Automation With Ansible

Introduction

Poll Question 1

- Which of the following statements best applies to you
  a) I am new to Ansible
  b) I have just learned the basics and now want to get more in-depth knowledge about Ansible
  c) I have been working with Ansible for some time already
  d) I consider myself very experienced with Ansible

Poll Question 2

- Did you attend my "Ansible Fundamentals" course on Safari Live?
  a) yes
  b) no

Automation With Ansible

About this Course

Agenda (subject to Change)

- Getting Started Lab
- Best Practices
- Using Roles
- Dynamic Inventory
- Advanced Features
- Managing Network Devices
- Managing Windows
- Managing Cloud (optional)
- Ansible Tower

Course Files

- The demo files used in this course are available on github
- Use **git clone https://github.com/sandervanvugt/ansible-advanced** to download

Lab Environment

Lab Environment

- ansible-control: CentOS system that is set up for managing all nodes but not itself
- ansible1: CentOS system that has been set up for management through Ansible
- ansible2: CentOS system that has been set up for management through Ansible
- windows: basic installation of Windows Server 2016 standard, not set up for management yet

Getting Started Lab

Lab - part 1

1. Create an Ansible configuration that sets up hosts Ansible1 and Ansible2 for automatic installation. Create custom facts for both hosts and use variable inclusion to realize this. To configure ansible1, use a host group with the name "file", to configure ansible2, use a host group with the name "lamp"
2. Create a file with the name custom.fact that defines custom facts. In this file, define two sections. The section package contains the following:
   smb_package = smb
   ftp_package=ftp
   db_package=mariadb
   web_package=http
   The section service contains service variables for the packages mentioned above. Use the name smb_service etc. and set the variable to the appropriate name of the service
3. Create a playbook with the name copy_facts.yml that copies these facts to all managed hosts. Define a variable with the name "remote_dir" and a variable with the name "fact_file" and use these. Use the file and copy modules.
4. Run the playbook and verify it worked

Lab - part 2

5. Create a variable inclusion file with the name ./vars/allvars.yml and set the following variables
   web_root: /var/www/html
   ftp_root: /var/ftp
6. Create a tasks directory in the project folder. In this directory, create two YAML files, one that installs, starts, and enables the LAMP services; and one that installs, starts, and enables the file services
7. Create the main playbook that will set up the lamp servers and the file servers with the packages they need, using inclusions to the previously-defined tasks file. Also, ensure that it opens the firewalld firewall to allow access to these servers. Finally, the web service should be provided with an index.html file that shows "managed by Ansible" on the first line
8. Run the playbook
9. Use ad hoc commands to verify the services have been started

Lab Solution

1. Create the inventory file (see lab-inventory)
2. see custom.fact
3. see lab-copy-facts.yml
4. ansible-playbook -i lab-inventory lab-copy-facts.yml; ansible -i lab-inventory all -m setup -a 'filter=ansible_local*'
5. see lab-vars/allvars.yml
6. see lab-tasks/lamp.yml and file.yml
7. see lab-playbook.yml
8. ansible-playbook lab-playbook.yml
9. ansible lamp -a 'systemctl status mariadb'; ansible file -a 'systemctl status vsftpd'

Files and Directories Best Practices

Organizing Ansible Contents

- Even simple projects should have their own directory structure
- Within that directory structure, you'll have an ansible.cfg, inventory as well as playbooks
- If the project grows bigger, variable files as well as includes may be used
- Roles can be used to standardize and easily re-use specific parts of Ansible
- Consider a role a complete project dedicated to a specific task that is going to be included from the main playbook

Directory Layout Best Practices

- Ansible Documentation describes best practices (https://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html)
- Some highlights:
  - On top in the directory, use site.yml as the master playbook
  - From site.yml, call specific playbooks, such as webservers.yml etc.
  - Use different inventory files to differentiate between production and staging
  - Use group_vars/ and host_vars/ to set host related variables
  - Use roles to standardize common tasks

Using Roles

Understanding Roles

- Ansible roles provide uniform ways to load tasks, handlers, and variables from external files
- A role typically corresponds to the type of service that is offered (web, database, etc.)
- The purpose is to keep the size of playbooks manageable
- Roles use a specific directory structure, with locations for defaults, handlers, tasks, templates, and variables
- Many Roles are provided through the Ansible Galaxy community and can be created manually

Using Roles

- While working with roles, generic profiles are defined in a role
- For specific (groups of) servers, specific playbooks may be created to include one or more roles
- To manage what should happen, default variables are set in the role, which can be overwritten at a playbook level
- Roles are defined in a roles directory, which is created in the project directory (and can have alternative locations also)
- Jinja2 templates are very useful in roles, as they allow working with flexible parameters that are set as variables or facts

