

CCIE DevNet

Technology Reference Guide

Nagaraj Ravinuthala

DevNet Expert

<https://in.linkedin.com/in/nagaraj-ravinuthala>

Section 1.0: Setting up and Getting Started-----	4
What would you need?-----	4
Section 1.1 - Terraform - Docker and ACI-----	5
What is Terraform-----	5
Installation-----	5
How does Terraform work?-----	5
Terraform workflow-----	5
Resource Blocks-----	6
Resource Syntax-----	6
Providers-----	6
Terraform Docker Config file example-----	7
Input Variables-----	10
Output Variables-----	17
Local Variables-----	19
Section 1.2 - YANG, Cisco NSO (Network Services Orchestrator)-----	21
What is Cisco NSO?-----	21
How does Cisco NSO Work?-----	21
Install Cisco NSO-----	21
Getting Started-----	23
Creating Services-----	26
Modifying a service-----	33
YANG-----	35
YANG Structure-----	37
YANG Data Types-----	40
Refer-----	41
Section 2.0: Python Programming Crash Course-----	42
What is Program?-----	42
What is Python?-----	42
Why Python?-----	42
Python Installation-----	42
Getting started-----	43
Data Types-----	43
Variables-----	44
Operators-----	45
Getting Input from the user-----	50

Data Types - Deep Dive-----	50
Conditional Statements-----	53
Loops-----	55
Functions-----	56
Libraries-----	59
Working with files-----	61
Python Object Oriented Programming-----	71
Working with REST APIs-----	73
Create REST API-----	76
Paramiko-----	80
Netmiko-----	82
Click CLI-----	86
Section 2.1: Python NCCLIENT, YANG, NETCONF, RESTCONF-----	90
NETCONF-----	90
Section 2.2: Python Libraries Jinja2, Scrapli-----	96
Jinja2-----	96
Scrapli-----	101
Section 3.1: Git, Gitlab, CI/CD Pipelines-----	103
Git-----	103
Gitlab-----	107
CI/CD Pipelines-----	108
Sample Pipeline 1-----	110
Sample Pipeline 2-----	111
Section 3.2: Network Test Automation with pyATS-----	113
How does pyATS work?-----	113
Section 3.3: Secret Management with Vault-----	118
Secret Management with HashicorpVault-----	118
Step 1: Download Vault-----	118
Step 2: Move Vault binary to /usr/local/bin-----	118
Step 3: Verify the Installation-----	118
Check that Vault is installed correctly by running:-----	118
Step 4: Create a Vault Configuration File-----	118
Create a configuration file for Vault, e.g., config.hcl:-----	118
Step 5: Start Vault in Dev Mode (for testing)-----	119
For testing purposes, you can start Vault in development mode. In a real production environment, you would configure Vault differently.-----	119
Once server starts successfully, you should see output as shown below:-----	119
Step 6: Access vault UI-----	119

Section 3.4: Ansible Crash Course	124
Introduction	124
How Ansible Works?	124
Ansible Terminology	124
Ansible with Network Devices	125
Ansible Installation	126
Ansible Getting Started	126
Host vars and Group vars	128
Ad hoc Commands	129
Playbooks	132
Playbook to run show commands	133
Playbook to run config commands	134
Ansible Playbook to read secrets from Hashicorp Vault	134
Section 4.1: Docker, Docker Compose	135
Docker	135
Docker Compose	137
Section 4.1: Zammad Python SDK	138

Section 1.0: Setting up and Getting Started

What would you need?

- DevNet VM Image with the following pieces set up
 - Python
 - Ansible
 - Docker
 - Terraform
- VMWare Workstation Player
- Code Editor/ IDE (Integrated Development Environment)
 - VS Code (Recommended)
 - Pycharm
 - Any other editor of your choice
- Download Links
 - VM Image
 - Octa Image:
https://drive.google.com/file/d/1a_xfj2wl3Ezbi0ElstYebPxpUC5cMulO/view
 - Username: student
 - Password: devnetstudent
 - Cisco Image
https://learningcontent.cisco.com/images/2022-04-09_DevNetExpert_CWS_Example.ova
 - If you download the Cisco image, what you get is a zip file, which you need to unzip into some folder and then open the OVF file with any virtualization software like VMWare Workstation
 - VMWare Player
 - <https://www.vmware.com/go/getplayer-win>
 - If clicking the link does not take you to the download page, copy the link and paste it in the browser
 - Visual Studio Code (VS Code)
 - <https://code.visualstudio.com/download>

Section 1.1 - Terraform - Docker and ACI

What is Terraform

- Terraform is an infrastructure as code tool
- It lets you build, change, and version infrastructure, both on cloud and on-prem, safely and efficiently
- Resources are defined in human-readable configuration files that can be versioned, reused, and shared
- Can manage both
 - Low level resources like compute, storage and networking
 - High level resources like DNS entries, SaaS features etc.

Installation

- If you have downloaded the DevNet VMImage mentioned above, Terraform is already installed.
- If you are using any other system, you can follow the below instructions to install Terraform.
 - <https://developer.hashicorp.com/terraform/downloads>

How does Terraform work?

- Providers enable Terraform to work with virtually any platform or service with an accessible API.
- HashiCorp and the Terraform community have already written **thousands of providers** to manage many different types of resources and services
- You can find all publicly available providers on the [Terraform Registry](#)
 - Explore the following providers on Terraform Registry
 - Docker
 - Cisco ACI
 - Azure
 - AWS

Terraform workflow

- Write
 - Create a config file in HCL (Hashicorp Config Language) which defines the infrastructure in a declarative way
- Plan
 - Terraform creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration.

- Apply
 - On approval, Terraform performs the proposed operations in the correct order, respecting any resource dependencies

Resource Blocks

- Resources are the most important element in the Terraform language
- Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records

Resource Syntax

- A resource block declares a resource of a given type ("`docker_container`") with a given local name ("`nginx_container`")
- The local name can only be used to refer this resource from elsewhere in the same module but has no significance outside

Providers

- Each resource type is implemented by a provider, which is a terraform plugin
- Providers are distributed separately from Terraform itself
- Terraform can automatically install most providers when initializing a working directory
- In order to manage resources, a Terraform module must specify which providers it requires
- Terraform usually automatically determines which provider to use based on a resource type's name
- By convention, resource type names start with their provider's preferred local name
- Every Terraform provider has its own documentation, describing its resource types and their arguments
- Most publicly available providers are distributed on the [Terraform Registry](#), which also hosts their documentation

Terraform Docker Config file example

```
terraform {
  required_providers {
    docker = {
      source = "kreuzwerker/docker"
      version = "~> 3.0.2"
    }
  }
}

provider "docker" {}

resource "docker_image" "nginx_image" {
  name          = "<image_name>:<image_version>"
  keep_locally = false
}

resource "docker_container" "nginx_container" {
  image = docker_image.nginx_image.image_id
  name   = "tutorial"
  ports {
    internal = 80
    external = 8000
  }
}

# initialize terraform
$ terraform init
```

If everything went well, you should see the following output.

```
Activities  Terminal ▾  Nov 12 13:14
expert@expert-cws: ~/temp/aci_terraform/HandsOn1

(main) expert@expert-cws:~/temp/aci_terraform/HandsOn1$ terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of kreuzwerker/docker from the dependency lock file
- Using previously-installed kreuzwerker/docker v3.0.2

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
(main) expert@expert-cws:~/temp/aci_terraform/HandsOn1$
```

check what is terraform going to create by generating a plan
\$ terraform plan

If everything went well, you should see the following output.

```
expert@expert-cws: ~/temp/aci_terraform/HandsOn1

(main) expert@expert-cws:~/temp/aci_terraform/HandsOn1$ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# docker_container.nginx_container will be created
+ resource "docker_container" "nginx_container" {
  + attach                = false
  + bridge                = (known after apply)
  + command               = (known after apply)
  + container_logs        = (known after apply)
  + container_read_refresh_timeout_milliseconds = 15000
  + entrypoint            = (known after apply)
  + env                   = (known after apply)
  + exit_code             = (known after apply)
  + hostname              = (known after apply)
  + id                    = (known after apply)
  + image                 = (known after apply)
  + init                  = (known after apply)
  + ipc_mode              = (known after apply)
  + log_driver            = (known after apply)
  + logs                  = false
  + must_run              = true
}
```

```

+ ports {
+   external = 8000
+   internal = 80
+   ip       = "0.0.0.0"
+   protocol = "tcp"
+ }
}

# docker_image.nginx_image will be created
+ resource "docker_image" "nginx_image" {
+   id           = (known after apply)
+   image_id     = (known after apply)
+   keep_locally = false
+   name         = "nginx:latest"
+   repo_digest  = (known after apply)
+ }

```

Plan: 2 to add, 0 to change, 0 to destroy.

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

```

# apply the plan to create resources
$ terraform apply

```

If everything went well, you should see the plan again, with a confirmation prompt this time. Say yes to continue.

```

}

# docker_image.nginx_image will be created
+ resource "docker_image" "nginx_image" {
+   id           = (known after apply)
+   image_id     = (known after apply)
+   keep_locally = false
+   name         = "nginx:latest"
+   repo_digest  = (known after apply)
+ }

```

Plan: 2 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

Plan: 2 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

```

docker_image.nginx_image: Creating...
docker_image.nginx_image: Still creating... [10s elapsed]
docker_image.nginx_image: Creation complete after 18s [id=sha256:c20060033e06f882b0f8e2db7d974d72e0887a3be5e554efdb0dcf8d53512647nginx:latest]
docker_container.nginx_container: Creating...
docker_container.nginx_container: Creation complete after 4s [id=862ef81a0df3c69f109e84a70f75acde04877aee49d44eee78092e7ce19813a6]

```

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
(main) expert@expert-cws:~/temp/aci_terraform/HandsOn1\$

```

# destroy the resources created by terraform
$ terraform destroy

```

```

Plan: 0 to add, 0 to change, 2 to destroy.

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes

docker_container.nginx_container: Destroying... [id=862ef81a0df3c69f109e84a70f75acde04877aee49d44eee78092e7ce19813a6]
docker_container.nginx_container: Destruction complete after 0s
docker_image.nginx_image: Destroying... [id=sha256:c20060033e06f882b0fbe2db7d974d72e0887a3be5e554efdb0dcf8d53512647nginx:latest]
docker_image.nginx_image: Destruction complete after 0s

Destroy complete! Resources: 2 destroyed.
(main) expert@expert-cws:~/temp/aci_terraform/HandsOn1$

```

Hands-On 1 - Provision Docker Container using Terraform

- Create an nginx docker container using terraform
- Expose port 80 on the container via port 8000 on host
- Ensure that default page comes up

Hands-On 1 Solution

- Refer the Terraform Docker example given above

If all went well, you should see the following output, in the browser.



Input Variables

- Variables let us customize aspects of Terraform modules without altering the module's source code
- This functionality allows you to share modules across different Terraform configurations
- If you're familiar with traditional programming languages, it can be useful to compare Terraform modules to function definitions
 - Input variables are like function arguments
 - Output values are like function return values
 - Local values are like a function's temporary local variables
- Each input variable accepted by a module must be declared using a **variable** block
- The label after the **variable** keyword is a name for the variable, which must be unique among all variables in the same module

Hands-On 2 - Parametrize Terraform config script using variables

- Using Cisco ACI provider, create an ACI tenant and an application profile
- Names of the tenant and the application profile should be taken from variables file set as default values

Hands-On 2 Solution

Contents of main.tf

```
terraform {
  required_providers {
    aci = {
      source  = "CiscoDevNet/aci"
      version = "0.7"
    }
  }
}

provider "aci" {
  # username has been set declared in variables file
  username = var.USER

  # password has been set declared in variables file
  password = var.PASS

  insecure = true

  # aci url has been set declared in variables file
  url      = var.aciUrl
}

resource "aci_tenant" "tenant1" {
  # tenant name has been set declared in variables file
  name = var.tenantName
}

resource "aci_application_profile" "appl" {
  tenant_dn = aci_tenant.tenant1.id

  # appname has been set declared in variables file
```

```
name      = var.appName
}
```

Contents of variables.tf

```
variable "aciUrl" {
  type = string
  default = "https://<ip address>/"
}

variable "tenantName" {
  type = string
  default = "tenant1"
}

variable "appName" {
  type = string
  default = "app1"
}

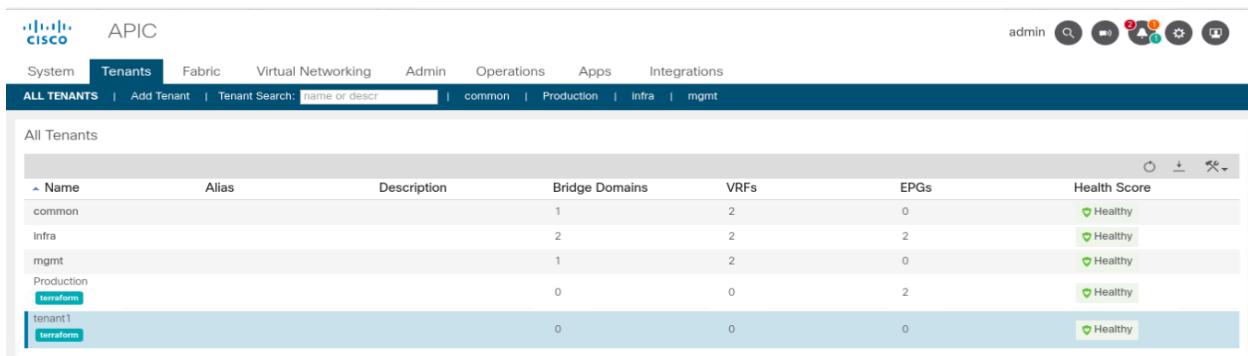
variable "USER" {
  type = string
}

variable "PASS" {
  type = string
}
```

Username and password have been set as environment variables as a best practice.

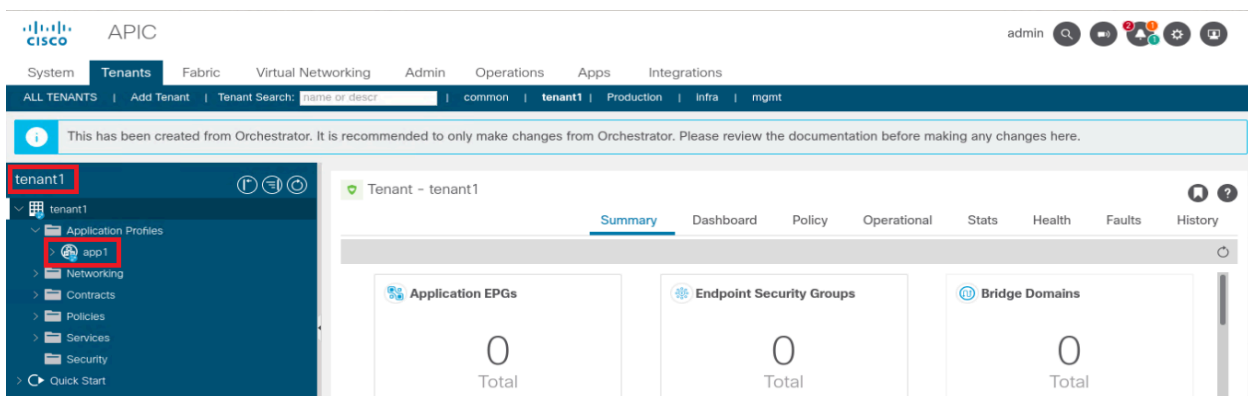
```
export TF_VAR_USER="admin"
export TF_VAR_PASS="Admin@1234"
```

On successful execution of `terraform apply`, you should see the following output in ACI.



The screenshot shows the Cisco APIC interface with the 'Tenants' tab selected. A table lists all tenants with columns for Name, Alias, Description, Bridge Domains, VRFs, EPGs, and Health Score. The tenants listed are common, infra, mgmt, Production, and tenant1. The 'tenant1' row is highlighted in blue.

Name	Alias	Description	Bridge Domains	VRFs	EPGs	Health Score
common			1	2	0	Healthy
infra			2	2	2	Healthy
mgmt			1	2	0	Healthy
Production			0	0	2	Healthy
tenant1			0	0	0	Healthy



The screenshot shows the Cisco APIC interface with the 'Tenants' tab selected. The 'tenant1' tenant is selected, and the 'Summary' tab is active. The configuration for 'tenant1' is displayed, showing a message that it was created from Orchestrator. The configuration includes Application EPGs, Endpoint Security Groups, and Bridge Domains, all with a total count of 0.

Message: This has been created from Orchestrator. It is recommended to only make changes from Orchestrator. Please review the documentation before making any changes here.

Tenant - tenant1

Summary | Dashboard | Policy | Operational | Stats | Health | Faults | History

Application EPGs: 0 Total

Endpoint Security Groups: 0 Total

Bridge Domains: 0 Total

Hands-On 3 - Create multiple ACI tenants using count meta-argument

- Declare a list variable with a list of tenant names
- Using for each construct, create multiple tenants with each name given in the list variable

Hands-On 3 Solution

Contents of main.tf

```

terraform {
  required_providers {
    aci = {
      source = "CiscoDevNet/aci"
      version = "0.7"
    }
  }
}

provider "aci" {
  # username has been set declared in variables file

```

```
username = var.USER

# password has been set declared in variables file
password = var.PASS

insecure = true

# aci url has been set declared in variables file
url      = var.aciUrl
}

resource "aci_tenant" "tenant1" {
  # tenant name has been set declared in variables file
  for_each = toset(var.tenants)
  name     = each.key
}
```

Contents of variables.tf

```
variable tenants {
  type = list
  default = ["tenant1", "tenant2", "tenant3"]
}

variable "USER" {
  type = string
}

variable "PASS" {
  type = string
}

variable "aciUrl" {
  type = string
  default = "https://<ip_address>/"
}
```

Username and password have been set as environment variables as a best practice.

```
export TF_VAR_USER="admin"
export TF_VAR_PASS="Admin@1234"
```

check what is terraform going to create by generating a plan
\$ terraform plan

Followed by

\$ terraform apply

If everything went well, you should see the following output.

```
(main) expert@expert-cws:~/temp/aci_terraform/HandsOn3$ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# aci_tenant.tenant1["tenant1"] will be created
+ resource "aci_tenant" "tenant1" {
  + annotation = "orchestrator:terraform"
  + description = (known after apply)
  + id         = (known after apply)
  + name       = "tenant1"
  + name_alias = (known after apply)
}

# aci_tenant.tenant1["tenant2"] will be created
+ resource "aci_tenant" "tenant1" {
  + annotation = "orchestrator:terraform"
  + description = (known after apply)
  + id         = (known after apply)
  + name       = "tenant2"
  + name_alias = (known after apply)
}

# aci_tenant.tenant1["tenant3"] will be created
+ resource "aci_tenant" "tenant1" {
  + annotation = "orchestrator:terraform"
  + description = (known after apply)
  + id         = (known after apply)
  + name       = "tenant3"
  + name_alias = (known after apply)
}

Plan: 3 to add, 0 to change, 0 to destroy.
```

Activate Windows
Go to Settings to activate Windows.

```
Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.
(main) expert@expert-cws:~/temp/aci_terraform/HandsOn3$
```



```
Plan: 3 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aci_tenant.tenant1["tenant1"]: Creating...
aci_tenant.tenant1["tenant2"]: Creating...
aci_tenant.tenant1["tenant3"]: Creating...
aci_tenant.tenant1["tenant1"]: Creation complete after 2s [id=uni/tn-tenant1]
aci_tenant.tenant1["tenant2"]: Creation complete after 2s [id=uni/tn-tenant2]
aci_tenant.tenant1["tenant3"]: Creation complete after 2s [id=uni/tn-tenant3]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
(main) expert@expert-cws:~/temp/aci_terraform/HandsOn3$
```

Name	Alias	Description	Bridge Domains	VRFs	EPGs	Health Score
common			1	2	0	Healthy
Infra			2	2	2	Healthy
mgmt			1	2	0	Healthy
Production			0	0	2	Healthy
tenant1			0	0	0	Healthy
tenant2			0	0	0	Healthy
tenant3			0	0	0	Healthy

Clean up the resources created using terraform with the command

```
$ terraform apply
```

If everything went well, you should see the following output.

```
Plan: 0 to add, 0 to change, 3 to destroy.
```

Do you really want to destroy all resources?

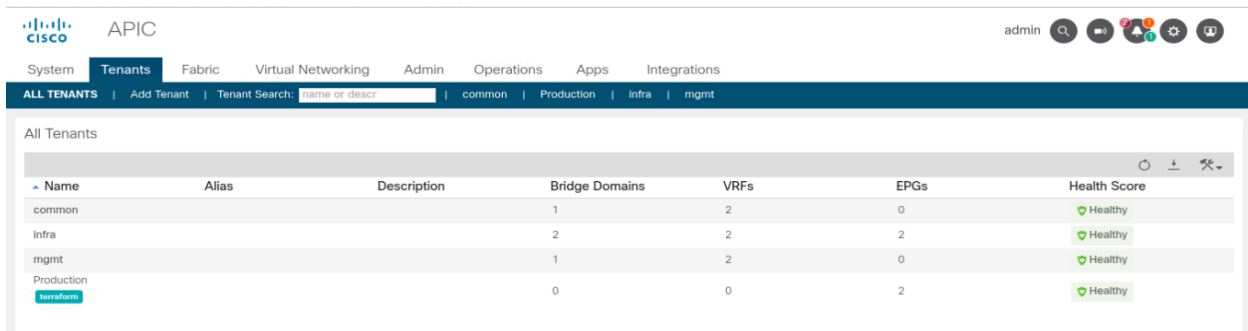
Terraform will destroy all your managed infrastructure, as shown above.
There is no undo. Only 'yes' will be accepted to confirm.

```
Enter a value: yes
```

```
aci_tenant.tenant1["tenant2"]: Destroying... [id=uni/tn-tenant2]
aci_tenant.tenant1["tenant1"]: Destroying... [id=uni/tn-tenant1]
aci_tenant.tenant1["tenant3"]: Destroying... [id=uni/tn-tenant3]
aci_tenant.tenant1["tenant3"]: Destruction complete after 0s
aci_tenant.tenant1["tenant2"]: Destruction complete after 0s
aci_tenant.tenant1["tenant1"]: Destruction complete after 1s
```

```
Destroy complete! Resources: 3 destroyed.
```

```
(main) expert@expert-cws:~/temp/aci_terraform/HandsOn3$
```



Name	Alias	Description	Bridge Domains	VRFs	EPGs	Health Score
common			1	2	0	Healthy
infra			2	2	2	Healthy
mgmt			1	2	0	Healthy
Production			0	0	2	Healthy

Output Variables

- Output variables in Terraform are used to extract and display information from your Terraform resources after a configuration has been applied.
- They can be used to:
 - Display essential resource details on the command line after terraform apply.
 - Pass values to external systems or modules.
- Lets take the example of pulling a docker image and creating a container.
- We can store information like image id, container id, container name etc. in output variables, as shown below.

```
# Specify the Docker provider
provider "docker" {
  host = "unix:///var/run/docker.sock" # Adjust if Docker uses a different host
}

# Pull a Docker image resource
resource "docker_image" "nginx_image" {
  name          = "nginx:latest"
  keep_locally = false
}

# Create a Docker container resource
resource "docker_container" "nginx_container" {
  name = "nginx_server"
  image = docker_image.nginx_image.latest
}
```

```

ports {
  internal = 80
  external = 8080
}

# Define output variables to extract information
output "container_id" {
  description = "The unique ID of the Docker container"
  value       = docker_container.nginx_container.id
}

output "container_name" {
  description = "The name of the Docker container"
  value       = docker_container.nginx_container.name
}

output "container_ports" {
  description = "Details about the container's port mappings"
  value       = docker_container.nginx_container.ports
}

output "image_id" {
  description = "The ID of the Docker image"
  value       = docker_image.nginx_image.id
}

```

- To see the output again we can use the command:
 - terraform output
- Or we can access specific variables using:
 - terraform output <output_variable_name>

Local Variables

- Local variables in Terraform are used to simplify and organize your configuration by allowing you to define reusable values.

