Automate Your Network

Introducing the Modern Approach to Enterprise Network Management

John Capobianco

Sr. IT Planner and Integrator – Canadian House of Commons

AnsibleFest

Who am I?

A Network Engineer turned Developer

Computer Programmer Analyst – St. Lawrence College (2000-2003)

Client / Server – 2000 – 2008 (A+, Network +, MCITP: EA/SA; ITIL)

Networks – 2008 – present (CCNA, CCNP, CCNP:DC, 5x Cisco Specialist)

Professor CNTS Program – St. Lawrence College (2010-2013)

Joined the House of Commons 2013 – "NG-PPN"

According to my e-mail archives I first became aware of **Ansible** in **April 2017**

What have I automated?

Everything!

Cisco:

Catalyst 2960 / 3560 / 3750 / 3850 / 4500 / 6500 / 9300

Nexus 9K / 7K / 5K / FEX

Integrated Services Routers 891FW / 4400

FXOS / ASA

Identity Services Engine

Prime Infrastructure

Umbrella

Cisco.com APIs

Servers:

Red Hat Enterprise

CentOS

Microsoft Windows Servers

VMWare

Clouds:

Azure

AzureStack

Storage:

Solidfire

Appliances:

F5 BIGIP

Forcepoint NGFW

Gigamon

What are my challenges?

Why did I turn to Ansible?

Ever growing complexity !

From VRFs to clouds to BGP to OSPF to 802.1x to Netflow to STP

Consistency at scale

How to ensure hundreds / thousands of devices and their interfaces are correctly configured to a corporate standard at scale

How do I **know** the network is functional

Agility

How to rise to the challenge of ever increasing demand for network services

Human operational limitations

My general workflow of submitting changes to be manually implemented by humans was becoming exceeding difficult

Why Ansible?

It's not just another new tool

Simple	Powerful	Agentless
Human readable automation	App deployment	Agentless architecture
No special coding skills needed	**Configuration management**	OpenSSH, WinRM, and HTTPS
Tasks executed serially	Workflow orchestration	No agents to exploit or update
Usable by every team	**Network automation**	Get started immediately
Get productive quickly	Orchestrate the app lifecycle	**More efficient & more secure**

Introduction to Ansible

What is Ansible ?
Why you should use Ansible ?
The basics of Ansible

What is Ansible?

It is *NOT* C++ for Networks !

The Ansible project is an open source community sponsored by Red Hat. It's also a **simple automation framework** that describes infrastructure in **Ansible playbooks**.

Under the hood Ansible is **Python** code, but this is **abstracted**

You do **NOT** need to be a software developer or programmer analyst

It's an **automation engine** that runs Ansible playbooks. **Not a programing language!**

The Ansible Advantage

CROSS PLATFORM – Linux, Windows, UNIX
Agentless support for all major OS variants, physical, virtual, cloud and network devices

HUMAN READABLE – YAML
Perfectly describe and document every aspect of your application environment

PERFECT DESCRIPTION OF APPLICATION
Every change can be made by playbooks, ensuring everyone is on the same page

VERSION CONTROLLED
Playbooks are plain-text. Treat them like code in your existing version control.

STATIC OR DYNAMIC INVENTORIES
Capture all the servers 100% of the time, regardless of infrastructure, location, etc.

ORCHESTRATION THAT PLAYS WELL WITH OTHERS
Homogenize existing environments by leveraging current toolsets and update mechanisms.

Why Automation is Important

Not in the future – in the present!

Your networks, services, and applications **are more than just collections of configurations**. They're a finely tuned and **ordered list** of tasks and processes that result in **your working application**.

Ansible can do it all:

Provisioning

App Deployment

Configuration Management

Multi-tier Orchestration

Documentation

On-Prem or Clouds

Use Case Examples

Some ideas to get you thinking

Configuration Management: Centralizing configuration file management and deployment

Security and Compliance: Auditing and remediation and integrate into automated processes

Continuous Delivery: Keep your applications properly deployed throughout their entire lifecycle

Provisioning: Helps streamline the process whether your deploying on bare-metal or the cloud

Feature deployment: Manage the entire feature lifecycle from development to production.

Orchestration: Bring multiple configurations together and manage them as a whole

Network Automation: device provisioning, performing compliance checks, validate network topology, upgrading software.

Updating Systems Documentation & Reporting: Created documentation derived from your code, create reports on multiple systems in a variety of formats.

How Does Ansible Work?

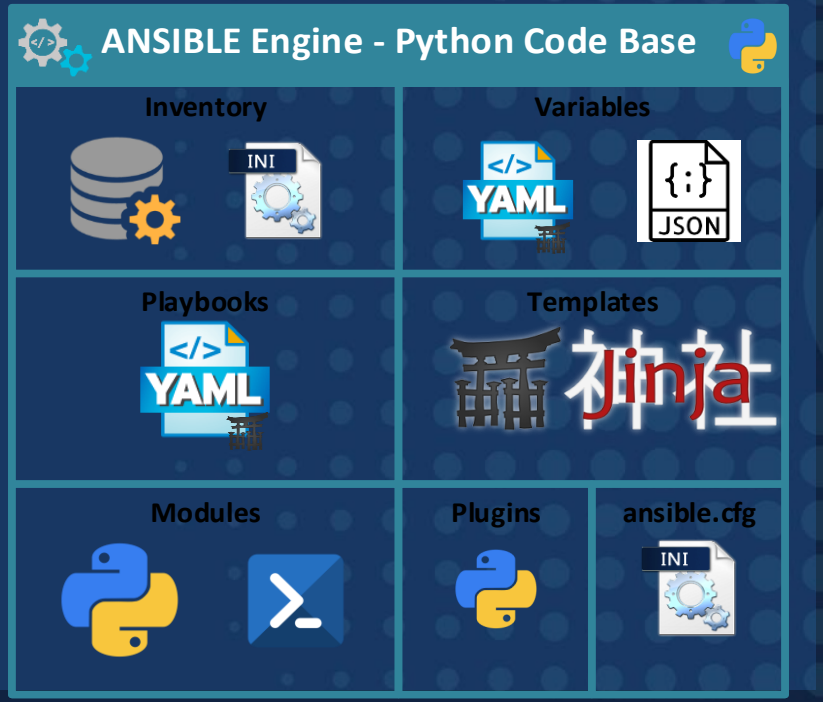
Inventory – ini file or dynamic:
- Contains hosts & group information

Playbooks - YAML with imbedded Jinja:
- Tasks executed sequentially, Invokes Modules
- Ansible tools: Plays, Tasks, loops, Handlers, Blocks, Tags, Check mode

Templates - jinja to generate:
- Configuration files
- Documentation (markdown/html)

Plugins – Python:
- Codes the plugs into the core engine
- Adaptability for various uses and platforms

Modules – Python & PowerShell:
- Tools in the toolkit, extend simplicity to entire stack
- Can be any language that returns json



Key Ansible Features

Idempotency:
The ability to run an operation which produces the same result whether run once or multiple times

Check-mode:
validate your code before actually making the change, build confidence!

Diff-Mode:
Some modules support diff mode, use in combination with check-mode to review before applying change!

Limit:
Dynamically, at run time, adjust the scope of your playbook

Ansible.cfg

Within the same folder as your Ansible **playbooks** there is an **Ansible.cfg** file

Certain settings in Ansible are adjustable via a configuration file (**ansible.cfg**). The stock configuration should be sufficient for most users, but there may be reasons you would want to change them.

