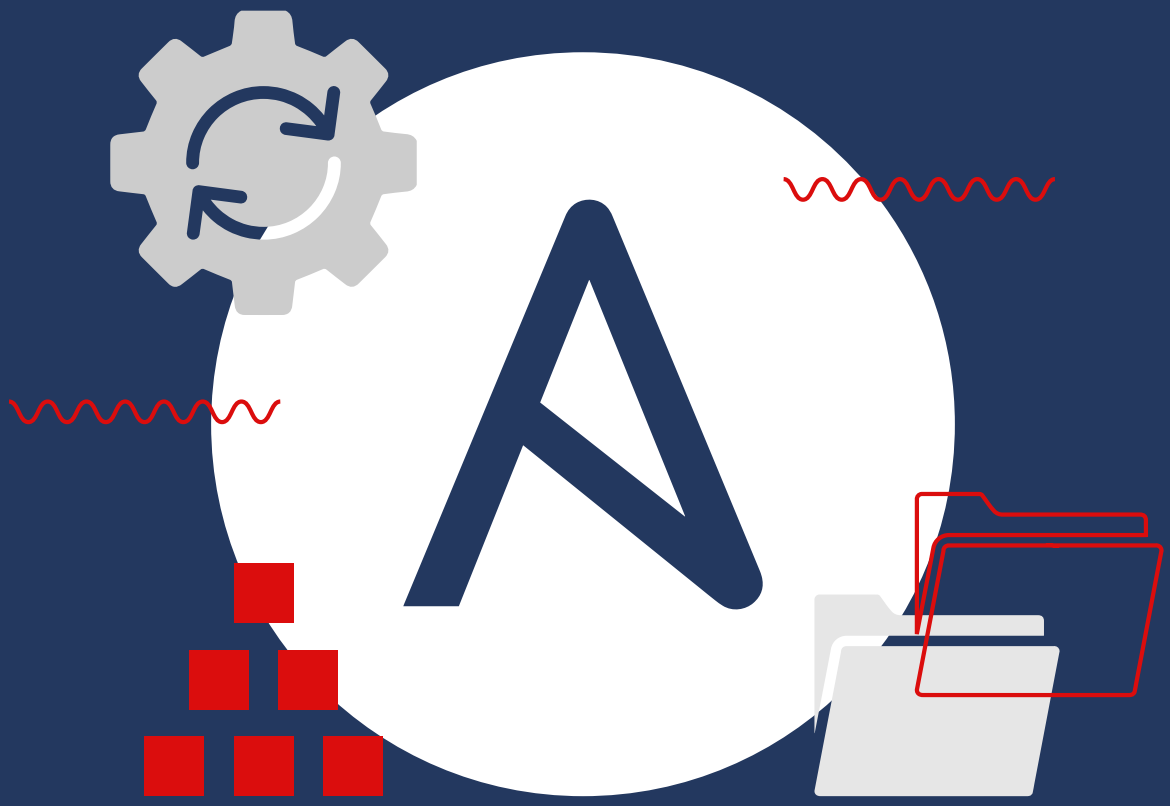
Marcin T. Ślęczek

THE WORLD OF AUTOMATION WITH

Red Hat Ansible



Open Virtualization Pro

2021

TABLE OF CONTENT:

01

ABOUT THE AUTHOR:
Marcin T. Ślęczek

Marcin works as CEO, Network Engineer and System Administrator at **networkers.pl**, which designs and implements IT Systems, Data Centers and DevOps environments. networkers.pl also sells software and hardware for building such environments and is a partner of well-known manufacturers such as Red Hat, Cisco Systems, IBM, Storware and VMware.

Marcin is Red Hat Certified Architect, Certified Kubernetes Administrator, Certified Kubernetes Security Specialist and also Cisco Certified Network Professional in the area of Data Center, Security and Enterprise Networks.

In 2006-2014 he conducted trainings in the area of GNU/Linux systems and Cisco Systems network solutions. He is also Cisco Certified Academy Instructor.

He has been working in the industry since 2000. Until 2013, he works with GNU/Linux systems, Enterprise Networks and Service Provider Networks. Between 2013 and 2017, he dealt mainly with Network Security, IP Telephony (VoIP), videoconferencing and Enterprise Networks. Since 2017, he works mainly with everything related to Data Centers and DevOps environments.

Open Virtualization Pro
Professional community focused on sharing knowledge about open-source technologies.

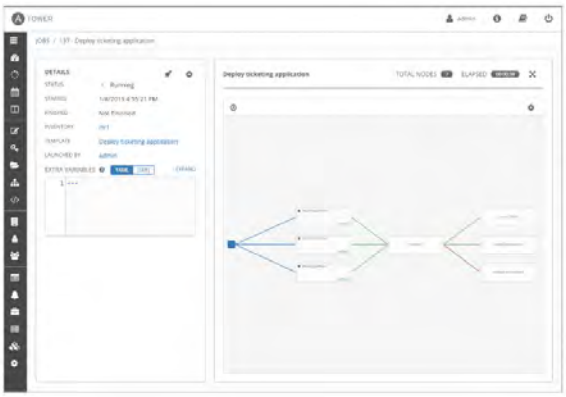
Portal Coordination and Editing: **Leszek Warzecha**

Portal Partners: storware Red Hat

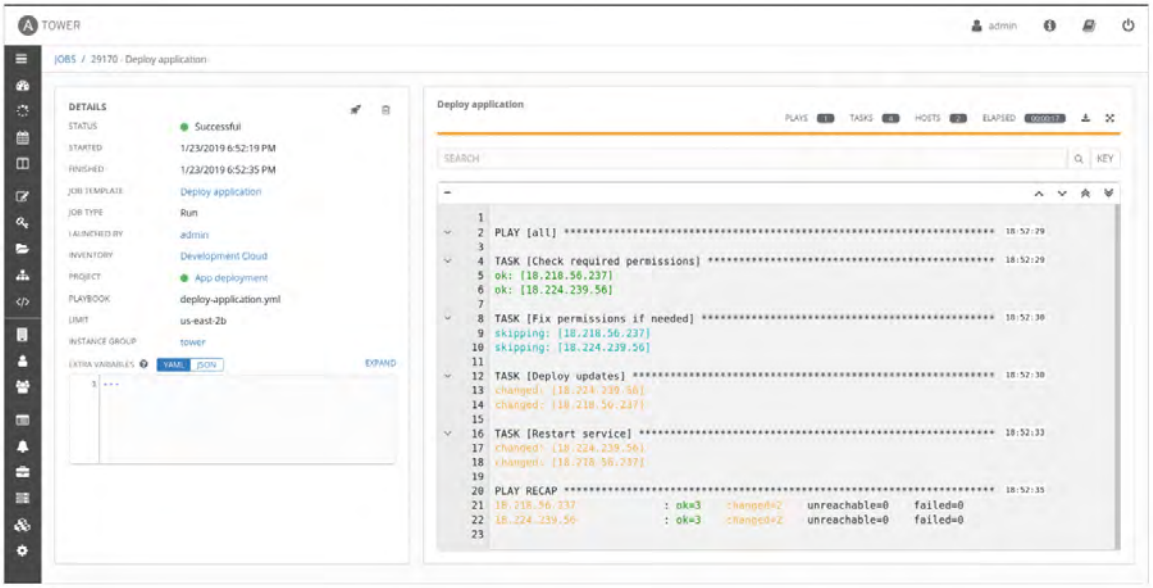
02

RED HAT ANSIBLE AUTOMATION
– Architecture and Features

Red Hat Ansible Automation is an open source automation platform. It is suitable for both low-level automation, which connects directly to each of the elements separately, and high-level, which is implemented through dedicated controllers from other manufacturers. In addition to the automation of other Red Hat Solutions, it also automates products and solutions from other manufacturers and binds automation within the entire infrastructure.



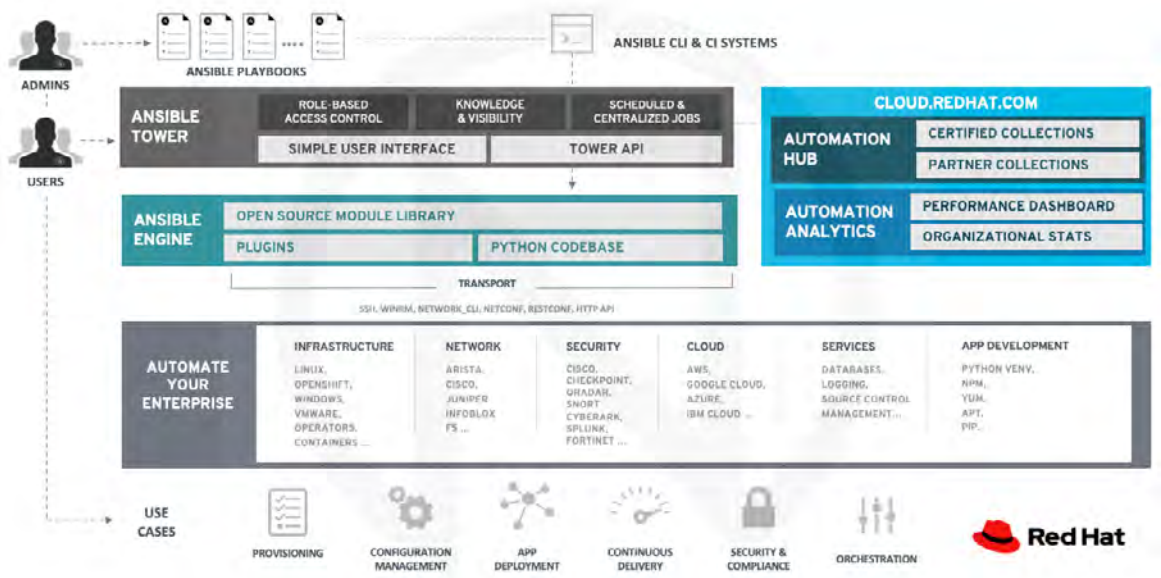
The platform provides CLI and WebUI as well as API for integration with external systems. It provides centralized insight into current and historical events. It can create very advanced task flows that can be dependent on each other and be conditionally performed. A schedule is also supported which enables cyclical tasks and facilitates compliance verification. Access to its individual elements can be differentiated and controlled using RBAC (Role Based Access Control). It also enables convenient integration with internal and external credential containers and user databases.



