Politecnico di Torino

Configuration management with Ansible

Francesco Borgogni, Alex Palesandro

January 27, 2021

License

Acknowledgments

The author would like to thank all the people who contributed to this document.

Contents

1 Introduction

This lab aims at practicing with Ansible, an open source software provisioning, configuration management, and application-deployment tool.

First we will learn how to install and configure Ansible. Ansible needs to be installed only on a *master* machine: this will manage your nodes using a simple SSH connection.

Later we will study some commands you can execute to perform actions on your managed hosts. Ansible's power, however, does not reside in the command line, but in Playbooks, YAML-formatted files which allows you to specify different tasks. We will see how to write a playbook, its main fields and their meaning and how to run it, on a single node or a group of them.

Finally we will explore some advanced features Ansible provides us, such as variables, facts (a particular type of variables), flow execution control (loops and conditionals), handlers (sort of 'functions' that are triggered by an event) and Jinja2[1] templates.

Note: this lab was inspired from the work available on the following websites:

- `https://mylabs.readthedocs.io/en/latest/calm/lab6.html`
- `http://people.redhat.com/grieger/summit2018_labs/getting_started_ansible.html`
- `https://www.redhat.com/en/blog/system-administrators-guide-getting-started-ans`

[1] `https://en.wikipedia.org/wiki/Jinja_(template_engine)`

2 Environment setup

You will have 3 VMs on Crownlabs: one Cloud Client VM will be the *master*, on which you will install Ansible; the others will be your managed hosts (you can use, for example, the template "Cloud Computing: Ansible").

`<user>` should be `netlab` for every host.

2.1 Install Ansible

Let's install Ansible. Ansible is an **agent-less** system: this means that it does not require any additional software to be installed on the client computers. This is one way that Ansible simplifies the administration of servers. Any server that has an SSH port exposed can be brought under Ansible's configuration umbrella, regardless of what stage it is at in its life cycle.

As said, you need to install Ansible **only on your *master*** machine. For an Ubuntu machine the commands are the following[1]

```
sudo apt update
sudo apt install software-properties-common
sudo apt-add-repository --yes --update ppa:ansible/ansible
sudo apt install ansible
```

N.B. Ansible is already installed on the Cloud Client VM in Crownlabs

2.1.1 Initialize SSH

Before going on using Ansible, let us create a new pair of ssh keys to have a passwordless access to all the hosts we want to configure, which are the hosts that have to be controlled on Ansible.

From the host that you will use to control your infrastructure (the Ansible *master* machine), you have to type the following command:

```
ssh-keygen
```

And then copy our key on every host using:

```
ssh-copy-id netlab@<host_IP>
```

[1] https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html?extIdCarryOver=true&sc_cid=701f2000001OH7YAAW

3 Ansible basics: configuration and usage

Now that you have configured your environment you can start to work with Ansible. First of all you need to configure Ansible's hosts: Ansible keeps track of all of the servers that it knows through a *hosts* file. We need to set up this file first before we can begin to communicate with our other computers.

```
sudo nano /etc/ansible/hosts
```

Note: you can use the program you prefer, with nano you just have to insert the content of your file and then save it (`ctrl + O`) and exit (`ctrl + X`).

You will see a file that has a lot of example configurations commented out. Keep these examples in the file to help you learn Ansible's configuration if you want to implement more complex scenarios in the future. The hosts file is fairly flexible and can be configured in several different ways. The syntax we are going to use looks something like this

```
[<group_name>]
<alias> ansible_ssh_host=<server_ip>
```

Use the IP you have previously noted. You should have something similar to this

```
[servers]
node1 ansible_ssh_host=10.0.0.5
node2 ansible_ssh_host=10.0.0.6
```

Ansible will, by default, try to connect to remote hosts using your current username. If that user does not exist on the remote system, a connection attempt will result in an error. Let's specifically tell Ansible that it should connect to servers in the *servers* group with a *user*. Create a directory in the Ansible configuration structure called group_vars.

```
sudo mkdir /etc/ansible/group_vars
```

Within this folder, we can create YAML-formatted files for each group we want to configure:

```
sudo nano /etc/ansible/group_vars/servers
```

Add this code to the file:

```
---
ansible_ssh_user: <user>
ansible_python_interpreter: /usr/bin/python3
ansible_ssh_private_key_file: /home/netlab/.ssh/id_rsa
```

<user> should be `netlab`. The second line of the file specifies the path to the python interpreter: if the default variable points to a version 2 interpreter, an annoying warning is displayed every time you type a command. If you have problems you can delete this line and try with the default interpreter. The third one specifies the `key.pem` file for the ssh connection.