- Instead of repeating the same values or calculations throughout your configuration, you can assign them to local variables and refer to those variables as needed.
- The following code shows usage of local variables

```
# Specify the Docker provider
provider "docker" {
    host = "unix:///var/run/docker.sock" # Adjust if Docker uses a different host
}

# Define local variables
locals {
    image_name      = "nginx:latest"      # Docker image name
    container_name  = "nginx_server"      # Container name
    internal_port   = 80                  # Internal container port
    external_port   = 8080                # External host port
    keep_image_local = false              # Whether to keep the image locally
}

# Pull a Docker image using local variables
resource "docker_image" "nginx_image" {
    name      = local.image_name
    keep_locally = local.keep_image_local
}

# Create a Docker container using local variables
resource "docker_container" "nginx_container" {
    name = local.container_name
    image = docker_image.nginx_image.latest
    ports {
        internal = local.internal_port
        external = local.external_port
    }
}
```

```
}  
  
# Define output variables for useful information  
output "container_id" {  
    value = docker_container.nginx_container.id  
}  
  
output "container_ports" {  
    value = docker_container.nginx_container.ports  
}  
  
output "image_used" {  
    value = local.image_name  
}
```

Section 1.2 - YANG, Cisco NSO (Network Services Orchestrator)

What is Cisco NSO?

- A Linux application which orchestrates the configuration life cycle of physical and virtual network devices
- NSO gathers, parses, and stores the configuration state of the network devices it manages in a configuration database (CDB)
- Users and applications can then "ask" NSO to create, read, update, or delete configuration in a programmatic way either with ad hoc commands or through customizable network services

How does Cisco NSO Work?

- NSO uses software packages called Network Element Drivers (NEDs)
- Using NEDs NSO facilitates telnet, SSH, or API interactions with the devices that it manages
- NED provides an abstraction layer that reads in the device's running configuration and parses it into a data-model-validated snapshot in the CDB
- NEDs can also do the reverse
 - creating network configuration from CDB data inputs
 - sending the configurations to the network devices

Install Cisco NSO

- Refer <https://developer.cisco.com/docs/nso/#!getting-and-installing-nso/installation> for detailed instructions on downloading and installing Cisco Network Services Orchestrator
- We will just do the local install of Cisco NSO. If you need to do the system install, please refer to the instructions given in the link above.
- Prerequisites
 - Java
 - `apt install default-jdk` (already installed on CWS)
 - Apache ANT
 - `apt install ant` (already installed on CWS)
 - Python
 - `apt install python3 python3-pip python3-setuptools` (already installed on CWS)
 - Development Tools
 - `apt install libxml2-utils` (already installed on CWS)

- Download NSO suitable for your OS
 - <https://software.cisco.com/download/home/286331591/type/286283941/release/6.0>
 - Locate the installer for your OS (e.g.: Cisco Network Services Orchestrator Linux Installer) and click download button on the right
 - Once downloaded,
 - `cd /home/expert/Downloads`
 - `sh nso-6.0-freetrial.linux.x86_64.signed.bin`
 - Files are unpacked
 - Locate your installer
 - E.g. `nso-6.0.linux.x86_64.installer.bin`
 - Use the following command to get help with the installation
 - `sh nso-6.0.linux.x86_64.installer.bin --help`
 - Use the following command to perform the local installation (recommended for installing NSO for development purpose on laptops)
 - `nso-6.0.linux.x86_64.installer.bin --local-install <LocalInstallDir>`

```
(main) expert@devnet-lab:~/Downloads$ sh nso-6.0.linux.x86_64.installer.bin --local-install ~/nso-6.0
INFO Using temporary directory /tmp/ncs_installer.4998 to stage NCS installation bundle
INFO Unpacked ncs-6.0 in /home/expert/nso-6.0
INFO Found and unpacked corresponding DOCUMENTATION_PACKAGE
INFO Found and unpacked corresponding EXAMPLE_PACKAGE
INFO Found and unpacked corresponding JAVA_PACKAGE
INFO Generating default SSH hostkey (this may take some time)
INFO SSH hostkey generated
INFO Generating self-signed certificates for HTTPS
INFO Environment set-up generated in /home/expert/nso-6.0/ncsrc
INFO NSO installation script finished
INFO Found and unpacked corresponding NETSIM_PACKAGE
INFO NCS installation complete
```

- Ensure that installation is successful as shown in the above screenshot
- Look at the contents of the `nso/packages/neds` directory:

```
(main) expert@devnet-lab:~/nso-6.0$ ls -l packages/neds/
total 40
drwxr-xr-x 8 expert expert 4096 Oct 3 2022 a10-acos-cli-3.0
drwxr-xr-x 7 expert expert 4096 Oct 3 2022 alu-sr-cli-3.4
drwxr-xr-x 8 expert expert 4096 Oct 3 2022 cisco-asa-cli-6.6
drwxr-xr-x 7 expert expert 4096 Oct 3 2022 cisco-ios-cli-3.0
drwxr-xr-x 7 expert expert 4096 Oct 3 2022 cisco-ios-cli-3.8
drwxr-xr-x 8 expert expert 4096 Oct 3 2022 cisco-iosxr-cli-3.0
drwxr-xr-x 8 expert expert 4096 Oct 3 2022 cisco-iosxr-cli-3.5
drwxr-xr-x 8 expert expert 4096 Oct 3 2022 cisco-nx-cli-3.0
drwxr-xr-x 8 expert expert 4096 Oct 3 2022 dell-ftos-cli-3.0
drwxr-xr-x 5 expert expert 4096 Oct 3 2022 juniper-junos-nc-3.0
(main) expert@devnet-lab:~/nso-6.0$ |
```

- Go to NSO installation folder and execute the following command to add NSO executable to the PATH
- Alternately you can also add it to the end of bash profile so that NSO executables are available on system boot up
 - `cd nso-6.0`
 - `source ncsrc`
- Now we are ready to start using NSO commands that start with its earlier name **ncs**
 - `ncs-setup --help`
- If we don't see a command not found error, we are good

Getting Started

- NSO needs a Deployment/Runtime Directory where the database files, logs, etc. are stored.
- An empty default directory can be created using the `ncs-setup` command.
- This helps in keeping the installation separate from the runtime
- This also enables us to create a project specific nso setup to keep the list of devices and their corresponding configuration separate
- Go to any directory created for working on a project, preferably outside the nso installation directory and set it up as NSO runtime directory as shown below

```
(main) expert@devnet-lab:~/nso-6.0$ cd ~/work
(main) expert@devnet-lab:~$ mkdir my_nso
(main) expert@devnet-lab:~/work$ cd my_nso
(main) expert@devnet-lab:~/work/my_nso$
```

- In this directory create a simple network with a couple of devices using netsim by choosing some ned available with the installation

```
(main) expert@devnet-lab:~/work/my_nso$ ncs-netsim create-network
cisco-ios-cli-3.8 2 ios
DEVICE ios0 CREATED
DEVICE ios1 CREATED
(main) expert@devnet-lab:~/work/my_nso$ ls
netsim
(main) expert@devnet-lab:~/work/my_nso$ ls netsim/ios/
ios0  ios1
(main) expert@devnet-lab:~/work/my_nso$
```

- Make this as NSO runtime using ncs-setup and pointing to netsim directory as inventory of devices

```
(main) expert@devnet-lab:~/work/my_nso$ ncs-setup --dest .
--netsim-dir netsim
Using netsim dir netsim
(main) expert@devnet-lab:~/work/my_nso$ ls
logs  ncs-cdb  ncs.conf  netsim  packages  README.ncs
README.netsim  scripts  state
(main) expert@devnet-lab:~/work/my_nso$
```

- Start the devices and then nso using the following commands

```
(main) expert@devnet-lab:~/work/my_nso$ ncs-netsim start
DEVICE ios0 OK STARTED
DEVICE ios1 OK STARTED
(main) expert@devnet-lab:~/work/my_nso$ ncs
(main) expert@devnet-lab:~/work/my_nso$ ncs --status | grep status
status: started
      db=running id=34 priority=1
path=/ncs:devices/device/live-status-protocol/device-type
(main) expert@devnet-lab:~/work/my_nso$
```

Managing the devices

- Connect to NSO CLI as admin using Cisco-Style
- Sync NSO with our simulated devices

```
(main) expert@devnet-lab:~/work/my_nso$ ncs_cli -u admin -C

admin connected from 192.168.110.1 using ssh on devnet-lab
admin@ncs# devices sync-from
sync-result {
    device ios0
    result true
}
sync-result {
    device ios1
    result true
}
admin@ncs#
```

- NSO always keeps a copy of the device configuration in CDB and you can easily check if devices are in sync or not

```
admin@ncs# devices device ios1 check-sync
result in-sync
admin@ncs#
```

Configuring the network

- From NCS CLI, get into config mode using the below command

```
(main) expert@devnet-lab:~/work/my_nso$ ncs_cli -u admin -C

User admin last logged in 2023-06-09T12:37:09.88296+00:00, to
devnet-lab, from 192.168.110.1 using cli-ssh
admin connected from 192.168.110.1 using ssh on devnet-lab
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)#
```

- Examine the existing configuration as follows

```
admin@ncs(config)# show full-configuration devices device ios1
config
devices device ios1
config
  tailfnd device netsim
  tailfnd police cirmode
  no service password-encryption
  no cable admission-control preempt priority-voice
  no cable qos permission create
  no cable qos permission update
  no cable qos permission modems
  ip source-route
  no ip cef
  ip vrf my-forward
  bgp next-hop Loopback1
```

- Configuration of a single device can be changes as follows
 - Get into device config mode using **devices device ios0 config**
 - Change hostname using **ios:hostname nso.cisco.com**
 - Move to parent prompt using **top**

- Verify the config change using **show configuration**
- Commit the configuration changes using **commit**

```
admin@ncs(config)# devices device ios0 config
admin@ncs(config-config)# ios:hostname nso.cisco.com
admin@ncs(config-config)# top
admin@ncs(config)# show configuration
devices device ios0
  config
    hostname nso.cisco.com
  !
!
admin@ncs(config)# commit
Commit complete.
admin@ncs(config)#
```

- At any point if you wish to connect to any device directly, you can do it as follows

```
(main) expert@devnet-lab:~/work/my_nso$ ncs-netsim cli-c ios0

User admin last logged in 2023-06-09T14:24:30.838836+00:00, to
devnet-lab, from 192.168.110.1 using cli-ssh
admin connected from 192.168.110.1 using ssh on devnet-lab
ios0# show running-config | include hostname
hostname nso.cisco.com
ios0#
```

Creating Services

- Service is a way of simplifying device configuration by automating service provisioning in NSO
- To add a service, you need to create a service package, which will have some pre-built files and directories
- We will use the NSO built in command **ncs-make-package** with the option **service-skeleton** to create a blank service

```
(main) expert@devnet-lab:~/work/my_nso$ cd packages/
(main) expert@devnet-lab:~/work/my_nso/packages$ ncs-make-package
--service-skeleton template simple-service
(main) expert@devnet-lab:~/work/my_nso/packages$ ls
cisco-ios-cli-3.8  simple-service
(main) expert@devnet-lab:~/work/my_nso/packages$ ls simple-service/
package-meta-data.xml  src  templates  test
```

- You can see 2 folder created above **src**, **templates**
- Check their contents and open simple-service.yang

```
(main) expert@devnet-lab:~/work/my_nso/packages/simple-service$ ls
src/yang/
simple-service.yang
(main) expert@devnet-lab:~/work/my_nso/packages/simple-service$
nano src/yang/simple-service.yang
```

```
module simple-service {
  namespace "http://com/example/simple-service";
  prefix simple-service;

  import ietf-inet-types {
    prefix inet;
  }
  import tailf-ncs {
    prefix ncs;
  }

  list simple-service {
    key name;

    uses ncs:service-data;
    ncs:servicepoint "simple-service";

    leaf name {
      type string;
    }

    // may replace this with other ways of referring to the
    devices.
    leaf-list device {
```

```

        type leafref {
            path "/ncs:devices/ncs:device/ncs:name";
        }
    }
    // replace with your own stuff here
    leaf dummy {
        type inet:ipv4-address;
    }
}
}

```

- Replace the part that says dummy as follows

```

// replace with your own stuff here
leaf dummy {
    type inet:ipv4-address;
}

```

Replace above section with the below section

```

leaf secret {
    type string;
    tailf:info "Enable secret for this device";
}

```

Go to the import section on top and add the following:

```

import tailf-common {
    prefix tailf;
}

```

- Basically we are trying to create a service to enable password on the devices
- Accordingly we are defining how the password should look in the yang data model
- We will now look at the xml template which specifies the configuration as defined in the yang model

```

(main) expert@devnet-lab:~/work/my_nso/packages/simple-service$ ls
templates/
simple-service-template.xml
(main) expert@devnet-lab:~/work/my_nso/packages/simple-service$
nano templates/simple-service-template.xml

```

```

<config-template xmlns="http://tail-f.com/ns/config/1.0"
                  servicepoint="simple-service">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <!--
        Select the devices from some data structure in the
service
        model. In this skeleton the devices are specified in a
leaf-list.
        Select all devices in that leaf-list:
      -->
      <name>{/device}</name>
      <config>
      <!--
        Add device-specific parameters here.
        In this skeleton the service has a leaf "dummy"; use that
        to set something on the device e.g.:
      -->
      <ip-address-on-device>{/dummy}</ip-address-on-device>
      -->
    </config>
  </device>
</devices>
</config-template>

```

- We need to modify the above sample template as shown below

```

<config-template xmlns="http://tail-f.com/ns/config/1.0"
                  servicepoint="simple-service">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>{./device}</name>
      <config>
        <enable xmlns="urn:ios">
          <password>
            <secret>{./secret}</secret>
          </password>
        </enable>
      </config>
    </device>
  </devices>
</config-template>

```

- After we change yang model, we have to invoke the command **make** to check for any errors and compile it
- For instance we made some mistake in yang file which is shown like below

```
(main) expert@devnet-lab:~/work/my_nso/packages/simple-service$
make -C src
make: Entering directory
'/home/expert/work/my_nso/packages/simple-service/src'
mkdir -p ../load-dir
/home/expert/nso/bin/ncsc `ls simple-service-ann.yang` > /dev/null
2>&1 && echo "-a simple-service-ann.yang" ` \
    --fail-on-warnings \
    \
    -c -o ../load-dir/simple-service.fxs yang/simple-service.yang
yang/simple-service.yang:34: error: premature end of file
make: *** [Makefile:26: ../load-dir/simple-service.fxs] Error 1
make: Leaving directory
'/home/expert/work/my_nso/packages/simple-service/src'
```

- After rectifying errors it should look like below

```
(main) expert@devnet-lab:~/work/my_nso/packages/simple-service$
make -C src
make: Entering directory
'/home/expert/work/my_nso/packages/simple-service/src'
/home/expert/nso/bin/ncsc `ls simple-service-ann.yang` > /dev/null
2>&1 && echo "-a simple-service-ann.yang" ` \
    --fail-on-warnings \
    \
    -c -o ../load-dir/simple-service.fxs yang/simple-service.yang
make: Leaving directory
'/home/expert/work/my_nso/packages/simple-service/src'
```

- We can also validate yang file correctness using python module for yang called pyang
- We will use to display yang file in tree format to better understand the hierarchy and relations

```
(main) expert@devnet-lab:~/work/my_nso/packages/simple-service$
pyang -f tree --tree-depth=2 src/yang/simple-service.yang
module: simple-service
  +--rw simple-service* [name]
    +---x check-sync
      |
      ...
```

```

+---x deep-check-sync
|   ...
+---x re-deploy
|   ...
+---x reactive-re-deploy
|   ...
+---x touch
|   ...
+--ro modified
|   ...
+--ro directly-modified
|   ...
+---x get-modifications
|   ...
+---x un-deploy
|   ...
+--ro used-by-customer-service*   ->
/ncs:services/customer-service/object-id
+--ro commit-queue
|   ...
+--rw private
|   ...
+--ro plan-location?              instance-identifier
+--ro log
|   ...
+--rw name                        string
+--rw device*                     -> /ncs:devices/device/name
+--rw secret?                     string

```

- We have now defined our service using yang data model and xml config template as a package
- We now need to connect to nso cli and reload the packages for the changes to take effect

```

(main) expert@devnet-lab:~/work/my_nso/packages/simple-service$
ncs_cli -u admin -C

User admin last logged in 2023-06-09T13:51:51.918629+00:00, to
devnet-lab, from 192.168.110.1 using cli-ssh
admin connected from 192.168.110.1 using ssh on devnet-lab
admin@ncs# packages reload

>>> System upgrade is starting.
>>> Sessions in configure mode must exit to operational mode.

```

```
>>> No configuration changes can be performed until upgrade has
completed.
>>> System upgrade has completed successfully.
reload-result {
    package cisco-ios-cli-3.8
    result true
}
reload-result {
    package simple-service
    result true
}
admin@ncs#
System message at 2023-06-09 16:58:29...
    Subsystem stopped: ncs-dp-1-cisco-ios-cli-3.8:IOSDp2
admin@ncs#
System message at 2023-06-09 16:58:29...
    Subsystem stopped: ncs-dp-2-cisco-ios-cli-3.8:IOSDp
admin@ncs#
System message at 2023-06-09 16:58:29...
    Subsystem started: ncs-dp-3-cisco-ios-cli-3.8:IOSDp2
admin@ncs#
System message at 2023-06-09 16:58:29...
    Subsystem started: ncs-dp-4-cisco-ios-cli-3.8:IOSDp
```

- We can verify the status of packages using the below command

```
admin@ncs# show packages package oper-status
packages package cisco-ios-cli-3.8
    oper-status up
packages package simple-service
    oper-status up
admin@ncs#
```

- Our service is now ready to start configuring the system
- Enter config more
- NSO CLI now understand our service and starts offering autocomplete hints

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# si
Possible completions:
    side-effect-queue  simple-service
```

```

admin@ncs(config)# simple-service ?
% No entries found
Possible completions:
  <name:string>
admin@ncs(config)# simple-service

```

- Use the below command to set the **password** on the device **ios0** to **mypasswd**
- Verify the config change by moving to parent prompt and using the command **show configuration**
- Commit the config changes using the **commit** command

```

admin@ncs(config)# simple-service pwdchange device ios0 secret
(<string>): mypasswd
admin@ncs(config-simple-service-pwdchange)# top
admin@ncs(config)# show configuration
simple-service pwdchange
  device [ ios0 ]
  secret mypasswd
!
admin@ncs(config)# commit
Commit complete.
admin@ncs(config)#

```

Modifying a service

- Once we have a service in place, making config modifications is very easy
- Just re invoke the service with the changed config
- For instance if we want to change the password, we can do it as below

```

admin@ncs(config)# simple-service pwdchange device ios0 secret
(<string>) (mypasswd): securepasswd
admin@ncs(config-simple-service-pwdchange)# commit
Commit complete.
admin@ncs(config-simple-service-pwdchange)# top
admin@ncs(config)# simple-service pwdchange get-modifications
cli {
  local-node {
    data
  }
}

```

- Check the modification using the below command

```
admin@ncs(config-simple-service-test1)# top
admin@ncs(config)# simple-service test1 get-modifications
cli {
    local-node {
        data
    }
}
```

- It is showing that what has been changed is the actual password, which is nothing but data
- We can now apply the changed configuration by redeploying the changed service

```
admin@ncs(config)# simple-service pwdchange re-deploy
admin@ncs(config)#
System message at 2023-06-09 17:13:31...
Commit performed by admin via ssh using cli.
admin@ncs(config)#
```

- If the service is no longer needed, we can delete it as below

```
admin@ncs(config)# no simple-service pwdchange
admin@ncs(config)# commit
Commit complete.
admin@ncs(config)#
```

NETCONF

- NETCONF is a configuration management protocol that defines a mechanism through which:
 - A network device can be managed
 - configuration data can be retrieved
 - new configuration data can be uploaded and manipulated
- NETCONF uses SSH for secure communication and data is encoded in XML
- This data should be consistent, standardized and as per certain agreed upon rules, if the sender and receiver have to be on the same page
- So it can be modeled using YANG

YANG

What is a data model?

- A data model is a well-understood and agreed-upon method to describe "something".
- E.g. Person:
 - **First name:** the name of the person.
 - **Last name:** the surname.
 - **Date of birth:** the day the person was born.
 - **Job Role:** such as admin, developer, or manager.
 - **Email:** the email address of the person.
- Sample YANG model to define a person would look like below

