FINDMORE
CONSULTING

RED HAT®
ANSIBLE®

Introduction

Who am I?

Hello! My name is Luís Nabais (https://www.luisnabais.com), I'm a Findmore Consulting employee.

I'm currently working on Out.Cloud, a Findmore backed project, and also on a Findmore's client company, called AXA, both where I work as a DevOps Engineer.

My specialties are Linux, Docker, Jenkins, DevOps, AWS and OpenSource.
I'm a Red Hat Certified Engineer and an ITIL Foundations certified professional. I'm also in the process of getting certified on AWS.

Notes

Feel free to interrupt for questions at any time
Please correct me if you think the pace is too fast or too slow.
All the content is publicly available (slides, code samples, scripts)

Contact

Mail: luis.nabais@findmore.pt

Welcome to this Findmore Consulting Workshop! Enjoy!

What will we talk about?

Ansible Introduction

Basics

- Inventory
- Ad-hoc Commands
- Modules
- Playbooks

Beyond the Basics

- Handlers
- Variables
- Vaults
- Conditionals
- Templates

Playbook Organization

- Imports/Includes
- Roles
- Ansible Galaxy

Real World Examples

Pre-requisites

Knowledge

Some Linux knowledge is advised, preferably more than just the basics.

Material/Equipment

Nothing!
Seriously, you can just listen, understand and try everything later, following the resources on the GitHub address on the bottom of this slide.

If you want to follow along, trying the examples live (Optional):

- A computer with Linux or macOS
- A Virtual Machine with Linux (CentOS, Ubuntu or Debian are recommended)
- An Internet connection

NOTE: All resources are available at https://github.com/luisnabais-courses/ansible2019

Ansible Introduction

What is Ansible?

Ansible is a very popular open source IT configuration management, provisioning and automation platform, which allows application deployment, infrastructure orchestration and provisioning, on local or remote servers.
Ansible allows to implement changes in parallel on multiple hosts.

On any given day, a systems administrator, developer or DevOps has many tasks, such as:

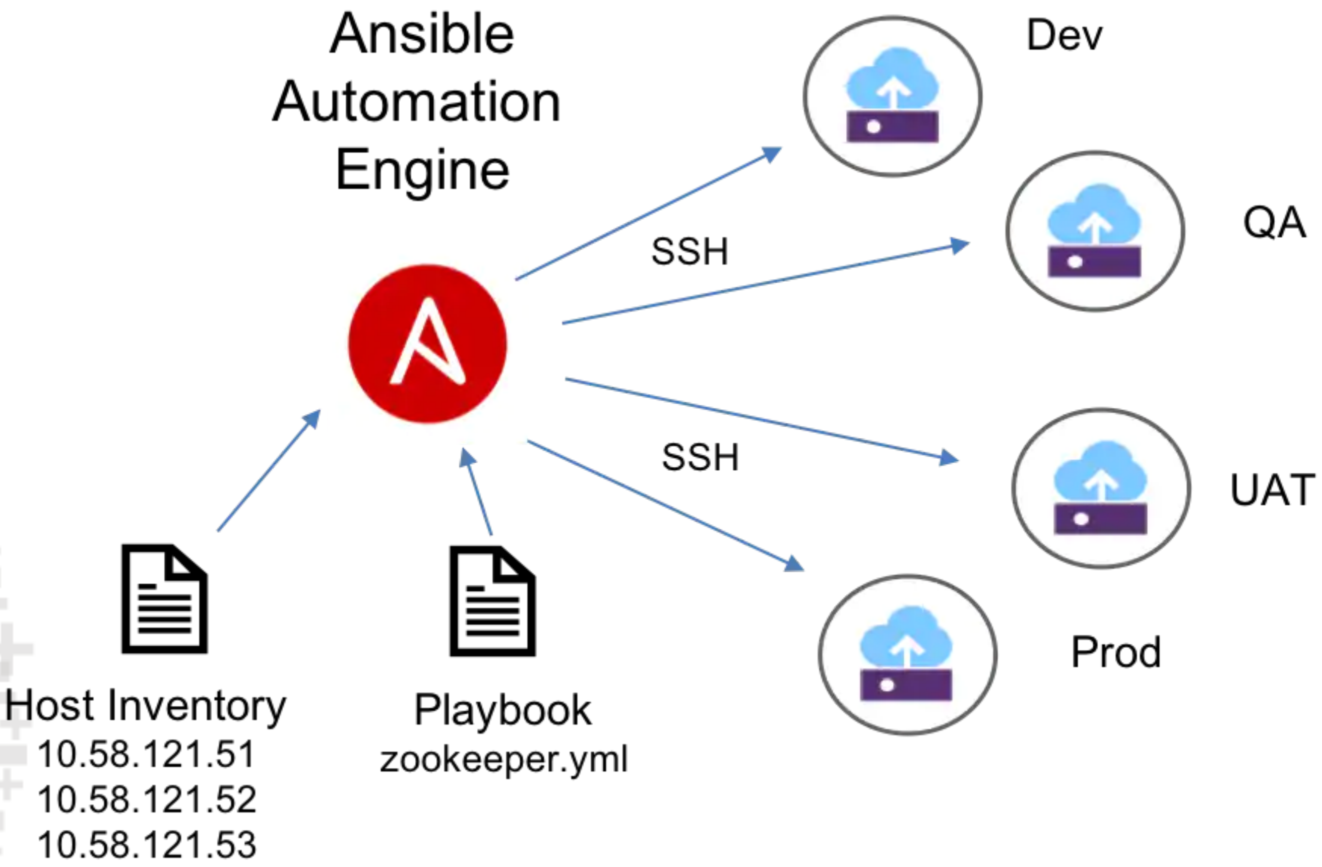
- Apply patches and updates via yum, apt, and other package managers
- Check resource usage (disk space, memory, CPU, swap space, network)
- Check log files
- Manage system users and groups
- Manage DNS settings, hosts files, etc
- Copy files to and from servers
- Create base servers (templates) for applications
- Deploy applications or run application maintenance
- Reboot servers
- Manage cron jobs

So it can be used for small tasks, such as copy a file, execute a command or change a value in a file.
But can also be used for advanced purposes, such as create and provision a complete local or remote infrastructure.

Ansible Introduction

How does it work?

Ansible is agentless and requires no extra software to be installed on servers. Ansible can be installed on a server, a laptop or a workstation.
Ansible works by pushing changes out to your servers (default behavior).



Ansible Basics

Inventory

Ansible uses a file, called the **inventory file**, to communicate with the servers. The inventory file's content is just a server list. Default file location on Linux hosts is /etc/ansible/hosts.

An example of an inventory file's content:

```
[localhost]
localhost

[app]
centos
debian
ubuntu

[db]
centos
```

Ansible Basics

Inventory

Some more advanced parameters can be used, such as:

```
[multi:children]
app
db

[multi:vars]
ansible_ssh_user=vagrant
```

Ansible allows to specify which inventory file to use, by setting the ANSIBLE_INVENTORY environment variable, like this:

```
export ANSIBLE_INVENTORY=/home/user/ansible_hosts
```

Ansible Basics

Ad-Hoc commands

Ansible allows to run Ad-Hoc commands to execute quick tasks.

If a simple SSH command to the servers work without password, Ansible should work as well. Example:

ssh username@centos

If something like the above command works, Ansible commands like these should work:

```
ansible multi -m ping -u [username]
ansible multi -a "free -m" -u [username]
ansible multi -a ”hostname" -u [username]
ansible multi -b -a "tail /var/log/messages” -u [username]
```

Note:

- Ansible assumes the connection to the servers is passwordless, that is key-based.
- It’s possible to override that, using parameters to force the password to be asked.
- The commands may not be run on each server in the order we’d expect. The commands run in parallel, using multiple process forks and will return depending on the server’s response time.
- It’s possible to specify how many servers run the command in parallel, using -f X (change X with the number of servers – 1 is the same as run the command in sequence on the servers)

Ansible Basics

Modules

Ansible Modules are standalone scripts than can simplify the commands, as they include validations and are easy to read and implement.