Role Directory Structure Contents

- **defaults**: contains a main.yml with default values for variables
- **files**: static files that are referenced by role tasks
- **handlers**: contains a main.yml with handler definitions
- **meta**: contains a main.yaml with information about the role, including author, license, platforms, and dependencies
- **tasks**: has a main.yml file with task definitions
- **vars**: has a main.yml file with role variable definitions

Role Locations

- Ansible will look for roles in different locations:
  - Current project directory
  - ~/.ansible/roles
  - /etc/ansible/roles
  - /usr/share/ansible/roles
- In case of conflict, the most specific location wins

Order of Execution

- Normally, tasks in a role execute before the tasks of the playbook using them
- Two solutions to override that
  - **pre_tasks** are performed before roles are applied
  - **post_tasks** are performed after completing all roles

Creating Roles

- Creating Roles involves the following steps
  - Create the role structure
  - Define the role content
  - Use the role in a playbook
  - Tip! Use the **ansible-galaxy --offline** utility to automate creating the role directory structure
- Each role has its own directory with specific subdirectories that exists in ~/roles and not in specific project directories
- Subdirectories that are not used may be empty

Using Roles: DEMO - part 1

- In this demo we're going to create a simple role that allows us to understand generic role structure and working.
- We'll create a role that specifies content for the /etc/motd file and call that from the generic project playbook
- To start with, we'll create the role structure
  - **mkdir -p ~/roles-lab/roles**
  - **cd ~/roles-lab/roles**
  - **ansible-galaxy init --offline motd**
  - **tree**

Using Roles - DEMO - part 2

- Now we need the main.yml file for the role

```
---
# tasks file for motd
- name: copy motd file
  template:
    src: templates/motd.j2
    dest: /etc/motd
    owner: root
    group: root
    mode: 0444
```

Using Roles - DEMO - part 3

- Next, we're using a Jinja2 template to define the contents of the motd file

```
Welcome to {{ ansible_hostname }}

This file was created on {{ ansible_date_time.date }}
Go away if you have no business being here

Contact {{ system_manager }} if anything is wrong
```

Using Roles - DEMO - part 4

- As we want to understand variable priority, we'll now define a default variable in the rol in defaults/main.yml
- If the same variable is defined at a lower level (playbook) it will be overwritten

```
---
# defaults file for motd
system_manager: anna@example.com
```

Using Roles - DEMO - part 5

- After defining this minimal role contents, we can write a playbook that's going to use it
- After writing the playbook, run it: **ansible-playbook motd.yml**

```
---
- name: use motd role playbook
  hosts: ansible2.example.com
  user: ansible
  become: true

  roles:
    - role: motd
      system_manager: amy@example.com
```

Using Ansible Galaxy Roles - DEMO

- Use **ansible-galaxy install geerlingguy.nginx**
- Create a simple playbook according to roles-lab/nginx-role.yml
- Run the playbook, using **ansible-playbook nginx-role.yml**

Using Dynamic Inventory

Understanding Inventory

- Inventory is a list of managed devices
- Inventory groups allow for grouping specific device types
  - Like Cisco IOS switches
  - Or Windows servers
- Static inventory works for small environments
  - Default in /etc/ansible/hosts
  - It's common to use project-based inventories specified in ansible.cfg or using the -i option
- Dynamic inventory is required in large and dynamic environments
  - Dynamic inventory requires an executable script to be started
  - Often there's an ini file that is used as the configuration file for the script

Using Inventory Parameters

- If hosts need specific parameters to connect to them, you can use specific inventory parameters
  - ansible_host: hostname of IP address to connect to
  - ansible_port: port to SSH to
  - ansible_user: user to SSH to
  - ansible_password: password to use for SSH connection
- Some less common parameters exist as well
- Defaults can be set in ansible.cfg, overrides for specific hosts can be set in static inventory

```
[vagrant]
vagrant1 ansible_host=127.0.0.1 ansible_port=2022
vagrant2 ansible_host=127.0.0.2 ansible_port=2023
```

Using Static Inventory in a Smart Way

- Define inventory groups based on host functionality, but also based on geographical location so that hosts can be addressed in different ways
- To apply the same configuration to different servers to distinguish between staging and production servers, you could use different host groups in inventory
- Best practice: instead of using host groups, it's better to use different inventory files and use -i to include the appropriate file