Save and close this file when you are finished. Now Ansible will always use the specified user for the servers group, regardless of the current user. If you want to specify configuration details for every machine, regardless of group association, you can put those details in a file at `/etc/ansible/group_vars/all`. Individual hosts can be configured by creating files under a directory at `/etc/ansible/host_vars`.

3.1 Execute command line tools

Now that we have our hosts set up and enough configuration details to allow us to successfully connect to our hosts, we can try out our very first `ansible` command[1]. Ping all the servers you configured by typing:

```
ansible -m ping all
```

You should see something like this

```
node1 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
node2 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

You can also ping a group (*servers*) or a single or multiple hosts by separating them with a colon:

```
ansible -m ping servers

ansible -m ping node1:node2
```

The `-m` option allows to specify the module to execute: if not specified, the default one is `command`. There are lots and lots of modules[2], for example `command` and `shell` are used to execute shell commands on the target machine[3]:

```
ansible -m shell -a 'ls -la' node1    # execute ls on node1

ansible -a 'ip addr' all     # show net information of all nodes
```

The `-a` option specifies the module arguments.

By now, you should have your *master* configured to communicate with the servers that you would like to control; with the `ansible` command you can execute simple tasks remotely. Although this is useful, we have not covered the most powerful feature of Ansible: Playbooks. You have configured a great foundation for working with your servers through Ansible, so your next step is to learn how to use Playbooks to do the heavy lifting for you.

[1] https://docs.ansible.com/ansible/latest/user_guide/command_line_tools.html
[2] https://docs.ansible.com/ansible/latest/modules/modules_by_category.html
[3] https://linux.die.net/man/8/ip

3.1.1 Ansible modules: shell vs command

In the most use cases `shell` and `command` modules lead to the same goal. The main differences between them are

- With the `command` module, the command will be executed without being proceeded through a shell. As a consequence some variables like `$HOME` are not available. Furthermore, stream operations like <, >, | and & will not work.
- The `shell` module runs a command through a shell, by default `/bin/sh`. This can be changed with the option `executable`. Piping and redirection are here available.
- The `command` module is more secure, because it will not be affected by the user's environment.

Note: before the usage of both modules you should check if there is not a more specific Ansible module for that task. It is always better to use a module instead of running a raw command, because the modules are designed to be idempotent and fulfill other standards like exception handling.[4]

As an example try these simple commands which redirect the output to a file

```
ansible -m shell -a 'ls -la > file' node1

ansible -m command -a 'ls -la > file' node1
```

As you can see, the former succeeds, the latter fails.

3.2 Ansible Playbooks

Playbooks are essentially sets of instructions (plays) that you send to run on a single target or groups of targets (hosts). They are written in YAML[5], a human-readable data serialization language, commonly used for configuration files. Playbooks start with the YAML three dashes (`---`) followed by:

- **name**: the name of the 'play', good for keeping the Playbooks readable
- **hosts**: identifies the target for Ansible to run against
- **become**: acquire super-user privileges before performing the listed tasks
- **tasks**: the operations to be performed by invoking Ansible modules with the necessary options.

Let's try to create a playbook to install and run nginx web server on one of our managed hosts.
Go to `/etc/ansible` folder and create a new file (name it as you prefer, e.g. *nginx.yaml*). Open it and paste this simple playbook

```
---
- name: Install nginx
  hosts: node1
  become: yes

  tasks:
  - name: Install nginx
    apt:
      name: nginx
      state: present
```

[4] https://blog.confirm.ch/ansible-modules-shell-vs-command/
[5] https://en.wikipedia.org/wiki/YAML

```
  - name: Start nginx
    service:
      name: nginx
      state: started
```

This code snipped is attached to the PDF file as "snippets/nginx.yaml"

Warning: YAML files strongly rely on indentation for their structure. Hence, do *not* copy the previous snippet directly from the PDF file, since it will totally scramble the indentation. Moreover, before moving on, verify (and fix, if necessary) the file layout using an editor of your choice.

Now from the command line move to the ansible folder and execute the playbook