It uses natively available and popular management methods, which include SSH (Secure Shell), WinRM (Windows Remote Management), NETCONF (RFC 6241), RESTCONF (RFC 8040) or API. Thanks to this, it can usually be used without major changes within the infrastructure, such as firewall reconfiguration or the installation of additional software in the form of indirect agents.

The Red Hat Ansible Automation platform includes several elements, which include:

- **Ansible Engine** – an engine used to perform playbooks and ad hoc commands.
- **Ansible Tower** – central service, control and management of automation within the organization.
- **Automation Hub** – a portal containing certified collections, roles and modules.
- **Automation Analytics** – analytics, reports and statistics within clusters consisting of many Ansible Towers.



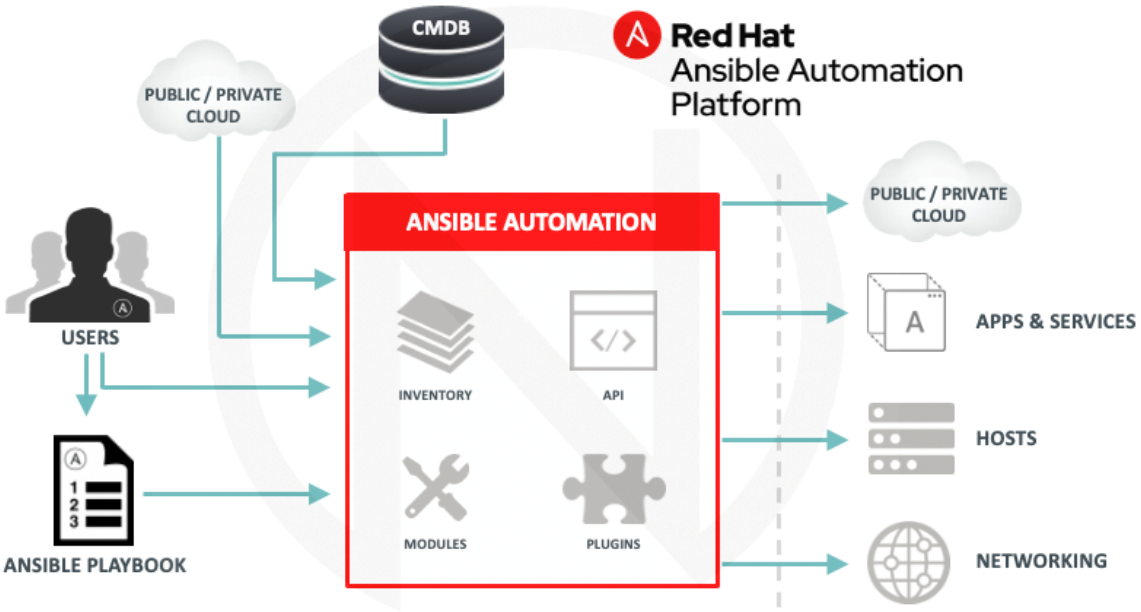
Red Hat Ansible Automation is a framework for automation that can be comfortably expanded with new functionalities. This is accomplished through additional plugins and **modules**. Their quantity is still growing.

The task of the Red Hat Ansible Automation platform is to bring devices, systems, files, applications or services to the desired state, which is described in a declarative language. This means that we only specify the state that we want the object to assume, without describing in detail how it must be done. The modules are already in charge of bringing it to a certain state. Depending on the initial state of the object, the module we use can perform more or less different operations on it, or even do not perform any operations at all, if it is not needed. This approach is characterized by **idempotence**, which here is the ability to repeatedly run the same module without changing the final state of the object.

It also means that we can safely execute the same module many times on a given object. If the object is already in the declared state, then nothing will happen, and if it is not, Ansible will bring the object to the desired state.

It should be remembered that idempotence is guaranteed only if we use specialized and built-in modules. If we create them ourselves or use other methods, such as **sending ad hoc commands**, then we have to take care of idempotence ourselves.

In addition to modules and commands, **we can also use Jinja2 templates**. They are great for personalizing the content of files based on the declared values of variables.



While the use of **Jinja2 templates** or the ability to issue CLI commands can be convenient and sometimes even necessary, these methods should be used as a last resort. If only a suitable specialized module is available, we should use it first.

In this way, we will not only be guaranteed idempotence, but also we can describe the entire infrastructure with one consistent code or language (IaC – Infrastructure as Code).

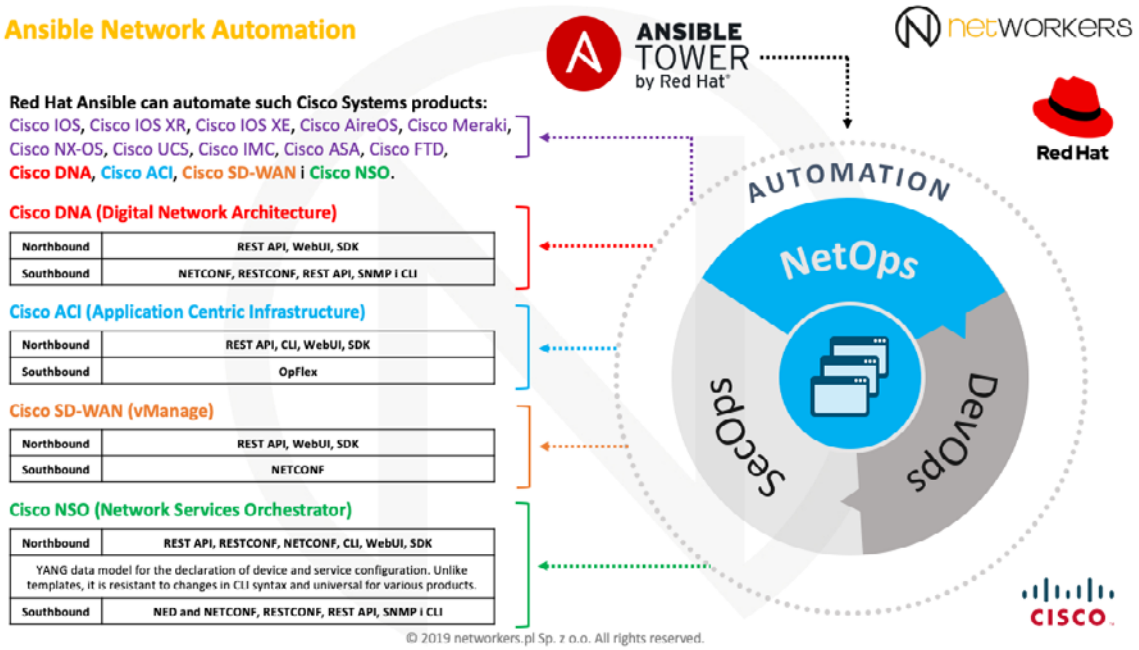
In the case of Ansible, the language is YAML (YAML Ain't Markup Language). With its help, we build playbooks, which are a sequential list of tasks to play on given objects, or in other words, the states that they should have after their execution. Please note, that various other operations may also be defined as part of the playbook, including copying the file, downloading something from somewhere, running specific CLI commands or using the Jinja2 templates. You can also use various loops, conditions, regular expressions, variables and handlers.

We also have the ability to perform ad hoc commands and tasks, as well as define more complex roles. Roles are a collection of files and playbooks that are necessary for a given device or system to perform a specific role or function in our environment.

Red Hat Ansible Automation supports both static and dynamic inventories. Static are created manually in INI or **YAML files**. Dynamic download data from other solutions, such as Amazon EC2, **Red Hat Satellite**, Cisco ACI (Application Centric

Infrastructure) or from the results of dynamically running scripts that download data from various databases.

Red Hat Ansible Automation is great for automating both small and large networks. In smaller networks, it can directly connect to devices, adapting them to specific requirements, and in larger networks, it can do so through dedicated systems, such as **Cisco NSO, Cisco SD-WAN, Cisco ACI or Cisco DNA**



The reason for using additional SDN (Software Defined Network) controllers in larger networks is not just automation and orchestration. These systems provide extensive diagnostic, analysis and visualization tools, which Ansible does not provide. Therefore, while Ansible cannot fully replace such solutions, it can cooperate with them by acting at a higher level. Manual clicking through a large number of tabs of various controllers is tedious, time-consuming and prone to errors and omissions. That is why it is much easier to automate it at the Ansible level.

In this way, **Red Hat Ansible Automation** can combine our entire infrastructure into one harmonious organism.

03

ANSIBLE, PART I:
– Basics and Inventory

Making changes manually is tedious, and problematic in terms of documenting. **Ansible** addresses both of these problems. It's great for what we don't like. It is doing the same thing on many elements. In addition, the resulting YAML file becomes a form of documentation. It certainly will not replace the typical documentation with diagrams, tables and additional notes. However, it is a good reference point when we want to check what has been configured.