Understanding Dynamic Inventory

- When using the **ansible** command, use **-i** followed by the name of the dynamic inventory script you want to run
- If the file is executable, it will be treated as dynamic inventory
- Alternatively, specify the location of the dynamic inventory in the ansible.cfg file
- Different scripts are available for different environments
  - see https://github.com/ansible/ansible/tree/devel/contrib/inventory
- It's possible to develop your own inventory scripts
  - Often written in Python
  - Make sure the script respects a --list argument

Dynamic Inventory Platforms

Scripts are available for multiple platforms

- Private clouds such as OpenStack
- Public Cloud: AWS, Azure, Google Compute Engine
- Virtualization platforms such as vSphere and oVirt
- PaaS solutions such as OpenShift
- Management solutions such as Spacewalk and Katello

Managing Multiple Inventories

- If a directory name is passed as the inventory, all files in that directory are used as the inventory files
- These can be dynamic inventory files mixed with static inventory files
- If using multiple inventory files, make sure there are no dependencies between files and all files are self-contained

Using **ansible-inventory**

- The **ansible-inventory** command can be used to show current inventory
- Use **ansible-inventory --list** to show what's in your current inventory (JSON format)
- Use **ansible-inventory --graph** to show the same in tree-shape format

Advanced Features: Using Handlers

Limitations to Handlers

- If a task triggers a handler, but a task after that task fails, the handler will never run
- Use force_handlers to overwrite that behavior
  - Use the --force-handlers command line option
  - Set force_handlers = True as a default in ansible.cfg
  - Use force_handlers: True in a play

Advanced Features: Using **when**

Combining Loops and Conditionals

- Ansible facts may present a dictionary with multiple values
- In that case, you can iterate through each value until a specific condition is met and use the result in a when statement

```
...
- name: install vsftpd if sufficient space on /var/ftp
  package:
    name: vsftpd
    state: latest
  with_items: "{{ ansible_mounts }}"
  when: item.mount == "/var/ftp" and item.size.available > 100000000
```

Advanced Features: Using
async and **waitfor**

Understanding Parallelism

- Running tasks in parallel will make Ansible faster
- Ansible can run tasks in parallel on all hosts
- By default, tasks can run on 5 hosts at once
  - Set **forks = nn** in /etc/ansible/ansible.cfg to increase this number
  - Alternatively, use the **--forks** option with the **ansible-playbook** or **ansible** command
- Use the **serial** keyword in a playbook to reduce the number of parallel tasks to a value that is lower than what is specified with the **forks** option

Understanding Asynchronous Tasks

- Normally, Ansible waits for completion of tasks before starting the next task
- Use the **async** keyword in a task to run the task in the background
  - **async: 3600** tells Ansible to give the task an hour to complete, note that this will be the maximum amount of time permitted for the job to run
  - **poll: 10** indicates that Ansible will poll every 10 seconds to see if the command has completed
- Using **async** allows the next task to be started so it will make playbooks more efficient
  - Recommended for backup jobs, yum updates, large file downloads, etc.

Using **wait_for**

- The **wait_for** module can be used in a task to check if a certain condition was met
- Using this module may be useful to verify successful restart of servers, etc.
- Use **poll: 0** in a task to tell Ansible not to wait for completion of this task, but to move on to the next task. Add ignore_errors as well, to prevent an error condition arising and have this task fail

Managing Network Devices

Managing Network Devices

- Network devices need to be identified in inventory
- Using dynamic inventory is suggested
- Connecting to network devices works on SSH in most cases

Understanding Network Device Modules

- Modules exist for different platforms, and module names are structured as <name>os_function, like ios_facts
- ***os_facts** devices are used to gather facts from network devices
- ***os_command** is used to issue commands on network devices
- ***os_config** is used to configure network devices
- ***os_l3_interface** is used to configure layer 3 interfaces

Managing Windows

Setting up a Windows Host

- Supported Windows versions
  - Windows 7, 8.1, and 10
  - Windows Server 2008, 2012, and 2016
- Required Software
  - Powershell 3.0 or later
  - .NET 4.0 or later
- A WinRM Listener should be created and activated
- Note that some modules may have different requirements

Managing Windows - 1

- Install Windows 2016 server standard, ensure there is a Windows user with admin privileges (will be created automatically)
- login as Admin
- Open powershell: **winrm quickconfig**
- Set up WinRM, using the script that is provided on docs.ansible.com/ansible/2.5/user_guide/windows_setup.html in WinRM Setup
- In Windows control panel, create Ansible user, set password and make this user an Administrator