```
ansible-playbook nginx.yaml
```

When the tasks are completed, you can check that nginx is running on the other node by opening the browser and navigate to web server ip address (e.g. *10.0.0.5*); another option is to use one of these commands (the first one only checks if the connection to the url is ok)

```
ansible localhost -m uri -a "url=http://<node1_ip>"

curl http://<node1_ip>
```

You can change the nginx default installation page by adding the following task to the playbook (obviously you need to have created an *index.html* file with some content in your working directory, here `/etc/ansible`)

```
- name: Insert index page
  copy:
    src: /etc/ansible/index.html
    dest: /var/www/html/index.html
```

If you rerun the playbook the output should look like as follows:

```
PLAY RECAP *********************************************************************
node1    : ok=3    changed=0    unreachable=0    failed=0    skipped=0
           rescued=0    ignored=0
```

This tells you that the nginx package was already installed, so nothing was changed. This is another powerful funcionalities of playbooks: they are **idempotent**. When dealing with complicated playbooks across many hosts, being able to identify the hosts that were different becomes very useful. For example, if you notice a host always needs a specific config updated, then there is likely a user or process on that host which is changing it. Without idempotence, this may never be noticed.

This was just a very short presentation of playbooks, but there are lots and lots of other features they provide. In the following section we will see some of them, for a complete guide visit the Ansible documentation[6].

[6] `https://docs.ansible.com/ansible/latest/user_guide/playbooks.html`

3.3 Ansible Console

Ansible offers a read-eval-print loop (REPL) language[7] for running ad-hoc tasks against a chosen inventory[8] (an inventory is a list of nodes, in brief the *hosts* file you have written in section 3). This environment is called Ansible console. You can enter it by typing:

```
ansible-console [-i <inventory>]
```

The `-i` option allows you to specify an inventory host path (different from the *hosts* file you wrote previously) or comma separated host list; if not specified, by default you will connect to all the machines.

Once entered in Ansible console you can use the normal Ansible modules. For example if you want to get the date on all the hosts, simply type

```
shell date
```

Remember: if you do not specify an Ansible module, `command`, the default one, will be used. Therefore, you could use the previous command without `shell` and you would have the same result.

If you press `<tab><tab>` you will see the (very long) list of modules you can use. A very useful one is `cd` which allows you to navigate through your groups or hosts: for example you could need to first execute a command on all your machines, then something specific only on the *servers* group (`cd servers`), afterwards something else only on *node1* (`cd node1`).

Warning: the `cd` command you have just used is *not* the Linux `cd` one you use to move between folders, even if its behaviour is pretty similar.

[7] A read eval print loop (REPL), also termed an interactive toplevel or language shell, is a simple, interactive computer programming environment that takes single user inputs (i.e., single expressions), evaluates (executes) them, and returns the result to the user. More information at `https://en.wikipedia.org/wiki/Read-eval-print_loop`

[8] `https://docs.ansible.com/ansible/latest/user_guide/basic_concepts.html`

4 Advanced features

What we have seen so far was nice, but the real power of Ansible is to apply the same set of tasks reliably to many hosts. First, try to extend the previous playbook (section 3.2) to run on 2 VM: this is quite simple since you have already configured different servers in your hosts file.
You just have to change the `hosts` field in the playbook:

```
hosts: servers
# (or)
hosts: node1:node2
```

Now let's see some advanced features of Ansible.

4.1 Ansible variables

Ansible supports variables to store values that can be used in Playbooks. Variables can be defined in a variety of places and have a clear precedence. Ansible substitutes the variable with its value when a task is executed. Variables are referenced in Playbooks by placing the variable name in double curly braces.

```
Here comes a variable {{ variable1 }}
```

The recommended practice is to define variables in files located in two directories named *host_vars* and *group_vars*: we have already defined variables for the group *servers* (in chapter 3), now repeat the previous steps to create host-specific variables.
Note: host variables take precedence over group variables.

Once you have created the `host_vars` directory, you need to create a file within it for one of your nodes; the file must be named as the node name (e.g. *node1*). Inside it, define a variable named `stage` and give it the value `prod` (write `stage: prod`). Define the same variable in the `group_vars/servers` file and give it the value `dev` (write `stage: dev` below the `ansible_*` variables).

Now create two index files: *index_dev.html* and *index_prod.html* and insert the following content

```
<body>
    <h1>This is a production webserver, take care!</h1>
</body>
```

This code snipped is attached to the PDF file as "`snippets/index_prod.html`"