```
list person {
  key email;

  leaf first-name {
    tailf:info "Person's first name";
    mandatory true;
    type string;
  }

  leaf last-name {
    tailf:info "Person's last name";
    mandatory true;
    type string;
  }

  leaf date-of-birth {
    tailf:info "Date of birth as dd/mm/yyyy";
    type string {
      pattern "[0-9][0-9]/[0-9][0-9]/[0-9][0-9][0-9][0-9]";
    }
  }

  leaf email {
    tailf:info "Contact email";
    mandatory true;
    type string;
  }

  leaf job-role {
    tailf:info "Persons job role within organization";
```

```

    type enumeration {
      enum admin;
      enum developer;
      enum manager;
    }
  }
}

```

- Data models form the foundation for model-driven APIs and network programmability
- They define the syntax, semantics, and constraints of the data interchanged
- They define data attributes and answer questions, such as the following:
 - What is the range of a valid VLAN ID?
 - Can a VLAN name have spaces in it?
 - Should the values be enumerated and only support "up" or "down" for an admin state?
 - Should the value be a string or an integer?
- Yet Another Next Generation (YANG) has become a widely used data modeling language in the networking industry
- Although it can describe any data model, it was originally designed for networking data models
- It is used to create models of configuration and state data
- It has a structured format that is also human-readable
- The below example shows how YANG defines a model of a network interface

```

module ietf-interfaces {
  import ietf-yang-types {
    prefix yang;
  }
  container interfaces {
    list interface {
      key "name";

      leaf name {
        type string;
        description
          "The name of the interface";
      }

      leaf description {
        type string;
        description
          "A textual description of the interface";
      }
    }
  }
}

```

```

    }

    leaf type {
        type identityref {
            base interface-type;
        }
        mandatory true;
        description
            "The type of the interface";
    }

    leaf enabled {
        type boolean;
        default true;
        description
            "The configured state of the interface";
    }
}
}

```

- This YANG model describes ethernet interfaces
- Each individual attribute of the interface is represented by a leaf statement
- Every interface that is modeled with this YANG model has a name, type and description, and can be enabled or disabled
- Interfaces are uniquely identified with the name. This is defined with the key attribute on the leaf named "name".

YANG Structure

Modules

- In YANG, all data modeling definitions reside inside YANG modules
- Modules organize related items and place them in groups
- Often a single module describes a complete part of a functionality, such as the Border Gateway Protocol (BGP) or interface IP configuration
- Other times, a device, especially a simpler one, could implement everything in a single module
- Module can be defined as shown below

Unset

```
module ip-access-list {  
    namespace "http://example.com/ns/yang/ip-access-list";  
    prefix acl;  
  
    // ...  
}
```

- We can add some additional information to this module as shown below

Unset

```
organization  
    "Example, Inc.";  
  
contact  
    "Example, Inc.  
    Customer Service  
    E-mail: cs-yang@example.org";  
  
description  
    "Access Control List (ACL) YANG model.";  
  
revision 2021-07-06 {  
    description  
        "Initial revision";  
}
```

- **organization**: Organization identifies the entity that created the module.
- **contact**: Contact identifies the contact responsible for the module.
- **description**: Description statement serves as an overview of the purpose of the module.
- **revision**: Revision statement is used for version control, with sub statements detailing the changes.

Containers and Leaves

- YANG uses different *nodes* to hold the actual data
- Think of a node as a *data structure* in a programming language
- The data inside a single node is also described using different data types

- To group several related nodes inside a single node, use a **container** node
- If you need a simple node for some basic data types, use a **leaf** node

Unset

```
container acl {
    description
        "Access Control Lists";
}
```

- A leaf node contains simple data such as an integer or a character string
- You can further restrict values of a particular type with the usage of the **pattern**, **range**, and **length** restriction statements
- A pattern is defined with the *regular expression* (regex), range, and length are both defined with **integers**

Unset

```
container acl {
    ...
    leaf acl-description {
        type string {
            length "0..64";
            pattern "[0-9a-zA-Z]*";
        }
        description "Purpose of ACL";
    }
}
```

- The `leaf-list` statement defines an array of values, which have the same, specific type

Unset

```
container acl {
    ...
    leaf acl-description {
        ...
    }
    leaf-list maintainers {
        type string;
        description "Maintainers working on the ACL";
    }
}
```

- The `list` node specifies a sequence of list entries and is similar to Python's *dictionary* data structure

YANG Data Types

Common Data Types

Type Name	Type	Description
binary	Text	Any binary data
bits	Text/Number	A set of bits or flags
boolean	Text	"true" or "false"
decimal64	Number	64-bit fixed point real number
empty	Empty	A leaf that does not have any value

enumeration	Text/Number	Enumerated strings with associated numeric values
identityref	Text	A reference to an abstract identity
instance-identifier	Text	References a data tree node
int8	Number	8-bit signed integer
int16	Number	16-bit signed integer
int32	Number	32-bit signed integer
int64	Number	64-bit signed integer
leafref	Text/Number	A reference to a leaf instance
string	Text	Human readable string
uint8	Number	8-bit unsigned integer
uint16	Number	16-bit unsigned integer
uint32	Number	32-bit unsigned integer
uint64	Number	64-bit unsigned integer
union	Text/Number	Choice of member types

Enumeration

- Used to define a finite set of values allowed
 - true/ false
 - enabled/ disabled
 - ipv4/ ipv6

Refer

- <https://developer.cisco.com/docs/nso/guides/#!/start/start>
- https://developer.cisco.com/learning/tracks/get_started_with_nso
- <https://developer.cisco.com/docs/nso/guides/#!/basic-operations/setup>

Section 2.0: Python Programming Crash Course

What is Program?

- A program is a **set of instructions** given to a computer to perform a specific operation using some data
- When the program is executed, **raw data is processed** into a desired output format
- These programs are written in high-level **programming languages** which are close to human languages
- They are then converted to **machine understandable low level languages** and executed

What is Python?

- Python is an **interpreted, object oriented**, high-level, scripting language used for general-purpose programming
- It has wide range of applications ranging from Web Development, Scientific and Mathematical Computing to Desktop Graphical User Interfaces
- It was created by Guido van Rossum and first released in 1991

Why Python?

- **Python interpreters** are available for many operating systems including network operating systems
- Cisco puts Python on many of its devices and releases many tools using Python
- This makes Python a **great choice for network engineers** looking to add programming to their skillset
- For network and IT infrastructure engineers, there is a well-established online community of infrastructure engineers using Python to deploy and maintain IT systems

Python Installation

- If you are using Cisco CWS, Python is already installed and ready for use the moment you connect to CWS as user **expert**
- If You want to set it up on any other VM or your laptop you can follow the steps below
- On Windows
 - Download the installer from <https://www.python.org/downloads/>
 - Invoke the installer executable and follow the steps in the installation wizard
- On Ubuntu
 - By default Ubuntu comes with Python older version that is python2
 - Now a days certain Ubuntu distributions come preinstalled with python3
 - Check which version is already installed using the below command

■ `python --version`

- If you get a message like command not found or you get the result showing python version as 2.x, you can proceed further
- If the result shows the version as 3.x, you are good to go, unless you want to update to the latest version
- To install the latest version, you can use the below command
 - `sudo apt install python3 python3-pip python3-setuptools`
- Or use the following link
 - <https://www.makeuseof.com/install-python-ubuntu/>
- Or any other link of your choice

Getting started

- Check which version of python are you using

```
(main) expert@devnet-lab:~$ python -V
Python 3.9.10
(main) expert@devnet-lab:~$
```

- Do not fret too much about the version you are using, as we need to focus on the basics first
- Python is an interpreted language (as opposed to being a compiled language)
- Check which interpreter you are using

```
(main) expert@devnet-lab:~$ which python
/home/expert/venvs/main/bin/python
(main) expert@devnet-lab:~$
```

- Start using interpreter in CLI mode

```
(main) expert@devnet-lab:~$ python
Python 3.9.10 (main, Jan 15 2022, 18:17:56)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Data Types

- All we are doing as part of our work is using and manipulating data
- Data comes in various types like:

- Names of network devices: Group of alphabets - Strings (str)
- Number of interfaces: Number (Integers) (int)
- Information like whether certain interface is enabled or not: True or False (Boolean) (bool)
- OS version of certain router or switch: Number (Decimals or Floating point numbers) (float)
- List of spine and leaf switches: Collection of names (List)
- A switch with all its attributes like: Name of the attribute mapped to its value
 - Make or manufacturer: Cisco/ Juniper etc.
 - Model: Cisco CSR 1000
 - OS Version: 16.9
 - Name: brch_rtr_01
 - And so on
- Looking at a piece of data it is easy for humans to understand its type, but for machines it is not
- So we need to explicitly let the systems know what is the type of data we are dealing with
- To do this every programming language uses some data types
- Python has the following basic data types
 - Text Type : Strings (str)
 - Numeric Types : Numbers (int, float, complex)
 - Sequence Types : Lists, Tuples, Set (list, tuple, set)
 - Mapping Type : Dictionaries (dict)
 - Boolean Type : bool

Variables

- Now that we are talking about different types of data, we need a way to store this data during the program (or code as we call it) execution
- We use variables for this purpose
- Variables are named locations used to store data in memory
- Think of them like containers that holds data
- Data held by the variable can change later
- When we create a variable, we are telling the system to keep some storage aside
- How much storage to keep aside depends on the type of data that is going to be stored in that variable
- Let us create few variables and display them on the console using an inbuilt utility or function called print()
- By the way, we can use Python Interpreter CLI to give any python commands and perform some basic operations
- In CLI mode you do not need to use print() function. Directly type the name of the variable and the value stored in it will be displayed

- Use suggestive names for the variables based on what they are used to store

```
(main) expert@devnet-lab:~$ python
Python 3.9.10 (main, Jan 15 2022, 18:17:56)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> hostname = "brch_rtr_01"
>>> make = "Nexus"
>>> model = "9300v"
>>> os_version = 9.3
>>> num_intfcs = 4
>>> print(hostname)
brch_rtr_01
>>> print(make)
Nexus
>>> num_intfcs
4
>>>
```

- We can check the type of data stored in a variable using another built in function called `type()`

```
>>> type(model)
<class 'str'>
>>> type(os_version)
<class 'float'>
>>> type(num_intfcs)
<class 'int'>
>>>
```

Operators

- Data manipulation requires that we are able to perform certain operations on the data
- We will use operators for this purpose
- Accordingly every programming language, including python has certain operators defined
- Depending on the types of operations they are classified as follows
 - Arithmetic Operators
 - Assignment Operators
 - Comparison Operators
 - Logical Operators

- Special Operators

Arithmetic Operators

- Used to perform arithmetic operations on variables

Operator	Operation	Example
+	Addition	$5 + 2 = 7$
-	Subtraction	$4 - 2 = 2$
*	Multiplication	$2 * 3 = 6$
/	Division	$4 / 2 = 2$
//	Floor Division	$10 // 3 = 3$
%	Modulo	$5 \% 2 = 1$
**	Power	$4 ** 2 = 16$

```
a = 7
b = 2

# addition
print ('Sum: ', a + b)

# subtraction
print ('Subtraction: ', a - b)

# multiplication
print ('Multiplication: ', a * b)

# division
print ('Division: ', a / b)

# floor division
print ('Floor Division: ', a // b)

# modulo
print ('Modulo: ', a % b)

# a to the power b
```

```
print ('Power: ', a ** b)
```

Assignment Operators

- Used to assign values to variables

Operator	Name	Example
=	Assignment Operator	a = 7
+=	Addition Assignment	a += 1 # a = a + 1
-=	Subtraction Assignment	a -= 3 # a = a - 3
*=	Multiplication Assignment	a *= 4 # a = a * 4
/=	Division Assignment	a /= 3 # a = a / 3
%=	Remainder Assignment	a %= 10 # a = a % 10
**=	Exponent Assignment	a **= 10 # a = a ** 10

```
# assign 10 to a
a = 10

# assign 5 to b
b = 5

# assign the sum of a and b to a
a += b      # a = a + b

print(a)

# Output: 15
```

Comparison Operators

- Used to compare the value or variables
- Result of the comparison operators is always true or false (boolean)

Operator	Meaning	Example
==	Is Equal To	3 == 5 gives us False
!=	Not Equal To	3 != 5 gives us True
>	Greater Than	3 > 5 gives us False
<	Less Than	3 < 5 gives us True
>=	Greater Than or Equal To	3 >= 5 give us False
<=	Less Than or Equal To	3 <= 5 gives us True

```

a = 5

b = 2

# equal to operator
print('a == b =', a == b)

# not equal to operator
print('a != b =', a != b)

# greater than operator
print('a > b =', a > b)

# less than operator
print('a < b =', a < b)

# greater than or equal to operator
print('a >= b =', a >= b)

# less than or equal to operator
print('a <= b =', a <= b)

```

Logical Operators

- Used to check truthiness or falseness of expressions

Operator	Example	Meaning
----------	---------	---------

and	a and b	Logical AND: True only if both the operands are True
or	a or b	Logical OR: True if at least one of the operands is True
not	not a	Logical NOT: True if the operand is False and vice-versa.

```
# logical AND
print(True and True) # True
print(True and False) # False

# logical OR
print(True or False) # True

# logical NOT
print(not True) # False
```

Special Operators

- Used to verify identity or check membership

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

```
x = 'Hello world'

# check if 'H' is present in x string
print('H' in x) # prints True
```

```
# check if 'hello' is present in x string
print('hello' not in x) # prints True
```

Getting Input from the user

- We have seen how to print something to the console, which we normally call output
- We can also take something from the console as input
- We use a built in function called `input()` for this
- This is useful in taking input from the console and assigning it to variables using the assignment operator
- By default the input from the console will be read as a string
- Let us read the following input from the console and store them in appropriate variables
 - Number of edge routers
 - Names of those edge routers
 - Their IP Addresses

```
>>> num_routers = input('Enter the number of edge routers:')
Enter the number of edge routers:2
>>> router1_name = input('Enter the name of the first router:')
Enter the name of the first router:edge_rtr_01
>>> router2_name = input('Enter the name of the second router:')
Enter the name of the second router:edge_rtr_02
>>> router1_ip = input('Enter the IP Address of the first router:')
Enter the IP Address of the first router:192.168.10.1
>>> router2_ip = input('Enter the IP Address of the second
router:')
Enter the IP Address of the second router:192.168.10.2
>>> print(router2_name)
edge_rtr_02
>>>
```

Data Types - Deep Dive

Strings

- Strings are sequence of characters enclosed by quotes
- Single or double quotes are accepted, but be consistent
- Create a variable called `hostname` and assign it the value of `"ROUTER1"`

```
>>> hostname = 'edge_rtr_01'
>>> type(hostname)
<class 'str'>
>>>
```

- Strings are very useful in Python as most of the time we end up working with strings
- Some of the built in methods on strings are shown below

Methods	Description
upper()	converts the string to uppercase
lower()	converts the string to lowercase
partition()	returns a tuple
replace()	replaces substring inside
find()	returns the index of first occurrence of substring
rstrip()	removes trailing characters
split()	splits string from left
startswith()	checks if string starts with the specified string
isnumeric()	checks numeric characters
index()	returns index of substring

Lists

- Lists are sequence data type used to store multiple values or objects
- It's one of the most frequently used and versatile data type
- It is an ordered sequence, meaning elements of a list are indexed by integers starting at 0
- Similar to arrays in other programming languages
- But elements of a list can be of any data type
- An example is shown below

```
>>> commands = ["interface Ethernet1/1", "switchport access vlan
```

```
10"]
>>> commands
['interface Ethernet1/1', 'switchport access vlan 10']
>>> type(commands)
<class 'list'>
>>>
```

- List is an ordered collection, which means that the elements in the list are indexed
- We can retrieve elements of a list using an index
- Indexing in lists starts from 0
- Lists also support negative indexing which helps us to traverse the list in reverse
- Some useful methods on lists are shown below

Method	Description
<u>append()</u>	add an item to the end of the list
<u>extend()</u>	add items of lists and other iterables to the end of the list
<u>insert()</u>	inserts an item at the specified index
<u>remove()</u>	removes item present at the given index
<u>pop()</u>	returns and removes item present at the given index
<u>clear()</u>	removes all items from the list
<u>index()</u>	returns the index of the first matched item
<u>count()</u>	returns the count of the specified item in the list
<u>sort()</u>	sort the list in ascending/descending order
<u>reverse()</u>	reverses the item of the list
<u>copy()</u>	returns the shallow copy of the list

Tuples

- A tuple in Python is similar to a list
- The difference between the two is that we cannot change the elements of a tuple once it is assigned

- We call this property as immutability
- A tuple is created by placing all the items (elements) inside parentheses '(', ')'
- Similar to lists, tuples can have any number of elements and of any type
- From memory management point of view tuples are more efficient than lists
- Also tuples are used where we do not expect the variable to be changed accidentally
- There are not too many built in methods are available for Tuples as they cannot be modified once created
- We just have **count()** and **index()** methods on tuples

Dictionaries

- Dictionaries are mapping data types where data is stored as combination of key, value pairs
- We can think of them as unordered lists
- Unlike lists instead being indexed by a number, dictionaries are indexed by a name known as key
- Also known with other names like hashes or associative arrays
- Let us create a simple dictionary to store the details of a network device

```
>>> dev_dict = {'vendor': 'cisco', 'hostname': 'csr2', 'ip':
'10.10.10.2'}
>>> dev_sub_dict = dev_dict["hostname"]
>>> dev_sub_dict
'csr2'
>>>
```

- But why do we need a new data type like a dictionary? Why can't we use one of the existing ones like a list?
- Let us relook at a list which stores similar details

```
>>> dev = ['sw1', '10.1.100.1/24', '00:00:00:00:00:01']
>>> # hostname is stored at index 0
>>> # mac address is stored at index 2
```

- Looking at the values stored in the list, we have to guess what they stand for
- Instead we can associate them with appropriate keys in the dictionary

```
>>> dev = {'hostname': 'sw1', 'mgmt_ip': '10.1.100.1/24',
'mac': '00:00:00:00:00:01'}
>>> type(dev)
<class 'dict'>
>>> dev
```

```
{'hostname': 'sw1', 'mgmt_ip': '10.1.100.1/24', 'mac':  
'00:00:00:00:00:01'}  
>>>
```

Conditional Statements

- Decision making is a crucial component of programming for building logic
- We can alter the course the program using this decision making feature
- if...elif...else statements are used in Python for decision making
- Syntax is
 if test expression:
 statement(s)
- The statement(s) are executed only if the test expression is evaluated to True (Python Boolean value)

if Statement

- As per the syntax seen above, the if statement should end with a colon (:) symbol
- The statements that follow must be indented (usually 4 by spaces)
- The combination of the if keyword and the statements forms a block
- TODO:
 - Check the os version of a device and add it to the list only if the version is equal to a predefined value

```
>>> facts = {'vendor': 'arista', 'mgmt_ip': '10.1.1.1', 'chipset':  
't2', 'hostname': 'NYC301', 'os': '6.1.2'}  
>>> devices = []  
>>> devices  
[]  
>>> if facts['os'] == '6.1.2':  
...     devices.append(facts['hostname'])  
...  
>>> devices  
['NYC301']  
>>>
```

if - else

- We have instructed the code what to do when the condition is satisfied
- We have seen that when the condition is not satisfied, the code does not do anything
- We will use if - else to rectify this

```
>>> command = ''
>>> if command:
...     print('Command to send', command)
... else:
...     print('No command to send')
...
No command to send
>>> #note that 0, empty string, None all evaluate to False
```

Loops

- In programming loops are used to repeat the execution of a block of code
- Two types of loops are used in Python
 - for loop
 - while loop
- For loop is used to iterate over a sequence or through a given set of objects
- While loop is used to iterate a block of code as long a test expression evaluates to True
- In that sense while is similar to if except that if stops execution after once whereas while continues execution till the condition continues to be True

for Loop

- Used to iterate through a given set of objects
- Uses a user defined variable to hold each object temporarily
- Syntax


```
for item in sequence:
    statement(s)
```
- TODO:
 - Create a list of routers and iterate through the list to print the name of each router

```
>>> routers = ['r1', 'r2', 'r3']
>>> for router in routers:
...     print(router)
...
r1
r2
r3
>>>
```

- Let us see another example

```
>>> interfaces = ['Eth1/1', 'vlan20', 'Eth4/4', 'loop10']
>>> for interface in interfaces:
```

```

...     if interface.lower().startswith('et'):
...         itype = 'ethernet'
...     elif interface.lower().startswith('vl'):
...         itype = 'svi'
...     elif interface.lower().startswith('lo'):
...         itype = 'loopback'
...     print(itype)
...
ethernet
svi
ethernet
loopback
>>>

```

while Loop Example

- Let us say we want to create 10 or 100 vlans
- We can use for loop for this
- But using a while loop would be much better

```

>>> vlan_count = 0
>>> while vlan_count < 10:
...     print(f"Creating vlan_{vlan_count}")
...     vlan_count += 1
...
Creating 0
Creating vlan_1
Creating vlan_2
Creating vlan_3
Creating vlan_4
Creating vlan_5
Creating vlan_6
Creating vlan_7
Creating vlan_8
Creating vlan_9
>>>

```

Functions

- Helps to break down large programs into small modular chunks
- Helps keep code organized and manageable
- More importantly helps us follow DRY (Don't Repeat Yourself) principle
- We write function once and call it multiple times

- Syntax

```
def function_name(parameters):  
    """docstring"""  
    statement(s)
```
- Functions are not new for us, as we have used quite a few of them already

A Few Built-in Functions

- `print()`, `len()`
- We can also chain the functions together
- `len()` returns the length which in turn can be passed to `print` for printing it to the console

```
command = 'sh ip int br'  
print(command)  
print(len(command))
```

Output:

```
sh ip int br  
12
```

Writing a Function

- Functions that we will create become user defined functions
- Follow the syntax shown above
- Starts with keyword `def` followed by a name to identify the function
- `()` for passing optional arguments followed by :
- Statements within the function will be indented uniformly
- Function definition ends with an optional `return` statement which returns some data to the caller
- Defining a function will not execute the code in it, unless it is called elsewhere

```
def get_vlans(device):  
    if device == 'R1':  
        vlans = [1, 5, 10, 11, 14, 15]  
    else device == 'R2':  
        vlans = [5, 10, 14, 15]  
    return vlans
```

```
>>> r1_vlans = get_vlans('R1')  
>>> print(r1_vlans)  
[1, 5, 10, 11, 14, 15]  
>>> r2_vlans = get_vlans('R2')  
>>> print(r2_vlans)
```

```
[5, 10, 14, 15]
```

Functions With Optional Arguments

- While defining a function, we can define it with some arguments
- But while calling the function, we can omit them
- In such case we must provide a default value for them

```
>>> def ip_mask(ip, mask = 24):  
...     print("IP is {}/{}".format(ip, mask))  
...  
>>> ip_mask('10.250.206.21', 32)  
IP is 10.250.206.21/32  
>>> ip_mask('10.250.206.21')  
IP is 10.250.206.21/24  
>>>
```