As an example, instead of running the command *yum install -y ntp* on each of the servers, we can use Ansible's yum module to do the same:

ansible multi -b -m yum -a "name=ntp state=present"

Instead of running *systemctl start ntpd* and *systemctl enable ntpd*, we can also use the service module:

ansible multi -b -m service -a "name=ntpd state=started enabled=yes"

Ansible Modules are much smarter ways of doing the same, as they check many things before executing the operations requested, such as check if a package is already installed.

Ansible Basics

Other useful modules

Manage Users:

- Group: *ansible app -b -m group -a "name=admin state=present"* (state=absent makes sure it is removed)
- User: *ansible app -b -m user -a "name=johndoe group=admin createhome=yes state=present"*

Manage Packages:

- Apt: *ansible app -b -m apt -a "name=git state=present"*
- Package: *ansible app -b -m package -a "name=git state=present"*

Manage Files:

- Copy file to hosts: *ansible multi -m copy -a "src=/etc/hosts dest=/tmp/hosts"*
- Create file/directory: *ansible multi -m file -a "dest=/tmp/test mode=644 state=directory"* (state=absent deletes file/directory)

Others:

- Shell: *ansible multi -m shell -a "tail /var/log/messages | grep ansible-command | wc -l"*
  *(*If there is no need to filter output, direct command can be used: *ansible multi -b -a "tail /var/log/messages")*
- Cron: *ansible multi -b -m cron -a "name='daily-cron-all-servers' hour=4 job='/path/to/daily-script.sh' state=present"*
- Git: *ansible app -m git -a "repo=git://example.com/path/to/repo.git \ dest=/opt/myapp update=yes version=1.2.4"*
- Ping: *ansible all -m ping*

Ansible Playbooks

An Ansible Playbook is a file, in YAML format, which contains organized instructions, just like a recipe or a blueprint. With it, we can execute complex operations in Ansible.

Let’s create a file, named it something like ntp-playbook.yml and add the following content to it:

```
- hosts: app
  become: yes

  tasks:
  - name: Ensure NTP is installed
    yum: name=ntp state=present

  - name: Ensure NTP is running
    service: name=ntpd state=started enabled=yes
```

Let’s go through the playbook, step by step.

Ansible Playbooks

1 ---

This first line is a marker showing that the rest of the document will be formatted in YAML.

2 - hosts: app

This line tells Ansible to which hosts this playbook applies.

3 become: yes

Since we need privileged access to install NTP and modify system configuration, this line tells Ansible to use sudo for all the tasks in the playbook (you’re telling Ansible to ‘become’ the root user with sudo, or an equivalent).

4 tasks:

All the tasks after this line will be run on all the hosts specified above.

Ansible Playbooks

```
- name: Ensure NTP daemon is installed.
yum: name=ntp state=present
```

This command is the equivalent of running yum install ntp, but is much more intelligent; it will check if ntp is installed, and, if not, install it. This is the equivalent of the following shell script:

```
if ! rpm -qa | grep -qw ntp; then
   yum install -y ntp
fi
```

Ansible Playbooks

```
- name: Ensure NTP is running.
service: name=ntpd state=started enabled=yes
```

This final checks and ensures that the ntpd service is started and running, and sets it to start at system boot. A shell script with the same effect would be:

```
# Start ntpd if it's not already running.
if ps aux | grep -q "[n]tpd”
then
   echo "ntpd is running." > /dev/null
else
   systemctl start ntpd.service > /dev/null
   echo "Started ntpd.”
fi

# Make sure ntpd is enabled on system startup.
systemctl enable ntpd.service
```

Ansible plays can be executed using the command ansible-playbook, which comes with Ansible:

```
ansible-playbook multi playbook.yml
```

Ansible Playbooks

One thing many SysAdmins find very attractive about Ansible is that it’s very easy to convert a Shell Script into a Playbook. As a basic example:

Shell Script	Ansible Playbook
# Install Apache. yum install --quiet -y httpd httpd-devel # Copy configuration files. cp httpd.conf /etc/httpd/conf/httpd.conf cp httpd-vhosts.conf /etc/httpd/conf/httpd-vhosts.conf # Start Apache and configure it to run at boot. service httpd start chkconfig httpd on	--- - hosts: all tasks: - name: Install Apache. command: yum install --quiet -y httpd httpd-devel - name: Copy configuration files. command: cp httpd.conf /etc/httpd/conf/httpd.conf - name: Start Apache. command: service httpd start - name: Configure Apache to run at boot command: chkconfig httpd on

Even if a SysAdmin knows Shell Script (which is just the basic commands organized) but doesn’t know the correspondent Ansible Modules, can use Ansible for his tasks.

Ansible Playbooks

And after a while, the SysAdmin can upgrade the script to something better, such as:

```
---
- hosts: all
  become: yes

  tasks:
  - name: Install Apache.
    yum:
      name:
        - httpd
        - httpd-devel
      state: present
  - name: Copy configuration files.
    copy:
      src: "{{ item.src }}”
      dest: "{{ item.dest }}”
      owner: root
      group: root
      mode: 0644
    with_items:
      - src: httpd.conf
        dest: /etc/httpd/conf/httpd.conf
      - src: httpd-vhosts.conf
        dest: /etc/httpd/conf/httpd-vhosts.conf
  - name: Make sure Apache is started now and at boot.
    service: name=httpd state=started enabled=yes
```

Ansible Playbooks: Handlers

Handlers are just like regular tasks, except that won’t run unless called/notified.

```
handlers:
 - name: restart apache
  service:
   name=apache2
   state=restarted
  - name: restart memcached
   service:
    name: memcached
    state: restarted

tasks:
 - name: Install Apache
  yum:
   name:
    - httpd
    - httpd-devel
   state: present
   notify:
    - restart apache
    - restart Memcached
```

Ansible Playbooks: Module Return Values

Ansible Modules include return values, which allow us to follow up on the status of each task and run other tasks accordingly. Common uses are input for playbook conditions and loops.
Return values are a key feature for monitoring and managing task execution.

Some common return values are:

- *stdout or stdout_lines*: Standard output of the command executed, in modules such as *raw*, *command* or *shell*.
- *stderr or stderr_lines*: The same output source as stdout, but for error message.
- *changed*: Indicates if the task has made a change to the target host. Returns a Boolean value of *True* or *False*.
- *failed*: Indicates if the task has failed or not. Returns a Boolean value.
- *skipped*: Indicates if the task has been skipped or not. Returns a Boolean value.
- *rc*: Returns the Return Code, which is generated by the command executed.
- *msg*: Contains the message

Ansible Playbooks: Module Return Values

```
--
- name: Restart Linux hosts if reboot is required after updates
  hosts: servers
  become: yes

  tasks:
    - name: check for updates
      apt: update_cache=yes

    - name: apply updates
      apt: upgrade=yes

    - name: check if reboot is required
      shell: "[ -f /var/run/reboot-required ]"
      failed_when: False
      register: reboot_required
      changed_when: reboot_required.rc == 0
      notify: reboot

  handlers:
    - name: reboot
      command: shutdown -r now "Ansible triggered reboot after system updated"
      async: 0
      poll: 0
      ignore_errors: true
```

Ansible Playbooks: Variables

Variables can be passed, when calling the *ansible-playbook* command, with the *--extra-vars* option:

```
ansible-playbook example.yml --extra-vars "foo=bar"
```

Variables can also be defined on the Playbook directly, can only be accessible by the Playbook and are unset after Playbook runs.

```
---
- hosts: localhost
  vars:
    foo: bar
  tasks:
    # Prints "Variable 'foo' is set to bar".
    - debug: msg="Variable 'foo' is set to {{ foo }}"
```

Note: Ansible gets a lot from Python, so it uses the Jinja2 template format for variables.

