DevOps Culture and Practices

Agenda

- Introduction to DevOps

- Implementing CI/CD and Continuous Deployment

- Continuous Integration (CI)

- Implementing Continuous Integration (CI)

- Continuous Delivery (CD)

- Continuous Deployment

- Understanding IaC Practices

- The IaC Topology

- IaC best practices

Introduction to DevOps

- DevOps is a culture different from traditional corporate cultures and requires a **change in mindset, processes, and tools**.

- It is often associated with **continuous integration (CI)** and **continuous delivery (CD)** practices, which are software engineering practices.

- It is also associated with **Infrastructure as Code (IaC)** which consists of codifying the structure and configuration of infrastructure.

Introduction to DevOps

- DevOps is a set of practices that combines software development and IT operations.

- It aims to **shorten the systems development life cycle** and provide continuous delivery with **high software quality.**

- The DevOps is the combination of two words

  - Development

  - Operations

- Practice to promote the **Development** and **Operation** process **Collectively.**

Getting started with DevOps

- The term DevOps was introduced in 2007-2009 by Patrick Debois, Gene Kim, and John Willis.

- DevOps represents the combination of **Development** (Dev) and **Operations** (Ops).

- It advocates **bringing developers and operations together within teams**.

- This is to be able to **deliver added business value** to users more quickly and hence be **more competitive in the market**.

Getting started with DevOps
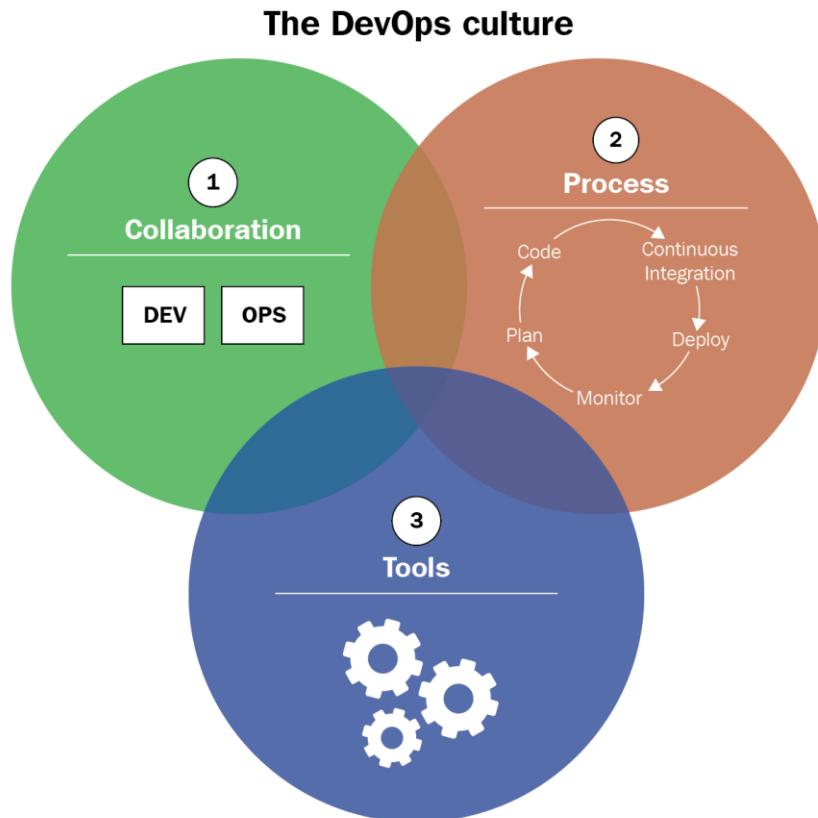
- DevOps culture is a set of practices that reduce the barriers between

- **Developers**: who want to **innovate and deliver faster.**

- **Operations**: who want to guarantee the **stability of production systems** and the **quality** of the system changes they make.

- DevOps culture is the **extension of agile processes** which make it possible to reduce delivery times and already involve developers and business teams, but are often hindered because of the **non-inclusion of Ops** in the same teams.

Getting started with DevOps

- To facilitate collaboration between Dev and Ops:

    - More **frequent** application deployments with integration and continuous delivery (called **CI/CD**).