Functions With Keyword Arguments

- When we call a function along with values, the values get assigned to arguments according to their position, as seen in the previous slide
- This allows us to change the order in which we pass the values
- But then, positional arguments have to come first, followed by keyword arguments

```
>>> def ip_mask(ip, mask):  
...     print("IP is {}/{}".format(ip, mask))  
...  
>>> ip_mask('10.250.206.21', 32)  
IP is 10.250.206.21/32  
>>> ip_mask(32, '10.250.206.21')  
IP is 32/10.250.206.21  
>>> ip_mask(mask=32, ip='10.250.206.21')  
IP is 10.250.206.21/32  
>>>
```

Functions With Arbitrary Arguments

- Sometimes the number of arguments to be passed is not known in advance
- In such cases we define the function with arbitrary or variable number of arguments
- We can mix variable length and fixed arguments, in which case fixed arguments must be keyword arguments
- The arbitrary argument defined to hold variable number of values is treated as an iterable item
- We can extract values from it using a for loop

```
>>> def router_vlan(*routers, vlan):
...     for router in routers:
...         print("Router {} belongs to VLAN {}".format(router,
vlan))
...
>>> router_vlan('r1', vlan=10)
Router r1 belongs to VLAN 10.
>>> router_vlan('r1', 'r2', 'r3', vlan=10)
Router r1 belongs to VLAN 10.
Router r2 belongs to VLAN 10.
Router r3 belongs to VLAN 10.
>>>
```

Libraries

- Modules
 - Collection of functions and global variables stored as a file with .py extension
 - It is an executable file
- Package
 - Simple directory containing collection of modules and sub packages (sub directories)
 - It has a `__init__.py` file which is used by Python interpreter to identify this directory as Python package

Built-in module - json

- We need to import json package before we can use it

- Once imported we can check the available methods in it using dir()

```
>>> import json
>>> dir(json)
['JSONDecodeError', 'JSONDecoder', 'JSONEncoder', '__all__',
 '__author__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__path__', '__spec__',
 '__version__', '_default_decoder', '_default_encoder', 'codecs',
 'decoder', 'detect_encoding', 'dump', 'dumps', 'encoder', 'load',
 'loads', 'scanner']
>>>
```

- As we are going to see further, json format is quite similar to Python dictionaries
- One use of json module is to pretty print Python dictionaries
- json objects are quite similar to Python dictionaries
- We “dump” Python objects into json string using dumps() method
- This process is called serialization

```
>>> device_facts = {'chipset': 't2', 'hostname': 'NYC301',
 'vendor': 'arista'}
>>> device_facts
{'chipset': 't2', 'hostname': 'NYC301', 'vendor': 'arista'}
>>> type(device_facts)
<class 'dict'>
>>> json_str = json.dumps(device_facts)
>>> json_str
'{"chipset": "t2", "hostname": "NYC301", "vendor": "arista"}'
>>> type(json_str)
<class 'str'>
>>> json_str = json.dumps(device_facts, indent=4)
>>> print(json_str)
{
    "chipset": "t2",
```

```

        "hostname": "NYC301",
        "vendor": "arista"
    }
>>> json_str = json.dumps(device_facts)
>>> print(json_str)
{"chipset": "t2", "hostname": "NYC301", "vendor": "arista"}
>>>

```

- We can also deserialize data and load them into the program during the run time
- We will use loads() method for this

```

>>> json_str = '{"chipset": "t2", "hostname": "NYC301", "vendor": "arista"}'
>>> json_obj= json.loads(json_str)
>>> json_obj
{'chipset': 't2', 'hostname': 'NYC301', 'vendor': 'arista'}
>>> type(json_obj)
<class 'dict'>
>>> json_obj['chipset']
't2'
>>>

```

Working with files

- So far we have either hard coded the data or read the data from the user as input
- What is more useful in network automation is reading the data from files
- Accordingly Python supports both reading data from and writing data to files
- We can read data from text files, XML files,JSON files, CSV files and so on
- All the above files are plain text format files, also termed as flat files
- We can also read data from Excel files using some special libraries

Working with text files

- Create a file with config commands as strings

```
$ cat conf.txt
en
conf t
hostname CR2
int g 0/1
ip address dhcp
no shut
```

- Open the file using the read() command and check the details

```
>>> conf_file = open('conf.txt')
>>> conf_file
<_io.TextIOWrapper name='conf.txt' mode='r' encoding='UTF-8'>
>>> dir(conf_file)
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__enter__', '__eq__', '__exit__',
 '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
 '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__',
 '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '_checkClosed', '_checkReadable',
 '_checkSeekable', '_checkWritable', '_finalizing', 'buffer',
 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno',
 'flush', 'isatty', 'line_buffering', 'mode', 'name', 'newlines',
 'read', 'readable', 'readline', 'readlines', 'reconfigure', 'seek',
 'seekable', 'tell', 'truncate', 'writable', 'write',
 'write_through', 'writelines']
>>>
```

- We have methods like read(), readline(), readlines()
- Let us try them and understand

```
>>> print(conf_data)
en
conf t
hostname CR2
int g 0/1
ip address dhcp
no shut

>>> type(conf_data)
<class 'str'>
>>>
```

- Try readline() now

```
>>> conf_line = conf_file.readline()
>>> conf_line
''
>>>
```

- We are getting blank line because we already read the data using read() method
- So the file pointer is at the bottom of the file now
- We can move the pointer back to the top or anywhere within the file using seek() method

```
>>> conf_file.seek(0)
0
>>> conf_line = conf_file.readline()
>>> print(conf_line)
en

>>> conf_line = conf_file.readline()
>>> print(conf_line)
conf t
```

```
>>>
```

- Let us go back to the top of the file and use `readlines()`

```
>>> conf_file.seek(0)
0
>>> conf_lines = conf_file.readlines()
>>> print(conf_lines)
['\n\n', 'conf t\n', 'hostname CR2\n', 'int g 0/1\n', 'ip address
dhcp\n', 'no shut\n']
>>> type(conf_lines)
<class 'list'>
>>>
```

- We can also write data to files using the write mode
- Open a file for writing using the "w" value instead "r" for reading

```
>>> out_file = open('interface.cfg', 'w')
>>> out_file.write("interface Eth1\n")
15
>>> out_file.write(" speed 100\n")
11
>>> out_file.write(" duplex full\n")
13
>>> out_file.close()
```

- Opened the file in write mode, wrote data and closed the file
- Closing the file is not mandatory but is recommended as a coding best practice
- Without using the context manager working with files opening them for read write operations and forgetting to close them can leave them in inconsistent state
- The **with** statement ensures that files is closed as soon as we are done working with it

```
>>> out_file = open('interface.cfg', 'w')
>>> out_file.write("interface Eth1\n")
15
>>> out_file.write(" speed 100\n")
11
>>> out_file.write(" duplex full\n")
13
>>> out_file.close()
```

Working with CSV files

- Create a CSV file “devices.csv” as follows

```
Device,IP
Router1,192.168.1.1
Switch1,192.168.1.2
```

- We can read the data from the CSV file using a built in library called **csv** as follows

```
# Read devices from CSV file
with open('devices.csv', 'r') as csv_file:
    reader = csv.reader(csv_file)
    header = next(reader) # Skip the header row
    device_idx = header.index('Device')
    ip_idx = header.index('IP')

    for row in reader:
        device_name = row[device_idx]
        device_ip = row[ip_idx]
```

- The csv library also offers a better method to read csv data directly as Python dictionaries as shown in the below example

```
# Read devices from CSV file
with open('devices.csv', 'r') as csv_file:
    reader = csv.DictReader(csv_file)
    for row in reader:
        device_name = row['Device']
        device_ip = row['IP']
```

- We can also write data to csv files using writer method
- However we must first arrange the data in CSV suitable format, like list of lists where each list in the outer list corresponds to a row in CSV file

```
# Data to be written
data = [
    ['John Doe', 'john@example.com'],
    ['Jane Smith', 'jane@example.com'],
    ['Bob Johnson', 'bob@example.com']
]

# Open the CSV file in write mode
with open('contacts.csv', 'w', newline='') as csv_file:
    writer = csv.writer(csv_file)

    # Write the header
    writer.writerow(['Name', 'Email'])

    # Write the data rows
    writer.writerows(data)
```

- If the data is available as dictionaries, we can also use DictWriter method to write data to CSV files

```
# Data to be written
data = [
    {'Name': 'John Doe', 'Email': 'john@example.com'},
    {'Name': 'Jane Smith', 'Email': 'jane@example.com'},
    {'Name': 'Bob Johnson', 'Email': 'bob@example.com'}
]

# Field names/columns
fieldnames = ['Name', 'Email']

# Open the CSV file in write mode
with open('contacts.csv', 'w', newline='') as csv_file:
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames)

    # Write the header
    writer.writeheader()

    # Write the data rows
    writer.writerows(data)
```

Working with JSON files

- We have already seen how to manipulate JSON data using the builtin **json** library
- Here we will see how to work with JSON files
- Create a sample json file called network_devices.json with the following content

```
{
  "devices": [
    {
      "name": "Router1",
      "ip": "192.168.1.1"
    },
    {
      "name": "Switch1",
      "ip": "192.168.1.2"
    },
    {
      "name": "Firewall1",
      "ip": "192.168.1.3"
    }
  ]
}
```

- We can read the data from json files as shown below

```
import json

# Read JSON file
with open('network_devices.json') as f:
    data = json.load(f)

# Access the devices
devices = data['devices']
for device in devices:
    print(f"Name: {device['name']}, IP: {device['ip']}")
```

- Writing new data to json file can be down as shown below

```
import json
# Add a new device
new_device = {
    "name": "Switch2",
    "ip": "192.168.1.4"
}
devices.append(new_device)

# Write updated data to JSON file
with open('network_devices.json', 'w') as f:
    json.dump(data, f, indent=4)
```

Working with YAML files

- YAML files have become de facto choice for representing configuration in the networking as well as devops world
- Python provides a library **yaml** to work with yaml files
- Create a yaml file called config.yaml with the content shown below

```
network:
  interface: eth0
  ip_address: 192.168.1.10
  subnet_mask: 255.255.255.0
hostname: router1
```

- We can read data from yaml files as shown below

```
import yaml

with open('config.yaml', 'r') as file:
    config_data = yaml.safe_load(file)

# Accessing specific values
interface = config_data['network']['interface']
ip_address = config_data['network']['ip_address']
```

- Writing new data to yaml file can be done as shown below

```
import yaml

# Create YAML data
config_data = {
    'network': {
        'interface': 'eth0',
        'ip_address': '192.168.1.10',
        'subnet_mask': '255.255.255.0',
    },
    'hostname': 'router1',
}

# Save the data to a YAML file
with open('config.yaml', 'w') as file:
    yaml.dump(config_data, file)
```

Python Object Oriented Programming

Introduction

- Object oriented programming is a programming paradigm or style which uses the concept of classes and object to solve the problems
- It helps us to model a given problem in terms of real life objects so that we can build programs or solutions around the objects and their interaction with other entities
- It focuses on modeling real-world entities as objects and defining their behavior through classes and objects
- It promotes code reusability, modularity, and maintainability

Classes

- Classes are blueprints or templates that define the structure and behavior of objects
- They encapsulate attributes (data) and methods (functions) that represent the characteristics and actions of objects
- For example, we can define NetworkDevice and NetworkInterface as classes

```
class NetworkDevice:
    def __init__(self, ip, username, password):
        self.ip = ip
        self.username = username
        self.password = password

    def connect(self):
        # Code to establish a connection with the device
        pass

    def configure(self, command):
        # Code to send configuration commands to the device
        pass
```

```
class NetworkInterface:

    def __init__(self, name, ip_address):

        self.name = name

        self.ip_address = ip_address

    def configure_ip(self, ip_address):

        # Code to configure the IP address of the interface

        pass
```

Objects

- Objects are instances of classes
- They are created based on the class blueprint and have their own unique state and behavior
- Objects represent specific entities or elements that you want to work with
- For example, device1, device2, and interface are objects created from the respective classes defined above

Attributes

- Attributes are variables that store data associated with an object
- They represent the state or characteristics of an object
- In the examples, attributes like ip, username, password, name, and ip_address are used to store specific information related to network devices and interfaces

Methods

- Methods are functions defined within a class that perform operations on objects
- They define the behavior or actions associated with the objects
- In the examples, methods like connect(), configure(), and configure_ip() are defined to establish connections, send configurations, and configure IP addresses for network devices and interfaces

Initialization (__init__ method)

- The __init__ method is a special method in Python classes that is automatically called when an object is created
- It is used to initialize the object's attributes
- In the examples, the __init__ method is used to set initial values for attributes like ip, username, password, name, and ip_address



Encapsulation

- Encapsulation is the process of bundling data (attributes) and methods together within a class
- It helps in organizing related functionality and restricting access to data, allowing controlled interactions with objects
- In the examples, the classes encapsulate attributes and methods related to network devices and interfaces

Working with REST APIs

Introduction

- REST (Representational State Transfer) is an architectural style for designing networked applications
- For example, in the context of Cisco ACI and Python, REST APIs are used to interact with the Cisco ACI fabric programmatically
- REST APIs provide a set of endpoints or URLs that allow you to perform various operations on the ACI fabric
- These operations can be as follows:
 - Retrieving information
 - Creating new resources
 - Updating existing resources
 - Deleting resources
- Each endpoint represents a specific functionality or resource within the ACI fabric

HTTP – REST

- REST APIs are developed taking advantage of the well known HTTP technology which does pretty much the same in the web world
- HTTP makes use of action words or verbs (GET, POST, PUT/ PATCH and DELETE) to manipulate web resources
- Unlike traditional HTTP which deals with the data being sent and received as XML request and response, REST APIs use JSON as data interchange format
- We can map the above operations to the HTTP methods as below
 - Retrieving information – GET
 - Creating new resources – POST
 - Updating existing resources – PUT/ PATCH
 - Deleting resources – DELETE

Authentication

- Just because REST APIs let us work with the resources directly via endpoints, it does not mean we can bypass the security
- REST APIs also need authentication and authorization like any other applications
- E.g. the Python code sets up authentication by sending a POST request with the user's credentials to obtain an authentication token
- This token is then included in subsequent requests' headers to authenticate and authorize the API calls

Working with Cisco ACI REST APIs

- To work with REST APIs in Python there are some libraries which we need like **requests** (to send requests, parse response and so on), **json** (to deal with json data being sent as request and received as response)
- We first need to import them to work with them
- We then need to define other details like ACI URL and username and password to connect to it
- To generate authentication token ACI authentication API expected the username and password to be sent in JSON format
- Once we sent the request in required format, we get the response which contains the token in JSON format
- We need to load this response into a variable as a Python dictionary and extract the required information using dictionary indexing

```
import requests
import json
import urllib3
urllib3.disable_warnings()
base_url = 'https://sandboxapicdc.cisco.com/api'
fabric_nodes_endpoint = '/class/fabricNode.json'
tenants_endpoint = '/node/class/fvTenant.json'
def get_token():
    auth_url = 'https://sandboxapicdc.cisco.com/api/aaaLogin.json'
    username = 'admin'
    password = '!v3G@!4@Y'
```

```

payload = {
    "aaaUser": {
        "attributes": {
            "name": username,
            "pwd": password
        }
    }
}

response = requests.post(auth_url, json=payload, verify=False)
token = response.json()['imdata'][0]['aaaLogin']['attributes']['token']
# print(token)
return token

def get_data(endpoint):
    url = base_url + endpoint
    token = get_token()
    headers = {
        'Cookie': f'APIC-cookie={token}'
    }

    response = requests.get(url, headers=headers, verify=False)
    # print(response)
    data = response.json()
    print(json.dumps(data, indent=4))

# get_data(fabric_nodes_endpoint)
get_data(tenants_endpoint)

```

- The method `get_token()` connects to ACI sandbox instance and fetches the token
- We wrote a generic method `get_data()` which accepts any endpoint and returns the data

Create REST API

- We can create a simple REST API using Python REST libraries as follows
- We can also generate REST API documentation using a tool called Swagger

```
from flask import Flask
from flask_restx import Api, Resource
from flasgger import Swagger

app = Flask(__name__)
api = Api(app, version='1.0', title='My API', description='API for greetings')
swagger = Swagger(app)

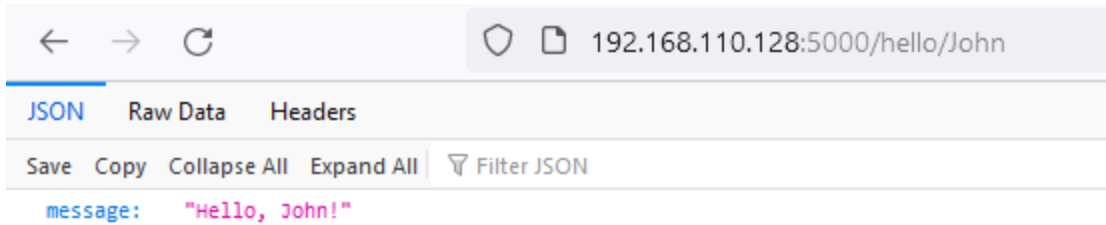
@api.route('/hello/<string:name>')
class HelloWorld(Resource):
    @api.doc(params={'name': 'Your name'})
    @api.response(200, 'Success')
    def get(self, name):
        """
        Get a hello message
        """
        return {'message': f'Hello, {name}!'}

@app.route('/doc')
def swagger_doc():
    return api.__schema__

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

- In the above example we are using a python library called flask to create a REST API
- We use an application called Swagger to document REST APIs
- We create a simple python function which accepts a string and returns the same concatenated with a Hello
- This method can be accessed via an endpoint called /hello by passing the input string as path parameter, meaning the parameter itself is part of the endpoint path

- E.g. When you hit the endpoint **`http://<server_hostname_or_ip>:<port>/hello/John`** in the browser, you get a response "Hello John"
- To get the API documentation as json you can use the endpoint **`http://<server_hostname_or_ip>:<port>/doc`**
- To get the documentation as Swagger UI, directly access **`http://<server_hostname_or_ip>:<port>/`** in a browser



←

→

↻

🛡️

📄

192.168.110.128:5000/doc

JSON

Raw Data

Headers

Save

Copy

Collapse All

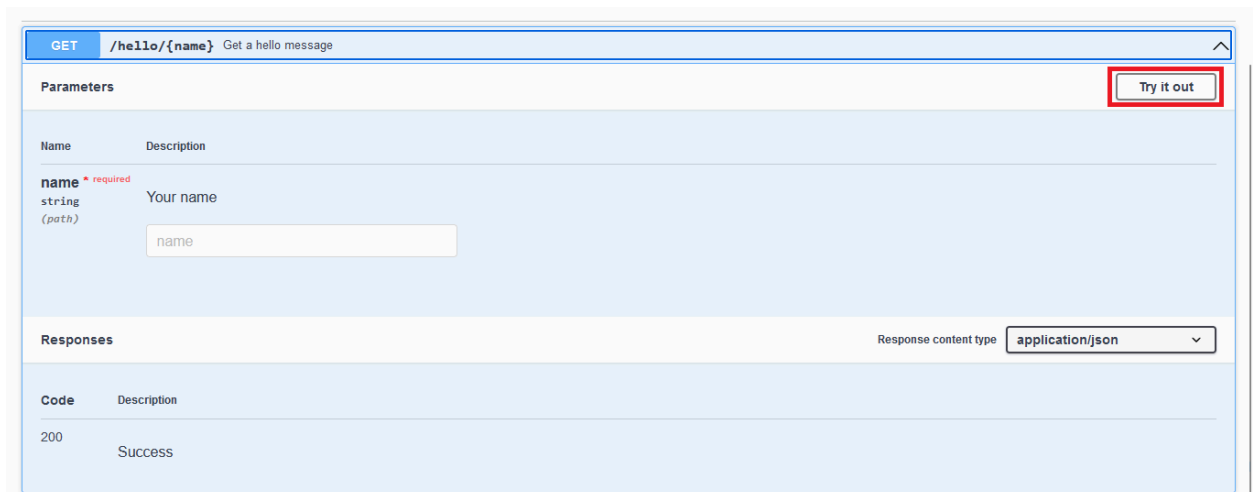
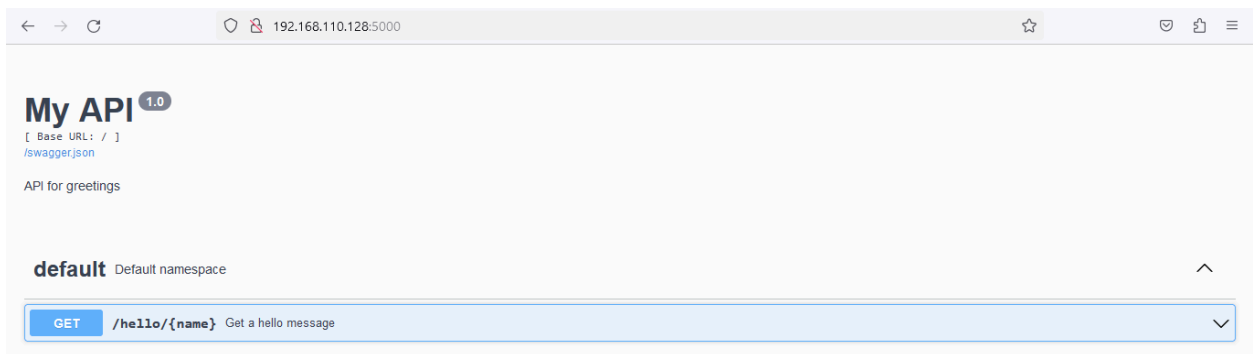
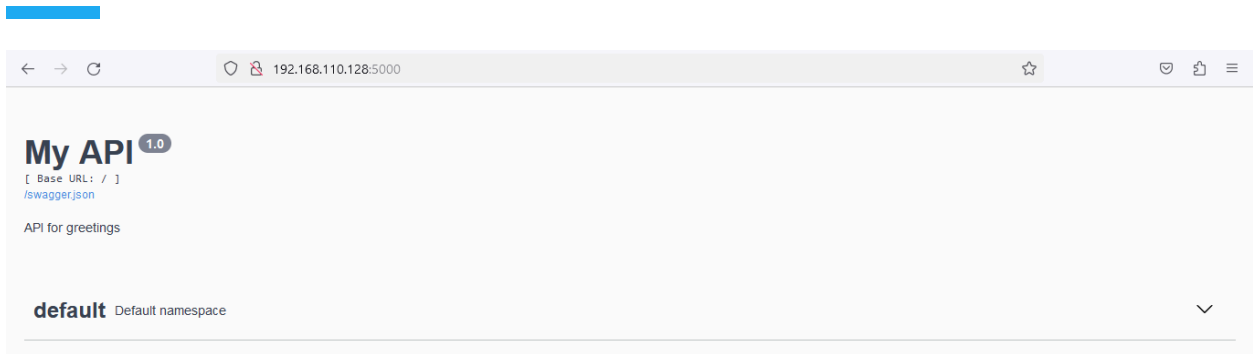
Expand All

🔍 Filter JSON