Managing Windows - 2

- On Ansible control, create windows project directory with an ansible.cfg and an inventory
- Set up /etc/hosts for host name resolution to the windows box - notice that Windows firewall disallows ping incoming
- On ansible control: **sudo pip install pywinrm**
- On ansible control: **ansible win -i inventory -m win_ping**
- **ansible-playbook playbook.yml**
- Verify on Windows by showing the users in the control panel

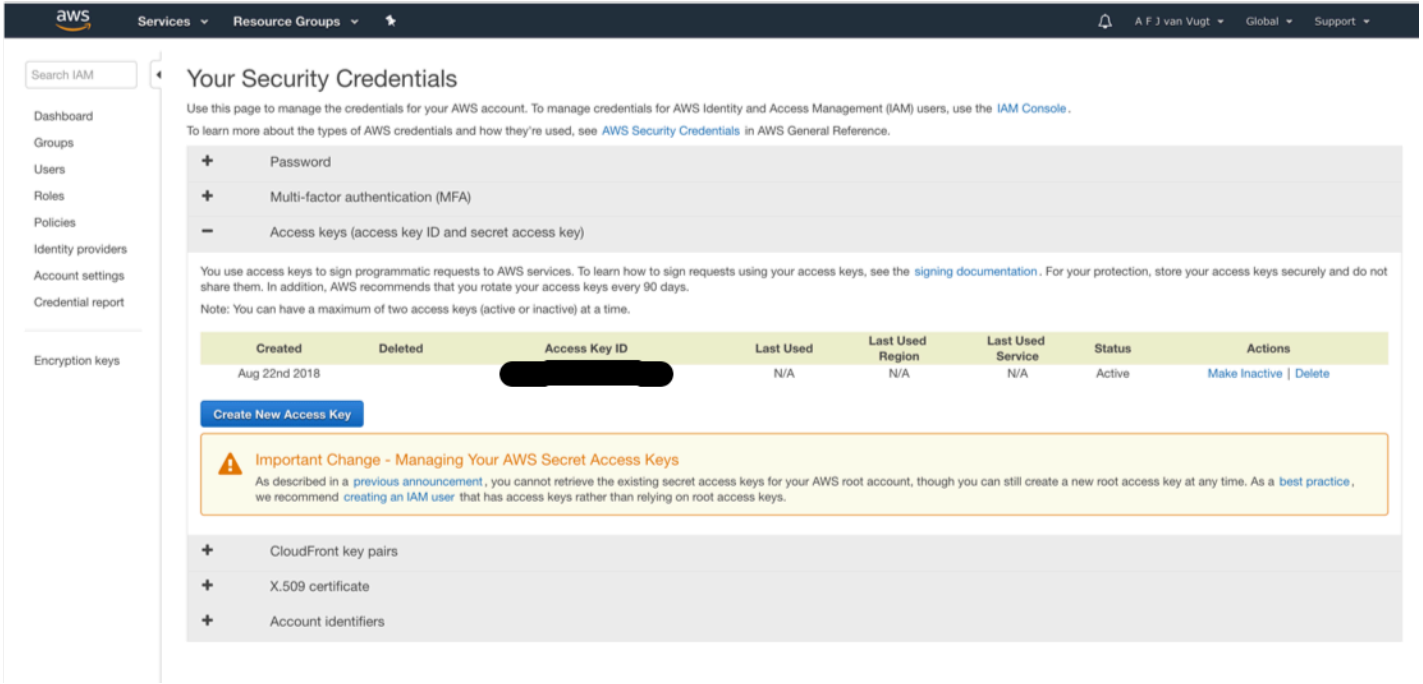
Managing AWS

Understanding the Process

- Cloud instances need to be discovered, using dynamic inventory
- Specific Ansible modules exist for working with instances in different public cloud environments
  - Hundreds of modules exist that relate to the different cloud platforms

Step 1: Defining Access Credentials

- Log in to AWS console, and in your account setting generate an access key



Step 1: Defining Access Credentials - continued

- Specify the AWS access keys as variables in the .bash_profile file in your home directory

```
export AWS_ACCESS_KEY_ID=123
export AWS_SECRET_ACCESS_KEY=abc
```

- At this point you can start using dynamic inventory as well as Ansible modules

Step 2: Installing Python Boto

- To access AWS, you need to install the Python Boto library
  - **pip install boto**
- Next, verify connectivity to EC2 using a python shell

```
[ansible@ansiblecontrol ~]$ python
Python 2.7.5 (default, Jul 13 2018, 13:06:57)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import boto.ec2
>>> conn = boto.ec2.connect_to_region("us-west-2")
>>> statuses = conn.get_all_instance_status()
>>> statuses
[InstanceStatus:i-07256f36804be4229]
>>>
```