Ansible allows you to describe the infrastructure and its services using code (IoC – Infrastructure as Code). If we additionally take care of appropriate comments and order in the file structure, Ansible can replace quite some elements of typical documentation, especially dynamic ones that require frequent updates. Creating such a code may seem like a big investment of time, but it is not at all, and it will pay off in the long run. When we need to configure further infrastructure elements in a similar way, it can easily be done. Without a thorough analysis of the configuration of what is already there and without fear of missing something. Of course, good knowledge of what we automate is important. No tool can replace it. That's why it is better, when this is done by experienced experts in a given field. Everything else related to using Ansible is very simple.

While Ansible can be used to automate various components, this must be done from the GNU/Linux system. In our example, this will be exactly **RHEL (Red Hat Enterprise Linux)**. In the RHEL8 system, this requires activation of the selected repository and a typical installation of the package:

```
[root@vm0-net ~]# subscription-manager repos --enable ansible-2-for-rhel-8-x86_64-rpms
[root@vm0-net ~]# yum install ansible
```

Before we start automating, we must define the inventory on which we will operate. In its simplest form, it can be a static text file in INI or YAML format. There is also the possibility of using a dynamic inventory. Inventory data is then taken from other solutions, such as **Red Hat Satellite** or through dynamic scripts from various databases.

The inventory contains a set of nodes, such as devices or servers, which can be grouped in any way. They can belong to many groups at the same time or not

belong to any of them. Other subgroups can also be members of one group. There are two built-in groups **“all”** and **“ungrouped“**. The “**all**” group implicitly contains all nodes from the inventory, and the “**ungrouped**” group implicitly contains all nodes from the inventory that are not members of any other group. Inside the inventory, you can define variables associated with groups and individual nodes. While we used here the term nodes, Ansible uses the term hosts in its place, regardless of whether these nodes are servers, a network device or something else.

You should not use “-” in group names, and the same host and group names in the inventory file. In the latter case, the group name will be omitted and only the host will be included. Group names should only use letters, numbers and “_”, the number cannot be the first character of the name.

Most often, the INI format file is used to build the inventory. We will also use it in most examples. Its simplest form is a list of IP addresses and domain names (one per line).

```
vm1-net.int.networkers.pl
vm2-net.int.networkers.pl
vm3-net.int.networkers.pl
vm4-net.int.networkers.pl
10.8.232.61
10.8.232.62
```

The INI format allows you to create sections by using square brackets “[]“. This is how you define groups. Each section is a separate group. The section name corresponds to the group’s name. Groups are used very often, because we usually want to perform specific tasks on a specific group of hosts. An example inventory file is shown below, which allows convenient operations on all servers with a **specific function** or **located in a given data centre** or **operating in a given environment**.

```
[web_servers]
web1-net.int.networkers.pl
web2-net.int.networkers.pl

[db_servers]
db1-net.int.networkers.pl
db2-net.int.networkers.pl

[dns_servers]
10.8.232.61
10.8.232.62

[east_datacenter]
web1-net.int.networkers.pl
db1-net.int.networkers.pl
10.8.232.61
```

```
[west_datacenter]
web2-net.int.networkers.pl
db2-net.int.networkers.pl
10.8.232.62

[production]
web1-net.int.networkers.pl
web2-net.int.networkers.pl
db1-net.int.networkers.pl
db2-net.int.networkers.pl
10.8.232.61
10.8.232.62

[development]
vm1-net.int.networkers.pl
vm2-net.int.networkers.pl
10.8.232.123
10.8.232.124
```

As you can see above, practically any grouping method is allowed. We also mentioned the possibility of creating nested groups, consisting not only of hosts but also of other groups. The "**:children**" suffix is used for this purpose, which is added to the group name.

An example of using nested groups can be seen below, where the "**web_servers**" and "**db_servers**" groups belong to the "**db_and_web**" group, and the "**web_servers**" and "**db_server**" groups and two hosts "**10.8.232.61**" and "**10.8.232.62**" belong to the "**production**" group.

```
[root@vm0-net ~]# cat inventory

vm1-net.int.networkers.pl
vm2-net.int.networkers.pl
10.8.232.123
10.8.232.124

[web_servers]
web1-net.int.networkers.pl
web2-net.int.networkers.pl

[db_servers]
db1-net.int.networkers.pl
db2-net.int.networkers.pl

[db_and_web:children]
web_servers
db_servers

[production]
10.8.232.61
10.8.232.62

[production:children]
web_servers
db_servers
[root@vm0-net ~]#
```

As you can see above, while the group can contain hosts and other groups, they are added in different sections.

Group membership can be verified using the command:

```
$ ansible -i <inventory_file> --list-hosts <group_name>
```

Below, we use the "-i" option to indicate the inventory file named "inventory", and the "–list-hosts" option to display hosts belonging to the given group. As you can see below, the "**web_servers**" and "**db_servers**" groups have 2 different hosts:

```
[root@vm0-net ~]# ansible -i inventory --list-hosts db_and_web
 hosts (4):
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
[root@vm0-net ~]#
```

Thus, 4 hosts belong to the "**db_and_web**" group, as the "**db_servers**" and "**web_servers**" groups are its members:

```
[root@vm0-net ~]# ansible -i inventory --list-hosts db_and_web
 hosts (4):
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
[root@vm0-net ~]#
```

The "**production**" group includes 6 hosts (2 from "**web_servers**", 2 from "**db_servers**" and 2 from "**production**"):

```
[root@vm0-net ~]# ansible -i inventory --list-hosts production
 hosts (6):
 10.8.232.61
 10.8.232.62
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
[root@vm0-net ~]#
```

You can also use several inventory files and then operate on a combined set. This applies to all groups.

```
[root@vm0-net ~]# cat inventory1
[production]
10.8.232.61
10.8.232.62

[root@vm0-net ~]# cat inventory2
[production]
10.8.232.63
10.8.232.64

[root@vm0-net ~]# ansible -i inventory1 -i inventory2 --list-hosts production
 hosts (4):
 10.8.232.61
 10.8.232.62
 10.8.232.63
 10.8.232.64
[root@vm0-net ~]#
```

Let's return to the aforementioned built-in groups: "**upgrouped**" and "**all**". The first contains **4 hosts** located at the very top of our inventory file, in the place where hosts not belonging to any groups are defined.

```
[root@vm0-net ~]# ansible -i inventory --list-hosts ungrouped
 hosts (4):
 vm1-net.int.networkers.pl
 vm2-net.int.networkers.pl
 10.8.232.123
 10.8.232.124
[root@vm0-net ~]#
```

And the second contains all the hosts that were defined in the inventory file.

```
[root@vm0-net ~]# ansible -i inventory --list-hosts all
 hosts (10):
 vm1-net.int.networkers.pl
 vm2-net.int.networkers.pl
 10.8.232.123
 10.8.232.124
 10.8.232.61
 10.8.232.62
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
[root@vm0-net ~]#
```

Our file can also be slightly simplified using the "**[START: END]**" range in the host name or address:

- 10.8.232.12**[0:9]** – IPv4 addresses from 10.8.232.120 to 10.8.232.129.
- 10.8.22**[4:5].[0:255]** – the entire subnet 10.8.224.0/23, IPv4 addresses from 10.8.224.0 to 10.8.225.255.

• 2001:db8:c1sc0::[a:f] – IPv6 addresses from 2001:db8:c1sc0::a to 2001:db8:c1sc0::f.

• web[1:20].networkers.pl – FQDN from web1.networkers.pl, through web8.networkers.pl to web20.networkers.pl.

• web[01:20].networkers.pl – FQDN from web01.networkers.pl, through web08.networkers.pl to web20.networkers.pl.

When comparing the last two examples, it is worth noting that leading zeros are left in the pattern.

The simplified version of our previous inventory file looks like this:

```
[root@vm0-net ~]# cat inventory

vm[1:2]-net.int.networkers.pl
10.8.232.12[3:4]

[web_servers]
web[1:2]-net.int.networkers.pl

[db_servers]
db[1:2]-net.int.networkers.pl

[production]
10.8.232.6[1:2]

[production:children]
web_servers
db_servers

[db_and_web:children]
web_servers
db_servers

[root@vm0-net ~]#
```

Verification of its correctness is below:

```
[root@vm0-net ~]# ansible -i inventory --list-hosts ungrouped
 hosts (4):
 vm1-net.int.networkers.pl
 vm2-net.int.networkers.pl
 10.8.232.123
 10.8.232.124
[root@vm0-net ~]# ansible -i inventory --list-hosts db_servers
 hosts (2):
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
[root@vm0-net ~]# ansible --inventory inventory --list-hosts db_and_web
```

```
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
[root@vm0-net ~]#
```

You can also add comments to the inventory file. Each comment line should start with a "#".

While the default location of the inventory file is "**/etc/ansible/hosts**", it is not used in practice. Usually, a file is created in the project directory and then selected by the "-i", "**–inventory**" options or through the "**ansible.cfg**" configuration file setting.

The location of the used configuration file can be checked using the "**ansible –version**" command. Below you can see that this is the "**/etc/ansible/ansible.cfg**" file.

```
[root@vm0-net ~]# ansible --version
ansible 2.9.7
 config file = /etc/ansible/ansible.cfg
 configured module search path = ['/root/.ansible/plugins/modules', '/usr/share/ansible/
plugins/modules']
 ansible python module location = /usr/lib/python3.6/site-packages/ansible
 executable location = /usr/bin/ansible
 python version = 3.6.8 (default, Dec 5 2019, 15:45:45) [GCC 8.3.1 20191121 (Red Hat
8.3.1-5)]
[root@vm0-net ~]#
```

This is the default file that can be overwritten with "**~/.ansible.cfg**". If we need a different configuration for each project, then they can be overwritten with the "**./ansible.cfg**" file, in the directory where the "ansible" command will be executed.