    - The implementation and automation of unitary and integration tests, with a process focused on Behavior-Driven Design (BDD) or Test-Driven Design (TDD).

    - The implementation of a means of **collecting feedback from users**.

    - **Monitoring** applications and infrastructure.

Three Axis of DevOps Culture

1. Collaboration

2. Process

3. Tools

The DevOps culture

Three Axis of DevOps Culture

- The DevOps movement is based on **three axes**:

  - **The culture of collaboration**: This is the very essence of DevOps—the fact that teams are no longer separated by specialization (developers, Ops, testers, and so on), but, on the contrary, these people are brought together by making multidisciplinary teams that have the **same objective: to deliver added value to the product as quickly as possible**

Three Axis of DevOps Culture

- The DevOps movement is based on three axes:

  - **Processes:** For rapid deployment, **teams must follow development processes from agile methodologies** with **iterative phases** that allow for better functionality quality and rapid feedback.

  - These processes should not only be integrated into the development workflow with **continuous integration** but also into the deployment workflow with **continuous delivery and deployment**.

  - The DevOps process is divided into several phases: The planning and prioritization of functionalities, Development, Continuous integration and delivery, Continuous deployment, Continuous monitoring.

Three Axis of DevOps Culture

- The DevOps movement is based on three axes:

  - **Tools:** The choice of tools and products used by teams is very important in DevOps.

  - When teams were separated into Dev and Ops, each team used their specific tools—deployment tools for developers and infrastructure tools for Ops—which further widened communication gaps.

Three Axis of DevOps Culture

- **Developers** need to **integrate with monitoring tools** used by Ops teams to detect performance problems as early as possible and with **security tools** provided by Ops to protect access to various resources.

- **Ops**, on the other hand, must **automate the creation and updating of the infrastructure and integrate the code into a code manager**; this is called **Infrastructure as Code**, but this can only be done in collaboration with developers who know the infrastructure needed for applications.

- Ops must also be integrated into application release processes and tools.

Getting started with DevOps

- The benefits of establishing a DevOps culture:

  - Better **collaboration** and **communication** in teams.

  - **Shorter lead times to production**, resulting in better performance and end user satisfaction.

  - **Reduced infrastructure costs** with IaC.

  - **Significant time saved with iterative cycles that reduce application errors** and **automation tools that reduce manual tasks**, so teams focus more on developing new functionalities with added business value.

Implementing CI/CD and Continuous Deployment

- One of the key DevOps practices is the process of integration and continuous delivery, also called CI/CD.

- In fact, behind the acronyms of CI/CD, there are three practices:

  - Continuous Integration (CI)

  - Continuous Delivery (CD)

  - Continuous Deployment

Continuous Integration (CI)

- "*Continuous Integration is a software development practice where* **members of a team integrate their work frequently.** *Each integration is* **verified by an automated build** *(including test)* **to detect integration errors** *as quickly as possible.*"

- CI is an automatic process that allows you to **check the completeness of an application's code** every time a team member makes a change.

- This verification must be done as quickly as possible.

Continuous Integration (CI)

- DevOps culture is seen in CI very clearly, with the spirit of collaboration and communication, because the **execution of CI impacts all members** in terms of work methodology and therefore collaboration.

- Moreover, CI requires the implementation of processes (**branch, commit, pull request, code review**, and so on) with automation that is done with tools adapted to the whole team (**Git, Jenkins, Azure DevOps**, and so on).

- CI must run quickly to **collect feedback on code integration** as soon as possible and hence be able to deliver new features more quickly to users.

Implementing Continuous Integration (CI)

- To set up CI, it is necessary to have a Source Code Manager (SCM) that will allow the centralization of the code.

- This code manager can be of any type: **Git, SVN (Subversion Repository), or Team Foundation Source Control (TFVC).**

- Git is a distributed version control system, whereas SVN is a centralized version control system.

- Next is **Automatic build manager** (CI server) which **supports continuous integration such as Jenkins, GitLab CI, TeamCity, Azure Pipelines**, **GitHub Actions, Travis CI, Circle CI**, and so on.

Implementing Continuous Integration (CI)

- Each team member will work on the application code daily, iteratively and incrementally.

- Each task or feature must be partitioned from other developments with the use of branches.

