filters, connection, inventory etc.)
  - Roles, Playbooks (Templates, Files, Vars etc.), Documentation
- Fully Qualified Collection Name (FQCN)
  - `namespace.collection.content_name`
  - `cisco.ios.ios_config`
- Public centralized repository at https://galaxy.ansible.com/
  - Can also install collections from any Git repository
- CLI tool: **ansible-galaxy**
  - Install, List, Verify Collections on the Ansible Control Host
  - Ansible Documentation on Using Collections

Developing Ansible Content Collections

- CLI tool: **ansible-galaxy**
  - Create a collection skeleton
  - Package the collection
  - Publish to Galaxy
- Ansible Documentation on [Developing Collections]
- Existing Collections on [GitHub]

```
collection/
├── docs/
├── galaxy.yml
├── meta/
│   └── runtime.yml
├── plugins/
│   ├── modules/
│   │   └── module1.py
│   ├── inventory/
│   └── .../
├── README.md
├── roles/
│   ├── role1/
│   ├── role2/
│   └── .../
├── playbooks/
│   ├── files/
│   ├── vars/
│   ├── templates/
│   └── tasks/
└── tests/
```

LAB TIME!

Labs: 11-12

Network to Code – Confidential

Ansible Inventory Plugins

Inventory Plugins

- How Ansible builds its inventory of managed devices
- Most common is the INI static file inventory with YAML based data (vars)

```
> ansible-doc -t inventory -l
auto                Loads and executes an inventory plugin specified in a YAML config
aws_ec2             EC2 inventory source
ini                 Uses an Ansible INI file as inventory source
netbox              NetBox inventory source
script              Executes an inventory script that returns JSON
toml                Uses a specific TOML file as an inventory source
tower               Ansible dynamic inventory plugin for Ansible Tower
virtualbox          virtualbox inventory source
vmware_vm_inventory VMware Guest inventory source
yaml                Uses a specific YAML file as an inventory source
... and many more!
```

LAB TIME!

Labs: 13-14

Thank you!

For direct feedback or questions about the course:
Cristian Sîrbu - cristian.sirbu@networktocode.com

Find me **@cmsirbu** on Twitter, GitHub, NTC Slack

General help: NetworkToCode Slack (#ansible #python channels)