You can also use the "**$ANSIBLE_CONFIG**" variable, which is useful when we use many configuration files and do not want to move between different directories.

The search order for the Ansible configuration file is summarized below:
1. "**$ANSIBLE_CONFIG**"
2. "**./ansible.cfg**"
3. "**~/.ansible.cfg**"
4. "**/etc/ansible/ansible.cfg**"

Thus, the default configuration file is only used if no other is found. The best practice is to use the "**ansible.cfg**" file in the directory from which we run the "ansible" command.

```
[msleczek@vm0-net projekt_A]$ pwd
/home/msleczek/projekt_A
[msleczek@vm0-net projekt_A]$ ansible --version | grep "config file"
 config file = /etc/ansible/ansible.cfg
[msleczek@vm0-net projekt_A]$ touch ~/.ansible.cfg
[msleczek@vm0-net projekt_A]$ ansible --version | grep "config file"
 config file = /home/msleczek/.ansible.cfg
[msleczek@vm0-net projekt_A]$ touch ansible.cfg
[msleczek@vm0-net projekt_A]$ ansible --version | grep "config file"
 config file = /home/msleczek/projekt_A/ansible.cfg
[msleczek@vm0-net projekt_A]$ touch /home/msleczek/ansible-projekt_A.cfg
[msleczek@vm0-net projekt_A]$ export ANSIBLE_CONFIG=/home/msleczek/ansible-projekt_A.cfg
[msleczek@vm0-net projekt_A]$ ansible --version | grep "config file"
 config file = /home/msleczek/ansible-projekt_A.cfg
[msleczek@vm0-net projekt_A]$ rm /home/msleczek/ansible-projekt_A.cfg
[msleczek@vm0-net projekt_A]$ ansible --version | grep "config file"
 config file = /home/msleczek/projekt_A/ansible.cfg
[msleczek@vm0-net projekt_A]$ unset ANSIBLE_CONFIG
[msleczek@vm0-net projekt_A]$
```

Please note, that the file indicated by the "$ANSIBLE_CONFIG" variable must exist. If **Ansible** can't find it, it will go to the next place, which is "./ansible.cfg". This can be seen above when we deleted this file before the "$ANSIBLE_CONFIG" variable was deleted.

Additionally, **Ansible** will not load the "ansible.cfg" file from the current directory if that directory has world-writable permission. This is for security reasons, because another user may have tossed a properly crafted file there. If for some reason, that we need to use such a directory, it will be required to put the path to it in the variable "$ANSIBLE_CONFIG". However, it is a good idea to think about it beforehand and consider the resulting security risk.

The **Ansible** configuration file is in INI format. It consists of sections defined by square brackets "[]", which contain settings in the "**key = value**" format. The two most commonly used sections are:

- **[defaults]** – default Ansible settings,
- **[privilege_escalation]** – defines how Ansible escalates permissions on managed hosts.

In practice, this file is created to indicate the inventory and the methods to connect to hosts within a given project. An example of the contents of the "ansible.cfg" file in this respect can be seen below.

```
[msleczek@vm0-net projekt_A]$ cat ansible.cfg
[defaults]
inventory = ./inventory
remote_user = user
ask_pass = false

[privilege_escalation]
become = true
become_method = sudo
become_user = root
become_ask_pass = false
[msleczek@vm0-net projekt_A]$
```

The names of the parameters are so intuitive, that they indicate their use directly:

- **inventory** – path to the inventory file (you can overwrite "**-i**" or "**–inventory**").
- **remote_user** – username used during login process (current user by default).
- **ask_pass** – whether it should ask for a password (if used public key for SSH, it should be "**false**").
- **become** – whether to escalate permissions automatically (default "**false**").
- **become_method** – method used to escalate permissions (default "**sudo**").
- **become_user** – user to which the escalation takes place on the managed host ("**root**" by default).
- **become_ask_pass** – whether to ask for a password during escalation (default "**false**").

Thanks to the appropriate configuration of this file, you don't have to enter all the parameters, which makes work more convenient. For example, it can be seen below that we no longer have to indicate the inventory.

```
[msleczek@vm0-net projekt_A]$ ansible --list-hosts production
 hosts (6):
 10.8.232.61
 10.8.232.62
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
[msleczek@vm0-net projekt_A]$
```

This is the syntax of the command we use to verify the inventory:

```
$ ansible [pattern] --list-hosts
```

Until now, we have provided single hosts or groups in "**[pattern]**". However, this can be any pattern, which will be further adapted to the objects defined in the

inventory. It may consist of many elements which are separated by ":" or ",". Special characters such as "**!**", "**&**" or "*" may also appear in it. In these cases, the pattern should be placed inside a single quotation mark – '**pattern**' – or preceded by special characters using the backslash character '\'. Actually, you can always specify the whole pattern in a single quotation mark – '**pattern**' – then it will be universal, and you won't encounter any problems.

Examples of commonly used matching patterns are:

- **host1:host2** or **host1,host2** – sum of hosts (multiple hosts).
- **web_servers:db_servers** or **web_servers,db_servers** – all hosts from both groups.
- **production:!db_servers** or **production,!db_servers** – all hosts from production, but without those from db_servers.
- **production:&db_servers** or **production,&db_servers** – all production hosts that are in db_servers.
- ***networkers.pl** – all inventory objects with the name "networkers.pl" at the end.
- **10.8.232.*** – all inventory items having the name "10.8.232." At the beginning.

Before we show you how these patterns work, let's check our inventory:

```
[msleczek@vm0-net projekt_A]$ grep inventory ansible.cfg
inventory = ./inventory3
[msleczek@vm0-net projekt_A]$ cat inventory3
[web_servers]
web1-net.int.networkers.pl
web2-net.int.networkers.pl

[db_servers]
db1-net.int.networkers.pl
db2-net.int.networkers.pl

[dns_servers]
10.8.232.61
10.8.232.62

[east_datacenter]
web1-net.int.networkers.pl
db1-net.int.networkers.pl
10.8.232.61

[west_datacenter]
web2-net.int.networkers.pl
db2-net.int.networkers.pl
10.8.232.62

[production]
web1-net.int.networkers.pl

web2-net.int.networkers.pl
```

```
db1-net.int.networkers.pl
db2-net.int.networkers.pl
10.8.232.61
10.8.232.62

[development]
vm1-net.int.networkers.pl
vm2-net.int.networkers.pl
10.8.232.123
10.8.232.124

[msleczek@vm0-net projekt_A]$
```

The list of hosts or groups in the '**pattern**' does not have to be limited to 2 objects:

```
msleczek@vm0-net projekt_A]$ ansible 10.8.232.61,db2-net.int.networkers.pl,10.8.232.62
--list-hosts
 hosts (3):
 10.8.232.61
 db2-net.int.networkers.pl
 10.8.232.62
[msleczek@vm0-net projekt_A]$
```

Below you can see the sum, difference and intersection of the two groups of sets:

```
[msleczek@vm0-net projekt_A]$ ansible 'production:db_servers' --list-hosts
 hosts (6):
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
 10.8.232.61
 10.8.232.62
[msleczek@vm0-net projekt_A]$ ansible 'production:!db_servers' --list-hosts
 hosts (4):
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 10.8.232.61
 10.8.232.62
[msleczek@vm0-net projekt_A]$ ansible 'production:&db_servers' --list-hosts
 hosts (2):
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
[msleczek@vm0-net projekt_A]$
```

The '*' sign enables convenient matching to parts of a given pattern. In this way, you can easily refer to hosts working within a specific domain or having specific IP addresses. The given patterns must match the objects in the inventory. So, if we give everything by name, then the attempt to search by IP addresses will fail.

Below is an example of the use of '*' in the formula with the name and IP address (please note that this is a special character):

```
[msleczek@vm0-net projekt_A]$ ansible \*networkers\* --list-hosts
 hosts (6):
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
 vm1-net.int.networkers.pl
 vm2-net.int.networkers.pl
[msleczek@vm0-net projekt_A]$ ansible '*networkers*' --list-hosts
 hosts (6):
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
 vm1-net.int.networkers.pl
 vm2-net.int.networkers.pl
[msleczek@vm0-net projekt_A]$ ansible '10.8.232.6*': --list-hosts
 hosts (2):
 10.8.232.61
 10.8.232.62
[msleczek@vm0-net projekt_A]$
```

Previously obtained results can be further narrowed, creating more complex patterns:

```
[msleczek@vm0-net projekt_A]$ ansible '10.8.232.6*:&east_datacenter' --list-hosts
 hosts (1):
 10.8.232.61
[msleczek@vm0-net projekt_A]$
```

Some of the more complex patterns can be quite useful. For example, we may want to perform operations on all web servers and databases that work in a production environment but are not located in a western data center:

```
[msleczek@vm0-net projekt_A]$ ansible 'web_servers:db_servers:&production:!west_
datacenter' --list-hosts
 hosts (2):
 web1-net.int.networkers.pl
 db1-net.int.networkers.pl
[msleczek@vm0-net projekt_A]$
```

You can also use regular expressions as part of the pattern. To use them, precede the pattern with '~'. Many special characters can be used in this pattern, so it's best to enter it in single quotation marks.

```
[msleczek@vm0-net projekt_A]$ ansible '~10.8.232.6[12]' --list-hosts
 hosts (2):
 10.8.232.61
 10.8.232.62
```

```
[msleczek@vm0-net projekt_A]$ ansible '~(web|db)_*' --list-hosts
 hosts (4):
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
[msleczek@vm0-net projekt_A]$
```

You can also refer the hosts by their position number within the given group. The square brackets "**[]**" are used for this purpose. This way you can indicate both the item number "**[POSITION]**" and the range "**[START: END]**". The position numbers within the group start at zero.

```
[msleczek@vm0-net projekt_A]$ ansible 'production' --list-hosts
 hosts (6):
 web1-net.int.networkers.pl
 web2-net.int.networkers.pl
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
 10.8.232.61
 10.8.232.62
[msleczek@vm0-net projekt_A]$ ansible 'production[0]' --list-hosts
 hosts (1):
 web1-net.int.networkers.pl
[msleczek@vm0-net projekt_A]$ ansible 'production[1]' --list-hosts
 hosts (1):
 web2-net.int.networkers.pl
[msleczek@vm0-net projekt_A]$ ansible 'production[-1]' --list-hosts
 hosts (1):
 10.8.232.62
[msleczek@vm0-net projekt_A]$ ansible 'production[2:]' --list-hosts
 hosts (4):
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
 10.8.232.61
 10.8.232.62
[msleczek@vm0-net projekt_A]$ ansible 'production[2:3]' --list-hosts
 hosts (2):
 db1-net.int.networkers.pl
 db2-net.int.networkers.pl
[msleczek@vm0-net projekt_A]$
```

Mastering the rules of using the inventory and the **Ansible** configuration file is a necessary basis for further work. Lack of knowledge in this area will be behind us all the time, which is why we recommend taking the proper amount of time to these topics before moving on to the next articles.

