Ansible Fundamentals

1. Understanding Ansible

Your Instructor

- Sander van vugt, author of a lot of open source related stuff with a focus on Ansible, Linux and Kubernetes
- Founder of the Living Open Source Foundation (https://livingopensource.net)
- mail@sandervanvugt.nl
- www.sandervanvugt.com
- Linkedin: @sandervanvugt

About this Course

- This course is an introduction to Ansible, targeting students with no current Ansible knowledge
- After attending this course, students can attend my RHCE8/EX294 course, or my Automating with Ansible course

Agenda

1. Understanding Ansible
2. Setting up an Ansible Managed Environment
3. Using Ad-hoc Commands and Modules
4. Writing Playbooks
5. Working with Facts and Variables
6. Working with Conditionals

Lab requirements

- 3 virtual machines with the names **control**, **ansible1** and **ansible2** are required with RHEL 8 or Centos 8.x installed
  - Cloud instances are not recommended
  - Containers are not supported
- Use "minimal installation" profile
- 1 GB RAM
- 20 GB disk space
- Fixed IP address on each machine
- Internet access
- After setting up the base nodes, use **git clone https://github.com/sandervanvugt/ansiblefundamentals**

Ansible Fundamentals

1. Understanding Ansible

Automating with Ansible

- Ansible is a solution that provides automation tools
- In Automation, configuration of multiple nodes can be centrally organized and pushed to the managed nodes
- Automation fits well in a DevOps way of working where integration of changes is made much easier
- Ansible itself can be used in a DevOps strategy, and working with Ansible follows the DevOps way
- Playbooks are used to work in a declarative way, where versions of playbooks are easily managed by putting them in a GitHub repository

Alternatives to Ansible

- Solutions similar to Ansible exist
  - Puppet
  - Chef
  - SaltStack
- Ansible excells by the easy way it can be used and the wide range of devices it can manage
  - Playbooks are written in YAML
  - Thousands of modules are available to manage a wide range of devices
  - Backed by Red Hat

Ansible Fundamentals

2. Setting up an Ansible Managed Environment

Setting up Ansible

- In an Ansible configuration there is the control node, as well as the managed nodes
- The Ansible software is installed on the Control node
- The Ansible software generates Python scripts that can be executed on the managed nodes
- Managed nodes just require remote access as well as Python to be available

Setting up the Managed Environment

- Create a dedicated user account. In this course we'll use the user **ansible**
- Ensure this user has **sudo** privileges on at least the managed nodes
- Ensure that SSH based access to the managed nodes has been configured
- Install Python as well as Ansible software
- Set up host name resolving
- Create an inventory file
- Configure the anisble.cfg file

Understanding the User Account

- To manage settings on remote hosts, a dedicated user account is needed
- This user account needs to be able to log in to managed hosts, using SSH
- Typically, SSH key-based access is used, but other options are possible
- To perform admin tasks on the managed hosts, **sudo** setup is required

Understanding Inventory

- Inventory is a file that identifies Ansible managed hosts
- In inventory, host groups can be defined
- Static as well as dynamic inventory can be used
- Dynamic inventory is a script (normally community provided) that explores inventory in dynamic environments. It contacts the central repository where Inventory host names are stored
- Static inventory is a configuration file that lists managed hosts
- Default inventory is in /etc/ansible/hosts and provides nice syntax examples

Understanding ansible.cfg

- ansible.cfg is the main configuration file
- It is used for default settings, including privilege escalation and remote user settings
- Default ansible.cfg is in /etc/ansible/ansible.cfg
- It is common to work with project-based ansible.cfg as well

Lab 1 – Setting up Ansible (1)

1. On all hosts, create a user **ansible** and set the password to **password**
2. On control, configure an /etc/hosts file to set up name resolution for each of the nodes
3. On CentOS, use **yum install –y epel-release** to setup repository access
4. On RHEL: use **subscription-manager repos --enable=ansible-2-for-rhel-8-x86_64-rpms** to add the Ansible repository
5. On control, type **yum install ansible**
6. Type **ansible --version** to verify software has been installed
7. On managed nodes: use **systemctl status sshd** to verify SSH is running and available