Here you set paths to certain folders; set environment variables; and add additional options and features

```
[defaults]
host_key_checking = False
inventory=../hosts
gathering=explicit
transport=local
retry_files_enabled = False
filter_plugins=../filter_plugins
library = ~/.local/lib/python3.8/site-packages/napalm_ansible
action_plugins = ~/.local/lib/python3.8/site-packages/napalm_ansible/plugins/action
roles_path    = ../roles

[persistent_connection]
connect_timeout = 380
command_timeout = 320
```

Inventory - hosts

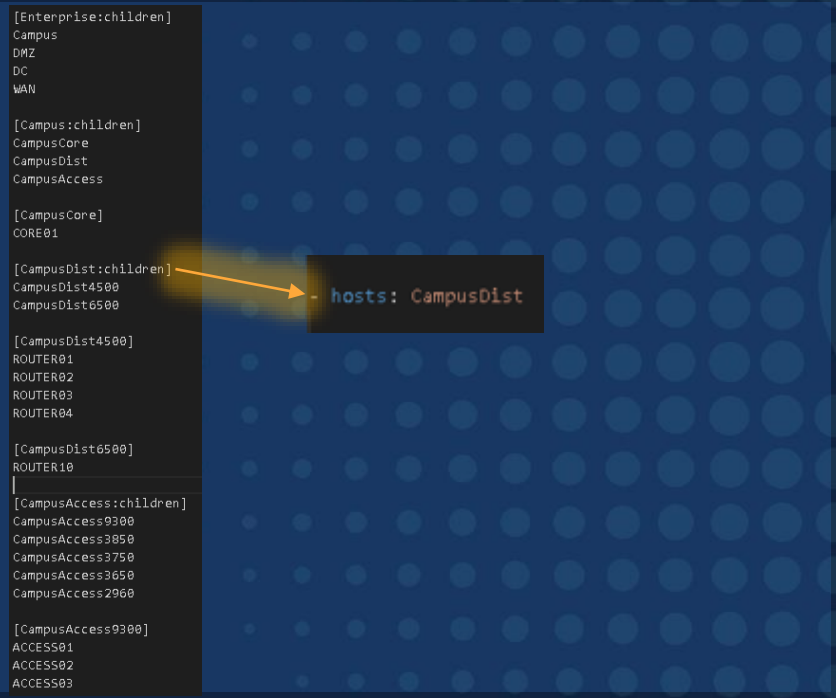
Ansible uses a **hosts** file to identify targets and scope of **playbooks**

hosts can be **static** or **dynamic** from an upstream source of truth (IPAM, inventory system, etc)

hosts can be organized into logical hierarchies, grouped by platform, function, or role; geographic, campus, data center, or cloud location

Custom groups of hosts can be created to identify the scope of a specific **playbook**

Completely flexible



Ansible Playbooks

- **YAML Ain't Markup Language (YAML)**
- Human-readable
- Simple or complex
- Serially executed
- Can be interactive (prompted input)
- Scope based on inventory (hosts) file

```yaml
---

- hosts: CampusDist

  vars_prompt:

    - name: ios_password_prompt
      prompt: "Enter local admin password"
      private: yes

  tasks:
    - set_fact:
        ios_password: "{{ios_password_prompt}}"
      no_log: true
      delegate_to: localhost
      run_once: true


    - name: run show ospf neighbors
      ios_command:
        commands: show ip ospf neighbor

      register: ospfneighbor_result

    - set_fact: ospfneighbor={{ ospfneighbor_result.stdout_lines | to_nice_yaml }}

    - name: create log file for all devices
      file: path=../../documentation/CAMPUS/DISTRIBUTION/ospf_neighbors.txt state=touch
      check_mode: no

    - name: log to file for all devices
      lineinfile:
        insertafter: EOF
        path: ../../documentation/CAMPUS/DISTRIBUTION/ospf_neighbors.txt
        line: '{{ item }}'
      with_items:
        - "#####{{ inventory_hostname }}#####"
        - "{{ ospfneighbor }}"
```

The Modern Enterprise Network Management Toolkit

Introducing the toolkit to maximize your productivity and chance of success

Brief introduction and overview of the key components

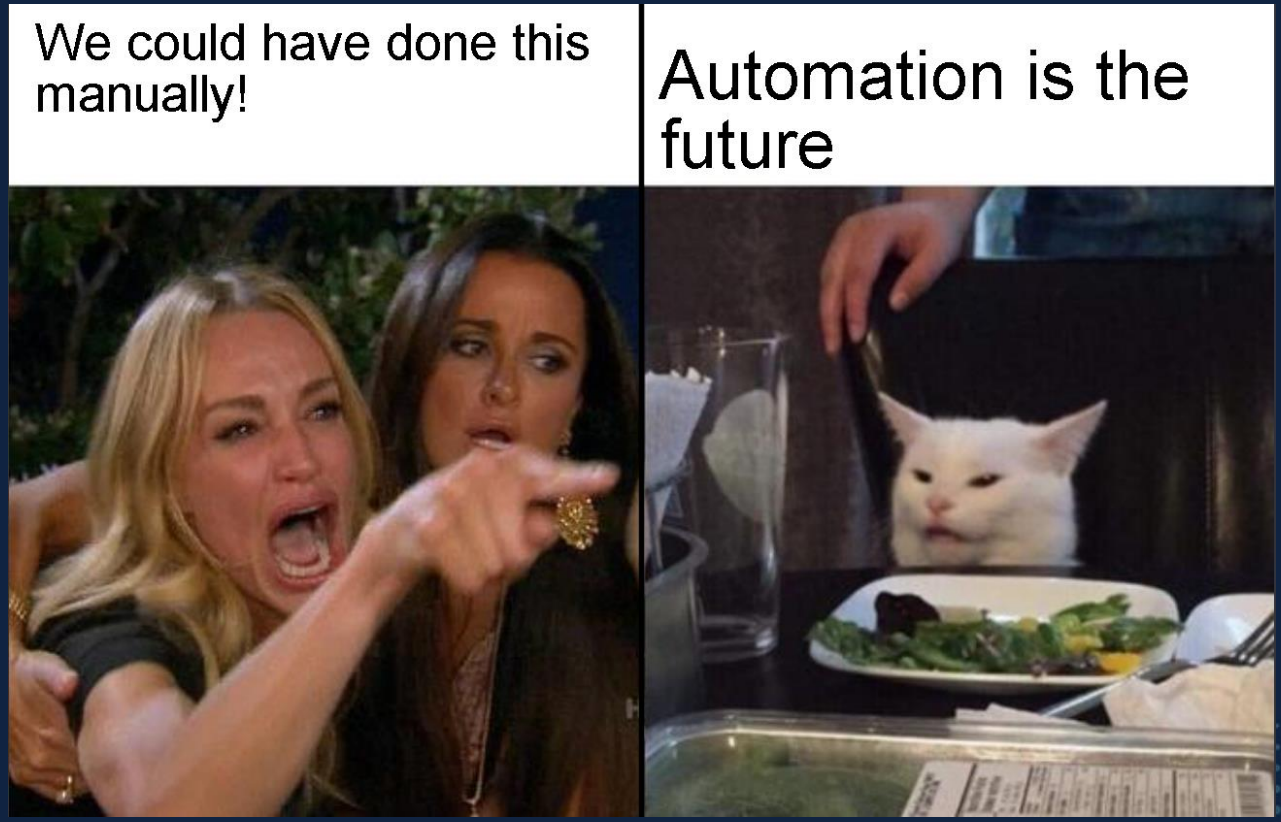
Toolkit – A Journey

Learn from my mistakes!

My first playbook

- Written in Notepad on my Windows workstation
- Transferred to Linux host with Ansible with WinSCP
- Back and forth trial and error (with emphasis on error)
- Eventually had “success” but I was clearly missing the big picture
- There had to be a better way
- Enter the Modern Enterprise Network Management Toolkit


CI/CD
State Validation
Full Configuration Management
Tactical Changes 1 Time Playbooks
Capture Network State Gather Information
Establish Connectivity
Build Toolkit


We could have done this manually!
Automation is the future

A Standard (Legacy) Toolkit

Old tools for old methodologies

A text editor, possibly “enhanced”

A console cable (or VTY port)

Putty

A file transfer, or several, program (FTP / SFTP / SCP / TFTP)

RAW CLI output gathered device-by-device

Infrastructure As Code

This revolution and evolution allows traditional system administrators and engineers work with **code** like a developer!

Accepting software development lifecycle practices are vital because **automation isn't about building a button** to press when its time.

Instead, create an **automation framework** that adjusts the infrastructure to your definition of it.