- Team members archive or commit their code and with small commits (trunks) that can easily be fixed in the event of an error.

- This will be integrated into the rest of the code of the application with all of the other commits of the other members.

Implementing Continuous Integration (CI)

- This integration of all the commits is the starting point of the CI process.

- This process, executed by the CI server, must be automated and triggered at each commit.

- The server will retrieve the code and then do the following:

  - **Build the application package**—compilation, file transformation, and so on.

 - **Perform unit tests** (with code coverage).
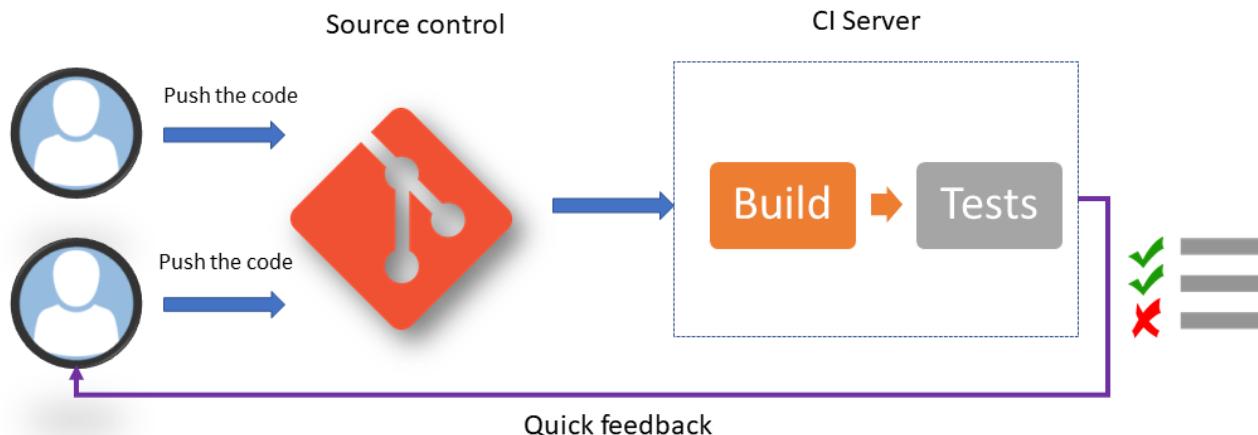
Implementing Continuous Integration (CI)

- This CI process must be optimized as soon as possible so that it can run fast and developers can have quick feedback on the integration of their code.

- Bad practices can result in the failure of tests in the CI.

- Such practice can have serious consequences when the errors detected by the tests are revealed in production.

- The time saved during CI will be lost on fixing errors with hotfixes and redeploying them quickly with stress.

- Instead of developing new features, we spend time correcting errors.

Implementing Continuous Integration (CI)

● With an optimized and complete CI process, the developer can quickly fix their problem and improve their code or discuss it with the rest of the team and commit their code for a new integration.

The continuous integration (CI) pipeline

Continuous Delivery (CD)

- After CI, next step is to **deploy the application automatically in one or more non-production environments**, which is called **staging**.

- This process is called **Continuous Delivery (CD)**.

- CD often starts with an application package prepared by CI, which will be installed according to a list of automated tasks.

- These tasks can be of any type: unzip, stop and restart service, copy files, replace configuration, and so on.

- The execution of **functional and acceptance tests** can also be performed during the CD process.

Continuous Delivery (CD)

- CD aims to **test the entire application with all of its dependencies**.

- This is very visible in microservices applications composed of several services and APIs.

- CI will only test the microservice under development while, once deployed in a staging environment, it will be possible to test and validate the entire application as well as the APIs and microservices that it is composed of.

- In practice, it is common to link CI with CD in an integration environment; that is, **CI deploys at the same time in an environment.**

Continuous Delivery (CD)

- Developers need to have the following at each commit

  - Execution of unit tests

  - **Verification of the application as a whole** (UI and functional), with the integration of the developments of the other team members.

- Important that the **package generated during CI**, **deployed during CD** is the same one that will be **installed on all environments until production.**

- There may be **configuration file transformations** that differ depending on the environment, but the application code (binaries, DLL, and JAR) must remain unchanged.