Lab 1 – Setting up Ansible (2)

1. Type **rpm –qa | grep python** to verify Python is installed
2. Type **firewall-cmd --list-all** to verify the SSH port is open in the Firewall
3. From the control machine as used **ansible**, type **ssh ansible1**. Repeat this this for **ansible2**
4. On control, as user ansible, type **ssh-keygen**
5. Distribute SSH keys using **ssh-copy-id ansible1** and **ssh-copy-id ansible2**.
6. As root on all nodes, use **echo "ansible ALL=(ALL) NOPASSWD: ALL" > /etc/sudoers.d/ansible**

Lab 1– Setting up Ansible (3)

- As user ansible on control, create a file with the name inventory and give it the following contents:
  ansible1
  ansible2
- Use **ansible all –i inventory --list-hosts**
- Use **ansible all –i inventory --graph**

Lab 1 – Setting up Ansible (4)

- As user Ansible, create a file with the name **anisble.cfg** and the following contents:

```
[defaults]
remote_user = ansible
host_key_checking = false
inventory = inventory

[privilege_escalation]
become = True
become_method = sudo
become_user = root
become_ask_pass = False
```

Ansible Fundamentals

3. Using Ad-hoc Commands and Modules

Understanding Ansible Modules

- Thousands of Ansible modules are provided, providing access to a wide range of tasks
- Modules are written in Python, and used in ad-hoc commands and Ansible playbooks
- **command**, **shell** and **raw** are pretty basic and non-specific, many specialized modules exist as well
- Use **ansible-doc –l** to get a list of all modules that are available
- Knowing Ansible = Knowing Modules

Using Modules in ad-hoc Commands

- An Ad-hoc command is a command where all required command parameters are provided on the command line
- Add-hoc commands have different elements
  - Name or names of target hosts
  - Name of the inventory file to be used
  - Name of the module to be used
  - Name of the module arguments between double quotes
- **ansible all –i inventory –m user –a "name=lisa"**

Lab 2: Running an Ad-hoc Command

- As user **ansible** on **control**, make sure you are in the directory that has inventory and ansible.cfg files
- Type **ansible all –m ping**
- Type **ansible all –m user –a "name=lisa"**
- Observe command output
- Type **ansible all –m command –a "id lisa"**
- Type **ansible all –m user "name=lisa state=absent"**

Exploring Essential Modules

- **command**: runs arbitrary commands without using a shell
- **shell**: runs arbitrary commands while using a shell
- **raw**: runs commands directly on top of SSH without using the Python layer
- **copy**: copies files or lines of text to files
- **yum**: manages packages on RHEL managed hosts
- **service**: manages current state of packages
- **ping**: verifies host availability

Lab 3: Command Versus Shell

1. Use **ansible ansible1 –m command –a "rpm –qa | grep python"**
2. Observe the command output, did it work?
3. Use **ansible ansible1 –m shell –a "rpm –qa | grep python"**.
4. Did you see any difference?
5. Use **ansible all –m copy –a 'content="hello world" dest=/etc/motd'**
6. Verify it worked
7. Use **ansible all –m package –a "name=nmap state=latest"**
8. Use **ansible all –m service –a "name=httpd state=started enabled=yes"**
9. Did that work?