04

ANSIBLE, PART II
– Modules and ad-hoc Commands

Sending ad-hoc commands is very simple and at the same time opens up many very useful possibilities. Thanks to them, we can conveniently issue single command on one or many nodes. Issuing such commands allows among others: querying for various statuses, compliance verification, configuration backup, node or service restart, file, package and user accounts management, software upgrade and even periodic password change.

To issue ad-hoc commands, use the "ansible" command:

```
$ ansible [pattern] -m [module] -a "[module options]"
$ ansible [pattern] --module-name [module] --args "[module options]"
```

- **[pattern]** – inventory matching pattern that was described in detail in the previous article.
- **[module]** – the name of the module.
- **[module options]** – options and arguments for the used module.

Ansible configuration file located below assumes that **we connect to all hosts using the "ansible" user without entering a password**. This requires that you first generate a key pair on the managing host for the SSH service and then send the public key to the appropriate location on the "**ansible**" user account of the managed nodes. After logging in, **the "root" user rights are escalated using "sudo" without entering any password**. Appropriate configuration of the sudo service on managed nodes is required for this purpose but as we focus only on Ansible, we skip these elements

```
[msleczek@vm0-net projekt_A]$ cat ansible.cfg
[defaults]
inventory = ./inventory
remote_user = ansible
ask_pass = false

[privilege_escalation]
become = true
become_method = sudo
become_user = root
become_ask_pass = false
[msleczek@vm0-net projekt_A]$
```

Our inventory file is very simple. It has 4 IPv4 addresses for servers with the RHEL8 system.

```
[msleczek@vm0-net projekt_A]$ cat inventory
10.8.232.121
10.8.232.122
10.8.232.123
10.8.232.124
[msleczek@vm0-net projekt_A]$
```

Most often, the INI format file is used to build the inventory. We will also use it in most examples. Its simplest form is a list of IP addresses and domain names (one per line).

```
vm1-net.int.networkers.pl
vm2-net.int.networkers.pl
vm3-net.int.networkers.pl
vm4-net.int.networkers.pl
10.8.232.61
10.8.232.62
```

Before we start, let's see if we can manage remote nodes. The "**ping**" module is used to verify the ability to manage remote nodes. This module checks whether it can login to them and escalate permissions, as well as verifies the availability of Python. In fact, it requires the remote node to have Python installed. If everything goes correctly, it returns "**pong**" by default. This module does not generate any ICMP packets.

```
[msleczek@vm0-net projekt_A]$ ansible all -m ping
10.8.232.122 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
10.8.232.123 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
10.8.232.121 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
10.8.232.124 | SUCCESS => {
```

```
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
[msleczek@vm0-net projekt_A]$
```

Ansible uses modules to perform specific tasks on hosts. When we use ad-hoc commands and do not provide the module name, the "**command**" module will be used by default. It executes a binary file on the remote node. Its operation is not affected by any shell environment variables. It does not use the remote system shell at all. It only executes the indicated binary file with the given arguments and returns the formatted result of its execution to us. Examples of explicit and implicit use of the "**command**" module are shown below:

```
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.123 -m command -a "df -h"
10.8.232.123 | CHANGED | rc=0 >>
Filesystem                      Size  Used Avail Use% Mounted on
devtmpfs                        440M     0  440M   0% /dev
tmpfs                           456M     0  456M   0% /dev/shm
tmpfs                           456M  6.2M  450M   2% /run
tmpfs                           456M     0  456M   0% /sys/fs/cgroup
/dev/mapper/rhel_vm3--net-root   21G  1.8G   20G   9% /
/dev/sda1                      1014M  185M  830M  19% /boot
tmpfs                            92M     0   92M   0% /run/user/1000
tmpfs                            92M     0   92M   0% /run/user/1001
[msleczek@vm0-net projekt_A]$ ansible all -a "uptime"
10.8.232.121 | CHANGED | rc=0 >>
 15:02:29 up 1 day, 20:38, 2 users, load average: 0.07, 0.02, 0.00
10.8.232.123 | CHANGED | rc=0 >>
 15:02:14 up 1 day, 19:58, 2 users, load average: 0.34, 0.08, 0.03
10.8.232.122 | CHANGED | rc=0 >>
 15:03:08 up 1 day, 20:15, 2 users, load average: 0.15, 0.03, 0.01
10.8.232.124 | CHANGED | rc=0 >>
 15:01:25 up 1 day, 19:18, 2 users, load average: 0.00, 0.00, 0.00
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.123 -a "free -h"
10.8.232.123 | CHANGED | rc=0 >>
              total        used        free      shared  buff/cache   available
Mem:           911Mi       212Mi       249Mi       6.0Mi       450Mi       547Mi
Swap:          2.0Gi       0.0Ki       2.0Gi
[msleczek@vm0-net projekt_A]$
```

When we log in via SSH, the commands entered in CLI are first interpreted by the shell. This is the reason why we need to precede some special characters using backslash "\" or use appropriate comments when we want to pass them as a command argument. These characters include "<", ">", "|", ";" **and** "**&**". The "**command**" module does not use the shell. Hence, any special shell characters **will be passed directly to the command as an argument**. Therefore, it is not

possible to use operators and pipes provided by the shell. This is seen in the example below, where both "|" and "**grep**" and "**sda1**" were passed as arguments to the "df" command. When we give additional arguments for the "**df**" command, it shows information about the file systems on which the files provided as command arguments are located.

```
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.123 -m command -a "df -h | grep sda1"
10.8.232.123 | FAILED | rc=1 >>
df: '|': No such file or directory
df: grep: No such file or directory
df: sda1: No such file or directorynon-zero return code
[msleczek@vm0-net projekt_A]$ df -h /boot /root
Filesystem                      Size  Used  Avail  Use% Mounted on
/dev/sda1                      1014M  185M   830M   19% /boot
/dev/mapper/rhel_vm0--net-root   29G  2.0G    28G    7% /

[msleczek@vm0-net projekt_A]$
```

In order to better direct our attention, Ansible uses different colors in the returned results. These include:

- "**task failed**",
- "**made changes**",
- "**nothing had to be done**".

The "**command**" module also has no access to any user shell environment variables, like "**$HOSTNAME**", so its operation cannot use them. This has the advantage that there is no negative impact of these variables and environment settings on the used commands. This is also the reason why it is the **safest and recommended** way to issue ad hoc commands.

In many places you can find that the environment variable "**$HOME**" will not be available for the "**command**" module which is not correct. Actually, some environment variables, such as "**$HOME**", "**$SHELL**" and "**$PATH**", are set by the "**login**" program in the GNU/Linux system and available to the "**sshd**" process while establishing an SSH session. They can be accessed even before we execute any command to activate the shell. Thus the "**command**" module has access to them. However, other variables that are activated after the shell has been started, as for example "**$HOSTNAME**" are not available for the "**command**" module.

Where we need pipes, operators or shell environment variables, we should use the "**shell**" module. This module uses the "**/bin/sh**" shell by default, but this can be changed. At the same time, it is worth remembering that the result of this module may depend on the values of shell environment variables. Thus, a person modifying shell environment files may intentionally or unintentionally contribute to something that should not happen.

That is why we recommend using the "**command**" module to ad-hoc commands wherever this is possible.

Keep in mind that the shell can be activated in a variety of ways, and this affects how environment variables are set. If during the login, the interactive shell is activated or we do it manually by issuing the command "**bash -l**" or "**bash –login**", then the file "**/etc/profile**" is executed first, followed by the first available file from the list with the order: "**~/.bash_profile**", "**~/.bash_login**" or "**~/.profile**". For an interactive shell that is not a login shell, only the "**~/.bashrc**" file is executed. However, there is also a non-interactive shell ("**bash -c**") to which Ansible get access through the "**shell**" module. At startup, it executes the file under the "**$BASH_ENV**" variable. This variable is empty by default, so many people are surprised when they tell the "shell" module to use the bash shell, and it doesn't see the variables defined in their environment files.

The "**command**" and "**shell**" modules require Python on the managed node. This is not always possible, although most server systems already have it right after installing the base system. The same applies to new good quality network devices. However, we will definitely come across some older devices or systems where Python will not be available, and **it would be good to manage them with the same tool**. This is possible thanks to the "raw" module, which bypasses the entire subsystem of **Ansible** modules. He issues the commands directly after establishing the SSH connection, and then sends the result back to us. This module does not make any attempt to interpret the result or error checking. We'll get whatever is thrown at the output of STDERR and STDOUT.