```
<body>
    <h1>This is a development webserver, have fun!</h1>
</body>
```

This code snipped is attached to the PDF file as "`snippets/index_dev.html`"

Modify the *insert index page* task in the playbook

```
- name: Insert index page
  copy:
    src: index_{{ stage }}.html
    dest: /var/www/html/index.html
```

Rerun the playbook and check the two nodes: the index page should be different.

4.2 Ansible facts

Ansible *facts* are variables that are automatically discovered by Ansible from a managed host. Facts are pulled by the setup module and contain useful information stored into variables that administrators can reuse.
To get the complete list of facts Ansible collects by default run

```
ansible node1 -m setup
```

This will return a lot of information, so you can use filters to limit the output to certain facts (the following one returns memory related facts)

```
ansible node1 -m setup -a 'filter=ansible_*_mb'
```

Exercise: try to retrieve the linux distribution and kernel version of all your managed nodes (Suggestion: pass the output of `setup` module to `grep` command).

Facts can be used in a Playbook like variables, using the proper naming, of course. Create this Playbook as `facts.yml` and run it

```
---
- name: Output facts within a playbook
  hosts: all
  tasks:
  - name: Prints Ansible facts
    debug:
      msg: The default IPv4 address of {{ ansible_fqdn }} is {{ ansible_default_ipv4.
          address }}
```

This code snipped is attached to the PDF file as "`snippets/facts.yaml`"

4.3 Ansible loops

Often you'll want to do many things in one task, such as

- create a lot of users
- repeat a polling step until a certain result is reached

Ansible supports loops to iterate over a set of values, preventing administrators from writing repetitive tasks that use the same module. Ansible supports three main types of loops: simple loops, list of hashes and nested loops[1]. In this lab we will have a quick look at the first two.

[1] `https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html`

4.3.1 Simple loops

Simple loops are a list of items that Ansible iterates over. They are defined by providing a list of items to the loop keyword. Create the following Playbook as `simple_loop.yml` and run it

```
---
- name: Simple loop demo
  hosts: node1
  tasks:
    - name: Ensure that the nginx and sshd services are started
      service:
        name: "{{ item }}"
        state: started
      loop:
      - nginx
      - sshd
```

This code snipped is attached to the PDF file as "snippets/simple_loop.yaml"

In the following example the array is embedded in the playbook and called `check_services`. Create this Playbook as `simple_loop2.yml` and run it: the output should be the same.

```
---
- name: Simple loop demo 2
  hosts: node1
  vars:
    check_services:
    - nginx
    - sshd
  tasks:
  - name: Check if service is started
    service:
      name: "{{item}}"
      state: started
    loop: "{{check_services}}"
```

This code snipped is attached to the PDF file as "snippets/simple_loop2.yaml"

4.3.2 Hashes

Ansible allows also to loop over a list of hashes: this is particularly useful when working with more complex data. The following Playbook shows how a list of hashes with key-value pairs is passed to the `user` module[2], a module to manage user accounts and user attributes. Create as `hash_loop.yml` and run it

```
---
- name: Hash demo
  hosts: node1
  become: yes
  tasks:
    - name: Create users from hash
      user:
        name: "{{ item.name }}"
        state: present
        groups: "{{ item.groups }}"
```

[2] https://docs.ansible.com/ansible/latest/modules/user_module.html#user-module

```
    loop:
      - { name: 'jane', groups: 'wheel'}
      - { name: 'joe', groups: 'root'}
```

This code snipped is attached to the PDF file as "snippets/hash_loop.yaml"

User *jane* should not be created, since the group *wheel* does not exist; to check if user *joe* has been created you can simply run this command

```
ansible -a 'getent passwd' node1 | grep joe
```

To remove the user you only need to modify and add these two lines in the previous playbook (under the `user` module)

```
state: absent
remove: yes
```

4.4 Ansible conditionals

Ansible can use conditionals to execute tasks or plays when certain conditions are met. To implement a conditional, the `when` statement must be used, followed by the condition to test. The condition is expressed using one of the available operators, for example the standard comparison ones:

==	equality
!=	inequality
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

As an example you would like to install an FTP server, but only on hosts that are not in production because the protocol is not secure (remember in section 4.1 we defined the `stage` variable). Create this Playbook as `ftp.yml`, run it and examine the output