Changing the code changes the infrastructure not pushing a button you built

A Modern Toolkit's Components

Linux

Ansible and Ansible Modules

Ansible Tower

Git

GitHub

Microsoft Team Foundation Server (TFS) / AzureDevOps

Visual Studio Code

VS Code Extensions

APIs
- **cURL**
- **Postman**
- **Ansible URI Module**

RegEx

YAML

Jinja2

Python

Markdown

CSV

Mind Maps

pyATS

Genie Parsers

Docker

JSON

XML

HTML

Linux

Open, typically CLI driven, Operating System

Several flavors including licensed and supported Red Hat Enterprise License (**RHEL**) versions or free versions like CentOS

Required to host your **Ansible** environment

Playbooks executed from Linux host

To install Ansible on RHEL or CentOS:

sudo yum install ansible

**** Linux host requires access to the network's management plane ****

Should be **secure**

May require firewall permissions to reach targets

Ansible Modules

Ansible supports all different kinds of devices with specific **modules** and anything with a REST API !

Look up the platform and find the available modules

There are specific modules for, example, banner or NTP, or more generic command and config modules

Some modules accept templates as the source of input (more on templates later)

Cisco:

ios command: Performs IOS commands on Catalyst devices in privileged exec mode

Ios config: Performs IOS commands in configuration modes

nxos command

nxos config

asa command

asa config

Git

Since we are now working with code – we need a version and source control system. Enter: **Git**

Git is a distributed version control system, meaning your local copy of code is a complete version control repository.

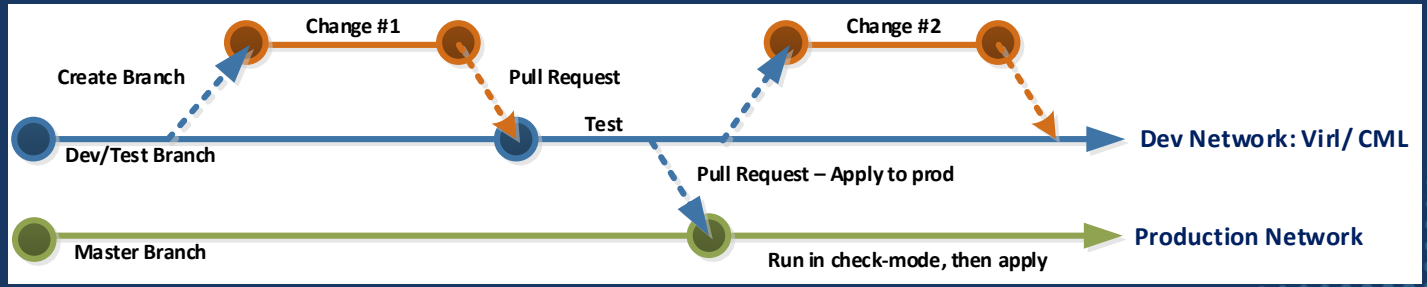
Git tracks changes and uses internal mechanisms such as **commits** and **pull requests** to merge changes back into **main**

Git Branching

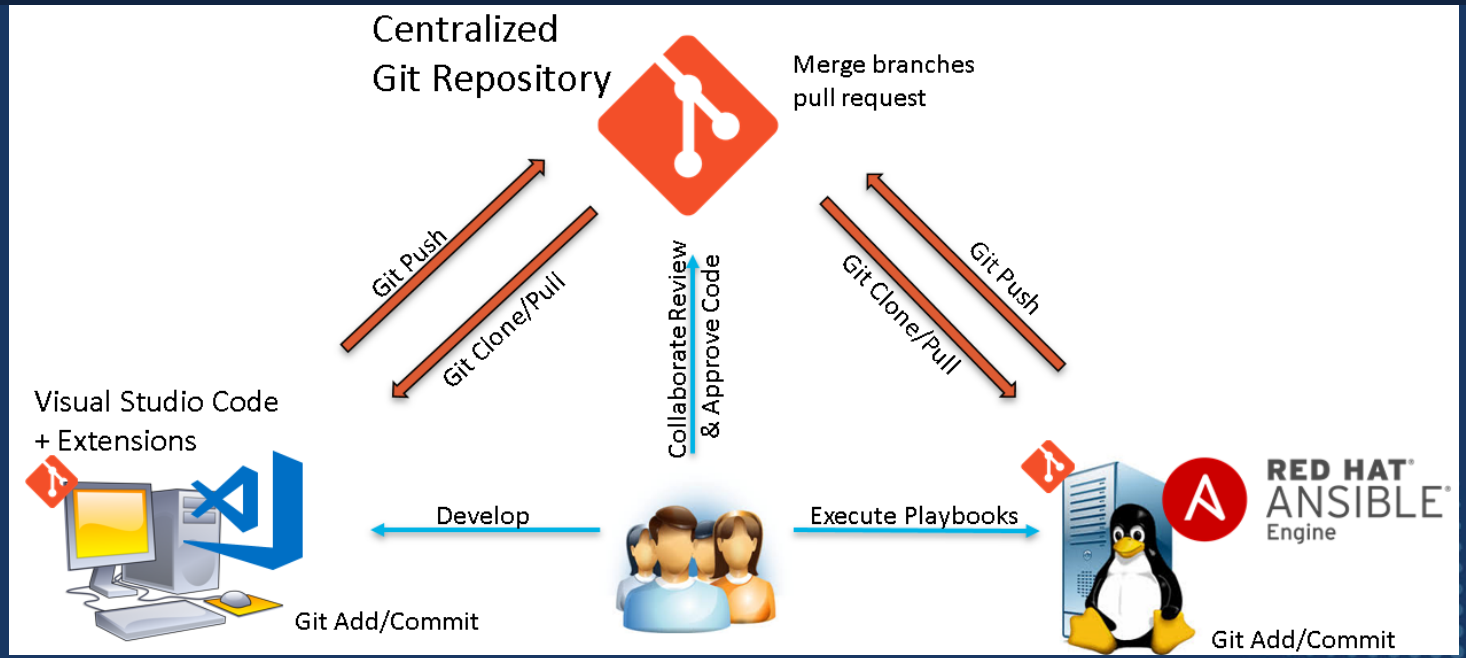
You will need to develop a **branching strategy** as a core component of Git is **branches**

The **main** branch, which represents valid, tested, known-good code, should be protected from direct changes

Any changes required, such as a new feature, new device, bug fix, or enhancement should be done under a working branch



Git Branching



Git Repositories

GitHub is the most famous example

Be very careful with what you store in GitHub repositories if they are public !

SSH keys, device secrets, public IP addresses, DNS records; many things should be handled with the utmost care.

There are private enterprise grade systems like Microsoft TFS or Azure Dev Ops to safely store Git repositories privately to your organization and secured behind RBAC and Kerberos controls.

Common Git Operations

git clone: Clone a Git repository locally either to your development system, likely Windows, using VS Code, or into your Ansible host using the Linux CLI

git pull: Refresh your local copy from the central repository to pull down any upstream changes

git add: Stage local changes to be committed

git commit: Commit local changes to the local repository

git push: Push locally committed changes into the central repository

Pull request: Pull changes from a branch into another branch, typically **main**

Visual Studio Code

VS Code blurs the lines between a powerful text editor and a full blown IDE and makes creating Ansible-related files *much* simpler

A vast library of powerful **extensions** further add to the power and native capabilities

Fully Git integrated and aware

New **Live Share** capabilities allow for teams to write code *together* sharing the editing space collaboratively with audio and whiteboard integrations

VS Code Extensions

Ansible	Markdown Preview Enhanced
Better Jinja	Markdown Lint
Chat	Open in Default Browser
Excel Viewer	Peacock
Git (integrated at OS level)	PowerShell
Git Graph	Python
Git Lens	Various Themes
Live Share	VS Code Icons
Live Share Audio	YAML

Application Programming Interface (API)

An **API** is a **web-service** that provides the ability to interact with a system using structured data.

You can retrieve stateful information using a representational state transfer (**REST) API** using a **GET** command.

Records can be added using a **POST** and updated using a **PUT** (which is idempotent)

Automating APIs

Traditionally APIs were interfaced with using **cURL** at the CLI.

A GUI client called **Postman** was released in 2012.