Below, using the "**shell**" and "**raw**" modules for our system gave a similar result. In both cases provided command was executed in the shell "**/bin/sh**". More differences should be observed in other examples.

```
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.123 -m shell -a "df -h | grep sda1"
10.8.232.123 | CHANGED | rc=0 >>
/dev/sda1 1014M 185M 830M 19% /boot
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.123 -m raw -a "df -h | grep sda1"
10.8.232.123 | CHANGED | rc=0 >>
/dev/sda1 1014M 185M 830M 19% /boot
Shared connection to 10.8.232.123 closed.

[msleczek@vm0-net projekt_A]$
```

Some **Ansible** modules require additional arguments and some don't. If you are not sure how to use the module, it's best to use the "**ansible-doc**" command. An example of the information it provides for the "**ping**" module can be seen below.

```
[msleczek@vm0-net projekt_A]$ ansible-doc ping
> PING (/usr/lib/python3.6/site-packages/ansible/modules/system/ping.py)

A trivial test module, this module always returns `pong' on successful contact.
It does not make sense in playbooks, but it is useful from `/usr/bin/ansible'
to verify the ability to login and that a usable Python is configured. This
is NOT ICMP ping, this is just a trivial test module that requires Python
on the remote-node. For Windows targets, use the [win_ping] module instead.
For Network targets, use the [net_ping] module instead.

* This module is maintained by The Ansible Core Team
OPTIONS (= is mandatory):

- data
 Data to return for the `ping' return value.
 If this parameter is set to `crash', the module will cause an exception.
 [Default: pong]
 type: str


SEE ALSO:
 * Module net_ping
 The official documentation on the net_ping module.
 https://docs.ansible.com/ansible/2.9/modules/net_ping_module.html
 * Module win_ping
 The official documentation on the win_ping module.
 https://docs.ansible.com/ansible/2.9/modules/win_ping_module.html


AUTHOR: Ansible Core Team, Michael DeHaan
 METADATA:
 status:
 - stableinterface
 supported_by: core

EXAMPLES:

# Test we can logon to 'webservers' and execute python with json lib.
# ansible webservers -m ping

# Example from an Ansible Playbook
- ping:

# Induce an exception to see what happens
- ping:
  data: crash


RETURN VALUES:

ping:
 description: value provided with the data parameter
 returned: success
 type: str
 sample: pong
```

According to the “**ansible-doc**” for the “**ping**” module, the default “**pong**” value can be changed with the “data” argument.

```
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.124 --module-name ping
10.8.232.124 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.124 --module-name ping --args data=JESTEM
10.8.232.124 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "JESTEM"
}
[msleczek@vm0-net projekt_A]$
```

Above we use extended arguments, where “**-a**” is equivalent to “**–args**” and “**-m**” to “**–module-name**“.

You can check the list of modules and plugins supported by your system by using:

```
$ ansible-doc -l
```

The list is quite long, but it can be searched, as shown below. Manually added modules are also visible in the list. Below you can see the “**cisco_webex**” module that we made, which allows you to send messages into the room at **Cisco Webex Teams**.

```
[msleczek@vm0-net projekt_A]$ ansible-doc -l | grep webex
cisco_webex                    Send a webexmsg to a Cisco Webex Teams Room or Individual
[msleczek@vm0-net projekt_A]$
```

It is also possible to generate an example configuration snip for a given module, which can be used as an introduction to further configuration. This is done using the command:

```
[msleczek@vm0-net projekt_A]$ ansible-doc -s cisco_webex
- name: Send a webexmsg to a Cisco Webex Teams Room or Individual.
  cisco_webex:
      personal_token:  # (required) Personal access token required to validate the Webex
API.
      recipient_id:    # (required) The unique identifier associated with `recipient_type'.
      recipient_type:  # (required) Messages can be sent to a room or individual (by ID or
E-Mail).
      webexmsg:        # (required) The webexmsg you would like to send.
      webexmsg_type:   # Specifies how you would like the webexmsg formatted.
```

Ansible tasks can be run once or cyclically on a scheduled basis. Thanks to the ability to send messages to **Cisco Webex Teams**, we can quickly notify the appropriate team about any failures or detected incompatibilities.

We will now take care of some useful options for the "**ansible**" command. First, we will go one level below in the directory tree structure, so that our default settings values from the "**ansible.cfg**" file no longer work.

```
[msleczek@vm0-net projekt_A]$ ansible --version | grep "config file"
 config file = /home/msleczek/projekt_A/ansible.cfg
[msleczek@vm0-net projekt_A]$ cd ..
[msleczek@vm0-net ~]$ ansible --version | grep "config file"
 config file = /home/msleczek/.ansible.cfg
[msleczek@vm0-net ~]$ ansible 10.8.232.124 -a whoami -i projekt_A/inventory
10.8.232.124 | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: msleczek@10.8.232.124: Permission
denied (publickey,gssapi-keyex,gssapi-with-mic,password).",
    "unreachable": true
}
[msleczek@vm0-net ~]$
```

Above you can see the error, because by default Ansible tries to use the current user's name when logging into the remote host. Earlier, we used the "**ansible**" user for this purpose. We can specify it manually using the "**-u**" or "**–user**" option.

```
[msleczek@vm0-net ~]$ ansible 10.8.232.124 -a whoami -i projekt_A/inventory -u ansible
10.8.232.124 | CHANGED | rc=0 >>
ansible
[msleczek@vm0-net ~]$ ansible 10.8.232.124 -a whoami -i projekt_A/inventory -u ansible -b
10.8.232.124 | CHANGED | rc=0 >>
root
[msleczek@vm0-net ~]$ ansible 10.8.232.124 -a whoami -i projekt_A/inventory --user ansible
--become
10.8.232.124 | CHANGED | rc=0 >>
root
[msleczek@vm0-net ~]$
```

After providing the correct user we can see that we do not get an error and the "**whoami**" command returns the user "**ansible**". When we add the "**-b**" or "**–become**" option, the "**whoami**" command will be issued only after the privilege escalation. As a result, in the next two examples above, it returns the user "**root**".

In some cases, it may be more readable or useful to put the entire result on one line. This can be done with the "**-o**" or "**–one-line**" option.

```
[msleczek@vm0-net projekt_A]$ ansible all -a "cat /etc/redhat-release"
10.8.232.122 | CHANGED | rc=0 >>
```

```
Red Hat Enterprise Linux release 8.2 (Ootpa)
10.8.232.124 | CHANGED | rc=0 >>
Red Hat Enterprise Linux release 8.2 (Ootpa)
10.8.232.123 | CHANGED | rc=0 >>
Red Hat Enterprise Linux release 8.2 (Ootpa)
10.8.232.121 | CHANGED | rc=0 >>
Red Hat Enterprise Linux release 8.2 (Ootpa)
[msleczek@vm0-net projekt_A]$ ansible all -a "cat /etc/redhat-release" --one-line
10.8.232.121 | CHANGED | rc=0 | (stdout) Red Hat Enterprise Linux release 8.2 (Ootpa)
10.8.232.122 | CHANGED | rc=0 | (stdout) Red Hat Enterprise Linux release 8.2 (Ootpa)
10.8.232.123 | CHANGED | rc=0 | (stdout) Red Hat Enterprise Linux release 8.2 (Ootpa)
10.8.232.124 | CHANGED | rc=0 | (stdout) Red Hat Enterprise Linux release 8.2 (Ootpa)
[msleczek@vm0-net projekt_A]$ ansible all -a "cat /etc/redhat-release" -o
10.8.232.121 | CHANGED | rc=0 | (stdout) Red Hat Enterprise Linux release 8.2 (Ootpa)
10.8.232.124 | CHANGED | rc=0 | (stdout) Red Hat Enterprise Linux release 8.2 (Ootpa)
10.8.232.123 | CHANGED | rc=0 | (stdout) Red Hat Enterprise Linux release 8.2 (Ootpa)
10.8.232.122 | CHANGED | rc=0 | (stdout) Red Hat Enterprise Linux release 8.2 (Ootpa)
[msleczek@vm0-net projekt_A]$
```

While the use of the "**command**", "**shell**" and "**raw**" modules is easy and convenient, we should avoid them wherever possible. Ultimately, we should use dedicated modules specialized for specific tasks, whose purpose is to bring devices, systems, files, applications or services to the desired state, which is described in a **declarative** language. This means that we only specify the state that we want the object to assume, without describing in detail how this should be done. The modules that are **appropriate for the specific tasks** are already dealing with the rest. Depending on the initial state of the object, such a module can perform more or less different operations on it, or even not perform any, if it is not needed. This approach is characterized by idempotence, which here is the ability to repeatedly run the same module without changing the final state of the object.

It also means that we can execute the same specialized module on a given object many times. If it is already in the desired state, then nothing will happen, and if it is not, it will be brought to desired state.

It should be remembered that idempotence is guaranteed only if we use specialized and built-in modules. If we create them ourselves or use other methods, such as sending commands by the "**command**", "**shell**" or "**raw**" modules, then we have to take care of idempotence ourselves.

Ansible supports a very large number of specialized modules, and their number is constantly growing. Full list is available in **the Ansible documentation**. **Red Hat** also provides additional modules and plugins in its **Automation Hub**. Let's look at just a few here.

Sometimes we need to update the entire file containing the list of NTP servers, the keys used for authentication or environment variables. Ansible is perfect for

this. Before doing anything, it checks if the file exists and its contents. If file exists and its content is the same, it does nothing. If not, it performs the appropriate operations. It is even able to make a copy of the previous version of the file.