```
---
- name: Install insecure FTP server as long as not in production
  hosts: node1:node2
  become: yes
  tasks:
    - name: Install FTP server if not in production
      apt:
        name: vsftpd
        state: latest
      when: stage != "prod"
```

This code snipped is attached to the PDF file as "snippets/ftp.yaml"

Important

- in a `when` statement, facts and variables are not enclosed in double curly braces like you would do elsewhere in the playbook
- the `when` statement must be placed "outside" of the module by being indented at the top level of the task.

Expected outcome: the task is skipped on `node1` because it has the `stage` variable set to `prod` and succeeds on `node2` which has `stage = dev`.

Exercise: write a playbook that installs `MariaDB` only if the host has more than 3 GB of RAM.

Find the fact (4.2) for `memtotal` in MB (look at the ad-hoc command output and feel free to use `grep`). Use this Playbook as a template

```
---
- name: MariaDB server installation
  hosts: all
  become: yes
  tasks:
    - name: Install latest MariaDB server when host RAM greater 3 GB
      yum:
        name: mariadb-server
        state: latest
      when: <fact> <comparison_operator> <value>
```

This code snipped is attached to the PDF file as "`snippets/mariadb.yaml`"

Note: the VMs from Openstack should have less than 2 GB of ram, therefore the above playbook should be skipped in all your machines. If you want to actually install MariaDB, you can modify the value of the requested ram or the `hosts` field.

4.5 Ansible handlers

Sometimes when a task makes a change to the system, a further task may need to be run. For example, a change to a service's configuration file may then require the service to be reloaded so that the changed configuration takes effect.

Here Ansible's handlers come into play. Handlers can be seen as inactive tasks that only get triggered when explicitly invoked using the `notify` statement.
As a an example, let's write a Playbook that:

- modifies nginx's configuration file (`/etc/nginx/sites-available/default`) on all hosts in the *servers* group
- restarts nginx when the file has changed

Create a file named *default* in your working directory and copy the following content in it.

```
##
# You should look at the following URL's in order to grasp a solid understanding
# of Nginx configuration files in order to fully unleash the power of Nginx.
# https://www.nginx.com/resources/wiki/start/
# https://www.nginx.com/resources/wiki/start/topics/tutorials/config_pitfalls/
# https://wiki.debian.org/Nginx/DirectoryStructure
#
# In most cases, administrators will remove this file from sites-enabled/ and
# leave it as reference inside of sites-available where it will continue to be
# updated by the nginx packaging team.
#
# This file will automatically load configuration files provided by other
# applications, such as Drupal or Wordpress. These applications will be made
# available underneath a path with that package name, such as /drupal8.
```

```
#
# Please see /usr/share/doc/nginx-doc/examples/ for more detailed examples.
##

# Default server configuration
#
server {
    listen 80 default_server;
    listen [::]:80 default_server;

    # SSL configuration
    #
    # listen 443 ssl default_server;
    # listen [::]:443 ssl default_server;
    #
    # Note: You should disable gzip for SSL traffic.
    # See: https://bugs.debian.org/773332
    #
    # Read up on ssl_ciphers to ensure a secure configuration.
    # See: https://bugs.debian.org/765782
    #
    # Self signed certs generated by the ssl-cert package
    # Don't use them in a production server!
    #
    # include snippets/snakeoil.conf;

    root /var/www/html;

    # Add index.php to the list if you are using PHP
    index index.html index.htm index.nginx-debian.html;

    server_name _;

    location / {
        # First attempt to serve request as file, then
        # as directory, then fall back to displaying a 404.
        try_files $uri $uri/ =404;
    }

    # pass PHP scripts to FastCGI server
    #
    #location ~ \.php$ {
    #   include snippets/fastcgi-php.conf;
    #
    #   # With php-fpm (or other unix sockets):
    #   fastcgi_pass unix:/var/run/php/php7.0-fpm.sock;
    #   # With php-cgi (or other tcp sockets):
    #   fastcgi_pass 127.0.0.1:9000;
    #}

    # deny access to .htaccess files, if Apache's document root
    # concurs with nginx's one
    #
    #location ~ /\.ht {
    #   deny all;
    #}
}


# Virtual Host configuration for example.com
```

```
#
# You can move that to a different file under sites-available/ and symlink that
# to sites-enabled/ to enable it.
#
#server {
#   listen 80;
#   listen [::]:80;
#
#   server_name example.com;
#
#   root /var/www/example.com;
#   index index.html;
#
#   location / {
#       try_files $uri $uri/ =404;
#   }
#}
```

This code snipped is attached to the PDF file as "snippets/default"

Now create the playbook `nginx_conf.yaml`

