Title: Automating SAP Deployments with Terraform and Ansible

Presenter: Darpan Patel
Kick Off Automated SAP S/4HANA Build
Project Thunder

- Join [here](#) if you are not a member so you don’t miss updates regarding this topic and many more!
- Blog Post on this topic will be released beginning of Q2
Agenda

- Why learn these tools?
- Getting Started with Terraform
- Getting Started with Ansible
- Merging Terraform and Ansible
Why Terraform and Ansible?
SAP Implementations

- Implementation Issues:
  - Technical Implementations can become very lengthy
  - They are a bottleneck for the project
  - Involve a very manual process
    - Both Provisioning servers and applications
  - Outdated in today’s quick paced and automated world

- How do we make implementations more efficient?
  - Automate
    - Provisioning Servers
    - Resources that attach onto those servers
    - The networks they reside in
    - OS configurations
    - Application Deployments
- Infrastructure as Code
- Use to Deploy the Hosts for SAP application servers
- Database server hosts
- Storage
- Networks, VPC’s, Subnets, routers
- Security Groups, NACL’s

- Deploy applications
- Install Packages for OS
- Configure your file system
- Mount Network locations
- Install SAP HANA
- Any task SWPM can do
- System Copy, System Rename, Distributed System, Standard System
Getting Started with Terraform
Terraform Implementation Considerations

- Terraform is logically split into two main parts:
  - Terraform Core
  - Terraform Plugins
- Terraform is Agentless
- Terraform can be installed on any platform
- Installation Options
  - Manual
  - OS Package Manager
- Terraform Core and Plugins are written in the GO Language
- Terraform uses CRUD (create, read, update, and delete) API's to communicate with providers

- Statically-compiled binary written in GO
- Terraform core is the command line tool
- Entry point for anyone using terraform
- Resource State Management
- Communicates with plugins

- Terraform plugins are executable binaries written in GO
- Plugins contain Providers and Provisioners
- Providers provide a service such as AWS
Terraform Execution Flow

- Executing a Terraform Script to provision in AWS:
  - 1 – Execute terraform against AWS.tf
  - 2 – Core makes Remote Procedure Call to Plugin to download AWS Provider and provide resources to provision
  - 3 – Plugin downloads AWS Libraries- ADD where the request goes
  - 4 – AWS Client library translates requests to API requests
  - 5 – API requests sent to AWS
  - 6 – Once provisioning is complete, state data is logged in terraform.tfstate
Terraform Providers

AWS.tf Terraform File:

```terraform
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "3.31.0"
    }
  }
}
provider "aws" {
  # Configuration options
}
```

- Terraform Providers:
  - To install a provider you must include provider configuration code into your terraform files
  - A list of available providers can be found [here](#) on the terraform website
  - Each Provider comes with its own set of documentation describing its resource types and their arguments
  - Terraform currently has 70 providers in their registry
  - You can create your own provider
Terraform Resources

Resources:

- Resources are the most important element in the Terraform language
- They represent some type of infrastructure object
  - virtual networks, compute instances, compute devices or DNS records.
- Terraform is platform dependent
- Each provider has different requirements for parameters
- AWS requires – ami ID, and Instance_type

Resource Template:

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG = ]
}
```

AWS.tf – Create an EC2 Instance:

```
resource "aws_instance" "example" {
  ami = "ami-0c09927562c939f41"
  instance_type = "t2.micro"
  tags { name = "TESTVM" }
}
```
Terraform Variables

AWS.tf Terraform File with Variables:

```tf
resource "aws_instance" "example" {
  ami = var.amiID
  instance_type = var.instanceType
  tags = { name = var.Tags }
}
```

Variables.tf File:

```tf
variable "amiID"{
  type = string
  default = "ami-0c09927662c939f41"
}

variable "instanceType"{
  type = string
  default = "t2.micro"
}

variable "Tags"{
  type = string
  default = "TEST_VM"
}
```

- **Variables:**
  - Variables can be used to make your terraform scripts re-usable
  - Avoid having to hardcode in your parameters by using variables
  - Easier to Maintain – Instead of maintaining script file, you just need to maintain variables file

- **Create Variables File:**
  - Create a file in the same directory as your provisioning script and title it variables
Terraform Best Practices for File Structure

Example File Structure:

```
darpan-test:/terraform # tree -1
.
  dev
  |   AWS.tf
  |   variables.tf
  prod
  |   AWS.tf
  |   variables.tf
  qa
  |   AWS.tf
  |   variables.tf
3 directories, 6 files
darpan-test:/terraform #
```

- **File Structure:**
  - Separate out clusters of Servers
  - An example would be breaking out your environments to include DEV, QA, and PROD
  - Each environment has its own execution script – AWS.tf
  - Can also create groups such as Active Directory Servers, DNS servers, Printer Servers, etc..

- **Drawbacks to this:**
  - Every time you upgrade or modify one AWS.tf, you will need to modify the others
**Terraform Modules**

- **Modules**
  - Don’t need to update provisioning scripts in multiple locations
  - All resource provisioning scripts will be put into the module
  - Useful when managing multiple environments with different types of servers
  - Modules can be shared to the community [here](#)

- **Use Case**
  - You want to build out another application server for your S/4 Dev Environment
  - Update `/dev/AWS.tf` to include the module `/modules/s4-server/application`
  - To execute – cd into `/dev` and run: `terraform apply`
Getting Started with Ansible
Ansible Implementation Considerations

- Types of Resources in Ansible
  - Control Node
  - Managed Node
- Ansible is Agentless
- Installation Options
  - Install using OS package Manager
    - RHEL
    - SUSE
  - Install with Python Package Manager `pip`
- Control Node uses `ssh` to communicate with Managed Nodes
- Ansible Modules/Libraries are written in python

Control Node

- Cannot be a Windows System
- Can be a laptop, shared desktop, or server
- RHEL, SUSE, macOS, Ubuntu, other Linux operating systems
- Control Node should be located on the same network as Managed Nodes
- Must have python installed, python 3 is recommended

Managed Node

- Can be any operating system, including Windows
- Must have python installed, python 3 is recommended
Ansible Execution Flow

- Authentication
  - SSH Keys or Passwords
- Hosts File
  - Contains the list of Managed Nodes
- Playbook
  - Contain Tasks to execute against the hosts in Hosts file
Ansible Hosts File

- Hosts File
  - Host file can be written in YAML or INI format
  - Managed nodes can use FQDN, Short Name, or IP Addresses
  - Group out your Servers for segregation of environments
  - Make groups of groups using :children suffix to make your execution calls more robust
  - Execute Ansible Playbooks against a server, a group of servers, and groups of groups of servers
  - More Information on Hosts files here
Ansible Playbook Files

- Playbook Files
  - Playbooks are written in YAML format
  - You must specify a host or a group of hosts to run the play against
  - Define your task by providing a name, module and parameters for each task
  - Playbooks define the execution flow of your Ansible tasks

- Execute Playbook command:
  - Ansible-playbook <playbook_name>

- Playbook Drawbacks
  - Can get messy when you have too many tasks

```yaml
- hosts: dev
tasks:
  - name: Install SAP Hana Prerequisite Packages for SUSE Linux 15
    zypper:
      name: libgcc_s1
      state: present
  - name: Install SAP Hana Prerequisite Packages for SUSE Linux 15
    zypper:
      name: libstdc++6
      state: present
  - name: Install SAP Hana Prerequisite Packages for SUSE Linux 15
    zypper:
      name: libatomic1
      state: present
```
Ansible Roles

- **Roles**
  - Provide a clean way to organize your Ansible scripts and not overpopulating your playbooks
  - Treat roles like functions in programming
    - Should perform one task or a group of similar tasks
  - Roles are reusable and can be shared to your team or the [Ansible Galaxy Community](https://galaxy.ansible.com)

- **Role Creation**
  - In the same directory your playbook can be found create a folder called roles and change directory into it
  - Run `- ansible-galaxy init <Role_Name>`
  - Folder structure for role is created

New look Playbook:

```yaml
- hosts: dev
  roles:
    - [ role: sap-configure ]
```

Role task file:

```yaml
---
# tasks file for sap-configure
- name: Install SAP Hana Prerequisite Packages for SUSE Linux 15
  zypper:
    name: libgcc_s1
    state: present

- name: Install SAP Hana Prerequisite Packages for SUSE Linux 15
  zypper:
    name: libstdc++6
    state: present

- name: Install SAP Hana Prerequisite Packages for SUSE Linux 15
  zypper:
    name: libatomic1
    state: present
```
Ansible Variable Files

- Variable File
  - Use variables to manage differences between systems
  - Prevents you from having to hard code in your playbook and role files
  - Create variables with YAML syntax, lists, or dictionaries
  - Define variables in playbooks, hosts file, roles, at the command line, or in separate vars files
  - Make all variables in vars file useable in your playbooks and roles

- Variable Drawbacks
  - Sensitive variables need to be secured as they are visible to users

Playbook File calling vars file:

```
- hosts: dev
  vars_files:
    - /etc/ansible-vars/ansible_vars.yml
  vars:
    ansible_ssh_private_key_file: "/{{ private_key }}"
  roles:
    - { role: sap-configure }
```

Vars File:

```
# Secret Variables for Playbook
{
  "private_key": "/etc/ansible/dpid_rsa"
}
```
Ansible Vault

- Vault
  - Encrypt and Decrypt your sensitive files with Ansible Vault
  - Before files are encrypted you must provide a password
  - When executing playbooks, if an ansible vault encrypted file exists, you will be prompted for the vault password
  - Files are decrypted during runtime only for Ansible to use in execution
  - Create multiple vault passwords for encrypting different sets of files

- Vault Encrypt Command
  - ansible-vault encrypt <file_path1> <file_path2> ....

Vars file before encryption:
```
{
    "private_key": "/etc/ansible/dpid_rsa"
}
```

Vars file after encryption:
```
1 | $ANSIBLE_VAULT:1.1;AES256
2 | 313351239661334393164313866139626666363333396446668643613636361365
3 | 393611319636566366164763643352336341383232a3836162626133556361323
4 | 326364353930965350326346661631636568637633730335368163792661238623
5 | 35353613134366261640a623438663937636261343336376434323623636336138314663
6 | 663613335930663613536386669061626432446466623665313656
7 | 386238623356632636566536135336603366134316566357838305126564
8 | 61643965333337272666123553378681375866394393731626585811313556341396
9 | 3932656264761663626663686136313832566131340383766438333962323332343
10 | 61343236137622372372372345666393263633983161365562627373565646593938
11 | 335595856531535334973960476664343238229963258335322621537363653
12 | 6432313365641363313353838467356561561363353330565363763733337232862326
13 | 3164396133861366638320328873935328333106136553253133265363623651413
14 | 61313266265666646623333629966343082686466337626392363663263934613033
15 | 6653231334530649633313233838656238593838536616135634329313862527383
16 | 38386646164161139386273766613132635333873153613264643483686353531316
17 | 3926644643563565366463336439135455333664346265388353866861696666
18 | 3464263838288644339966163563432653953595965553527642956
```

23
Ansible File Structure

- **File Structure**
  - Ansible can be run from anywhere
  - Bare minimum you need a playbook and hosts file
  - Group Vars gives you the ability to associate variables with a specific group of hosts
  - Host Vars gives you the ability to associate variables to specific hosts
Use Ansible to Call Terraform

Combining Terraform and Ansible File Structures:

sap-configure.yml calls Terraform:

```yaml
- hosts: localhost
  tasks:
    - name: init terraform
      shell: terraform init
      args:
        chdir: "/etc/ansible/terraform/dev"
    - name: apply terraform script
      terraform:
        project_path: "/etc/ansible/terraform/dev"
        state: present

- hosts: dev
  vars_files:
    - /ansible/vars/ansible_vars.yml
  vars:
    ansible_ssh_private_key_file: "{{ private_key }}"
  roles:
    - { role: sap-configure }
```
How Ansible Performs SAP Installations

Hdblcm In Batch Mode and SWPM Unattended:

- Config File needs to be maintained with parameters used for HANA and S/4 Installations
- You can write a playbook that picks up this file and inputs it into an hdblcm command or sapinst command
- No need to reinvent the wheel – download the following ansible roles
  - SAP HANA
  - SAP S/4HANA
- For SWPM, follow SAP Note - [2230669](https://support.sap.com) to generate config files

Config File for SWPM:

```yaml
# Standard system with AS ABAP only: ASCS instance number. Leave empty for default.
NW_CI_Instance.ascsiInstanceNumber = {{ sap_s4hana_deployment_ascsi_instance_nr }}

# Standard system with AS ABAP only: Virtual host name for the ASCS instance. Leave empty for default.
NW_CI_Instance.ascsiVirtualHostname = {{ ansible_hostname }}

# Instance number of the primary application server instance. Leave empty for default.
NW_CI_Instance.ciInstanceNumber = {{ sap_s4hana_deployment_pos_instance_nr }}
```