Ansible Playbooks: Variables

Variables can be passed, using an external file:

```
---
- hosts: app
  vars_files:
    - "apache_default.yml"
    - "apache_{{ ansible_os_family }}.yml"
  tasks:
    - service: name={{ apache }} state=running
```

```
# apache_default.yml
apache: apache2

# apache_redhat.yml
apache: httpd
```

Ansible Playbooks: Variables

Host Variables and Group Variables can be passed, using the inventory file:

```
[group]
host1 admin_user=jane
host2 admin_user=jack
host3

[group:vars]
admin_user=john
```

Ansible Playbooks: Variables

Variables can be registered dynamically from the result of tasks, for later use:

```
---
- hosts: localhost

  tasks:
  - name: teste
    shell: echo "Teste"
    register: foo_result
    ignore_errors: True

  - name: condicao
    shell: echo "Sucesso"
    when: foo_result.stdout == "Teste”
```

Ansible Playbooks: Variables

Simple variables can be accessed like we’ve seen before:

```
- command: /opt/my-app/build {{ my_environment }}
```

It’s very common that variables are structured as lists/arrays. Take as an example the following playbook:

```
tasks:
  - debug: var=ansible_eth0
```

The elements of those variables can be accessed using:

```
{{ ansible_eth0.ipv4.address }}
{{ ansible_eth0['ipv4']['address'] }}
```

Ansible Playbooks: Facts

When we run an Ansible Playbook, Ansible first gathers information (facts), about each host in the play.

```
$ ansible-playbook playbook.yml

PLAY [group] ********************************************************

GATHERING FACTS *****************************************************
ok: [host1]
ok: [host2]
ok: [host3]
```

Facts include host IP addresses, CPU type, disk space, operating system information, network interface information, among many others.

All facts can be shown with the setup module:

```
$ ansible localhost -m setup
```

Ansible Playbooks: Facts

Some common facts include *ansible_os_family*, *ansible_hostname*, and *ansible_memtotal_mb,* usually used with *when*, to determine whether to run certain tasks.

If you don't need facts and would like to save a few seconds per-host when running playbooks, just set *gather_facts: no* in the playbook.

- *hosts: db*
- *gather_facts: no*

Ansible Playbooks: Vault

If you use Ansible to fully automate the provisioning and configuration of your servers, chances are you will need to use passwords or other sensitive data for some tasks, whether it's setting a default admin password, synchronizing a private key, or authenticating to a remote service.

Ansible Vault is built in into Ansible and store encrypted passwords and other sensitive data alongside the rest of the playbooks.

Ansible Vault works much like a real-world vault:

1. You take any YAML file you would normally have in your playbook (e.g. a variables file, host vars, group vars, role default vars, or even task includes!), and store it in the vault.
2. Ansible encrypts the vault ('closes the door'), using a key (a password you set).
3. You store the key (your vault's password) separately from the playbook in a location only you control or can access.
4. You use the key to let Ansible decrypt the encrypted vault whenever you run your playbook.

Ansible Playbooks: Vault

Let’s see how it works in practice. Here’s a playbook that connects to a service’s API, and requires a secure API key to do so:

```
---
- hosts: appserver

  vars_files:
    - vars/api_key.yml

  tasks:
    - name: Connect to service with our API key.
      command: connect_to_service
      environment:
        SERVICE_API_KEY: "{{ myapp_service_api_key }}"
```

The vars_file, which is stored alongside the playbook, in plain text, looks like:

```
---
myapp_service_api_key: "yJJvPqhqgxyPZMispRycaVMBmBWPqYDf3DFanPxAMAm4UZcw"
```

Ansible Playbooks: Vault

This is convenient, but it’s not safe to store the API key in plain text. Even when running the playbook locally on an access-restricted computer, secrets should be encrypted.