```
---
- name: Manage nginx configuration
  hosts: node1
  become: yes
  tasks:
    - name: Copy Nginx configuration file
      copy:
        src: default
        dest: /etc/nginx/sites-available/default
      notify:
      - restart_nginx
  handlers:
  - name: restart_nginx
    service:
      name: nginx
      state: restarted
```

This code snipped is attached to the PDF file as "snippets/nginx_conf.yaml"

The `notify` section calls the handler only when the copy task changed the file. The `handlers` section defines a task that is only run on notification.

Run the Playbook. We didn't change anything in the file yet so there should not be any changed lines in the output and of course the handler shouldn't have been fired.
Now change the `listen 80` line in *default* file to:

```
listen 8080
```

Run the Playbook again: now the output should be a lot more interesting:

- *default* should have been copied over
- the handler should have restarted nginx

Nginx should now listen on port 8080; you can easily verify it by using the commands suggested in the previous section (3.2).

4.6 Ansible templates

Ansible uses Jinja2 templating to modify files before they are distributed to managed hosts; Jinja2 is one of the most used template engines for Python[3].

When a template for a file has been created, it can be deployed to the managed hosts using the `template` module, which supports the transfer of a local file from the control node to the managed hosts. As an example of using templates you will customize the *motd*[4] (the message shown by Ubuntu at login) file to contain host-specific data.
In the `/etc/ansible` directory create the template file *motd_facts.j2*

```
Welcome to {{ ansible_hostname }}.
{{ ansible_distribution }} {{ ansible_distribution_version}}
deployed on {{ ansible_architecture }} architecture.
```

This code snipped is attached to the PDF file as "snippets/motd_facts.j2"

Now create and run the *motd_facts.yml* playbook

```
---
  - name: Fill motd file with host data
    hosts: node1
    become: yes
    tasks:
      - template:
          src: motd_facts.j2
          dest: /etc/motd
          owner: root
          group: root
          mode: 0644
```

This code snipped is attached to the PDF file as "snippets/motd_facts.yaml"

To understand what has changed login to `node1` via SSH and check the 'motto of the day' message: you should see how Ansible replaces the variables with the facts it discovered from the system.

Exercise: change the template to use the FQDN (Fully Qualified Domain Name) hostname (Suggestion: use ansible facts (4.2)).

[3] https://palletsprojects.com/p/jinja/
[4] https://en.wikipedia.org/wiki/Motd_(Unix)

5 Ansible in a complex use case

Now that we have explored different features Ansible gives us, we can try to build a more complex playbook that setups a complex service that includes a firewall, a web service, a database service, and the associated loggers.

First of all, rename your managed hosts so that their name are significant: we need a *webserver*, a *dbserver* and a *logserver*. We need 3 services, but we only have 2 managed hosts: in order to run the following playbook you can

- use also the master machine, adding it to the *hosts* file[1] (in the footnote you can find a guide to do it)
- use one machine for two services (e.g. *webserver* + *dbserver*)

In the following we will assume you have 3 entries in your *hosts* file, something similar to this

```
[servers]
webserver ansible_ssh_host=10.0.0.5
dbserver ansible_ssh_host=10.0.0.5
logserver ansible_ssh_host=10.0.0.6
```

We want to create a playbook to:

- set firewall rules on our *servers*
- install and start `nginx` on *webserver*
- install `MariaDB` on *dbserver*
- enable `rsyslog` on *logserver* to receive messages from the other servers

In the following we will guide you in writing it; you can try by yourself if you prefer and then check the solution.

First let's create a playbook (*firewalld.yaml*) to install and start `firewalld` service[2]. The default firewall manager in ubuntu is `ufw`[3] so we also need to disable it.