```

basePath: "/"
consumes:
  0: "application/json"
info:
  description: "API for greetings"
  title: "My API"
  version: "1.0"
paths:
  /hello/{name}:
    get:
      operationId: "get_hello_world"
      parameters:
        0:
          description: "Your name"
          in: "path"
          name: "name"
          required: true
          type: "string"
      responses:
        200:
          description: "Success"
          summary: "Get a hello message"
          tags:
            0: "default"
      produces:
        0: "application/json"
      responses:
        MaskError:
          description: "When any error occurs on mask"

```



GET /hello/{name} Get a hello message

Parameters

Name	Description
name * required string (path)	Your name

John

Execute

Responses

Response content type: application/json

Code	Description
200	Success

Responses

Response content type: application/json

Curl

```
curl -X 'GET' \
  'http://192.168.110.128:5000/hello/John' \
  -H 'accept: application/json'
```

Request URL

```
http://192.168.110.128:5000/hello/John
```

Server response

Code	Details
200	Response body <pre>{ "message": "Hello, John!" }</pre> Response headers <pre>content-length: 34 content-type: application/json date: Sun, 12 Nov 2023 16:50:32 GMT server: Werkzeug/1.0.1 Python/3.9.18</pre>

Paramiko

Paramiko is a pure-Python implementation of the SSHv2 protocol, providing both client and server functionality. It is a popular library for managing SSH connections in Python applications. Paramiko can be used to perform a variety of tasks, such as:

- Running remote shell commands
- Transferring files
- Port forwarding
- Creating SSH tunnels



Paramiko is a powerful library that can be used to manage SSH connections in a variety of ways. However, it is important to note that it is a low-level library, and it can be complex to use. For common SSH tasks, such as running remote shell commands or transferring files, it is recommended to use a higher-level library such as Fabric.

Advantages of Paramiko:

- It is a pure-Python library, so it can be used on any platform that has Python installed.
- It is a mature library with a large community of users.
- It is a powerful library that can be used to perform a wide variety of tasks.

Drawbacks of Paramiko:

- It is a low-level library, and it can be complex to use.
- It is not as well-documented as some other SSH libraries.
- It may not be the best choice for common SSH tasks, such as running remote shell commands or transferring files.

Getting Started

- Install paramiko using the following command
 - `pip install paramiko`
- The following example shows connecting to a server using username and password and executing some commands

```
import paramiko

command = "df"

host = "192.168.110.133"
username = "expert"
password = "1234QWer!"

client = paramiko.client.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect(host, username=username, password=password)
_stdin, _stdout, _stderr = client.exec_command("df")
_stdin.close()
```

```
print(_stdout.read().decode())
client.close()
```

- `ssh.exec_command` function, it contains what are known as three data streams
- These are the input, the output, and any errors that may appear. This is known as a 3-tuple. The syntax is:
 - `stdin, stdout, stderr = ssh.exec_command("command")`
- You can use any variable names you like in place of `stdin`, `stdout`, and `stderr`
- All three of these streams are returned whenever the `ssh.exec_command` function is executed
- They contain the following information:
 - `stdin` - the input stream. This is a file-like object connected to the standard input (`stdin`) of the remote command. You can write to this object if your command expects input during its execution. In the context of the example, `stdin` is not used, but you might need it if your command requires interactive input.
 - `stdout` - the output stream. This is a file-like object connected to the standard output (`stdout`) of the remote command. You can read from this object to get the output of the executed command.
 - `stderr` - any errors that may have appeared are stored here. This is a file-like object connected to the standard error (`stderr`) of the remote command. Like `stdout`, you can read from this object to get any error output produced by the command.

Netmiko

- Paramiko is a Python implementation of SSHv2 protocol
- Netmiko is a multi-vendor library to simplify Paramiko SSH connections to network devices
- Netmiko simplifies:
 - Establishing ssh connection to various devices
 - Execution of show commands used to retrieve config data
 - Execution of configuration commands sent to the device
 - Working with a broad range of devices from different vendors

Getting Started

- Install netmiko using the following command
 - `python -m pip install netmiko`

```
expert@devnet-lab: ~/venvs
(test) expert@devnet-lab:~/venvs$ python -m pip install netmiko
Collecting netmiko
  Downloading netmiko-4.2.0-py3-none-any.whl (213 kB)
    |████████████████████| 213 kB 2.3 MB/s
Collecting paramiko>=2.9.5
  Downloading paramiko-3.2.0-py3-none-any.whl (224 kB)
    |████████████████████| 224 kB 2.4 MB/s
Collecting scp>=0.13.6
  Downloading scp-0.14.5-py2.py3-none-any.whl (8.7 kB)
Collecting textfsm>=1.1.3
  Downloading textfsm-1.1.3-py2.py3-none-any.whl (44 kB)
    |████████████████████| 44 kB 1.9 MB/s
Collecting pyserial>=3.3
  Using cached pyserial-3.5-py2.py3-none-any.whl (90 kB)
Collecting ntc-templates>=2.0.0
  Downloading ntc_templates-3.4.0-py3-none-any.whl (377 kB)
    |████████████████████| 377 kB 2.2 MB/s
Collecting pyyaml>=5.3
  Using cached PyYAML-6.0-cp39-cp39-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_12_x86_64.manylinux2010_x86_64.whl (661 kB)
Collecting bcrypt>=3.2
  Downloading bcrypt-4.0.1-cp36-abi3-manylinux_2_28_x86_64.whl (593 kB)
    |████████████████████| 593 kB 1.2 MB/s
Collecting pynacl>=1.5
```

- Import "ConnectHandler" from the Netmiko library
- ConnectHandler acts as your main entry point into the library
- It picks the right class for you, creates a Netmiko object based on that class, and establishes an SSH connection to the remote device.

```
(main) expert@devnet-lab:~$ python
Python 3.9.10 (main, Jan 15 2022, 18:17:56)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> from netmiko import ConnectHandler
>>>
```

- You would then use ConnectHandler to pick the class and establish the SSH connection
- ConnectHandler needs the following details to connect to the device
 - device_type
 - host (hostname or IP)
 - username
 - password

```
from netmiko import ConnectHandler
conn = ConnectHandler(
    device_type="cisco_ios",
    host = "192.168.110.136",
    username = "admin",
    password = "cisco")
```

```
print(conn)
```

Output:

```
$ python netmiko_test.py
```

```
<netmiko.cisco.cisco_ios.CiscoIosSSH object at 0x7fa598a7ff40>
```

- We are now ready to start issuing commands to the device from our code
- Let us see which prompt we are currently at

```
print(conn.find_prompt())
```

Output:

```
csr01#
```

- Executing show commands

```
output = conn.send_command("show ip arp")
print(output)
```

Output:

```
$ python netmiko_test.py
```

Protocol	Address	Age (min)	Hardware Addr	Type	Interface
Internet	192.168.110.2	0	0050.56ee.d381	ARPA	GigabitEthernet1
Internet	192.168.110.128	4	000c.29f3.6e1e	ARPA	GigabitEthernet1

```
Internet 192.168.110.136 - 5000.0002.0000 ARPA
GigabitEthernet1
```

- Let us now try to send config commands to the device

```
cfg_list = [
    "ip access-list extended TEST1",
    "permit ip any host 1.1.1.1",
    "permit ip any host 1.1.1.2",
    "permit ip any host 1.1.1.3",
    "permit ip any host 1.1.1.4",
    "permit ip any host 1.1.1.5",
]

cfg_output = conn.send_config_set(cfg_list)
print(cfg_output)
```

Output:

```
$ python netmiko_test.py
configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
csr01(config)#ip access-list extended TEST1
csr01(config-ext-nacl)#permit ip any host 1.1.1.1
csr01(config-ext-nacl)#permit ip any host 1.1.1.2
csr01(config-ext-nacl)#permit ip any host 1.1.1.3
csr01(config-ext-nacl)#permit ip any host 1.1.1.4
csr01(config-ext-nacl)#permit ip any host 1.1.1.5
csr01(config-ext-nacl)#end
csr01#
```

Click CLI

- Click is a Python package for creating beautiful command line interfaces
- Stands for “Command Line Interface Creation Kit”
- Aims to make the process of writing command line tools quick and fun
- Using click you’ll be able to create your own command line interfaces that can pass context, with commands that can take mandatory or optional arguments, and nested commands
- The three main points of Python Click are:
 - Arbitrary nesting of commands
 - Automatic help page generation
 - Supporting lazy loading of subcommands at runtime
- Click is easy to use and it provides the necessary capabilities to create complex command line interfaces
- Install click using
 - `pip install click` or `python -m pip install click`

Getting Started with Click

- Create a command line program which just says “Hello World”
- This program is a simple CLI which just has one command **hello**
- When invoked, the CLI calls the function **hello** which prints a message to the console

```
import click
@click.command()
def hello():
    click.echo('Hello World!')

if __name__ == '__main__':
    hello()
```

```
$ python click_test2.py
Hello World!
```

- It also show built in default help message which can be customized as needed

```
$ python click_test2.py --help
Usage: click_test2.py [OPTIONS]

Options:
  --help  Show this message and exit.
```

Adding Arguments

- Click allows you to add arguments and options to your CLIs.
- Arguments are required positional parameters, while options are optional named parameters.
- To add an argument, you can use the `click.argument()` decorator
- For example, the following code adds an argument to the `hello()` command that allows you to specify a name

```
import click
@click.command()
@click.argument('name')
def hello(name):
    click.echo('Hello, {}!'.format(name))

if __name__ == '__main__':
    hello()
```

- Try to run the program now, like before

```
$ python click_test2.py
Usage: click_test2.py [OPTIONS] NAME
Try 'click_test2.py --help' for help.

Error: Missing argument 'NAME'.
```

- Note that now, we get an error in a nicely formatted help message
- Since arguments are required positional parameters, we must pass a name while invoking this CLI

```
$ python click_test2.py John
Hello, John!
```

- When invoked with a name parameter, it shows the expected output

Adding Options

- To add an option, you can use the `click.option()` decorator.
- For example, the following code adds an option to the `hello()` command that allows you to specify a greeting

```
import click
@click.command()
@click.argument('name')
@click.option('-g', '--greeting', default='Hello')
def hello(name, greeting):
    click.echo('{} {}'.format(greeting, name))

if __name__ == '__main__':
    hello()
```

- Check the help now, it should give us much more useful information now about the simple CLI we built

```
$ python click_test2.py --help
Usage: click_test2.py [OPTIONS] NAME

Options:
  -g, --greeting TEXT
  --help                Show this message and exit.
```

- The CLI now expects a mandatory parameter name and an optional parameter greeting
- If the greeting is not passed, by default **Hello** will be used as the greeting

```
$ python click_test2.py John
Hello John
$ python click_test2.py John -g "How are you"
How are you John
```

- Here is a more concrete example using Netmiko, putting it all together
- This example builds a CLI which provides a command called connect which accepts hostname, username and password to **connect** to a network device using the netmiko library

```
#!/home/expert/venvs/main/bin/python

import click
from netmiko import ConnectHandler

@click.command()
@click.option('--hostname', prompt="Enter hostname", help='Number of greetings.')
```

```
@click.option('--username', prompt='Enter username', help='The person to greet.')
@click.option('--password', prompt='Enter password', hide_input=True, help='The
person to greet.')
def connect(hostname, username, password):
    """Simple program that greets NAME for a total of COUNT times."""
    conn = ConnectHandler(
        device_type = "cisco_ios",
        host = hostname,
        username = username,
        password = password
    )
    print(conn.find_prompt())

if __name__ == '__main__':
    connect()
```

Section 2.1: Python NCCLIENT, YANG, NETCONF, RESTCONF

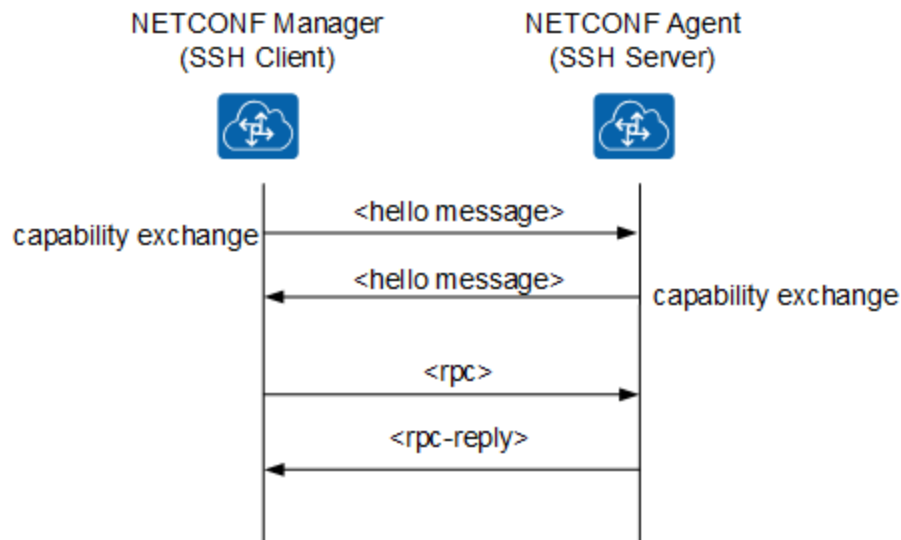
NETCONF

Introduction

- NETCONF (Network Configuration Protocol) is a network management protocol used for configuring and managing network devices.
- It is an industry-standard protocol developed by the Internet Engineering Task Force (IETF).
- NETCONF operates over the Secure Shell (SSH) protocol and uses Extensible Markup Language (XML) to encode data.
- It enables network administrators to remotely manage and configure network devices in a programmatic and standardized manner.

Workflow

- The client, written in any programming language or a network automation/ management system establishes connection with a NETCONF enabled device
- Typically SSH is used to secure this connection
- Once the session is established, the client and device can exchange XML-based messages defined using YANG data models
- NETCONF defines the following operations to manage the configuration on the devices
 - `get`: Used to retrieve configuration and operational data from the device.
 - `get-config`: Retrieves the current configuration of a specific part of the device's configuration data store.
 - `edit-config`: Allows the client to modify the configuration of the device.
 - `delete-config`: Removes a specific configuration data store.
 - `copy-config`: Creates or replaces a configuration data store with the contents of another data store.
 - `commit`: Confirms the changes made to the configuration data store.
 - `discard-changes`: Discards the uncommitted changes made during the current session.



YANG and NETCONF

- YANG is the de facto standard for NETCONF data modeling
- YANG is used to create data models that define the structure, hierarchy, and attributes of the data exchanged between the client and the network device via NETCONF
- YANG allows network administrators, device vendors, and standardization organizations to define a schema that precisely describes the data elements, their types, constraints, and relationships
- YANG provides a mechanism to define constraints and validation rules for the data exchanged via NETCONF

NETCONF Examples

- Retrieving Configuration Data (get-config):
- Before trying to connect via code we need to enable netconf-yang and netconf over ssh on the device as follows

```
CSR03#conf t
Enter configuration commands, one per line. End with CNTL/Z.
CSR03(config)#netconf-yang
CSR03(config)#netconf ssh
CSR03(config)#end
CSR03#wr
Building configuration...
[OK]
CSR03#
```

- Once this configuration is done on the device, check connectivity using the following command

```
$ ssh -s cisco@192.168.110.131 -p 22 netconf
Password:
<?xml version="1.0" encoding="UTF-8"?><hello
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"><capabilities><capa
bility>urn:ietf:params:netconf:base:1.0</capability><capability>urn
:ietf:params:netconf:capability:writeable-running:1.0</capability><
capability>urn:ietf:params:netconf:capability:startup:1.0</capabili
ty><capability>urn:ietf:params:netconf:capability:url:1.0</capabili
ty><capability>urn:cisco:params:netconf:capability:pi-data-model:1.
0</capability><capability>urn:cisco:params:netconf:capability:notif
ication:1.0</capability></capabilities><session-id>3896733280</sess
ion-id></hello>]]>]]>
```

- If you see a message like this, then your connection is successful
- We can extract device config as using netconf as shown below

```
import ncclient.manager

# Establish NETCONF session
with ncclient.manager.connect(
    host="192.168.110.131",
    port=22,
    username="cisco",
    password="cisco",
    hostkey_verify=False,
    device_params={"name": "csr"},
) as m:

    # Retrieve configuration using get-config operation
    # Send the NETCONF message and receive the response
    # netconf_response = m.get_config('running', filter=NETCONF_FILTER)
    netconf_response = m.get_config('running')

    # Parse and handle the response
    xml_response = netconf_response.xml
    print(xml_response)
```

RESTCONF

- RESTCONF (RESTful Network Configuration Protocol) is a protocol that allows the configuration and management of network devices using Representational State Transfer (REST) principles
- Provides a standardized way to interact with network devices over HTTP(S) using RESTful APIs
- RESTCONF is based on the YANG data modeling language and is supported by many network devices
- Similar to REST APIs RESTCONF supports the following methods to read and manipulate device configuration programmatically
 - GET: Used to retrieve information from the network device
 - POST: Used to create a new resource on the network device
 - PATCH: Used to modify a specific resource on the network device
 - PUT: Used to replace or overwrite a specific resource on the network device
 - DELETE: Used to remove a specific resource from the network device
- Importing libraries and establishing connection

```
import requests

HOST = 'ios-xe-mgmt-latest.cisco.com'
PORT = 443
USER = 'admin'
PASS = 'C1sco12345'

# disable urllib3 warning
requests.packages.urllib3.disable_warnings()

headers = {'Content-Type': 'application/yang-data+json',
           'Accept': 'application/yang-data+json'}

url =
f"https://{HOST}/restconf/data/Cisco-IOS-XE-interfaces-oper:interfaces/interface=
GigabitEthernet1"
```

- GET example

```
response = requests.get(url, auth=(USER, PASS), headers=headers, verify=False)
print(response.text)
```

- POST Example

```
loopback_data = {
    "name": "Loopback100",
    "description": "New Loopback Interface"
}

response = requests.post(url, headers=headers, auth=(USER, PASS),
                        json=loopback_data, verify=False)

if response.status_code == 201:
    print("Loopback interface created successfully.")
else:
    print(f"Request failed with status code: {response.status_code}")
```

- PATCH Example

```
patch_data = {
    "Cisco-IOS-XE-native:description": "Updated interface description"
}

response = requests.patch(url, headers=headers, auth=(USER, PASS),
                        json=patch_data, verify=False)

if response.status_code == 204:
    print("Interface description updated successfully.")
else:
    print(f"Request failed with status code: {response.status_code}")
```

- PUT Example

```
put_data = {
    "name": "GigabitEthernet1",
    "description": "New Interface Description",
    "ip": {
        "address": {
            "primary": {
                "address": "192.168.1.1",
                "mask": "255.255.255.0"
            }
        }
    }
}
```

```

        }
    }
}

response = requests.put(url, headers=headers, auth=(USER, PASS), json=put_data,
verify=False)

if response.status_code == 204:
    print("Interface configuration updated successfully.")
else:
    print(f"Request failed with status code: {response.status_code}")

```

- DELETE Example

```

interface_name = "Loopback100"
restconf_url =
f"https://{csr_ip}/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback={i
nterface_name}"

response = requests.delete(url, headers=headers, auth=(USER, PASS), verify=False)

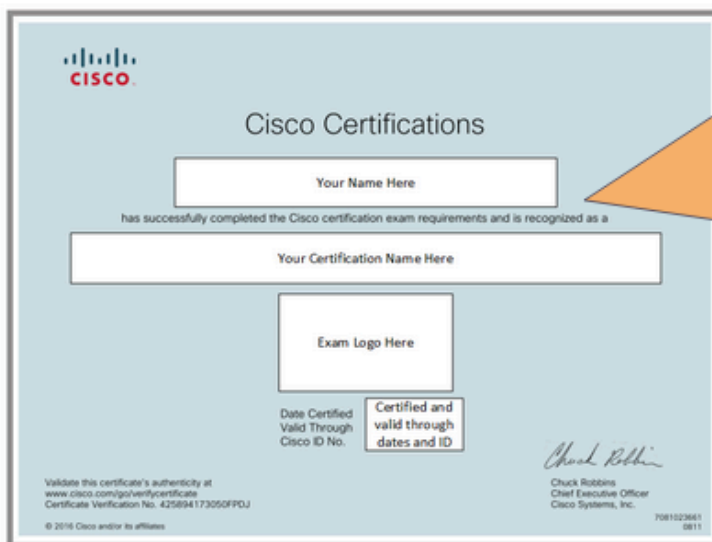
if response.status_code == 204:
    print("Loopback interface deleted successfully.")
else:
    print(f"Request failed with status code: {response.status_code}")

```

Section 2.2: Python Libraries Jinja2, Scrapli

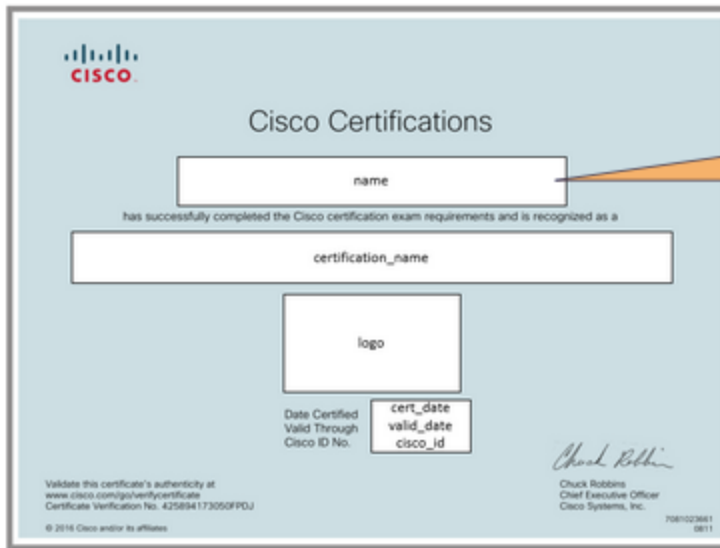
Jinja2

- Jinja2 is a feature rich templating language widely used in the Python ecosystem
- Let's take the below example
- Cisco has to generate multiple certificates for multiple candidates based on the exam they clear
- In each certificate, the outline is fixed with the specific details about the candidate, the exam, validity etc changing from one to another
- So we can create a template with the fixed parts and supply the changing parts as variables to be substituted into the placeholders in the template



- Imagine Cisco having to print thousands of such certificates for the vast number of certifications they offer
- They create a template with placeholders for the name of the person, certification, logo, dates etc.
- The things that tend to change from one certificate to the other
- The fixed part, along with placeholders becomes a template

- Replace the placeholders with variable names as shown below



- Assign values to variables as per the desired output

name

Ehsan Momeni Bashusqeh

certification_name

Cisco Certified Network Associate Routing and Switching

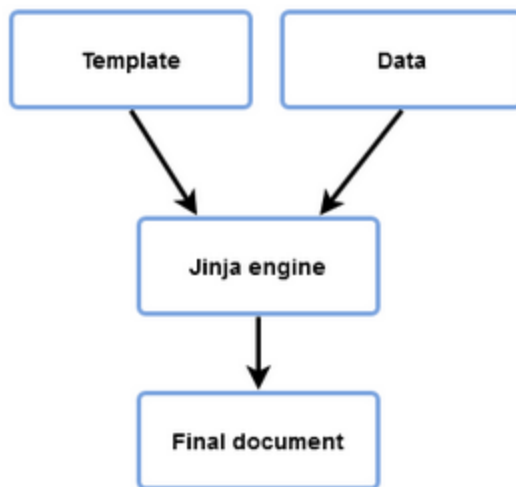
logo



cert_date
valid_date
cisco_id

August 6, 2016
August 6, 2019
CSCO13007220

- Jinja engine takes the template and data and creates the desired output file



- We can extend this to our use case as shown below
- Suppose we need BGP config files for multiple devices with differing values
- The config looks as shown below, with fixed and changing parts
- We replace changing parts with variables

```

router bgp 45000
router-id 172.17.1.99
bgp log-neighbor-changes
neighbor 192.168.1.2 remote-as 40000
neighbor 192.168.3.2 remote-as 50000
address-family ipv4 unicast
neighbor 192.168.1.2 activate
network 172.17.1.0 mask 255.255.255.0
exit-address-family

```

Sample target config we want to generate

```

router bgp {{ local_asn }}
router-id {{ router_id }}
bgp log-neighbor-changes
neighbor {{ neighbor_id_1 }} remote-as {{ remote_asn_1 }}
neighbor {{ neighbor_id_2 }} remote-as {{ remote_asn_2 }}
address-family ipv4 unicast
neighbor {{ neighbor_id_1 }} activate
network {{ network }} mask {{ net_mask }}
exit-address-family

```

Actual values replaced with variables.
This becomes a template now.