Let's assume that we want to update or standardize environment variables on a huge number of servers. Just in case, we also want to make a copy of the previous version of the file if its contents were different.

```
[msleczek@vm0-net projekt_A]$ cat environment
LANG=en_US.utf-8
LC_ALL=en_US.utf-8

[msleczek@vm0-net projekt_A]$ ansible all -m command -a 'cat /etc/environment' -o
10.8.232.122 | CHANGED | rc=0 | (stdout)
10.8.232.124 | CHANGED | rc=0 | (stdout) LANG=en_US.utf-8\nLC_ALL=en_US.utf-8
10.8.232.123 | CHANGED | rc=0 | (stdout) LANG=en_US.utf-8\nLC_ALL=en_US.utf-8
10.8.232.121 | CHANGED | rc=0 | (stdout)
[msleczek@vm0-net projekt_A]$ ansible all -m copy -a "src=./environment dest=/etc/
environment owner=root group=root mode=0644 backup=yes"
10.8.232.124 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "checksum": "0fa89fe380b8dc91e6bb56679f04a8759d5500d6",
    "dest": "/etc/environment",
    "gid": 0,
    "group": "root",
    "mode": "0644",
    "owner": "root",
    "path": "/etc/environment",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 37,
    "state": "file",
    "uid": 0
}
10.8.232.123 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "checksum": "0fa89fe380b8dc91e6bb56679f04a8759d5500d6",
    "dest": "/etc/environment",
    "gid": 0,
    "group": "root",
    "mode": "0644",
    "owner": "root",
    "path": "/etc/environment",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 37,
    "state": "file",
    "uid": 0
}
10.8.232.121 | CHANGED => {
```

```
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "backup_file": "/etc/environment.36908.2020-05-01@20:48:44~",
    "changed": true,
    "checksum": "0fa89fe380b8dc91e6bb56679f04a8759d5500d6",
    "dest": "/etc/environment",
    "gid": 0,
    "group": "root",
    "md5sum": "f14f07073c6735cc233511404a860097",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 37,
    "src": "/home/ansible/.ansible/tmp/ansible-
tmp-1588358960.2073026-15260-256434093911547/source",
    "state": "file",
    "uid": 0
}
10.8.232.122 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "backup_file": "/etc/environment.13295.2020-05-01@20:49:22~",
    "changed": true,
    "checksum": "0fa89fe380b8dc91e6bb56679f04a8759d5500d6",
    "dest": "/etc/environment",
    "gid": 0,
    "group": "root",
    "md5sum": "f14f07073c6735cc233511404a860097",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 37,
    "src": "/home/ansible/.ansible/tmp/ansible-
tmp-1588358960.2120671-15262-147935961544848/source",
    "state": "file",
    "uid": 0
}
[msleczek@vm0-net projekt_A]$ ansible all -m command -a 'cat /etc/environment' -o
10.8.232.122 | CHANGED | rc=0 | (stdout) LANG=en_US.utf-8\nLC_ALL=en_US.utf-8
10.8.232.121 | CHANGED | rc=0 | (stdout) LANG=en_US.utf-8\nLC_ALL=en_US.utf-8
10.8.232.123 | CHANGED | rc=0 | (stdout) LANG=en_US.utf-8\nLC_ALL=en_US.utf-8
10.8.232.124 | CHANGED | rc=0 | (stdout) LANG=en_US.utf-8\nLC_ALL=en_US.utf-8
[msleczek@vm0-net projekt_A]$
```

You can see above, that on two servers **Ansible** didn't have to do anything. On the other two, the content of the desired file has been unified.

Instead of pointing to the source file with the "**src**" option, you can directly define the content that the file should have. The "**content**" suboption is used for this purpose.

```
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.121 -m copy -a 'content="\nSerwer
obsługiwany przez networkers.pl\n\n" dest=/etc/motd'
10.8.232.121 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": true,
    "checksum": "0bcc7021129a6c1564e9d26983b7ad52c710ec28",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "2fda36d59048989bd169842be2c843e1",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 42,
    "src": "/home/ansible/.ansible/tmp/ansible-
tmp-1588359294.0033875-15420-226606317748867/source",
    "state": "file",
    "uid": 0
}
[msleczek@vm0-net projekt_A]$
```

Below you can see what the user sees after logging in to our servers:

```
ms-net:~ msleczek$ ssh -l msleczek 10.8.232.121
msleczek@10.8.232.121's password:

Serwer obsługiwany przez networkers.pl

Last login: Fri May 1 15:37:10 2020 from 10.8.64.128
[msleczek@vm1-net ~]$
```

Now let's assume that we want to create a directory with specific permissions on all managed servers, where we will put backup copies of selected files in the future:

```
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.121 -m file -a "path=/root/backup
state=directory mode=0700"
10.8.232.121 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": true,
    "gid": 0,
    "group": "root",
    "mode": "0700",
    "owner": "root",
    "path": "/root/backup",
    "secontext": "unconfined_u:object_r:admin_home_t:s0",
    "size": 6,
    "state": "directory",
    "uid": 0
}
[msleczek@vm0-net projekt_A]$
```

A lot depends on our ingenuity and knowledge of what we automate. For example, the same module can be used to disable the history of commands issued in the MySQL/MariaDB database:

```
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.121 -m file -a "src=/dev/null dest=/root/.
mysql_history state=link"
10.8.232.121 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": true,
    "dest": "/root/.mysql_history",
    "gid": 0,
    "group": "root",
    "mode": "0777",
    "owner": "root",
    "secontext": "unconfined_u:object_r:admin_home_t:s0",
    "size": 9,
    "src": "/dev/null",
    "state": "link",
    "uid": 0
}
[msleczek@vm0-net projekt_A]$
```

As a result of the above two tasks, the "**backup**" directory and the "**.mysql_history**" symbolic link to "/**dev/null**" appeared on the server:

```
[root@vm1-net ~]# ls -ald .mysql_history backup
drwx------. 2 root root 6 May 1 20:58  backup
lrwxrwxrwx. 1 root root 9 May 1 20:59  .mysql_history -> /dev/null
[root@vm1-net ~]#
```

Finally, we encourage you to familiarize yourself with the "**setup**" module, which collects and provides a very large amount of information about each of the managed nodes. There is so much information, that we do not put it in the article. For this we refer you to check it yourself. To do this, you can use the command:

```
$ ansible all -m setup
```

The information provided by the "**setup**" module can be conveniently filtered and narrowed using the appropriate suboptions. The best way is to read the "**ansible-doc**" for this module. Below is an example of a filter that allows you to check the operating system family and processor type.

```
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.121 -m setup -a filter="ansible_os_family"
10.8.232.121 | SUCCESS => {
    "ansible_facts": {
        "ansible_os_family": "RedHat",
```

```
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false
}
[msleczek@vm0-net projekt_A]$ ansible 10.8.232.121 -m setup -a filter="ansible_processor"
10.8.232.121 | SUCCESS => {
    "ansible_facts": {
        "ansible_processor": [
            "0",
            "GenuineIntel",
            "Intel Xeon E312xx (Sandy Bridge)"
        ],
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false
}
[msleczek@vm0-net projekt_A]$
```

Ad-hoc commands are just as easily run on one or hundreds of nodes.

05

ANSIBLE, PART III
– YAML and Playbooks

In the **previous article**, we showed how useful are ad-hoc commands. They allow us to run individual tasks on multiple nodes. Ad hoc commands are used on an ad hoc basis, hence their name. Their limitation is that they can only perform one module with one set of arguments at a time. It happens, that we need to perform a whole set of different tasks both on a large number of nodes operating at the given moment, and new ones that will appear in the future.

These tasks are much easier to write, modify and maintain in the form of a text file, which is the so-called playbook. **Playbook is a set of tasks to be performed on specific sets of nodes**.

By using **YAML** (**YAML Ain't Markup Language**) to create playbooks, the whole is much more **readable and understandable** than single-line ad hoc commands, especially those with a lot of arguments. It should be remembered that the larger the scale, the more important is simplicity, transparency and readability. Their absence leads to increased failure rates and security problems.

Ansible Playbook is a text file with the extension "**.yml**" or "**.yaml**". The latter is the extension recommended by the standard. **Indentations are a very important element of its file structure**. While their exact number does not matter, data at the same hierarchy level must have the same number of indentations (spaces), and children must have more indentation (spaces) than the parent object.

I am writing about a space character intentionally, as the tab character cannot be used for indentations. The reason for this is the different interpretations of the tab character by different text editors. The first line of playbook must contain three lines "—", and the last line should contain three dots "...". This last line is often omitted.

The place for "**plays**", which are **sets of tasks to be played on specific groups of hosts**, is located between "—" and "...". Before you start writing playbooks, it is worth spending some time getting to know the syntax and structure of YAML, as it will be the syntax and structure of our playbooks.

YAML is used to describe and represent data in an ordered and structured manner. Compared to XML, the YAML record is more concise and transparent, as well as more convenient for humans to work.

You can use comments within the YAML document. The hash "#" character at the beginning of a new line or the space and hash " #" characters after the data are used for this purpose. Everything to the right of this sign is treated as a comment. Comments can also be placed before the "—" sign.

```
# This is the comment
product: 'Red Hat Ansible Automation' # This is also the comment
```

Variables or keys are separated from the **value** by a colon and space ": ":

```
name: 'Marcin Ślęczek'
work: 'networkers.pl'
```

A more compact variable mapping format is also available with curly brackets "{}":

```
{name: 'Marcin Ślęczek', work: 'networkers.pl'}
```

The string value of the key does not have to be placed inside quotation marks, even if it contains spaces. However, you must watch out for special characters that are specific to YAML syntax, such as the colon ":". Therefore, we recommend using quotation marks for text variables. Double quotation marks "" allow the use of some special characters by preceding them with the backslash "****", such as the newline character **"\n**". A single quotation mark " prevents the use of special characters.