Postman or cURL can be used to validate an API call, or string, and then the API activity can be *automated* with the **Ansible URI** module

```
# Get Device ID from Hostname #
    - name: GET Device Entity ID's
      uri:
        url: "{{ enterprise_defaults.servers.api.protocol }}://{
        user: "{{ enterprise_defaults.servers.api_user }}"
        password: "{{ api_password }}"
        method: GET
        return_content: yes
        validate_certs: no
        force_basic_auth: yes
        follow_redirects: all
        body_format: json
      register: device_id
      delegate_to: localhost
```

eXtensible Markup Language (XML)

A common form of structured data an API often returns is in **XML** format

Extensible Markup Language (**XML**) is a markup language that defines a set of rules for encoding documents in a format that is both **human-readable** and **machine-readable**

XML can be challenging to work with

Some legacy APIs only return XML

```
<Configuration>
  <BGP>
    .
    .
    .
    <BGPEntity>
      <NeighborTable>
        <Neighbor>
          <Naming>
            <NeighborAddress>
              <IPV4Address>10.0.101.6</IPV4Address>
            </NeighborAddress>
          </Naming>
          <RemoteAS>
            <AS_XX>
              0
            </AS_XX>
            <AS_YY>
              6
            </AS_YY>
          </RemoteAS>
        </Neighbor>
        <Neighbor>
          <Naming>
            <NeighborAddress>
              <IPV4Address>10.0.101.7</IPV4Address>
            </NeighborAddress>
          </Naming>
          <RemoteAS>
            <AS_XX>
              0
            </AS_XX>
            <AS_YY>
              6
            </AS_YY>
          </RemoteAS>
        </Neighbor>
      </NeighborTable>
    </BGPEntity>
    .
    .
    .
  </BGP>
```

JavaScript Object Notation (JSON)

JSON (JavaScript Object Notation) is a lightweight data-interchange format.

It is easy for humans to read and write. It is easy for machines to parse and generate

JSON is often preferred to XML

Ansible can use **json query** functions to do SQL-like lookups against JSON data

More on YAML

What is YAML :
A **human-readable data serialization language** for all programing languages

Similar to XML and JSON, but way **easier to read and write**

It is **line-oriented**, therefore makes it easier to compare / diff (useful when reviewing code / pull request)

YAML Syntax

Get to know the rules

Indentation matters: **spaces only**, **no tabs** allowed

Comments start with “#”

Start of document with “---”

End of document with “…” (optional)

Scalar values: string, number, Boolean

Special Character need to be quoted “”:
(:, {, }, [,], ,, &, *, #, ?, |, -, <, >, =, !, %, @, underscore)

Multi-line, start with “|” or “>”, improves readability – must be indented under |

```yaml
# This is a comment, starts with "#"

# Start of a yaml document with "---"
---

# end of yaml document with "..."
...

#Scalar values
scalar values:
  - hello   #string
  - 99      #number
  - True    #boolean
  - http://docs.ansible.com # URL can be a string

#special string should be quoted when it contains these special charaters:
special: "YAML: is a simple language"


# multi-line, start with "| or >" , to improve readability
multi_line:
  - |
    multi-line string
    indentation matters!
```

Lists and Key-Pair Values

The building blocks

LISTS

Ordered collection of values also know as sequences in YAML

Accessed by absolute position of an item (zero based in YAML)

Each value could be a simple value, list, object.

One item per line

Every line start with a dash “- “

Item value is the rest of the line

```yaml
List:
  - zero
  - one
  - two
  - three
  - four
```

Don’t use comments in-line, as they are treated differently by various parsers!

Shorthand with bracket “[ ]” , but not as easy to read!

Key-Pair Values

A dictionary also know as mappings in YAML is represented in a simple key: value form , The colon must be followed by a space “: “

It is accessed by the key content, and is not sorted!

Values can be any scalar value: string, number, Boolean or It can also contain a complex list, variables

Key should be valid python variable: ASCII letters, numbers, underscore) **no dashes “-”**

```yaml
ioscli:
  username: root
  password: H3ll0w0r1d!@
  host: "{{inventory_hostname}}"
  port: 22
```

More YAML!

Yes, more **YAML**!

Along with Ansible **playbooks** you will eventually start to model your data in **data models**

Data models are YAML files that contain key-pair values and lists that describe the *intended configuration* of an entity on a specific device, group of devices, or every device

The values in the data model become **variables** available to the **templates** and can be used to generate configurations or documentation

There are **variables** available at the **group**, **role**, and **host** level

There are matching YAML files for the groups and hosts you add to the **hosts** file

An example of a **group** variable in a file called **Enterprise.yml** in the **group vars** folder

```yaml
enterprise_defaults:
  enable_secret: MySecretPassw0rd
  domain: my.domain.com
  native_vlan: 99
```

An example of a **host** variable in a file called **ROUTER01.yml** in the **host vars** folder

```yaml
host_vlans:
  10:
    name: Blue_VLAN

  20:
    name: Red_VLAN
```

Jinja2

Templating language used by Ansible and Python

Use with Ansible Filter or Custom Filter

Delimiters:
{% ... %} for Statements

{{ ... }} for Expressions to print to the template output

{# ... #} for Comments not included in the template output

Syntax documentation:
http://jinja.pocoo.org/docs/2.10/templates/

Jinja2

Jinja templates will contain **logic**, such a **If** statements and **For loops**, and be simple or complex.

The variables in the templates reference **group, role,** and **host** data models that complete the template at playbook runtime

The output file generated by **Ansible *must*** match the ***exact*** syntax (CSV, markdown, **IOS / NXOS**, **JSON**, etc) of the file format being created

Jinja2 – A Sample Template – Host VLANS

Index:

Green text: Comments, not processed

Pink text: Logic

Light Blue text: **Variables** called from **data models**

White text: Hardcoded text

```
{# VLANs #}
{%  if host_vlans is defined %}
{%      for host_vlan in host_vlans|natural_sort() %}
vlan {{ host_vlan }}
 name {{ host_vlans[host_vlan].name }}
{%     endfor %}
{%  endif %}
```

Python

"**Python** is powerful... and fast; plays well with others; runs everywhere; is friendly & easy to learn; is Open."

Python's design philosophy emphasizes code readability

Python is **object-oriented**

You can write your own **filters** to enhance **Ansible's** base functionality but you do **NOT** need to know **Python** to get started with **Ansible**

A custom natural sort **filter** written in **Python** that can be called in **Ansible**

```
# natural_sort filter
#
from jinja2 import TemplateError

import re

class FilterModule(object):

  def natural_sort(self,l):
    convert = lambda text: int(text) if text.isdigit() else text.lower()
    alphanum_key = lambda key: [ convert(c) for c in re.split('([0-9]+)', str(key)) ]
    return sorted(l, key = alphanum_key)

  def filters(self):
    return {
      'natural_sort' : self.natural_sort,
    }
```

Referenced in **Jinja2 template**:

```
{# LOOPBACK INTERFACES #}
{%  for host_loopback_interface in host_loopback_interfaces|natural_sort() %}
```

HyperText Markup Language (HTML)

Most of us are familiar with *consuming* HTML, web pages, and browsers leaving creation of these pages to web developers

Jinja2 templates can be used to create interactive HTML pages based on your **data models** for quick and easy, and most importantly, *automated* documentation stored in your **Git** repository

So you might want to learn how to write your own HTML and create dynamic artifacts from your **intent-based source of truth**

Markdown

Markdown is a lightweight markup language with plain-text-formatting syntax

Think of **markdown** as **HTML-lite**

Despite it's simplicity extremely powerful, automated, easy to read documents can be created, again, from **Jinja templates** sourcing **data models** (do you see a trend here?)