To encrypt the file with Vault, run:

```
$ ansible-vault encrypt api_key.yml
```

Enter a secure password for the file, and Ansible will encrypt it. If you open the file now, you should see something like:

```
$ANSIBLE_VAULT;1.1;AES256
653635363963663439383865313262396665353063663839616266613737616539303
```

Ansible Playbooks: Vault

Next time you run the playbook, you will need to provide the password you used for the vault so Ansible can decrypt the playbook in memory for the brief period in which it will be used. If you don’t specify the password, you’ll receive an error:

```
$ ansible-playbook test.yml
ERROR: A vault password must be specified to decrypt vars/api_key.yml
```

There are a number of ways you can provide the password, depending on how you run playbooks. Providing the password at playbook runtime works well when running a playbook interactively:

```
# Use --ask-vault-pass to supply the vault password at runtime.
$ ansible-playbook test.yml --ask-vault-pass Vault password:
```

After supplying the password, Ansible decrypts the vault (in memory) and runs the playbook with the decrypted data.

Ansible Playbooks: Vault

You can edit the encrypted file with *ansible-vault edit*.
You can also:

- *rekey* a file (change its password)
- *create* a new file
- *view* an existing file
- or *decrypt* a file.

You can also supply the vault password using a file.
Example:

Create the file *~/.ansible/vault_pass.txt* with your password in it, set permissions to 600, and tell Ansible the location of the file when you run the playbook:

```
$ ansible-playbook test.yml --vault-password-file ~/.ansible/vault_pass.txt
```

Ansible Playbooks: Conditionals

Many tasks need to only run in certain circumstances.

When

```
tasks:
  - name: "shut down Debian flavored systems"
    command: /sbin/shutdown -t now
    when: ansible_os_family == "Debian”
```

We can also force a change or a failure, using *change_when* or *failed_when*:

```
---
- hosts: localhost
  tasks:
    - shell: echo "teste"
      register: myoutput
      failed_when: myoutput.stdout == "teste"
```

Notes:
You can use Jinja2 and Python expressions, which allow much more powerful validations.

Ansible Playbooks: Templates

Sometimes, values in files, like configurations, vary from one remote machine to another, even if the rest of the file/settings remain the same.

Creating static file for each of the machines is not an effective solution. That’s why templates were created.

A template in Ansible is a file which contains all your configuration parameters, but the dynamic values are given as variables.

Templates in Ansible work like in other Python Framework template systems, like Django or Flask, they use Jinja2 templating engine. Using Jinja2, we have much more than just replacing variables, like conditional statements, loops, write macros, filters for transforming the data, do arithmetic calculations, etc.

(I really advise you to check Jinja2 Documentation website to see the potential for yourself)

To use templates, we create the template files in j2 format (.j2) and, of course, add it to our playbook.

Template on the playbook side is very much like Copy.

Let’s see an example.

Ansible Playbooks: Templates

```
# playbook.yml

---
- hosts: all
  become: true
  vars:
    ntp_servers:
      - "0.pt.pool.ntp.org"
      - "1.pt.pool.ntp.org"

  tasks:
  - name: install ntp
    apt: name=ntp state=installed update_cache=yes

  - name: write ntp.conf
    template:
      src=templates/ntp.conf.j2
      dest=/etc/ntp.conf mode=644 owner=root group=root

  - name: start ntp
    service: name=ntp state=restarted
```

Ansible Playbooks: Templates

```
# templates/ntp.conf.j2


driftfile /var/lib/ntp/drift

statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable

{% for item in ntp_servers %}
server {{ item }}
{% endfor %}

restrict default nomodify notrap nopeer noquery

restrict 127.0.0.1
restrict ::1
```

Ansible Playbooks: Organization

So far, we’ve been using fairly straightforward examples. Most of them are intended for a single server and are defined in one single playbook.

However, Ansible is very flexible when it comes to organizing tasks. Playbooks can be much more maintainable, reusable and powerful. We can split tasks so they can be much more efficient.