Using **ansible-doc**

- **ansible-doc** is the authoritative information about module use
- Notice it lists all available arguments, as well as some examples of how to use the modules in a playbook
- Use **ansible-doc –l** for a list of all modules
- Use **ansible-doc modulename** for all information about **modulename**
- Use **ansible-doc –s modulename** to see arguments that can be used while working with a module

Ansible Fundamentals

4. Writing Playbooks

Understanding Playbooks

- A *Playbook* is a collection of plays
- Each *play* contains at least one *task* that uses an Ansible module to get things done
- Each play has a header, that specifies on which hosts the tasks should be executed
- Playbooks are written in YAML
- Playbooks typically start with --- and may end with …

Understanding YAML

- In YAML, indentation is used to identify the relation between parent and child objects, which are specified as key: value pairs
- Data elements at the same level in the hierarchy must have the same indentation
- Items that are children (properties) are indented more than the parents
- If a specific module argument can have multiple values, these values typically start with a hyphen
- Indentation happens using spaces. Using tabs is not allowed
- Configure **vi** for intentation by adding the following to ~/.vimrc: **autocmd FileType yaml setlocal ai ts=2 sw=2 et**

Exploring an Example Playbook

- Have a look at **install_and_start_httpd.yaml**
- Run the playbook with **ansible-playbook install_and_start_httpd.yaml** and observe the output
- On playbook execution many things happen
  - Fact gathering
  - Task execution
  - Play Recap
- Notice that a good playbook is written to be *idempotent*
- Verify playbook syntax using **ansible-playbook –syntax-check**
- Perform a dry-run using **ansible-playbook –C**
- Increase verbosity using **–v** up to **–vvvvv**

Lab 4: Writing a Playbook

1. Write a playbook that meets the following requirements. Use **ansible-doc** for additional information if needed
    a. Install, start and enable the **vsftpd** service on all managed nodes
    b. Open the **firewalld** firewall to allow access to this service
2. Run the playbook, and observe working of the playbook
3. Run the playbook one more time, and again, observe the working of the playbook

Exploring Multi-Play Playbooks

- A playbook can define one or more plays
- Defining multi-play playbooks makes sense in the following cases:
  - To write the playbook in a modular way
  - To limit the number of tasks that run in one play
  - To run some tasks on one host, while others are running on another host
- Have a look at **run_and_test_httpd.yaml** and run it
- Have a look at **multi_play.yaml** and run it

Lab 5: Analyzing a Playbook

- Run **playbook_with_errors.yaml**. Analyze the errors and fix them

Ansible Fundamentals

5. Working with Facts and Variables

Understanding Variables

- Using Variables makes it possible to separate static code from dynamic code
- Variables can be provided in multiple ways
  - As ansible facts
  - Defined in a playbook
  - Defined in inventory
  - Specified as a command line option
  - Using **register**
  - Magic Variables are system defined and automatically set
- Have a look at **variables_example.yaml**

Using Variables

- To refer to a variable, include the variable name between double braces: **{{ user }}**
- If a value starts with a variable name, the entire value string needs to be between double quotes as well: **"{{ user }} is created"**

Understanding Ansible Facts

- Ansibel facts are system variables that contain information about the managed system
- Each playbook starts with an implicit task to gather facts
- To use fact gathering from an ad-hoc command, use the **setup** module
- Additional modules for fact gathering are available, like **package_facts**
- To disable fact gathering, use **gather_facts: no** in the play header
- Have a look at **gather_facts.yaml** to see how it works

Adressing Facts

- Facts are stored in a multi-valued variable with the name **ansible_facts**, which is organized as a dictionary
- To address specific values in this dictionary, two formats can be used:
  - **ansible_facts['default_ipv4']['address']**
  - **ansible_facts.default_ipv4.address**
- Of these two, the first one is preferred
- On old versions of Ansible, facts were stored as individual variables (injected variables): **ansible_default_ipv4** instead of **ansible_facts['default_ipv4']**
- Some modules only support the old way of referring to facts
- Have a look at **old_facts.yaml** and **new_facts.yaml**

Using Custom Facts

- Custom Facts are used to provide a host with arbitrary values that Ansible can use to change the behavior of plays
- Custom facts are provided as static files in INI or JSON format, which have the extension .fact and which are stored in the /etc/ansible/facts.d/ directory
- Have a look at **custom.fact** and **use_custom_facts.yaml**
- Notice the use of the [header] in custom.fact, which you will also see in the variable you refer to in the playbook