RAW **markdown**

```
# ACCESS01

## VLANS

| VLAN | Name |
| ---- | ---- |
| 99 | NativeVLAN |
| 10 | Blue_VLAN |
| 20 | Red_VLAN |
```

Rendered **markdown**

ACCESS01

VLANS

VLAN	Name
99	NativeVLAN
10	Blue_VLAN
20	Red_VLAN

Comma-Separated Values (CSV)

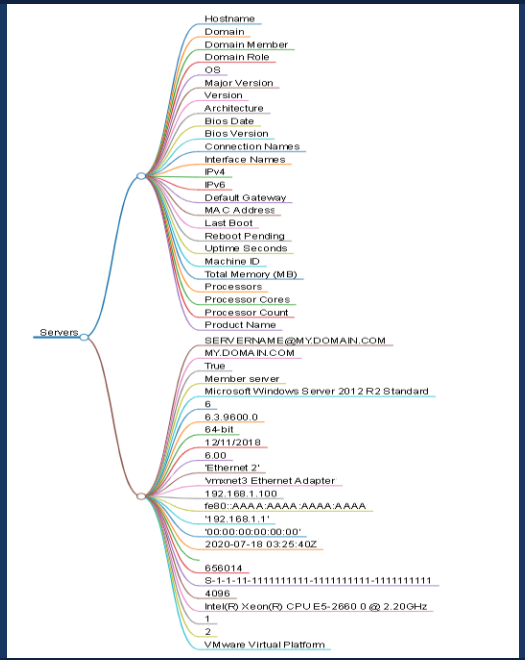
Like **HTML, markdown, JSON, IOS / NXOS config, XML;** powerful **comma-separated value** (**CSV**) files can be **templated** in **Jinja2**

These spreadsheets can be consumed upstream to create powerful artifacts in **Excel** or brought into systems like **Power BI** or other analytics

Mind Maps

A relatively new format, known as a **mind map**, can be automatically generated from a valid **markdown** file

A **mind map** is an **interactive HTML** page that creates keys and branches reflecting the data in a **CSV-like** format



RegEx

Regular Expression, often referred to as **RegEx**, is a sequence of characters that define a **search pattern**

Meme from 1997: Some people, when confronted with a problem, think "I know, I'll use **regular expressions**." ***Now they have two problems***

```
lineinfile:
  line: "{{distribution_cdp.stdout | map('regex_findall','(.*).my.domain.com.*\\n.*( ... .\\/.\\/.).*C65...') | first | unique | map('join') | list}}"
```

Genie Parser

Parsing data has become easier, and integrated with **Ansible**, with **Genie**

Genie returns **structured data** as opposed to **raw** command output

As the returned data is **JSON** you can work with it like **code** instead of **raw** output

Doc

```
Parser for
        * show arp
```

Schema

```
{
Any  (str) *: {
  'ipv4': {
    'neighbors': {
      Any  (str) *: {
        Optional  (str) ip: <class 'str'>,
        Optional  (str) prefix_length: <class 'str'>,
        'link_layer_address': <class 'str'>,
        'age': <class 'str'>,
        },
      },
    },
  },
}
```

pyATS

pyATS provides tests & verification automation

Ansible integrated to scale your testing

Works with **YAML testbed** files and **Python** tests

Provides **real-time** results ***from the perspective of the target host***

The Automation Progression Pyramid

Now that you have put together a modern toolkit – where and how do you start automating your network?

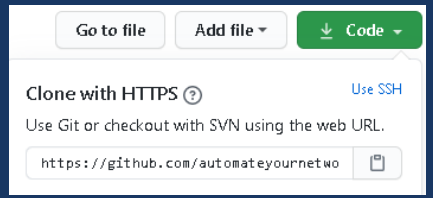
Personal reflection and more hard lessons learned


f5
CISCO
vmware
>_
API

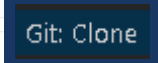
Git Repository and Cloning your Repo

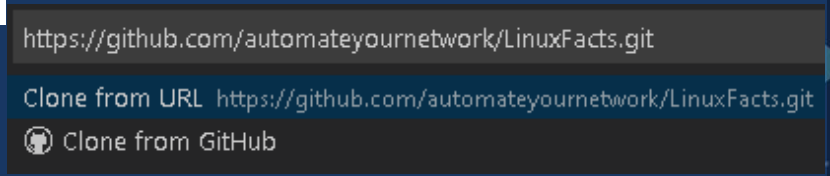
For this example we will use **GitHub**. Create an account and create a repository and initialize it with a README file

Copy the link to your repository



Git clone the repository in your development workstation (likely Windows 10) where your **VS Code** installation resides, create a folder (**C:repositories**) and save the repository here





Open the folder and begin to populate with your base folder structure

Base Folder structure

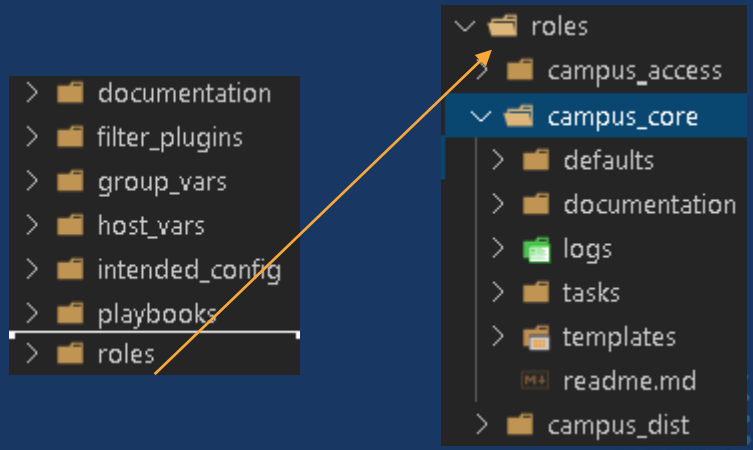
Now that we have all our tools it's time to take our first steps – a well organized based folder structure

Some folders are **required** by **Ansible** while other folders, such as folders to store output or results, are arbitrary and discretionary

Some folders you will need include:
Group vars, **host vars**, **roles**, **playbooks** and **filter plugins**

Your **roles** and **playbooks** folders should have a well organized sub-folder structure reflecting your logical environment

In the sample structure **documentation** is an arbitrary folder to store formatted output while **intended config** stores the RAW **IOS / NXOS / JSON** configs being pushed to devices



Build Hosts file, Group Vars, Host Vars

Next we will add our **hosts** file

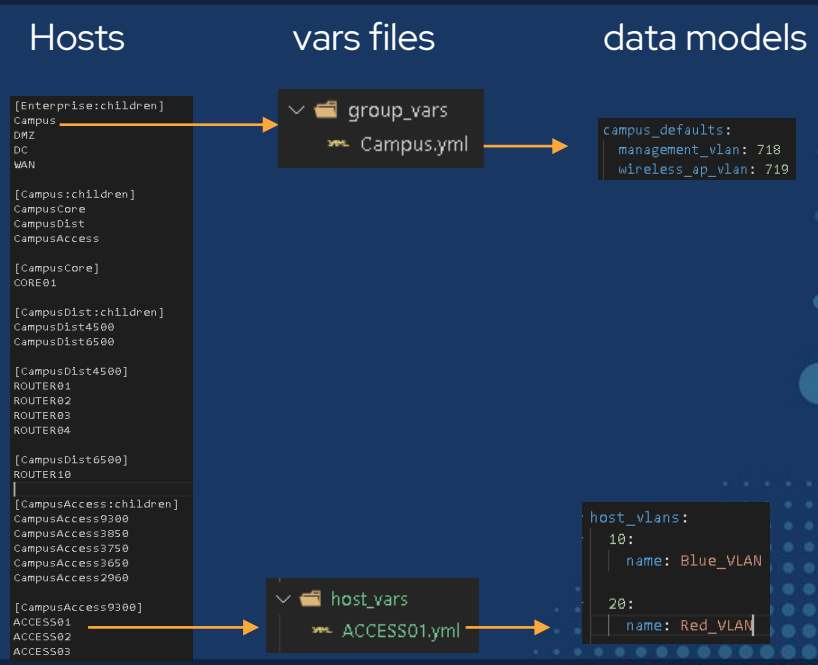
I recommend you spend some quality time crafting your hierarchy but if your network follows a Core / Distribution / Access topology you could start here

Remember your groups are in square brackets **[group]**

If your group has sub-groups you add the **children** key
[group:children]

This can nest down to an actual individual **host**

These groups and hosts are then reflected in group and host **vars** folders and **YAML** files where we keep our intent in the form of variable-based data models



- **Establish secure connectivity to hosts**

Establish Connectivity

Reminder that your **Linux** host running **Ansible** requires network connectivity to your target **hosts** as well as your **Git repository**