Let’s look at some examples

Ansible Playbooks: Organization - Imports

Imports

We’ve already seen an example of including other files, when we imported a file with variables, instead of using all the variables inline:

```
- hosts: localhost
  vars_files:
    - vars.yml
```

Tasks can easily be included the same way:

```
tasks:
- import_tasks: user.yml
```

Notes:

- Variables from vars.yml would be readable in user.yml, which helps reusability.
- Imported files can also import other tasks
- Imports are static and pre-processed, at the time the playbooks are parsed, before the task execution

A good example where imports are good is when we usually want outside our main playbook is handlers:

```
handlers:
  - import_tasks: handlers.yml
```

Ansible Playbooks: Organization - Includes

Includes

If we need to include tasks that are dynamic - that do different things depending on how the rest of the playbook runs – then we need to use *include_tasks*. Includes are executed during runtime, not during parsing time.

```
---
- name: Check for existing log files in dynamic log_file_paths variable.
  find:
    paths: "{{ item }}"
    patterns: '*.log'
  register: found_log_file_paths
  with_items: "{{ log_file_paths }}"
```

If log_file_paths is not statically defined earlier, import_tasks won't work, as it will fail knowing the variable's value, because it will defined only during runtime.

We can also include variables specific to an operating system, using includes and variables. For example:

```
- name: Include OS specific variables
  include_vars: "vars/{{ ansible_os_family }}.yml"
```

This would be the same as including the variables from a Debian.yml or a RedHat.yml file, existing in vars directory.

Ansible Playbooks: Organization - Import Playbooks

If we need to import other playbooks, we can use *import_playbook*. As playbooks are not dynamic like tasks, the only option is *import_playbook*, there is no *include_playbook*.

```
- hosts: all
  remote_user: root

  tasks:
    [...]

- import_playbook: web.yml
- import_playbook: db.yml
```

This way, we can create playbooks to configure all the servers in our infrastructure. We just need to create a master playbook that includes each of the individual playbooks.
When we want to initialize our infrastructure, make changes across the entire fleet of servers, or check to make sure their configuration matches the playbook definitions, we can run just one ansible-playbook command!

Ansible Playbooks: Organization - Complete example

The main playbook for a complete Drupal LAMP server can be less than 20 lines, using includes and/or imports. Just split each of the sets of tasks into their own files, and we’ll end up with a main playbook like this:

```
---
- hosts: all

  vars_files:
    - vars.yml

  pre_tasks:
    - name: Update apt cache if needed.
      apt: update_cache=yes cache_valid_time=3600

  handlers:
    - import_tasks: handlers/handlers.yml

  tasks:
    - import_tasks: tasks/common.yml
    - import_tasks: tasks/apache.yml
    - import_tasks: tasks/php.yml
    - import_tasks: tasks/mysql.yml
    - import_tasks: tasks/drupal.yml
```

Ansible Playbooks: Organization - Roles

Including playbooks inside other playbooks can make playbook organization a little more sane, but once we start wrapping up our entire infrastructure's configuration in playbooks, we might end up with something resembling Matryoshka dolls.

What if there was a way to take bits of related configuration, and package them together nicely?
Additionally, what if we could take these packages (often configuring the same thing on many different servers) and make them flexible so that we can use the same package throughout our infrastructure, with slightly different settings on individual servers or groups of servers?

Ansible Roles can do all that and more!

Ansible Playbooks: Organization - Roles

Instead of requiring certain files and playbooks to be explicit included in a role, Ansible automatically includes any main.yml files inside specific directories that make up the role.

There are only two directories required to make a working Ansible role:

```
role_name/
  meta/
  tasks/
```

If we create a directory structure like the one shown above, with a main.yml file in each directory, Ansible will run all the tasks defined in tasks/main.yml if we call the role from our playbook using the following syntax:

```
---
- hosts: all
  roles:
    - role_name
```

Roles can live in a couple different places: the default global Ansible role path (configurable in /etc/ansible/ansible.cfg), or a roles folder in the same directory as the main playbook file.

Ansible Playbooks: Organization - Roles

meta/main.yml

The main.yml file inside the meta folder is where the meta information for the role is defined. In simple roles, usually only contains the role dependencies, such as other roles which are required for the role to work.