Lab 6: Working with Facts

- Write a simple playbook that works with facts and meets the following requirements:
  - The fact file has a header [software] and defines two facts: package=httpd and service=httpd
  - The playbook uses the values of the facts to install the software and start and enable the software, using the names of the facts, correctly referred to as variables

User Defined Variables

- User variables can be defined at multiple locations
  - In a **vars** section in the play header
  - In a file, referred to by **vars_files** in the play header
  - As host or host group variables, using **host_vars** and **group_vars** directories in the project directory
- Have a look at **myvars.yaml** and **vars_file.yaml**
- When using host_vars, ensure the directory host_vars contains a file with the name of the host
- When using group_vars, ensure the directory group_vars contains a file with the name of the host group

Lab 7: Working with Variables

1. Create an inventory file that defines a host group webservers, as well as a group dbservers
2. Create the file group_vars/webservers with the following contents
   web_package: httpd
   web_service: httpd
3. Create a simple playbook that uses the **debug** module to show the current value of the variables web_package and web_service

Understanding **hostvars**

- The system defined **hostvars** variable contains all current host settings
- Use **ansible localhost –m debug –a 'var=hostvars["ansible1"]'** to discovery hostvars for ansible1

Capturing Command Output Using Register

- To capture command output, **register** can be used
- Register stores command output in a variable, which name you can define yourself
- The registered command output is particularly useful in conditionals
- Have a look at **register.yaml**

Ansible Fundamentals

6. Working with Conditionals

Control Structures Overview

- Loops and Items allow Ansible to process over a list of values
- When can be used to perform an action based on the result of a test
- Handlers can be used to run a task only if another task has triggered it

Understanding Loop

- A loop – previously known as with_* - can be used to iterate over a list of items
- This is useful for performing operations on many packages, users and more
- An example is in loop_packages.yaml
- The items can be provided as a list, but also as the value of a variable. An example is in loop_on_vars.yaml, which includes the users-list variable file

Understanding **when**

- **when** can be used to work on the result of a conditional test
  - **varname is defined**
  - **varname is not defined**
  - **ansible_distribution in distributions**
  - **variabe**
  - **not variable**
  - **key == "string"**
  - **key == int**
  - **key >= int**
  - **key < int**
  - **key != int**
- See **when_test.yaml** for an example

Understanding **when** - 2

- **when** statements can also be used to perform more complex tests
- See **when_complex.yaml** for an example
- **when** statements also are useful in combination with **register**
- See **when_register.yaml** for an example

Understanding Handlers

- A handler is a task that is called by another task
- In order to be called, the calling task must have triggered a change – handlers are not called if the triggering task generates an OK
- Handlers are executed after all other tasks have successfully been executed
- If one of the other tasks fail, the handler will not run
- Use **ignore_errors** to ensure play execution on a host doesn't stop if an error was encountered
- Use **force_handlers** to force handlers to be run, even if some tasks are failing
- See **using_handlers.yaml** for an example

Lab 8: Using Conditionals

- Write a playbook that meets the following requirements
  - The kernel must be updated, if the CentOS distribution is used
  - If the kernel was successfully updated, the managed server needs to be restarted

Ansible Fundamentals

Q&A / additional learning

Topics Not Covered in this Course

- Managing templates and files
- Working with roles
- Working with includes and imports
- Troubleshooting
- Working with blocks
- Using filters
- Advanced Usage Examples

Related Courses

- Live Courses
  - Automating with Ansible
  - Ansible Tower
  - RHCE/EX294
- Recorded Courses
  - Ansible Fundamentals
  - Automating with Ansible
  - RHCE/EX294
- Upcoming Book
  - RHCE8 Cert Guide (estimated august 2020) – preview on learning.oreilly.com estimated june 2020