Work with your IT Security and Network Connectivity teams to establish an RBAC controlled, highly secure, well monitored and logged host in a secure zone in your data centre

You may need new credentials or service accounts to provide an authentication mechanism

Establish Connectivity

Ansible is **agent-less** and uses **SSH**, **WinRM**, or **REST APIs** to establish connectivity and the ability to **resolve** (**DNS**) the **hosts**

There are various ways to **authenticate** against a device:

Pass hard-coded credentials

Prompt for input at runtime

SSH keys

Here is an example of a connection string using a hard-coded username and a prompted password

This is located in the **group var** or **host var** file

```
---

# Network provider:
ansible_connection: network_cli
ansible_network_os: ios
ansible_user: ansible
ansible_ssh_pass: "{{hostvars[inventory_hostname]['ios_password']}}"
```

This is using the **network cli** connection against a Cisco **IOS** device

All devices in this **group** will use this mechanism to authenticate

On Authentication

If you choose hard coded authentication strings be aware these are stored in the **Git** repository

Ansible Vault can be used to encrypt your secrets so they are safe to store in your **Git** repo

You may need a service account

Prompted passwords are secure but do not lend themselves to full scheduled non-interactive automation

- Ansible **facts**
- Safe **show** commands


SYSLOG
SNMP
SHOW RUN
STATE OF MY NETWORK
SYSLOG
SNMP
ANSIBLE
SHOW RUN
STATE OF MY NETWORK
STATE OF MY NETWORK
ANSIBLE

Your first playbook

We will start with gathering **network state** information or **network configuration** information gathering.

This approach has some benefits:

It is **safe** – we are **not** configuring / changing / reloading / upgrading anything

It provides a **baseline** at **scale**

Begin building a dynamic library of **facts** and **documentation** about your network

Management not likely to baulk at this approach

How to run a playbook

Running an **Ansible playbook** is a simply command:

ansible-playbook <playbook.yml>

Capture Current State / Gather Information

Ansible has the ability to gather **facts** from targeted host(s).

Facts are useful, truthful, stateful variables with pieces of information about a host

The **setup** module is used against Linux and Windows hosts while specific ***operating system facts*** modules exist for specific platforms like IOS, NXOS, F5, Azure, or VMWare

All **facts** or subsets of **facts** can be specified to be gathered by the module

A sample **playbook** using **ios facts**

```
- hosts: CampusDist
# Prompt for Username and Password #
  vars_prompt:
    - name: IOS_Password_Prompt
      prompt: "Enter Windows Password"
      private: yes
  tasks:
# Register Password #
    - set_fact:
        ios_password: "{{ IOS_Password_Prompt }}"
      no_log: true
      delegate_to: localhost
      run_once: true
# Gather Ansible Facts About Host #
    - name: Gather Ansible Facts About Host
      ios_facts:
        gather_subset:
          - all
    - debug:
        msg: "{{ ansible_facts }}"
      You, 4 minutes ago • campus dist facts
# Create JSON file with Ansible Facts #
    - name: Create raw JSON file
      copy:
        content: |
          {% for host in ansible_play_hosts %}
          {{ ansible_facts | to_nice_json }}
          {% endfor %}
        dest: ../documentation/CAMPUS/DISTRIBUTION/distribution_facts.json
      delegate_to: localhost
      run_once: true

# Create YAML file with Ansible Facts #
    - name: Create raw YAML
      copy:
        content: |
          {% for host in ansible_play_hosts %}
          {{ ansible_facts | to_nice_yaml }}
          {% endfor %}
        dest: ../documentation/CAMPUS/DISTRIBUTION/distribution_facts.yml
      delegate_to: localhost
      run_once: true
```

Fact output

JSON


```
"net_hostname": "DISTRIBUTION01",
"net_image": "bootflash:cat4500e-universalk9.SPA.03.11.00.E.152-7.E.bin",
"net_interfaces": {
    "FastEthernet1": {
        "bandwidth": 10000,
        "description": null,
        "duplex": null,
        "ipv4": [],
        "lineprotocol": "down",
        "macaddress": "0000.0000.0000",
        "mediatype": null,
        "mtu": 1500,
        "operstatus": "down",
        "type": "RP management port"
    },
    "Port-channel1": {
        "bandwidth": 10000000,
        "description": "Core Uplink",
        "duplex": null,
        "ipv4": [
            {
                "address": "192.168.1.1",
                "subnet": "30"
            }
        ],
        "lineprotocol": null,
        "macaddress": "0001.0001.0001",
        "mediatype": "N/A",
        "mtu": 1500,
        "operstatus": "up",
        "type": "EtherChannel"
    },
    "Port-channel1.10": {
        "bandwidth": 10000000,
        "description": "Subinterface for VRF Blue",
        "duplex": null,
        "ipv4": [
            {
                "address": "192.168.2.1",
                "subnet": "30"
            }
```


YAML


```
net_gather_subset:
- hardware
- default
- interfaces
- config
net_hostname: DISTRIBUTION01
net_image: bootflash:cat4500e-universalk9.SPA.03.11.00.E.152-7.E.bin
net_interfaces:
    FastEthernet1:
        bandwidth: 10000
        description: null
        duplex: null
        ipv4: []
        lineprotocol: down
        macaddress: 0000.0000.0000
        mediatype: null
        mtu: 1500
        operstatus: down
        type: RP management port
    Port-channel1:
        bandwidth: 10000000
        description: Core Uplink
        duplex: null
        ipv4:
        -   address: 192.168.1.1
            subnet: '30'
        lineprotocol: null
        macaddress: 0001.0001.0001
        mediatype: N/A
        mtu: 1500
        operstatus: up
        type: EtherChannel
    Port-channel1.10:
        bandwidth: 10000000
        description: Subinterface for VRF Blue
```

Show commands

Modules like **ios command** and **nxos command** can be used to send privileged exec commands and send the output to text files

Any command can be sent so use your imagination but here are some ideas:

Show vlan sh ip arp sh mac address-table
sh ip ospf neighbor sh ip bgp summary sh ip int brief
sh int status sh int show spanning-tree
sh spanning-tree blocked dir sh run sh start

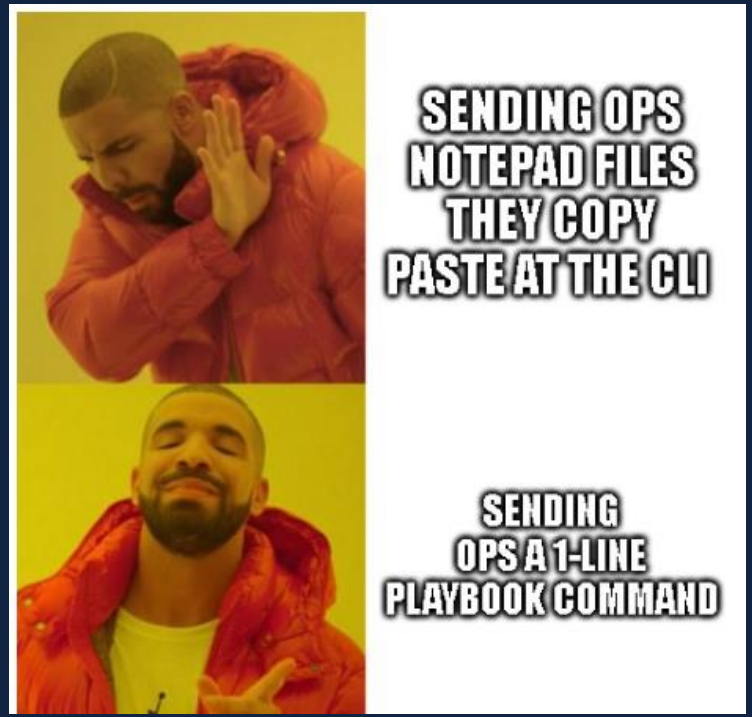
Show ip ospf neighbors

```
- name: run show ospf neighbors
  ios_command:
    commands: show ip ospf neighbor
```