A basic main.yml on the meta filder contains something like:

```
---
dependencies: []
```

tasks/main.yml

This file will contain the tasks exactly like the playbook would:

```
- name: Ensure NTP is installed.
yum: name=ntp state=present

- name: Ensure NTP is running.
service: name={{ ntp_daemon }} state=started enabled=yes
```

It would run exactly the same way and get the same output as without roles, except it contains the role name in a prefix for the task being run:

```
ntp | [Ensure NTP is installed].
```

Ansible Playbooks: Organization - Roles

So the final structure for the basic role should be something like:

```
playbook.yml
roles/
  ntp/
    meta/
      main.yml
    tasks/
      main.yml
```

Using the same structure, we can add much more, like files, handlers, templates or variables, among others.
And just like this we have a complete Ansible Role, ready to be reused.

Ansible Playbooks: Organization - Roles

Reusing roles is very easy and very powerful.

Imagine this: Adding a timezone variable to the ntp role, with the default value ‘UTC’. Then, creating a second playbook.yml file, with a different name and a different timezone value, with hosts for a different country in another continent.

Or imagine a playbook like this:

```
---
- hosts: appservers
  roles:
    - yum-repo-setup
    - firewall
    - nodejs
    - app-deploy
```

Each of the roles live in its own isolated world, and can be shared with other servers and groups of servers in our infrastructure.

- A yum-repo-setup role could enable certain repositories and import their GPG keys.
- A firewall role could have options for ports and services to allow or deny.
- An app-deploy role could deploy your app to a directory (configurable per-server) and set certain app options per-server or per-group.

Ansible Galaxy

Ansible Galaxy is a repository of community-contributed roles for common Ansible content.
There are hundreds of roles available on Ansible Galaxy, which can configure and deploy common applications and they are all available through the *ansible-galaxy* command.

Galaxy offers the ability to add, download, and rate roles. With an account, you can contribute your own roles or rate others' roles (though you don't need an account to use roles).

We can visit the official website (https://galaxy.ansible.com) to search the roles, manage and even see the files and details about them.

Ansible Galaxy

Getting a role from Galaxy

Roles must be downloaded before they can be used in playbooks.

We can download a role using the command:

```
ansible-galaxy install <role_name>
```

We can download the roles to create a LAMP server, just by running:

```
ansible-galaxy install geerlingguy.apache geerlingguy.mysql geerlingguy.php geerlingguy.php-mysql
```

We can then create an Ansible playbook named lamp.yml with the following content:

```
---
- hosts: all
  become: yes

  roles:
    - { role: geerlingguy.repo-epel, when: ansible_os_family == "RedHat" }
    - geerlingguy.mysql
    - geerlingguy.apache
    - geerlingguy.php
    - geerlingguy.php-mysql
```

If we add in a few variables, we can configure virtualhosts, PHP configuration options, MySQL server settings, etc.

Ansible Galaxy

Helpful Galaxy commands

Some other helpful ansible-galaxy commands you might use from time to time:

ansible-galaxy list displays a list of installed roles, with version numbers
ansible-galaxy remove <role_name> removes an installed role
ansible-galaxy init <role_name> can be used to create a role template suitable for submission to Ansible Galaxy

Ansible Galaxy

Contributing to Ansible Galaxy

If you’ve been working on some useful Ansible roles, and you’d like to share them with others, all you need to do is make sure they follow Ansible Galaxy’s basic template. To get started, use *ansible-galaxy init* to generate a basic Galaxy template, and make your own role match the Galaxy template’s structure.

Then push your role up to a new project on GitHub and add a new role while logged into https://galaxy.ansible.com, under the ‘My Content’ tab.

Ansible - Real World Examples

Provision Infrastructure (on the software side)

- Create user
- Install packages
- Ensure services are up: NTP

Docker

- Install Docker and Docker Compose
- Allow access to users

NGINX

- Install NGINX
- Add custom homepage

Ansible - Challenges

- Do all of the previous Playbooks, but using Ansible Galaxy
- Cover your own needs and start your own roles

Q&A

Thank you

Delivering talent.

Feel free to give feedback on this Workshop (or on possible future ones) or ask questions on luis.nabais@findmore.pt
Don’t be shy!