```

local_asn: 45000
router_id: 172.17.1.99
neighbor_id_1: 192.168.1.2
neighbor_id_2: 192.168.3.2
remote_asn_1: 40000
remote_asn_2: 50000
network: 172.17.1.0
net_mask: 255.255.255.0

```

Variable Set 1

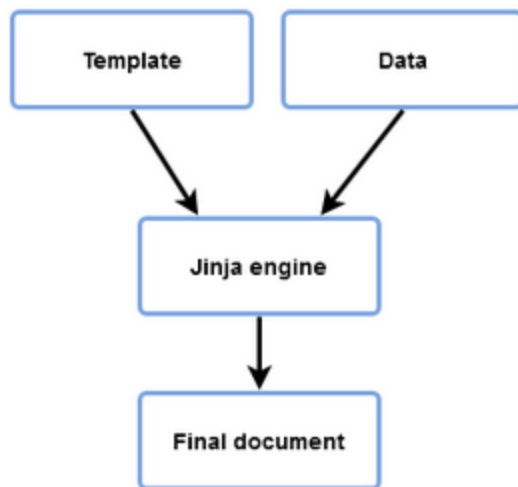
- When we substitute Variable Set 1 into the template, we get first set of BGP config commands
- Similarly when we substitute Variable Set 2 into the template, we get second set of BGP config commands

Variable Set 2

```

local_asn: 95000
router_id: 172.17.1.200
neighbor_id_1: 192.168.10.200
neighbor_id_2: 192.168.20.200
remote_asn_1: 70000
remote_asn_2: 80000
network: 172.17.1.0
net_mask: 255.255.255.0

```



```

router bgp 45000
router-id 172.17.1.99
bgp log-neighbor-changes
neighbor 192.168.1.2 remote-as 40000
neighbor 192.168.3.2 remote-as 50000
address-family ipv4 unicast
neighbor 192.168.1.2 activate
network 172.17.1.0 mask 255.255.255.0
exit-address-family
  
```

Target config 1

Target config 2

```

router bgp 95000
router-id 172.17.1.200
bgp log-neighbor-changes
neighbor 192.168.10.200 remote-as 70000
neighbor 192.168.20.200 remote-as 80000
address-family ipv4 unicast
neighbor 192.168.10.200 activate
network 172.17.1.0 mask 255.255.255.0
exit-address-family
  
```

- So you get the idea
- You pass 2 sets of data to get 2 sets of config
- Pass 'n' sets of data to get 'n' sets of config

- Let us see another simple example
- Sample template file

```

{% for name in hostnames -%}
hostname: {{ name }}
{% endfor -%}
{% for ip in ip_addresses -%}
IPAddress: {{ ip }}
  
```

```
{% endfor -%}  
device_type: {{ os_type }}
```

- Sample variables file

```
hostnames:  
  - CSR04  
  - CSR05  
ip_addresses:  
  - 10.10.2.2  
  - 10.10.2.3  
os_type: IOS_XE
```

- Sample code to render the output from the template and variables

```
from jinja2 import Template  
import yaml  
  
with open('./templates/test.j2') as template_file:  
    template_str = template_file.read()  
  
template = Template(template_str)  
# print(template)  
  
yaml_data = yaml.safe_load(open("./host_vars/vars1.yml"))  
# print(yaml_data)  
  
rendered_output = template.render(yaml_data)  
print(rendered_output)
```

Scrapli

- scrapli -- scrap(e c)li -- is a python library focused on connecting to devices, specifically network devices (routers/switches/firewalls/etc.) via SSH or Telnet.
- The name scrapli -- is just "scrape cli" (as in screen scrape) squished together!
- scrapli's goal is to be as fast and flexible as possible, while providing a thoroughly tested, well typed, well documented, simple API.

- An example to establish connection, send read commands and config commands is shown below
- Scrapli is pretty much similar to netmiko that we have already seen

```
from scrapli.driver.core import IOSXEDriver

with IOSXEDriver(
    host="192.168.110.131",
    auth_username="cisco",
    auth_password="cisco",
    auth_strict_key=False,
    transport="paramiko",
) as conn:
    prompt_result = conn.get_prompt()
    response = conn.send_command("show run")
    responses = conn.send_commands(["show run", "show ip int brief"])
    config_response = conn.send_configs(["interface loopback123", "description
configured by scrapli"])
print(prompt_result)
print(response.result)
print(response.elapsed_time)
print(responses.result)
print(config_response.result)
```

Section 3.1: Git, Gitlab, CI/CD Pipelines

Git

- Git is a distributed version control system
- Git allows developers to track changes in their source code
- Also allows them to collaborate with others on software development projects
- It enables multiple developers to work on the same codebase simultaneously,
- Keeps track of changes made by each developer, and merges their work together.

Install Git

- We can download the client and install on various platforms
 - <https://git-scm.com/downloads>
- Mostly git is preinstalled on linux systems
- If not we can just follow the simple steps mentioned below (On debian/ ubuntu systems)
 - `sudo apt update`
 - `sudo apt install git`

Getting Started with Git

- We either have an existing project directory or starting work in a new directory, we first have to tell git to start monitoring it

A screenshot of a Windows command prompt window. The title bar shows the path 'MINGW64:/c:/Users/NagaRaj/temp/git_work/project1'. The prompt is 'NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1'. The user has entered the command '\$ pwd' and the output is '/c:/Users/NagaRaj/temp/git_work/project1'. The prompt is now '\$ |'.

- Configure git by setting user name and email so that each can update can be traced back to a user
- Open git bash and give the following commands
 - `git config --global user.name "John Doe"`
 - `git config --global user.email johnndoe@example.com`

```

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1
$ git config --global user.name "John Doe"

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1
$ git config --global user.email johndoe@example.com

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/etc/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=main
user.email=johndoe@example.com
user.name=John Doe

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1
$ |

```

- Check if the current directory is already a git repo or not using
 - git status

```

MINGW64:/c/Users/NagaRaj/temp/git_work/project1

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1
$ git status
fatal: not a git repository (or any of the parent directories): .git

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1
$

```

- The above error message shows that current directory is not tracked by git and hence git does not know about it
- Any directory can be made as git working directory using the command
 - git init

```
NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1
$ git init
Initialized empty Git repository in C:/Users/NagaRaj/temp/git_work/project1/.git/

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1 (main)
$
```

- At any point check the status using
 - git status

```
NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1 (main)
$ git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1 (main)
$
```

- If we already have some files, git says it found some untracked files
- Use the command
 - git add .

```

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1 (main)
$ touch test.txt

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1 (main)
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
       test.txt

nothing added to commit but untracked files present (use "git add" to track)

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1 (main)
$ git add .

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1 (main)
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
       new file:   test.txt

```

- All files are added to staging area which acts as local cache
- Use the command
 - `git commit -m "appropriate commit message"`
- Once committed, the files are backed up into the local repo with a unique commit ID
- All the new files are now being tracked by git and are backed up into local repo with a unique version or commit ID

```

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1 (main)
$ git commit -m "first checkin"
[main (root-commit) a6d4966] first checkin
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 test.txt

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1 (main)
$ git status
On branch main
nothing to commit, working tree clean

NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1 (main)
$ git log
commit a6d49668af68c9382ccb6b4234d1bba6bcfa5c7d (HEAD -> main)
Author: John Doe <johndoe@example.com>
Date:   Fri Nov 17 01:22:48 2023 +0530

    first checkin

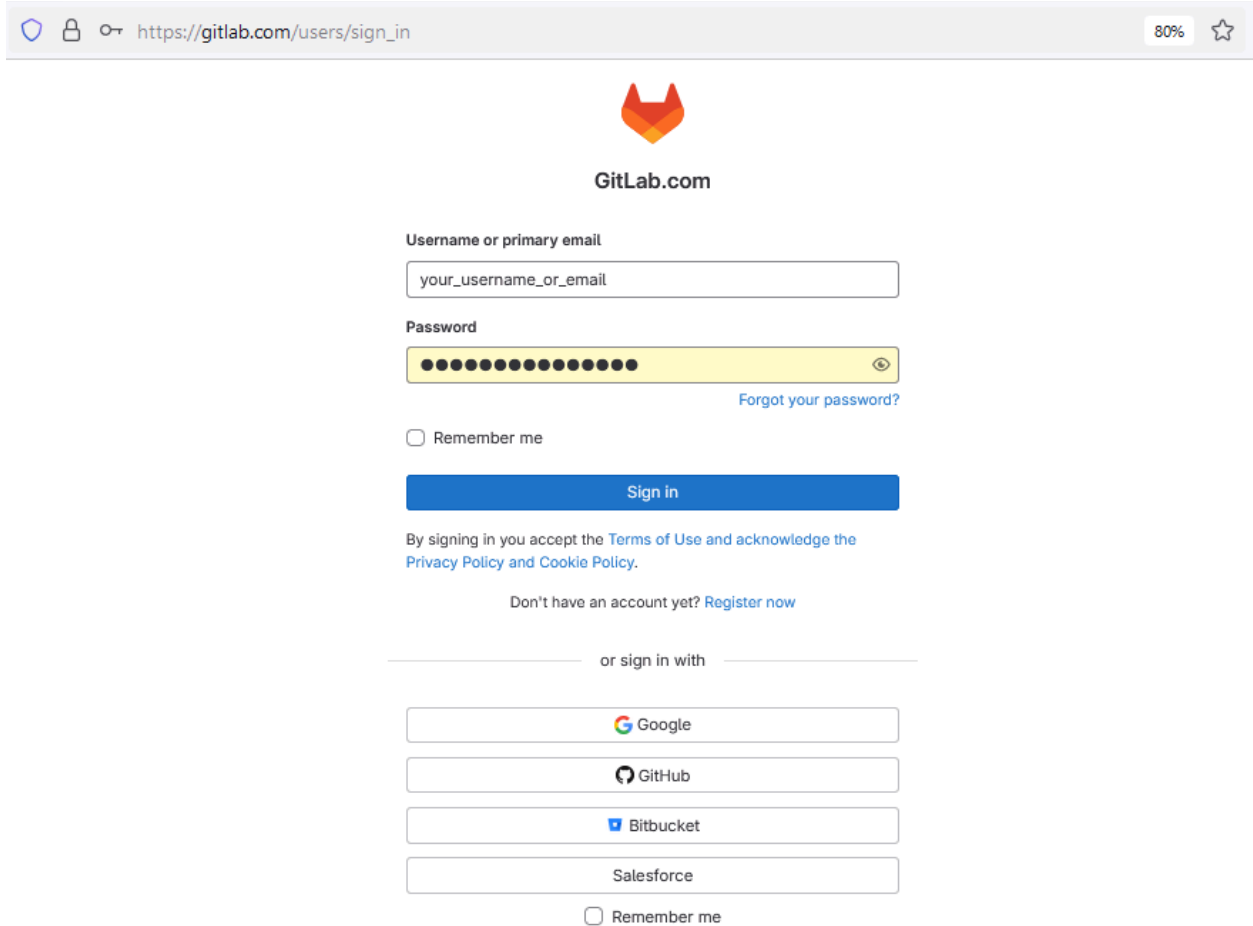
NagaRaj@DESKTOP-QTPSLK5 MINGW64 ~/temp/git_work/project1 (main)
$ |

```

- Any further changes to the files have to go through the same process of git add and git commit

Gitlab

- GitLab is a hosted version control system based on Git.
- We can sync local repo we created above with the remote repo (project) created in GitLab
- GitLab can be downloaded and installed in your local data center or could server
- It can also be used for free (for non commercial usage) or in a subscription model by creating an account or by logging in if you are already registered
 - https://gitlab.com/users/sign_in



The image shows the GitLab.com sign-in page. At the top, the browser address bar displays 'https://gitlab.com/users/sign_in' with a 80% zoom level. The GitLab logo is centered above the 'GitLab.com' text. Below this, the 'Username or primary email' field contains 'your_username_or_email'. The 'Password' field is masked with dots and has a 'Forgot your password?' link to its right. A 'Remember me' checkbox is present. A blue 'Sign in' button is centered. Below the button, a message states: 'By signing in you accept the Terms of Use and acknowledge the Privacy Policy and Cookie Policy.' A link 'Register now' is provided for users without accounts. A horizontal line separates this section from the 'or sign in with' section, which features buttons for Google, GitHub, Bitbucket, and Salesforce, along with another 'Remember me' checkbox.

Username or primary email

your_username_or_email

Password

Forgot your password?

☐ Remember me

Sign in

By signing in you accept the [Terms of Use](#) and acknowledge the [Privacy Policy](#) and [Cookie Policy](#).

Don't have an account yet? [Register now](#)

or sign in with

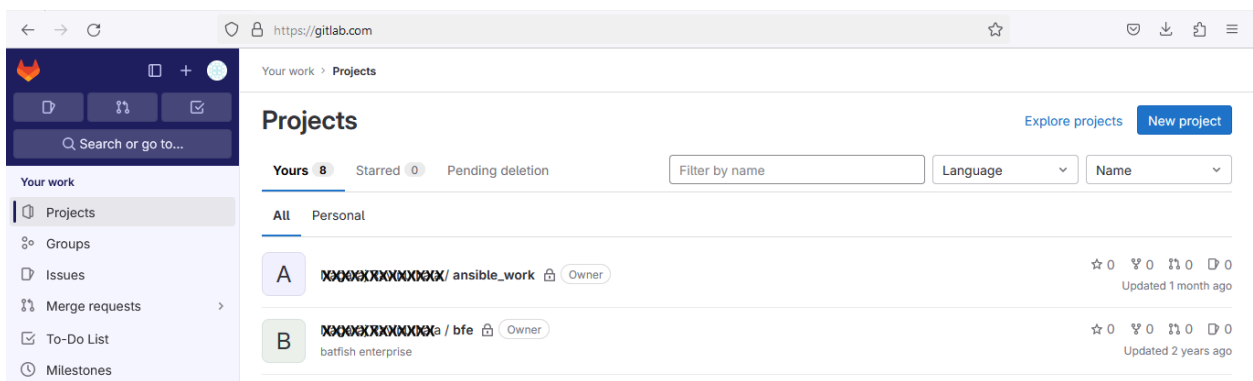
Google

GitHub

Bitbucket

Salesforce

☐ Remember me



The image shows the GitLab 'Projects' page. The left sidebar contains a 'Your work' section with links to Projects, Groups, Issues, Merge requests, To-Do List, and Milestones. The main content area is titled 'Projects' and includes a search bar and filters for 'Yours' (8), 'Starred' (0), and 'Pending deletion'. There are also filters for 'Filter by name', 'Language', and 'Name'. Below the filters, two project cards are visible. The first card, labeled 'A', shows a repository named 'ansible_work' owned by 'XXXXXXXXXX', with 0 stars, 0 forks, 0 issues, and 0 pull requests, updated 1 month ago. The second card, labeled 'B', shows a repository named 'bfe' owned by 'XXXXXXXXXXa / bfe', with 0 stars, 0 forks, 0 issues, and 0 pull requests, updated 2 years ago.

Projects

Explore projects New project

Yours 8 Starred 0 Pending deletion

Filter by name Language Name

All Personal

A XXXXXXXXXXXX / ansible_work Owner 0 stars 0 forks 0 issues 0 pull requests Updated 1 month ago

B XXXXXXXXXXXXa / bfe Owner 0 stars 0 forks 0 issues 0 pull requests Updated 2 years ago

CI/CD Pipelines

- GitLab CI/CD (Continuous Integration/Continuous Deployment) pipelines are a powerful feature of GitLab that allow you to automate the building, testing, and deployment of your software projects

- CI/CD pipelines are defined using a configuration file called `.gitlab-ci.yml`, which is stored in the root directory of your project repository.
- This file specifies the stages, jobs, and scripts that should be executed during the CI/CD process
- Stages:
 - A pipeline is divided into one or more stages.
 - Each stage represents a logical phase of the CI/CD process, such as "build," "test," and "deploy."
 - Jobs within a stage run in parallel, and the pipeline proceeds to the next stage only if all jobs in the current stage succeed.
- Jobs:
 - Each stage contains one or more jobs.
 - A job is a single task or set of tasks that should be executed, such as compiling code, running tests, or deploying the application.
 - Jobs are executed on runners, which can be shared runners provided by GitLab or dedicated runners set up by you.
- Runners:
 - Runners are responsible for executing the jobs defined in the pipeline.
 - GitLab provides shared runners that are available to all projects, and you can also set up your own runners if you have specific requirements.
- Artifacts:
 - Jobs can produce artifacts, which are files or directories generated during the CI/CD process.
 - Artifacts can be passed between jobs or stages, allowing you to share build outputs or test results with subsequent steps.
- Cache:
 - GitLab allows you to cache certain files or directories between CI/CD pipeline runs.
 - Caching can significantly speed up the build process by reusing dependencies from previous runs
- Triggers and Dependencies:
 - Pipelines can be triggered manually or automatically based on specific events, such as code pushes or merge requests.
 - You can also define dependencies between jobs, ensuring that a job only runs after its dependencies have completed successfully.
- Environments:
 - GitLab allows you to define environments for different stages of your pipeline, such as "staging" and "production."
 - This facilitates deploying your application to different environments based on the branch or tags.

Sample Pipeline 1

.gitlab-ci.yml

```
image: python:latest

job0:
  stage: build
  script:
    - echo "Build phase"
    - uname -a

job1:
  stage: test
  script:
    - echo "Test phase"
    - python hello.py

job2:
  stage: deploy
  script:
    - echo "Ready to be deployed"
```

hello.py

```
print("Hello World!!!")
```

Nagaraj Ravinuthala > my py proj > Pipelines

All 2 Finished Branches Tags

Clear runner caches CI lint **Run pipeline**

Filter pipelines

Status	Pipeline	Triggerer	Stages
passed 00:01:21 31 minutes ago	Update file .gitlab-ci.yml #939043076 main 8c78a4ed latest		✓✓✓
passed 00:01:22 34 minutes ago	Update file .gitlab-ci.yml #939038730 main 8c78a4ed latest		✓✓✓

Nagaraj Ravinuthala > my py proj > Jobs

All 6 Finished

Filter jobs

Status	Job	Pipeline	Stage	Name	Duration	Coverage
passed 00:00:26 1 hour ago	#4701313354 main 8c78a4ed	#939043... created by	deploy	job2		
passed 00:00:27 1 hour ago	#4701313351 main 8c78a4ed	#939043... created by	test	job1		
passed 00:00:27 1 hour ago	#4701313350 main 8c78a4ed	#939043... created by	build	job0		

Sample Pipeline 2

Another sample pipeline is shown below:

```
stages:
  - build
  - test
  - deploy

variables:
  APP_NAME: my-app
  IMAGE_TAG: latest

before_script:
  - echo "Setting up environment..."

build:
```

```
stage: build
script:
  - echo "Building the application..."

test:
  stage: test
  script:
    - echo "Running tests..."