When we need to write long text, consisting of many lines, it is best to use the "|" or "**>**" operator for this purpose. The first of these "|" preserves the newline and all trailing spaces. The leading blank characters of each line from the left are removed. An example of its use can be seen below:

```
data: >
  One long line of text
  written in such a way
  for better readability.

about: >
  networkers.pl provides complete solutions for IT systems, data centers
  and DevOps environments. It has been on the market since 2009. It sells
  individual products as well as complete solutions.
```

The values of logical variables (**booleans**) can be defined in various ways, such as: **True/False, true/false, yes/no** or **1/0**. It is worth choosing one of the methods and consistently use it.

Each element of the same **list** has the same number of indents, followed by a dash and space "–":

```
- firewall
- router
- switch
- hub
```

A more compact list format is also available with square brackets "**[]**":

```
[firewall, router, switch, hub]
```

At this stage, we don't need more information about YAML syntax. We have gone through the most necessary basics. We did not describe all possible ways to write the same things. We also did not refer to everything that we will need further. This is not our goal at the moment.

While there are many different ways to write the same thing, **it's a good idea to adopt a standard within the project or even the entire organization and then stick to it**. Then for everyone the whole will be easier to interpret and use, which will translate into more efficient management and fewer human mistakes.

Before we start creating playbooks, let's look at two ad hoc commands again:

```
$ ansible all -m command -a "df -h"
$ ansible all -m copy -a "src=./environment dest=/etc/environment owner=root group=root
mode=0644"
```

You can see that they are perfect for situations when we need to check something quickly or do it on an ad hoc basis. Using them to describe individual elements of infrastructure is rather uncomfortable, and in the long run difficult to maintain and prone to errors. Nevertheless, they have their application and are needed in some situations, so it is worth being able to use them efficiently.

Now we will deal with a slightly different use of Ansible, which is describing individual infrastructure elements or automating more tasks on different groups of nodes. Playbooks are much more suitable for such purposes.

From an Ansible point of view, a single "**play**" is a set of tasks to play (perform) on a given group of nodes, and the file in which one or more such "**play**" is located

is called a "**playbook**". Each "play" can perform different tasks on a different group of nodes, but in total they form a whole, which facilitates **orchestration** of larger and more complex projects.

Each "**play**" can specify different settings regarding how to connect to managed nodes, including connection and escalation method, and username. If they are not specified, then the values from the Ansible configuration file, discussed **in earlier articles** or default values are used. The names of these parameters inside the playbook file are exactly the same. These parameters are defined at the same level (the same number of indentations) as the parameters: "**hosts**" and "**tasks**".

As an argument to the "**hosts**" parameter, you must specify "pattern" to identify the nodes from the inventory on which the tasks will be performed.

It was discussed in more detail in the **first article on Ansible**.

The "**tasks**" parameter contains a list of tasks to be performed. Individual tasks are performed sequentially in order of arrangement within the playbook. However, they can be performed in parallel on many nodes at the same time. The number of tasks to be performed in parallel is configurable with the "**-f**" or "**–fork**" option – it defaults to 5. For better readability, you can leave a blank line between tasks in the list.

The "**name**" parameter is a label that describes the task or the entire "**play**" in a way we understand. As this label is for us and is visible during the operation of the playbook, it is worth ensuring that it contains clear and concise content. It should be relatively short, and at the same time reflect the purpose of the task or play. It is optional, however it is highly recommended to use it.

Below you can see a playbook with one "play", which corresponds to the second of the above-mentioned ad hoc command:

```
---
- name: 'ALL servers file unification'
  hosts: all
  tasks:
  - name: 'Copy ./environment file to /etc/environment'
    copy:
      src: ./environment
      dest: /etc/environment
      owner: root
      group: root
      mode: 0644
...
```

To run the playbook, use the command:

```
$ ansible-playbook <playbook_file> [-i INVENTORY]
```

Before starting the playbook, it is good practice to check the syntax. This can be done with the "**–syntax-check**" option. If such verification fails, we will receive information about the error and its location.

```
$ ansible-playbook <playbook_file> [-i INVENTORY] --syntax-check
```

If the syntax is correct, you can additionally run the playbook in "dry run" mode. Then, we will get all the information about the result of the playbook as if it had been made. This way you can see what has changed and what has not changed. Of course, there are no modifications on the managed nodes in "dry run" mode. For this purpose, a "-C" or "–check" option is used.

```
$ ansible-playbook <playbook_file> [-i INVENTORY] --check
```

In addition, you can increase the amount of information generated at the output of the "**ansible**" and "**ansible-playbook**" commands. The "**-v**", "**-vv**", "-**vvv**" and "**-vvvv**" options are used for this. Each of the following increases the debug level by 1, and thus the amount of information displayed. The default debug level is 0.

Below is a playbook containing two sets of tasks to play called "**Play 1: Verification**" and "**Play 2: Unification**". Each of them can define many tasks to be played on different host groups. In our example, both sets of tasks will be performed **on all nodes**.

```
[msleczek@vm0-net projekt_A]$ cat playbook.yml
---
- name: 'Play 1: Verification'
  hosts: all
  tasks:
  - name: 'Verify connectivity'
    ping:

- name: 'Play 2: Unification'
  hosts: all
  gather_facts: false
  tasks:
  - name: 'Copy ./environment file to /etc/environment'
    copy:
      src: ./environment
      dest: /etc/environment
      owner: root
      group: root
      mode: 0644
```

```
      backup: yes
  - name: 'Create backup destination'
    file:
      path: /root/backup
      state: directory
      mode: 0700
```

Let's follow the syntax verification result and launch of the above playbook. We created three tasks in it, but **at the bottom you can see four**. The initial "**Gathering Facts**" task is implicitly created by running the "**setup**" module, which collects a large amount of information about each managed node. It allows you to perform certain tasks based on collected values, that reflect the characteristics and configuration of the node. For example, if the node's operating system belongs to the "RedHat" family, then we should use the "**yum**" module for package management, and if it is "Debian", then the "**apt**" module. In this way, we can build more universal and dynamic playbooks.

Gathered in this way information are called facts. If they are not needed for anything, it is worth to turn them off. In this way, our playbook will perform much faster. This can be done by setting the "**gather_facts**" parameter to "**false**". This was done in "**Play 2: Unification**". Please note that unless explicitly turned off, this information will be collected by default at the beginning of each set of tasks to play.

```
[msleczek@vm0-net projekt_A]$ ansible-playbook --syntax-check playbook.yml

playbook: playbook.yml
[msleczek@vm0-net projekt_A]$ ansible-playbook playbook.yml

PLAY [Play 1: Verification] ****************************************************
**

TASK [Gathering Facts] *********************************************************
***
ok: [10.8.232.124]
ok: [10.8.232.123]
ok: [10.8.232.122]
ok: [10.8.232.121]

TASK [Verify connectivity] *****************************************************
***
ok: [10.8.232.124]
ok: [10.8.232.121]
ok: [10.8.232.122]
ok: [10.8.232.123]

PLAY [Play 2: Unification] *****************************************************
**

TASK [Copy ./environment file to /etc/environment] *****************************
**
```

```
ok: [10.8.232.124]
ok: [10.8.232.122]
changed: [10.8.232.121]
changed: [10.8.232.123]

TASK [Create backup destination] ***********************************************************
***
changed: [10.8.232.122]
ok: [10.8.232.123]
ok: [10.8.232.124]
changed: [10.8.232.121]

PLAY RECAP *********************************************************************************
***
10.8.232.121      : ok=4   changed=2   unreachable=0   failed=0   skipped=0   rescued=0
ignored=0
10.8.232.122      : ok=4   changed=1   unreachable=0   failed=0   skipped=0   rescued=0
ignored=0
10.8.232.123      : ok=4   changed=1   unreachable=0   failed=0   skipped=0   rescued=0
ignored=0
10.8.232.124      : ok=4   changed=0   unreachable=0   failed=0   skipped=0   rescued=0
ignored=0


[msleczek@vm0-net projekt_A]$
```

In the **previous article**, we have already mentioned idempotence. Every playbook task should work in such way and if we only use specialized modules that are supplied with **Ansible**, it is so. However, when we use some of our own modules or use more universal modules, such as "**command**", "**shell**" or "**raw**", we should take care of it by ourselves. The expressions available as part of the playbook allow this, but it requires additional work. In the end, if only we followed the rules, then playbook can be run many times on the same set of nodes, without any negative effects.

So is with our playbook, which on some nodes performed all tasks, on another part of the nodes several tasks, and on the rest of the nodes did not have to do anything at all, because they were already in the right condition. Despite this, it was launched without fear for all nodes.

In order to better direct our attention, **Ansible** uses different colors in the returned results. These include:

- "**task failed**",
- "**made changes**",
- "**nothing had to be done**".

When it is possible to use several colors, the one that refers to the events to which we should focus our attention will be used. At the bottom is a summary of all task sets – "**PLAY RECAP**".

WHAT IS THE OVP PORTAL

Open Virtualization Pro Portal is a professional community focused on sharing knowledge about open-source technologies such as virtualization, containerization and automation.

Together with industry experts the OVP portal is organizing meetups and webinars (you can watch our post-webinar recordings **here**).

Experts from open-source related companies are also writing articles on various topics, touching the latest news and technologies from the open world.

Our experts Jacek Skorzynski and Zbigniew Parys from Red Hat at OpenVirtualization.Pro Meetup Online #1.

Portal Partners: storware Red Hat