Output:

```
##### DISTRIBUTION01 #####
-   - Neighbor ID     Pri   State           Dead Time   Address         Interface
    - 1.1.1.1           0   FULL/  -        00:00:32    192.168.1.2.1   Port-channel1
    - 1.1.1.30          0   FULL/  -        00:00:38    192.168.1.2.1   Port-channel1.30
    - 1.1.1.20          0   FULL/  -        00:00:32    192.168.1.2.1   Port-channel1.20
    - 1.1.1.10          0   FULL/  -        00:00:34    192.168.1.2.1   Port-channel1.10
    - 1.1.1.40          0   FULL/  -        00:00:34    192.168.1.2.1   Port-channel1.40

-   - Neighbor ID     Pri   State           Dead Time   Address         Interface
    - 1.1.1.1           0   FULL/  -        00:00:37    192.168.2.2.1   Port-channel1
    - 1.1.1.85          0   FULL/  -        00:00:36    192.168.2.2.1   Port-channel1.85
    - 1.1.1.30          0   FULL/  -        00:00:38    192.168.2.2.1   Port-channel1.30
    - 1.1.1.10          0   FULL/  -        00:00:35    192.168.2.2.1   Port-channel1.10
    - 1.1.1.40          0   FULL/  -        00:00:33    192.168.2.2.1   Port-channel1.40

-   - Neighbor ID     Pri   State           Dead Time   Address         Interface
    - 1.1.1.1           0   FULL/  -        00:00:31    192.168.3.2   Port-channel1
    - 1.1.1.10          0   FULL/  -        00:00:36    192.168.3.2   Port-channel1.10
    - 1.1.1.40          0   FULL/  -        00:00:35    192.168.3.2   Port-channel1.40
```

- Tactical changes
- 1 time changes


SENDING OPS NOTEPAD FILES THEY COPY PASTE AT THE CLI
SENDING OPS A 1-LINE PLAYBOOK COMMAND

Tactical / 1 Time Changes

Tactical (or 1 time) changes are playbooks with a specific set of orchestrated tasks to execute on one more many devices

Generally speaking:

Corporate standards
Compliancy jobs
New feature requests
Bug fixes / corrections
Point-in-time changes

Add / Remove / Set *
vlans, vrfs, ntp
routes, ACLs
“anything”

A sample **ios config** to clear and set the boot image variable on all devices at the Distribution layer

```
---

- hosts: CampusDist
  vars_prompt:
    - name: ios_password_prompt
      prompt: "Enter local admin password"
      private: yes

  tasks:
    - set_fact:
        ios_password: "{{ios_password_prompt}}"
      no_log: true
      delegate_to: localhost
      run_once: true

- name: SETTING BOOT IMAGE
  ios_config:
    lines:
      - no boot system
      - boot system flash bootflash:{{new_image}}
```

Tactical / 1 Time Changes

Lines are executed serially ‘as-is’

Syntax is important

Without **parents** the command is assumed to be issued at **config terminal** (conf t) level of a Cisco device

Parents can be used to issue the commands from a sub-configuration mode such as **interface**, **sub-interface**, **port-channel**, **RADIUS** servers, etc

Tasks can be orchestrated together serially in a pipelined automated chain

with items, an **Ansible loop** mechanism, can be used in concert with **parents** to modify multiple elements in a single **task**

An example of **parents** adding an IP and description to a specific interface:

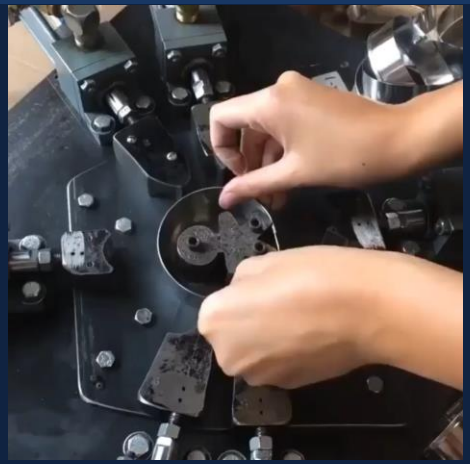
```
- name: configure interface settings
  ios_config:
    lines:
      - description test interface
      - ip address 172.31.1.1 255.255.255.0
    parents: interface Ethernet1
```

An example adding ip helpers to multiple interfaces using **with items**

```
- name: configure ip helpers on multiple interfaces
  ios_config:
    lines:
      - ip helper-address 172.26.1.10
      - ip helper-address 172.26.3.8
    parents: "{{ item }}"
  with_items:
    - interface Ethernet1
    - interface Ethernet2
    - interface GigabitEthernet1
```

- Full configuration management

- Data model: Your intended form
- Template: The hammers making the shape
- Raw metal: Your device
- Playbook: Machine in motion



Data Models

At the **group** level **variables** are shared by all downstream devices in the group and are common variables.

role level **variables** apply to all devices in that **role**

host level **variables** are **unique** and **specific** to that host

Enterprise (group) -> Campus(group) -> Campus Distribution (group) -> Campus Distribution Role (role) -> Campus Distribution Host (host)

Group Vars

Enterprise.yml
Ansible connection string, domain, native vlan, standard common VLANs, common VRFs, SNMP, syslog, IP helpers, RADIUS / TACACS+, NTP, AAA, 802.1x, ACLs

Campus.yml
Campus specific VLANs

DC.yml
DC specific VLANs

ACCESS01.yml
Device specific / host level: VLANs, vrfs, port-channels, interfaces, location, SNMP-ID, software

You can also “override” a group level variable

On Variables

Your **variables** will be **key-pair values** and **lists** inside your **YAML group**, **role**, or **host var** files

Tips: For readability and ease of locating ***where*** a variable can be found, I prefix my variables with the data model they are associated with

group_ **role_** **host_**

Have a **_defaults** section for common elements (management IP, location, corporate standards)

Templates

Templates are a skeletal outline of an intended configuration with placeholders in the from of **variables** that are filled in at **playbook** runtime using **logic**

The easiest way to **create** a template is to get a **valid**, **tested**, **working** configuration from the **platform** you want to template

*** You do not need to do full coverage when you begin ***

Scale vertically in a config (full coverage) or horizontally (VLANs across a whole role or platform)

Templates

For a network device I break down the **template** into 2 distinct sections: the **global config** and the **interface** configs

Global config + VRFs

```
{# GLOBAL CONFIG #}
no service pad
service tcp-keepalives-in
service tcp-keepalives-out
service timestamps debug datetime msec localtime
service timestamps log datetime msec localtime
service password-encryption
service compress-config
service counters max age 5
no service dhcp
hostname {{ inventory_hostname }}

{# VRFs #}
{% for host_vrf in host_vrfs|natural_sort() %}
{%     if host_vrf == "global" %}
{%     else %}
vrf definition {{ host_vrf }}
{%     endif %}
{%     if host_vrf == "global" %}
{%     else %}
{%         if host_vrf == "Transit" %}
 address-family ipv4
 exit-address-family
{%         else %}
 vnet tag {{ host_vrfs[host_vrf].tag }}
 address-family ipv4
 exit-address-family
{%         endif %}
{%     endif %}
{%     if host_vrfs[host_vrf].multicast is defined %}
ip multicast-routing vrf {{ host_vrf }}
{%     endif %}
{% endfor %}
```

An unused interface

```
{# HOST INTERFACES #}

{# UNUSED INTERFACES #}
{%  for host_interface in host_interfaces|natural_sort() %}
{%      if host_interfaces[host_interface].type is defined %}
{%          if host_interfaces[host_interface].type == "unused" %}
interface {{ host_interface }}
 description Unused
 switchport
 spanning-tree portfast edge
 switchport access vlan 99
 shutdown
 no cdp enable
{%          else %}
```

Automated Documentation

Templates are extremely **flexible** and **powerful** and you can re-use working logic that generates **intended configurations** you **push** to devices in their native syntax (IOS, NXOS, etc) and create **automated**, **source-of-truth**, **intent-based** documentation files

Then inside your **Git** repo you have **version controlled human readable** documentation illustrating your **intent** in **CSV**, **markdown**, **HTML**, **mind map**, **JSON**, **YAML**, or whatever filetype you need!