deploy:
  stage: deploy
  script:
    - echo "Deploying the application..."
```

1. `stages`: This defines the different stages of your CI/CD pipeline. In this example, there are three stages: `build`, `test`, and `deploy`. Jobs within the same stage run in parallel, and stages run sequentially.
2. `variables`: This section allows you to define variables that can be used throughout the pipeline. Here, we have two variables: `APP_NAME` and `IMAGE_TAG`. These can be referenced later in the pipeline.
3. `before_script`: This is a script that runs before every job. In this example, it's a simple echo statement, but you might use this section to set up your environment or install dependencies.
4. `build`: This is a job that belongs to the `build` stage. It has a script section that contains the commands to build your application. In this case, it's just an echo statement, but you would replace it with the actual build commands for your project.
5. `test`: This is a job that belongs to the `test` stage. Similar to the build job, it has a script section where you'd put the commands to run tests for your application.
6. `deploy`: This is a job that belongs to the `deploy` stage. It has a script section with commands to deploy your application. Adjust this section based on your deployment requirements.

Section 3.2: Network Test Automation with pyATS

- pyATS (Python Automation Testing System) is a framework developed by Cisco to facilitate network automation and testing.
- It provides a set of Python libraries and tools that enable network engineers to automate their network infrastructure and perform various testing tasks.

How does pyATS work?

- Testbed Definition:
 - A YAML file that describes the network devices, their connectivity, and other relevant information.
 - Includes details such as device names, IP addresses, connection parameters (e.g., SSH credentials), and interface configurations.
- Test Case Creation:
 - A test case is a Python class that contains individual test methods.
 - Each test method defines a specific test or action to be performed on the network devices.
- Test Execution:
 - Test cases are executed using the pyATS test runner.
 - The test runner reads the testbed file, establishes connections to the devices, and executes the test methods defined in the test cases.
- Device Interaction:
 - PyATS provides libraries for interacting with network devices, such as establishing SSH connections, executing commands, retrieving output, and configuring devices.
 - These libraries abstract the underlying device-specific APIs, making it easier to write cross-platform automation code.
- Test Result Reporting:
 - pyATS generates detailed test reports that include information about the test execution status, test case results, and any failures encountered.
 - These reports help you identify issues and troubleshoot problems in your network infrastructure.
- Sample code snippets are given below

Example to connect to the device and execute commands

```
from pyats.topology import loader
# load testbed
```

```

testbed = loader.load('testbed.yaml')
# get the device from testbed
csr03 = testbed.devices['CSR03']
# connect to the device
csr03.connect()
# send show command
csr03.execute('show run')
# send config command
csr03.configure('''
    interface GigabitEthernet4
        ip address 10.10.10.1 255.255.255.0
''')

# send show command to verify the config
csr03.execute('sh ip int br')

```

Example to show setup, test and clean up sections

```

from pyats import aetest
from pyats.topology import loader

# test data
loopback_interface = 'Loopback0'
loopback_ip = '192.168.1.1'
loopback_subnet = '255.255.255.0'

#####
###                      COMMON SETUP SECTION                      ###
#####

class Setup(aetest.CommonSetup):

    @aetest.subsection
    def connect(self):
        # Load the testbed file
        testbed = loader.load('./testbed.yaml')

```

```

# Get the device object
device = testbed.devices['CSR03']

# Connect to the device
device.connect()

# store the connection reference in instance variable
self.device = device
self.loopback_interface = 'Loopback0'
self.parent.parameters.update(device=device)

@aetest.subsection
def configure(self):
    # perform configuration
    self.device.configure([
        f'interface {loopback_interface}',
        f'ip address {loopback_ip} {loopback_subnet}'
    ])

#####
###                                TESTCASES SECTION                                ###
#####

class config_check(aetest.Testcase):

    @aetest.test
    def check_loopback_configuration(self):
        device = self.parent.parameters.get('device')
        # Get the loopback interface configuration
        output = device.execute('show ip interface brief | include Loopback0')
        print('*****')
        print(output)
        print('*****')

        expected_output = f'{loopback_interface}                                {loopback_ip}
YES manual up                                up'
        print('*****')

```

```

print(expected_output)
print('*****')

    assert expected_output in output, f'Loopback interface configuration is
incorrect'

@aetest.cleanup
def cleanup(self):
    device = self.parent.parameters.get('device')
    # clear the configuration
    device.configure([
        f'no interface {loopback_interface}',
    ])

if __name__ == '__main__':
    aetest.main()

```

- Typically the automated unit tests are supposed to be idempotent in nature, so that they can be run multiple times with each execution not depending on the previous one or affecting the next one
- Test cases should also be self sufficient, so that they can be executed on any new environment with minimal setup with ease
- For this reason, we divide the test cases into 3 sections
 - Setup
 - Test case
 - Tear down
- **Setup**
 - Will take care of the necessary setup required for the test cases execution
 - In the example above which tests the loopback configuration, the test cases will not assume the device to have the loopback configuration done already
 - Instead the loopback configuration is done in the setup section
- **Test case**
 - The test case will just focus on performing the test step and validating the expected vs actual output
 - The configuration required for the test case is already done by the setup section
- **Tear down**

- 
- The setup done for the test case execution will impact the further execution of the same test case or other test cases, unless cleared up
 - Hence this section will clear up any configuration done in the setup section, leaving the test environment in the same way as it was found

Section 3.3: Secret Management with Vault

Secret Management with HashicorpVault

HashiCorp Vault is a tool designed for managing secrets and protecting sensitive data within a modern data center or cloud environment. It provides a secure and centralized way to manage access to tokens, passwords, certificates, encryption keys, and other sensitive data.

Step 1: Download Vault

Visit the [official Vault releases page](#) and find the version you want to install. Copy the download link for the appropriate version and architecture. Open a terminal on your Ubuntu machine. Download and unzip Vault:

```
wget
https://releases.hashicorp.com/vault/<version>/vault_<version>_linux_amd
64.zip
unzip vault_<version>_linux_amd64.zip
```

Step 2: Move Vault binary to /usr/local/bin

```
sudo mv vault /usr/local/bin/
```

Step 3: Verify the Installation

Check that Vault is installed correctly by running:

```
$ vault -v
Vault v1.15.2 (cf1b5cafa047bc8e4a3f93444fcb4011593b92cb), built
2023-11-06T11:33:28Z
```

Step 4: Create a Vault Configuration File

Create a configuration file for Vault, e.g., `config.hcl`:

```

storage "file" {
  path = "/tmp/vault"
}

listener "tcp" {
  address = "0.0.0.0:8300"
  tls_disable = 1
}

```

Step 5: Start Vault in Dev Mode (for testing)

For testing purposes, you can start Vault in development mode. In a real production environment, you would configure Vault differently.

```

vault server -dev -config=config.hcl

```

Once server starts successfully, you should see output as shown below:

```

2023-11-19T18:20:40.449Z [INFO] core: loaded wrapping token key
2023-11-19T18:20:40.449Z [INFO] core: successfully setup plugin runtime catalog
2023-11-19T18:20:40.449Z [INFO] core: successfully setup plugin catalog: plugin-directory=""
2023-11-19T18:20:40.451Z [INFO] core: successfully mounted: type=system version="v1.15.2+builtin.vault" path=sys/ namespace="ID: root. Path: "
2023-11-19T18:20:40.452Z [INFO] core: successfully mounted: type=identity version="v1.15.2+builtin.vault" path=identity/ namespace="ID: root. Path: "
2023-11-19T18:20:40.452Z [INFO] core: successfully mounted: type=cubbyhole version="v1.15.2+builtin.vault" path=cubbyhole/ namespace="ID: root. Path: "
2023-11-19T18:20:40.454Z [INFO] core: successfully mounted: type=token version="v1.15.2+builtin.vault" path=token/ namespace="ID: root. Path: "
2023-11-19T18:20:40.455Z [INFO] rollback: Starting the rollback manager with 256 workers
2023-11-19T18:20:40.455Z [INFO] core: restoring leases
2023-11-19T18:20:40.456Z [INFO] rollback: starting rollback manager
2023-11-19T18:20:40.456Z [INFO] expiration: lease restore complete
2023-11-19T18:20:40.456Z [INFO] identity: entities restored
2023-11-19T18:20:40.456Z [INFO] identity: groups restored
2023-11-19T18:20:40.457Z [INFO] core: post-unseal setup complete
2023-11-19T18:20:40.457Z [INFO] core: vault is unsealed
2023-11-19T18:20:40.467Z [INFO] core: successful mount: namespace="" path=secret/ type=kv version=""
WARNING! dev mode is enabled! In this mode, Vault runs entirely in-memory
and starts unsealed with a single unseal key. The root token is already
authenticated to the CLI, so you can immediately begin using Vault.

You may need to set the following environment variables:

$ export VAULT_ADDR='http://127.0.0.1:8200'

The unseal key and root token are displayed below in case you want to
seal/unseal the Vault or re-authenticate.
Unseal Key: +YQxi0cVADaX3G0S4boq5oGpK03bBf+jJ/atK2RX7p0=
Root Token: hvs.st4AoWg2g3b8Scwgk292KzFj

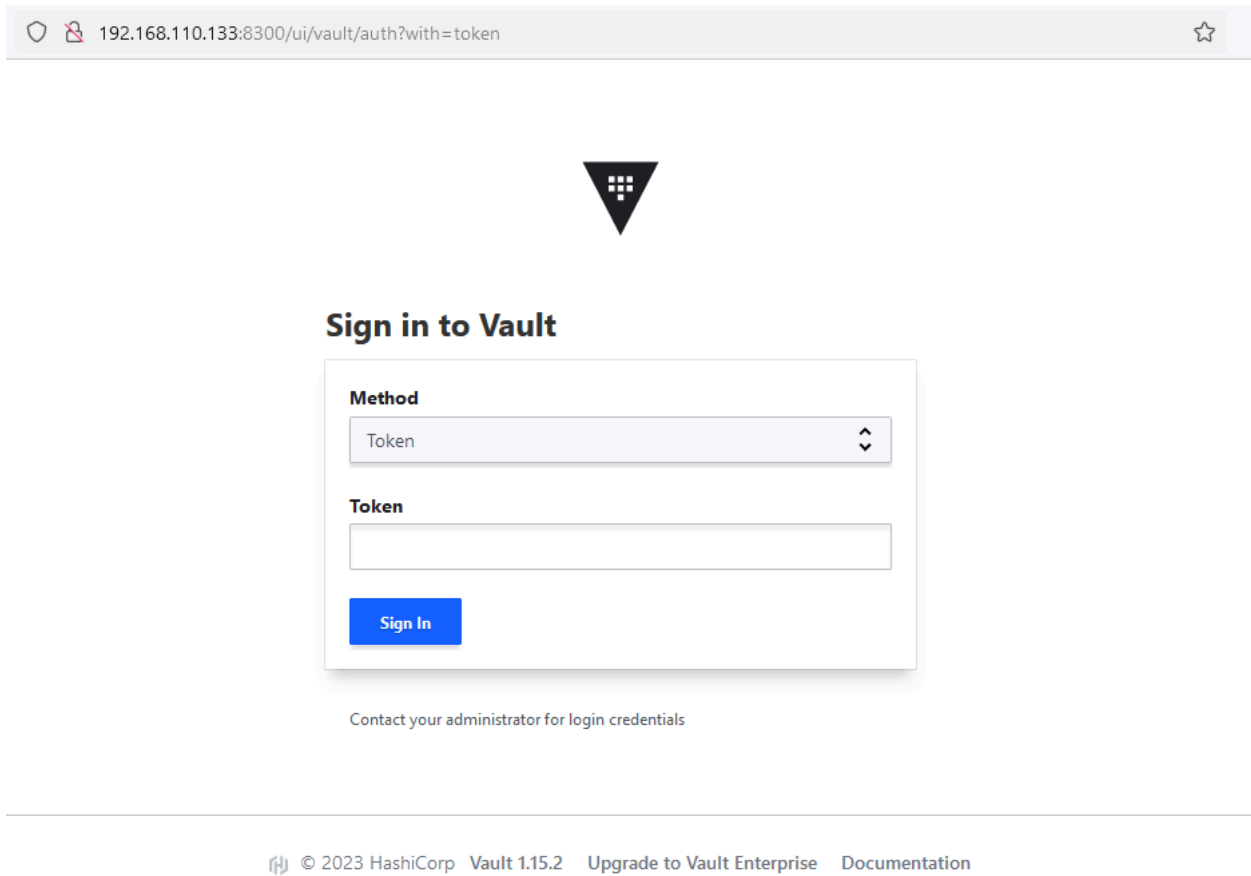
Development mode should NOT be used in production installations!

```

Make note of the Unseal key and Root token, highlighted above and save them somewhere handy.

Step 6: Access vault UI

Access Vault UI by going to `http://<server_ip>:8300` (as mentioned in config file). If everything went well, you should see the below screen.



The screenshot shows a web browser window with the address bar displaying `192.168.110.133:8300/ui/vault/auth?with=token`. The page features the Vault logo, a heading "Sign in to Vault", and a login form. The form includes a "Method" dropdown menu set to "Token", a "Token" input field, and a blue "Sign In" button. Below the form is a link to "Contact your administrator for login credentials". The footer contains the HashiCorp logo, copyright information for 2023, the version "Vault 1.15.2", and links to "Upgrade to Vault Enterprise" and "Documentation".

192.168.110.133:8300/ui/vault/auth?with=token

Sign in to Vault

Method

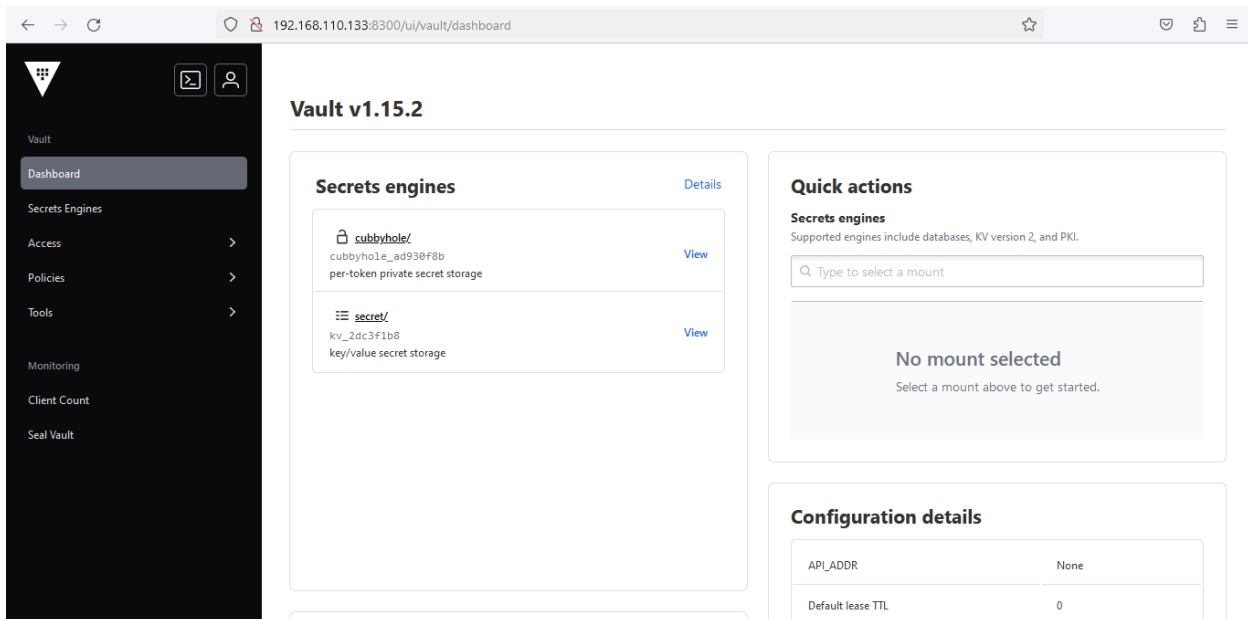
Token

Token

Sign In

[Contact your administrator for login credentials](#)

© 2023 HashiCorp Vault 1.15.2 [Upgrade to Vault Enterprise](#) [Documentation](#)



Or you can also follow the instructions shown below:

- Install Vault on Linux
 - <https://developer.hashicorp.com/vault/downloads>
- Accessing Hashicorp Vault UI
 - <https://developer.hashicorp.com/vault/tutorials/getting-started/getting-started-ui>
- Create a config file config.hcl

```
ui = true
disable_mlock = true

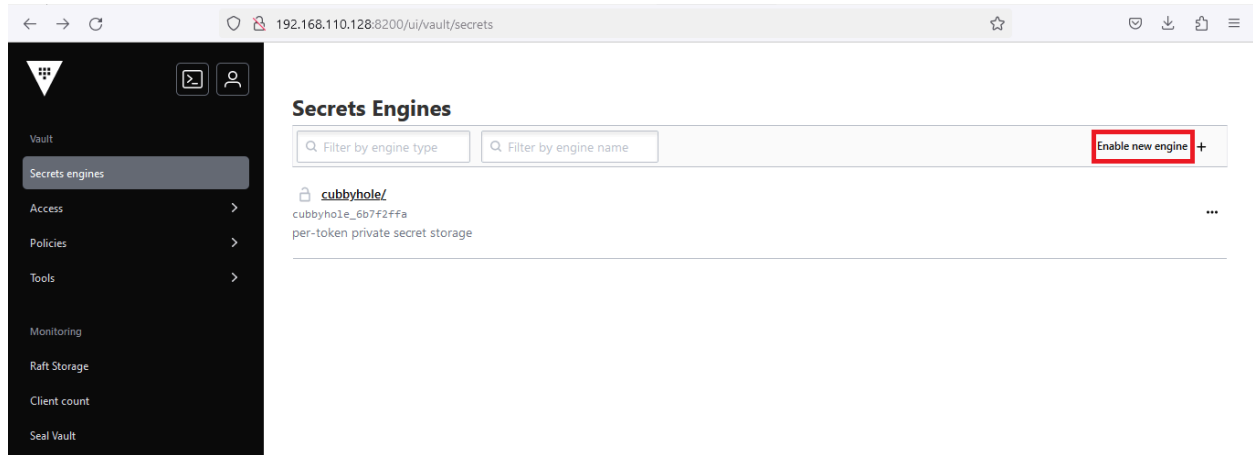
storage "raft" {
  path = "./vault/data"
  node_id = "node1"
}

listener "tcp" {
  address = "0.0.0.0:8200"
  tls_disable = "true"
}

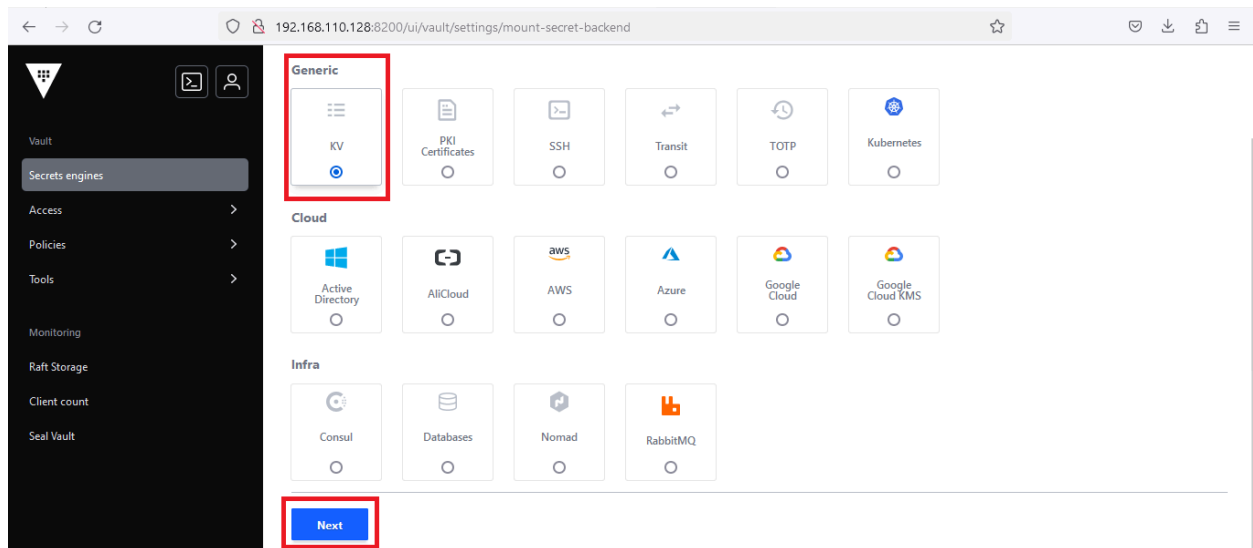
api_addr = "http://127.0.0.1:8200"
```

```
cluster_addr = "https://127.0.0.1:8201"
```

- Start vault server
 - vault server -config=config.hcl
- Enter the number of unseal keys needed to unseal followed by the token to login
- Click on Enable new engine



- Select kv and click next



- Enter path as secret and click Enable Engine

192.168.110.128:8200/ui/vault/settings/mount-secret-backend

Enable a Secrets Engine

Path
secret

Maximum number of versions
The number of versions to keep per key. Once the number of keys exceeds the maximum number set here, the oldest version will be permanently deleted. This value applies to all keys, but a key's metadata settings can overwrite this value. When 0 is used or the value is unset, Vault will keep 10 versions.
0

☐ **Require Check and Set**
If checked, all keys will require the cas parameter to be set on all write requests. A key's metadata settings can overwrite this value.

☐ **Automate secret deletion**
A secret's version must be manually deleted.

Method Options

Enable Engine Back

- Install python library hvac if not already installed
 - pip install hvac
- Write the following code to create a client and interact with the vault engine to create and retrieve secrets

```
import hvac
import sys

client = hvac.Client(
    url = "http://192.168.110.128:8200",
    token = "hvs.CbYIEPGkS5xMR23819IJgnwA"
)

print(client)

create_response = client.secrets.kv.v2.create_or_update_secret(
    path='my-secret-password',
    secret=dict(password='Hashi123'),
)

print(create_response)