Step 3: Using Dynamic Inventory

- At this point, you can download the Inventory files
  - **wget https://raw/githubusercontent.com/ansible/ansible/devel/contrib/inventory/ec2.py**
  - **wget https://raw/githubusercontent.com/ansible/ansible/devel/contrib/inventory/ec2.ini**
- Put them in the directory "inventory" and add the execute permission to the ec2.py file
- Run the inventory, using the command **./ec2.py --list**. Output will be presented in Json format

Understanding Inventory Caching

- Gathering Inventory information takes time, which is why an inventory cache is created in ~/.ansible/tmp
- To force a cache refresh, use **./ec2.py --refresh-cache**
- Make sure to do this after adding or destroying instances

Managing Instances in EC2

- EC2 instances come with a default user name, and as host keys are not known in advance for new instances, it's useful to add 2 custom parameters to ansible.cfg to manage Instances in EC2
  - **remote_user = yourinstanceuser**
  - **host_key_checking = False**
- Next, find image (AMI) ID's available for your region
  - **aws ec2 describe-images --region us-west-2**
- Then, use the ec2 module to deploy instances

Demo

- Sample playbook: ec2/ubuntu.yaml
  - image: the AMI image
    - Finding images: login to EC2 Management Console, select Images >AMIs, from the drop down list, select Public images, use a filter to filter on Ubuntu. Select source: aws-marketplace to restrict the number of results.
- Alternatively: from the aws-client software, use **aws ec2 describe-images --region us-west-2 --filters Name=name,Values=ubuntu/images/hvm-ssd/ubuntu-bionic-18.04-amd64***
    - Note that some software requires a subscription. if that is the case, a URL will be displayed. Follow the link and subscribe before you continue
  - region: your region
  - instance_type: look for free tier!
  - key_name: create the key first in EC2 and use it
  - group: the security group in EC2
  - instance_tags: optional tags to be used for grouping in EC2

Managing EC2 Instances

- After setting up Ansible EC2 connections, many modules are available to do many things
  - List instances: use **ec2_instance_facts**
  - Terminate instances: use **ansible localhost -m ec2 -a 'instance_id=i-1234 state=absent'**
  - Manage keys: **ec2_key**
  - Manage Security Groups: **ec2_group**
  - Find the latest ami: **ec2_ami_find**

Working with Tower

Understanding Tower

- Tower is used to bring Ansible to the enterprise
- Web-based management
- API access
- Users and credentials management, connecting to external sources
- Scheduled jobs
- Multi-playbook workflows
- Audit trails of all that has happened
- Based on the AWX open source project

Installing Tower

- Use a dedicated VM
  - 4GB RAM (more if possible)
  - 2 CPU's
  - At least 10GB in /var
  - Red Hat / CentOS / Ubuntu are supported
- On CentOS: add the EPEL repository
- Download and extract evaluation version
- Set passwords in inventory file
- Run **./setup.sh**
- Start a browser, connect to https://tower.example.com and log in with admin credentials

Preparing the Environment

- Request a demo license and import it
- Check organizations, you'll have one default organization containing users, teams, projects and inventories
  - Self-support license doesn't allow creation of additional organizations
- From here, work with the following items
  - Credentials: how to connect to managed hosts
  - Inventory: which hosts to connect to
  - Project: where to get the playbooks from
  - Job Template: what to run

Understanding the Process

- Create credentials: this is the alternative to ansible.cfg, where you specify as which user to connect and how to escalate permissions. Copy the ansible user private key here and make sure the user specified here exists on all managed hosts
- Create an inventory: this is where you add all hosts that should be managed
- Create a project: this is where you connect to the source of your playbooks. (GitHub is preferred)
- Create a template: this is where you specify the job that you want to run
- Run the job

Understanding Projects

- A project is a collection of playbooks
- Projects can connect to Github, or read playbooks from a local directory on the Ansible tower server
- Review how the demo project connects to a Github repository
- Before you can use a project, it must be synced. Click Get latest SCM revision to do that
  - After a successful sync, you'll see the last updated timestamp
  - New projects will sync automatically after creation

Creating a Job Template

- A job template connects the playbook from a project with the settings required to launch it, so the job template runs the playbook
- Check the demo template, where you can see the Yaml file that comes from the project, as well as the demo credential that allows to connect to managed servers
- Click the Rocket icon to launch the job in this template
- Observe in the Jobs view if it worked out well

Where to go Next

Next Steps

- Start working with Ansible - a lot
- Have a look at my recorded video courses on Safaribooksonline
  - Ansible Fundamentals
  - Automating with Ansible
  - Ansible Certification