Templates

A markdown template

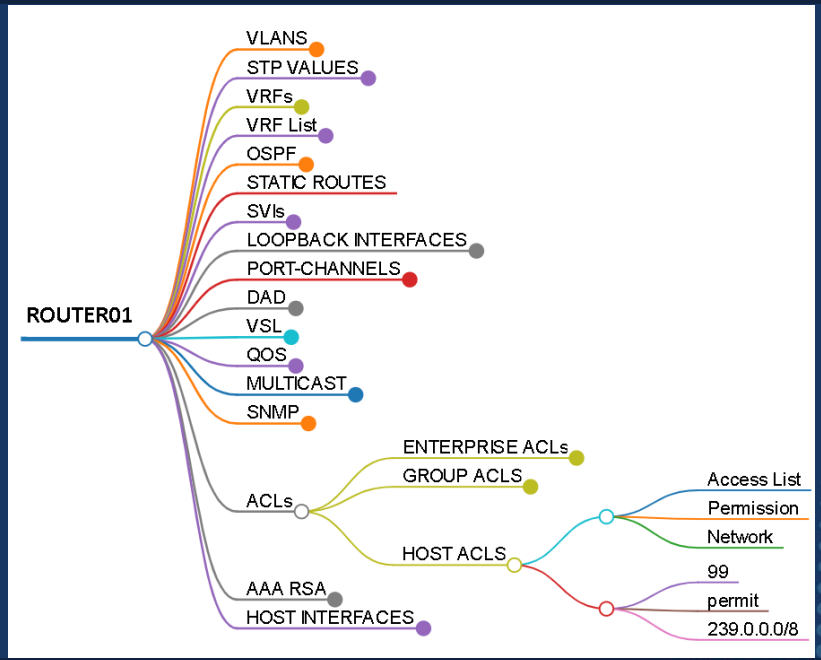
```
## {{ inventory_hostname }}

### Interfaces

* **LAN Interfaces**

| interface     | description  | ip address   |
| ------------- | ------------ | ------------ |
{% for lan_interface in lan_interfaces %}
| {{ lan_interface.interface }} | {{ lan_interface.description }} | {% if lan_inte
{% endfor %}
```

A CSV template

```
Interface,Host,Description,Mode,VLANs,Profile,Port-Channel,Speed
{% for interface in interfaces|natural_sort() %}
{{ interface }},{{ interfaces[interface].host }},{{ interfaces[interface].description }},
```

Templates

From the **markdown** we can generate a **mind map** an **interactive HTML** page



The goal: **idempotency**

When our **intended config** and the **running-config** are **idempotent**
Ansible will **only** push **changes**

- - check mode and see if your **intent** is **idempotent**

Does the **playbook** have changes to push ?
(playbook results displayed in **yellow** at runtime; change = true)

Or is it idempotent?
(results displayed in **green;** change = false)

The **BIG** picture

Once you achieve **full configuration idempotency** – every line, of every device, matching in your **Git intent-repository** and the **running-configuration** your **main Git branch** now represents your entire **source of truth** and **intent**

Now you can start to work using **Git branches** to represent all **changes to intent** – Moves, Adds, Changes, and Deletions – will all be **simple** changes to either the **data model** or **template**

Nobody is using the CLI anymore !

- State Validation

State Validation

Once you have achieved **idempotency** and **intent** for **configuration management**, arguably the “easy” part, we ***know*** the network is **configured** correctly –

But is it ***working*** properly based on our intent ?

Ansible and pyATS

We can go a step further with **pyATS** which is fully integrated with **Ansible** to perform network tests from the actual **host** on the network

This has incredible possibilities and power as you no longer are testing if a third party monitoring tool can reach your device or get SNMP traps or syslog

You take on the persona of the device and test from its CLI

Ansible and pyATS

```
+------------------------------------------------------------------------------+
|                            Starting section test                             |
+------------------------------------------------------------------------------+
+..............................................................................+
:            Starting STEP 1: Looking for ping failures ROUTER-1               :
+..............................................................................+
+..............................................................................+
:    Starting STEP 1.1: Checking Ping from ROUTER-1 to 208.67.222.222          :
+..............................................................................+
 Failed reason: Device ROUTER-1 had 0% success pinging 208.67.222.222
The result of STEP 1.1: Checking Ping from ROUTER-1 to 208.67.222.222 is => FAILED
+..............................................................................+
:       Starting STEP 1.2: Checking Ping from ROUTER-1 to 8.8.8.8              :
+..............................................................................+
 Failed reason: Device ROUTER-1 had 0% success pinging 8.8.8.8
The result of STEP 1.2: Checking Ping from ROUTER-1 to 8.8.8.8 is => FAILED
+..............................................................................+
:       Starting STEP 1.3: Checking Ping from ROUTER-1 to 1.1.1.1              :
+..............................................................................+
 Failed reason: Device ROUTER-1 had 0% success pinging 1.1.1.1
The result of STEP 1.3: Checking Ping from ROUTER-1 to 1.1.1.1 is => FAILED
The result of STEP 1: Looking for ping failures ROUTER-1 is => FAILED
+..............................................................................+
:          Starting STEP 2: Looking for ping failures ROUTER-1                 :
+..............................................................................+
+..............................................................................+
:    Starting STEP 2.1: Checking Ping from ROUTER-1 to 208.67.222.222          :
+..............................................................................+
 Failed reason: Device ROUTER-1 had 0% success pinging 208.67.222.222
The result of STEP 2.1: Checking Ping from ROUTER-1 to 208.67.222.222 is => FAILED
+..............................................................................+
:       Starting STEP 2.2: Checking Ping from ROUTER-1 to 8.8.8.8              :
+..............................................................................+
 Failed reason: Device ROUTER-1 had 0% success pinging 8.8.8.8
The result of STEP 2.2: Checking Ping from ROUTER-1 to 8.8.8.8 is => FAILED
+..............................................................................+
:       Starting STEP 2.3: Checking Ping from ROUTER-1 to 1.1.1.1              :
+..............................................................................+
 Failed reason: Device ROUTER-1 had 0% success pinging 1.1.1.1
The result of STEP 2.3: Checking Ping from ROUTER-1 to 1.1.1.1 is => FAILED
The result of STEP 2: Looking for ping failures ROUTER-1 is => FAILED
+------------------------------------------------------------+
|                        STEPS Report                        |
+------------------------------------------------------------+
STEP 1 - Looking for ping failures ROUTER-1     Failed
STEP 1.1 - Checking Ping from ROUTER-1 to 208.67.222.222    Failed
STEP 1.2 - Checking Ping from ROUTER-1 to 8.8.8.8    Failed
STEP 1.3 - Checking Ping from ROUTER-1 to 1.1.1.1    Failed
STEP 2 - Looking for ping failures ROUTER-1     Failed
STEP 2.1 - Checking Ping from ROUTER-1 to 208.67.222.222    Failed
STEP 2.2 - Checking Ping from ROUTER-1 to 8.8.8.8    Failed
```

- Continuous Integration
- Continuous Delivery

Continuous Integration / Continuous Delivery (CI/CD)

The ultimate destination of our automation journey is a complete **automation pipeline**

Continuously integration – ongoing pull requests

Continuously delivery - automated builds and releases to the network

We have to use our imaginations but it is possibly to create this **pipeline**

A Git branch is created
Clone locally; development occurs
Tested and approved to be Pulled into main
The successful Pull request triggers an automated pipeline
Generate intent and documents
Build Docker container image
Test the solution against a lab or virtual (vIRL) environment
Capture network state
Push to environment
Validate network state
Destroy Docker container image
Close or rollback change

In summary…


CI/CD
State Validation
Full Configuration Management
Tactical Changes 1 Time Playbooks
Capture Network State Gather Information
Establish Connectivity
Build Toolkit

Thank you

John Capobianco can be reached on Twitter, GitHub, and LinkedIn and would love to connect with you.

twitter.com/John_Capobianco

linkedin.com/in/john-capobianco-644a1515

github.com/automateyournetwork

AnsibleFest

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

youtube.com/user/RedHatVideos

linkedin.com/company/Red-Hat

facebook.com/ansibleautomation

twitter.com/ansible