```
---
- name: Install firewalld
  hosts: servers
  become: true
  tasks:
    - name: Install firewalld
      apt:
        name: firewalld
        state: present

    - name: Start firewalld
      service:
```

[1] https://www.middlewareinventory.com/blog/run-ansible-playbook-locally/
[2] https://en.wikipedia.org/wiki/Firewalld
[3] https://en.wikipedia.org/wiki/Uncomplicated_Firewall

```
        name: firewalld
        state: started

    - name: Disable ufw
      service:
        name: ufw
        state: stopped
```

This code snipped is attached to the PDF file as "snippets/firewalld.yaml"

As you can see, this playbook needs to be run on all the *servers* group, but our goal is to write a playbook with different *plays* executed on different hosts. Ansible, in order to write reusable playbooks, allows us to import a playbook inside another[4].
Furthermore, we can also include single *tasks*: to use this feature we need to write a yaml file with only a list of tasks to execute, and then use the `include_tasks` module in the main playbook.
Try to write a file (*nginx_tasks.yaml*) with only the tasks to install and start `nginx`: you should have a file like this

```
- name: Install nginx
  apt:
    name: nginx
    state: present

- name: Start nginx
  service:
    name: nginx
    state: started
```

Now we can write our main playbook in which we will also use the importing feature and some new modules

Note: you have to insert the ip address of your *logserver* in PLAY 5

```
---
# PLAY 1: firewalld on all servers
- name: Install firewalld           # by importing this playbook, firewalld is
  import_playbook: firewalld.yaml   # installed on all our servers at the beginning

# PLAY 2: nginx on webserver
- hosts: webserver
  become: yes
  tasks:
    - name: Install nginx
      include_tasks: nginx_task.yaml   # in order to reuse code we can include
                                       # the tasks we have already written

    - name: open firewall port
      firewalld:
        service: http
        immediate: true
        permanent: true
        state: enabled

    - name: set content directory
      file:
        path: /var/www/html
```

[4] https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_includes.html

```
        state: directory
        mode: u=rwx,g=rwx,o=rx,g+s

    - name: create default page content
      copy:
        content: "Welcome to {{ ansible_fqdn }} on {{ ansible_default_ipv4.address
            }}\n"
        dest: /var/www/html/index.html
        mode: u=rw,g=rw,o=r

# PLAY 3: MariaDB on dbserver
- hosts: dbserver
  become: yes
  tasks:
    - name: install MariaDB server
      apt:
        name: mariadb-server
        state: latest

    - name: enable and start MariaDB server
      service:
        name: mariadb
        enabled: yes
        state: started

# PLAY 4: rsyslog on logserver
- hosts: logserver
  become: yes
  tasks:
    - name: configure rsyslog remote log reception over udp
      lineinfile:                        # a module to change a single line in a file
        path: /etc/rsyslog.conf
        line: "{{ item }}"
        state: present
      with_items:
        - '$ModLoad imudp'
        - '$UDPServerRun 514'
      notify:
        - restart rsyslogd

    - name: open firewall port
      firewalld:
        port: 514/udp
        immediate: true
        permanent: true
        state: enabled

  handlers:
    - name: restart rsyslogd
      service:
        name: rsyslog
        state: restarted

#PLAY 5: rsyslog config on webserver and dbserver
- hosts: webserver:dbserver
  become: yes
  tasks:
    - name: configure rsyslog
      lineinfile:
        path: /etc/rsyslog.conf
```

```
        line: '*.* @<logserver_ip>:514'
        state: present
      notify:
        - restart rsyslogd

  handlers:
    - name: restart rsyslogd
      service:
        name: rsyslog
        state: restarted
```

This code snipped is attached to the PDF file as "snippets/service_chain.yaml"

This playbook is quite long and may be difficult to understand, therefore it has been split into 5 plays

- PLAY 1: executed on all *servers*, it installs `firewalld` by importing the playbook we wrote before
- PLAY 2: executed on *webserver*, it installs `nginx`, opens firewall ports so that `http` traffic can pass and creates a custom index page
- PLAY 3: executed on *dbserver*, it installs and enable `mariadb`
- PLAY 4: executed on *logserver*, it configures `rsyslog`[5] using `lineinfile`[6] module and opening firewall ports. After having modified the configuration file it restart `rsyslog` service by notifying a handler
- PLAY 5: executed on *webserver* and *dbserver*, it configures them so that log messages are sent to *logserver*. **Modify with your *logserver* ip**

You can verify the webserver hosts using `curl` as you have already done (section 3.2) and remote logging using the `logger` command on *webserver* and *dbserver* hosts and afterwards check *logserver*

```
ansible -m command -a 'logger hurray it works' webserver:dbserver

ansible -m command -a "grep 'hurray it works$' /var/log/syslog" logserver
```

[5] `https://en.wikipedia.org/wiki/Rsyslog`
[6] `https://docs.ansible.com/ansible/latest/modules/lineinfile_module.html`