# Reading a secret
read_response = client.secrets.kv.read_secret_version(path='my-secret-password')

print(read_response)
```

```
password = read_response['data']['data']['password']

if password != 'Hashi123':
    sys.exit('unexpected password')

print('Access granted!')
```

Section 3.4: Ansible Crash Course

Introduction

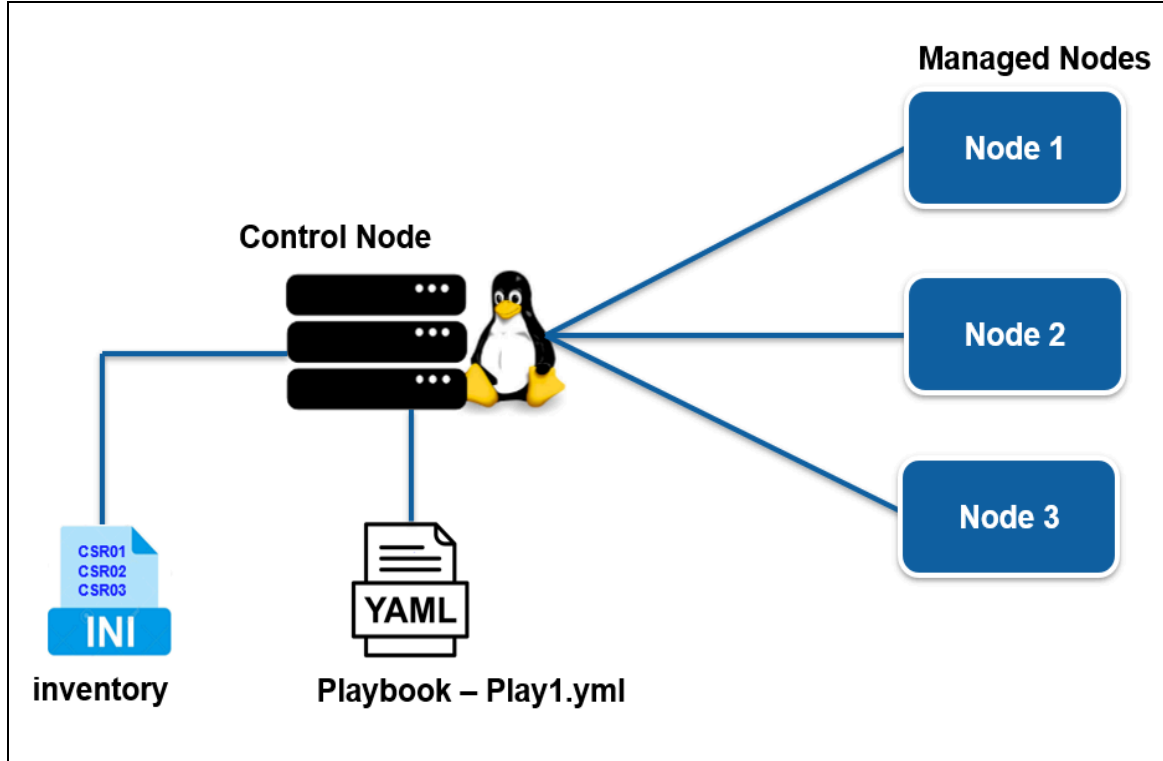
- An open source IT automation tool
- Can automate:
 - Provisioning
 - Configuration management
 - Application deployment
 - Orchestration
 - Other such manual IT processes
- Based on Python
- Can be extended by writing custom modules

How Ansible Works?

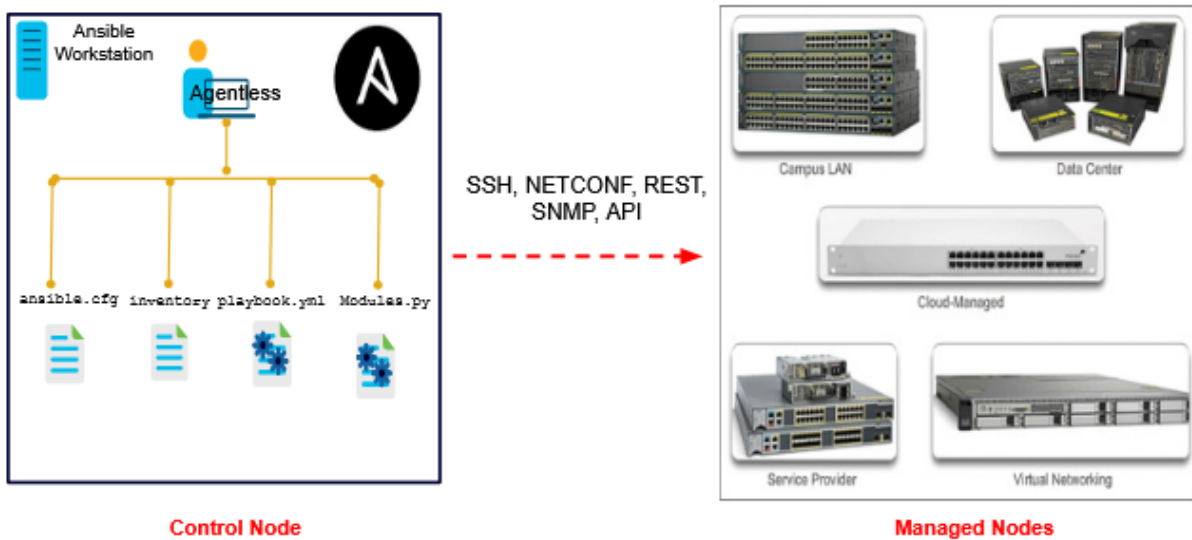
- Written in Python and uses Python programs called Modules to carry out automation tasks
- Connects to the remote systems (servers, network devices etc.) typically via SSH
- If target machine has Python installed, it pushes small programs called modules to execute the instructions written in YAML
- If the target machine does not have python installed, it executes these modules remotely to carry out automation tasks

Ansible Terminology

- Control Node - The machine from which we run Ansible commands
- Managed Nodes - Also known as hosts. These are the target devices (servers, network appliances or any other machines)
- Inventory - List of managed nodes provided by some inventory source or a flat file known as host file



Ansible with Network Devices



Ansible Installation

- <https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-ansible-on-ubuntu-22-04>

Ansible Getting Started

- Create ansible.cfg

```
$ sudo nano ansible.cfg

[defaults]
inventory=./hosts
```

- Create inventory file
- Add hosts to inventory

```
$ sudo nano hosts

192.168.110.129
192.168.110.130
192.168.110.131
192.168.110.132
csr01.example.com
csr02.example.com
csr03.example.com
```

- Check if inventory is properly defined

```
$ ansible all --list-hosts

hosts (7):
    192.168.110.129
    192.168.110.130
    192.168.110.131
    192.168.110.132
    csr01.example.com
    csr02.example.com
    csr03.example.com
```

```
$ ansible-inventory --list
{
  "_meta": {
    "hostvars": {}
  },
  "all": {
    "children": [
      "ungrouped"
    ]
  },
  "ungrouped": {
    "hosts": [
      "192.168.110.129",
      "192.168.110.130",
      "192.168.110.131",
      "192.168.110.132",
      "csr01.example.com",
      "csr02.example.com",
      "csr03.example.com"
    ]
  }
}
```

- Define groups in inventory

```
[cisco_routers]
192.168.110.129
csr01.example.com
csr02.example.com
csr03.example.com

[cisco_switches]
192.168.110.130

[arista_routers]
192.168.110.131
```

```
[arista_switches]
192.168.110.132

[cisco_devices:children]
cisco_routers
cisco_switches

[arista_devices:children]
arista_routers
arista_switches

[network_devices:children]
cisco_devices
arista_devices

[dc1:children]
cisco_routers
arista_switches

[dc2:children]
arista_routers
cisco_switches
```

- Run the same commands as above to see that inventory information is displayed group wise

Host vars and Group vars

- We can use variables (vars) to add additional information about the hosts in the inventory
 - Hostname and IP Address
 - Username and password etc.

```
[cisco_routers]
csr03 ansible_host=192.168.110.131
```

```
[cisco_routers:vars]
ansible_user=cisco
ansible_password=cisco
ansible_network_os=ios
```

- Display the info and check

```
$ ansible-inventory -i hosts --list
{
  "_meta": {
    "hostvars": {
      "csr03": {
        "ansible_host": "192.168.110.131",
        "ansible_network_os": "ios",
        "ansible_password": "cisco",
        "ansible_user": "cisco"
      }
    }
  },
  "all": {
    "children": [
      "cisco_routers",
      "ungrouped"
    ]
  },
  "cisco_routers": {
    "hosts": [
      "csr03"
    ]
  }
}
```

Ad hoc Commands

- Use ad hoc commands to connect to a Cisco router and execute show commands

- Ad hoc commands are like raw commands we can give at the command line by invoking the ansible executable
- These are similar to the commands we give at Linux shell or Python interpreter directly
- Can be used to achieve some quick tasks or extract information
- Let us ping the device we specified in the inventory using an ad hoc command
- Remember that this is not the usual ICMP ping command
- Rather this is a python program written and being used as Ansible module

```
[cisco_routers]
csr03 ansible_host=192.168.110.131

[cisco_routers:vars]
ansible_user=cisco
ansible_password=cisco
ansible_network_os=ios
# ansible_connection=network_cli
```

```
$ ansible csr03 -m raw -a "sh ip int br"
csr03 | CHANGED | rc=0 >>
```

Interface	IP-Address	OK?	Method	Status	Protocol
GigabitEthernet1	192.168.110.131	YES	DHCP	up	up
GigabitEthernet2	unassigned	YES	NVRAM	administratively down	down
GigabitEthernet3	unassigned	YES	NVRAM	administratively down	down
GigabitEthernet4	unassigned	YES	NVRAM	administratively down	down

Shared connection to 192.168.110.131 closed.

- While we can use the raw module to execute any command, it is not recommended as it bypasses ansible's control mechanisms
- Hence we need to use the modules provided for specific purpose like executing read commands (cli_command or ios_command) and executing config commands (ios_config)

```
[cisco_routers]
csr03 ansible_host=192.168.110.131

[cisco_routers:vars]
ansible_user=cisco
ansible_password=cisco
```

```
ansible_network_os=ios
ansible_connection=network_cli
```

- Using `ios_command`

```
$ ansible csr03 -m ios_command -a 'commands="sh ip int br"'
csr03 | SUCCESS => {
  "ansible_facts": {
    "discovered_interpreter_python": "/usr/bin/python3"
  },
  "changed": false,
  "stdout": [
    "Interface          IP-Address  OK? Method Status
Protocol\nGigabitEthernet1  192.168.110.131 YES DHCP    up
up  \nGigabitEthernet2      unassigned  YES NVRAM  administratively down down
\nGigabitEthernet3        unassigned  YES NVRAM  administratively down down
\nGigabitEthernet4        unassigned  YES NVRAM  administratively down down"
  ],
  "stdout_lines": [
    [
      "Interface          IP-Address  OK? Method Status
Protocol",
      "GigabitEthernet1  192.168.110.131 YES DHCP    up
up  ",
      "GigabitEthernet2      unassigned  YES NVRAM  administratively
down down ",
      "GigabitEthernet3        unassigned  YES NVRAM  administratively
down down ",
      "GigabitEthernet4        unassigned  YES NVRAM  administratively
down down"
    ]
  ]
}
```

- Using `cli_command`

```
$ ansible csr03 -m cli_command -a 'command="sh ip int br"'
```

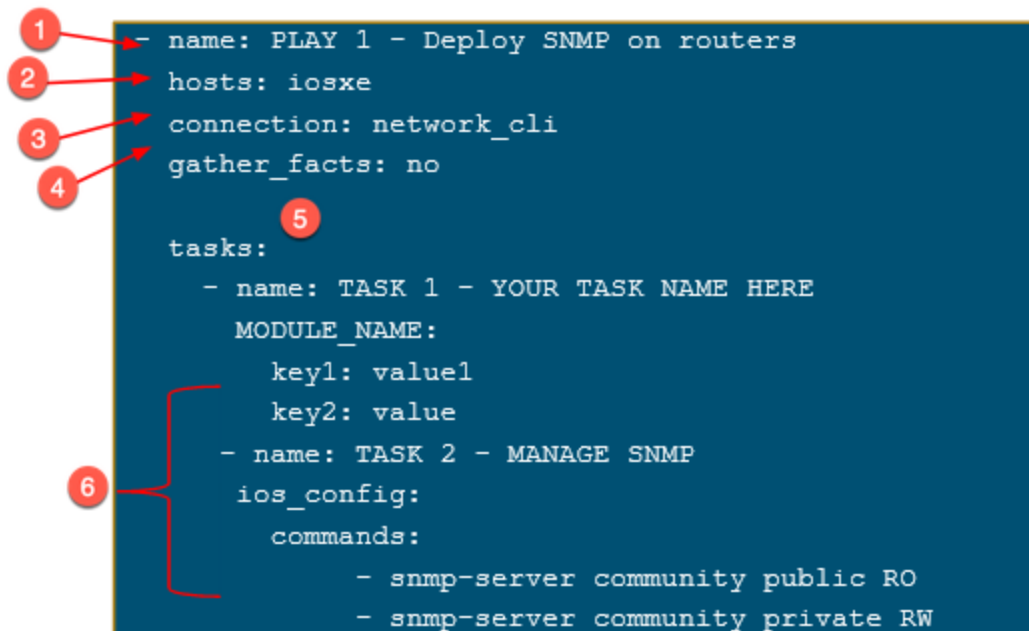
```

csr03 | SUCCESS => {
  "ansible_facts": {
    "discovered_interpreter_python": "/usr/bin/python3"
  },
  "changed": false,
  "stdout": "Interface          IP-Address  OK? Method Status
Protocol\nGigabitEthernet1    192.168.110.131 YES DHCP    up
up  \nGigabitEthernet2        unassigned  YES NVRAM   administratively down down
\nGigabitEthernet3          unassigned  YES NVRAM   administratively down down
\nGigabitEthernet4          unassigned  YES NVRAM   administratively down down",
  "stdout_lines": [
    "Interface          IP-Address  OK? Method Status
Protocol",
    "GigabitEthernet1    192.168.110.131 YES DHCP    up
up  ",
    "GigabitEthernet2        unassigned  YES NVRAM   administratively down down
",
    "GigabitEthernet3          unassigned  YES NVRAM   administratively down down
",
    "GigabitEthernet4          unassigned  YES NVRAM   administratively down
down"
  ]
}

```

Playbooks

- While we can work with ad hoc commands, writing scripts will let us reuse scripts as needed and also give us more flexibility
- Ansible Playbooks are written in YAML (stored as .yaml or .yml files)
- Generic structure of Ansible playbook is as follows



1. Name of the play
2. Specifying hosts
3. Connection mode (optional, can be skipped if it is mentioned in hosts file)
4. A default task that runs for every playbook to gather some useful info about the hosts. Can be turned off by setting it to no.
5. Tasks section
6. Task specification. Each task invokes an Ansible module. Depending on the nature of the task, we need to pass the appropriate arguments

Playbook to run show commands

- Playbook to run show commands

```
---
- name: playbook to run show commands
  hosts: csr03
  gather_facts: no

  tasks:
    - name: running show command using ios_command module
      ios_command:
        commands:
          - sh ip int br
```

```
register: output
```

```
- name: display output
  debug:
    var: output
```

Playbook to run config commands

- Playbook to run config commands

```
---
- name: playbook to run config commands
  hosts: csr03
  gather_facts: no

  tasks:
    - name: running config command using ios_config module
      ios_config:
        lines:
          - description test interface
          - ip address 172.31.2.1 255.255.255.0
        parents: interface GigabitEthernet3
      register: output

    - name: display output
      debug:
        var: output
```

Ansible Playbook to read secrets from Hashicorp Vault

```
---
- name: playbook to run config commands
  hosts: csr03
  gather_facts: no
```

```
tasks:
  - name: get secret from hashicorp vault
    debug:
      msg: "{{ lookup('hashi_vault', 'secret=secret/data/my-secret-password
token=hvs.CbYIEPGkS5xMR238l9IJgnwA url=http://192.168.110.128:8200') }}"
```

Section 4.1: Docker, Docker Compose

Docker

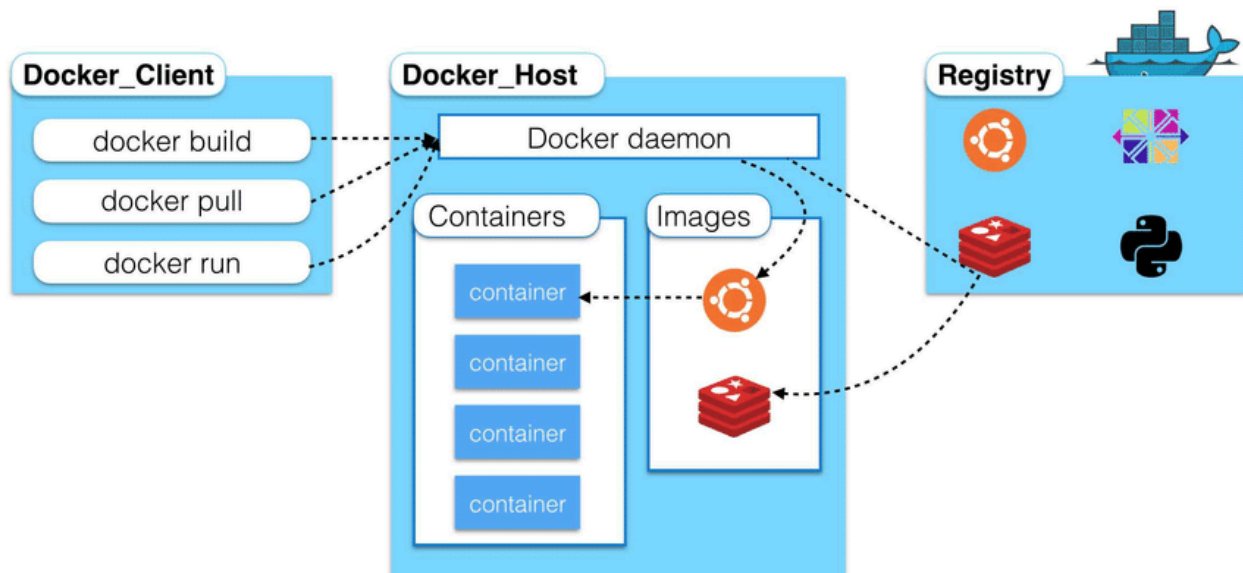
What is Docker?

- Docker is an open-source platform that allows developers to automate the deployment, scaling, and management of applications inside lightweight, portable containers
- Containers are a form of virtualization technology,
- They enable applications and their dependencies to run in isolated environments
- This ensures consistency across different systems and reduces conflicts between different software stacks.

Docker Components

- Docker Engine:
 - The Docker Engine is the core component of Docker.
 - It is a client-server application responsible for building, running, and managing Docker containers.
 - The Docker client communicates with the Docker daemon (server) through a REST API or a command-line interface (CLI).
- Container:
 - A container is a lightweight, standalone executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, and system tools.
 - Containers are isolated from the host system and other containers, making them highly portable and consistent across different environments.
- Image:
 - An image is a read-only template used to create containers.
 - It contains the application code, runtime, libraries, and other dependencies needed to run the application.
 - Images are built based on a Dockerfile, which specifies the instructions for creating the image.

- Dockerfile:
 - A Dockerfile is a text file that contains a series of commands and instructions for building a Docker image.
 - It defines the base image, sets up the environment, installs software packages, and copies the application code into the image.
- Registry:
 - Docker images can be stored and shared in a Docker registry.
 - The Docker Hub is a popular public registry that hosts a vast collection of pre-built images.
 - Organizations can also set up private registries to store their proprietary images securely.
- Containerization:
 - The process of creating and running containers based on Docker images is known as containerization.
 - Containers are isolated from each other and from the host system, ensuring that applications can run consistently across different environments.



- Dockerfile to build Python docker image on top of Ubuntu base image

```
FROM ubuntu:20.04

RUN apt-get update && apt-get install -y --no-install-recommends \
    python3.8 \
    python3-pip \
    && \
    apt-get clean && \
```

```
rm -rf /var/lib/apt/lists/*

RUN pip3 install nibabel pydicom matplotlib pillow
RUN pip3 install med2image
```

- Dockerfile to build a Python App image using Python image as base

```
FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD ["python3", "-m", "flask", "run", "--host=0.0.0.0"]
```

Docker Compose

- Docker Compose is a tool provided by Docker that allows you to define and manage multi-container applications.
- It uses a simple YAML file to define the services, networks, and volumes required for your application, making it easier to orchestrate complex deployments.
- Using a single docker-compose.yml file, you can define all the services required for your application, specifying the Docker images, environment variables, volumes, network settings, and other configuration details for each service.
- Sample docker-compose.yml is as follows which stitches a web server and a database container together required for deploying web applications

```
version: '3.9'
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
  database:
    image: mysql:latest
    environment:
```

```

    MYSQL_ROOT_PASSWORD: secret
volumes:
  - db_data:/var/lib/mysql
volumes:
  db_data:

```

- To start the application defined in the docker-compose.yml file, would use the following command
 - `docker-compose up`

Section 4.1: Zammad Python SDK

Note: Zammad installation is out of the scope of this document. Refer to the official documentation for various installation options and detailed installation instructions for the same. <https://docs.zammad.org/en/latest/about/zammad.html>

- Zammad is a free helpdesk or issue tracking system. It offers the connection of various channels like email, chat, telephone, Twitter, or Facebook.
- Zammad is developed in the programming languages Ruby and JavaScript. The name Zammad comes from the Bavarian language and means "together".
- A REST API is a secure way to integrate channels and applications to your Zammad helpdesk.
- We offer an easy way by using JSON (JavaScript Object Notation) as a data format.
- It ensures efficient and reliable data exchange between both sources and allows you to tailor your helpdesk with your own integrations.
- With Zammad, "API First" applies. This means that you can do anything with the REST API that can be done through the interface.
- Python API client for zammad is available on Pypi and can be installed using
 - `pip install zammad-py==0.1.6`
- Once installed you can import into python code as follows:
 - `from zammad_py import ZammadAPI`
- Here is a quick start script to check connection to Zammad server and extract user details like login, firstname, lastname

Get User Script

```
from zammad_py import ZammadAPI

HOST="http://dev-zammad.ppm.example.com/api/v1/"

USERNAME='expert@ppm.example.com'

PASSWORD='1234QWer!'

client = ZammadAPI(url=HOST, username=USERNAME, password=PASSWORD)

print(client.user.me())
```

Get User Output:

```
$ python zammad1.py
User Login: expert@ppm.example.com
First Name: expert
Last Name: expert
```

Create TicketScript

```
# Example: Create a ticket

params = {

    "title": "Help me!",

    "group": "Users",

    "customer": "sara.stone@ppm.example.com",

    "article": {

        "subject": "My subject",

        "body": "I am a message!",

        "type": "note",

        "internal": "false"

    }

}
```

```
}  
  
new_ticket = client.ticket.create(params=params)  
  
print(json.dumps(new_ticket,indent=2))
```

Create Ticket Output:

```
{  
  "id": 28,  
  "group_id": 1,  
  "priority_id": 2,  
  "state_id": 1,  
  "number": "48028",  
  "title": "Help me!",  
  "owner_id": 1,  
  "customer_id": 6,  
  "create_article_type_id": 10,  
  "create_article_sender_id": 1,  
  "article_count": 1,  
  "preferences": {},  
  "updated_by_id": 4,  
  "created_by_id": 4,  
  "created_at": "2023-11-25T05:08:17.470Z",  
  "updated_at": "2023-11-25T05:08:17.543Z",  
  "article_ids": [  
    28  
  ],  
  "ticket_time_accounting_ids": []  
}
```