Continuous Delivery (CD)

- If changes (improvements or bug fixes) are to be made to the code following verification in one of the environments, once done, the modification will have to go through the CI and CD cycle again.

- Tools set up for CI/CD:

    - **A package manager:** This constitutes the storage space of the packages generated by CI and recovered by CD. Must support feeds, versioning, and different types of packages. Tools in the market, such as Nexus, ProGet, Artifactory, and Azure Artifacts.

    - **A configuration manager:** This allows you to manage configuration changes during CD. Most CD tools include a configuration mechanism with a system of variables.

Continuous Delivery (CD)
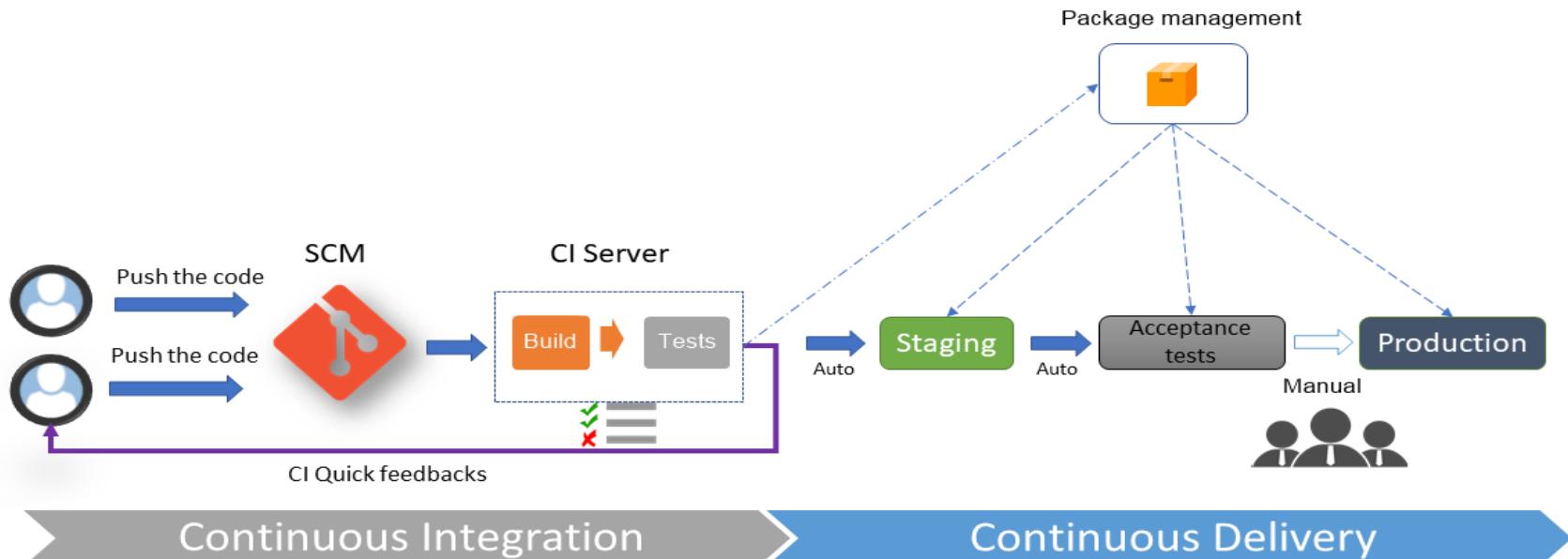
- In CD, the deployment of the application in each staging environment is triggered as follows:

- **Triggered Automatically**, following a successful execution on a previous environment.

- For example: Deployment in the pre-production environment is automatically triggered when the integration tests have been successfully performed in a dedicated environment.

- **Triggered Manually**, for sensitive environments such as the production environment, following a manual approval by a person responsible for validating the proper functioning of the application in an environment.

Continuous Delivery (CD)

- What is important in a CD process is that the deployment to the production environment, that is, to the end user, is triggered manually by approved users.

- CD process is a continuation of the CI process.

- The chain of CD steps are automatic for staging environments but manual for production deployments.

- Package is generated by CI and is stored in a package manager.

- The same package is deployed in different environments.

Continuous Delivery (CD)

The continuous delivery (CD) pipeline
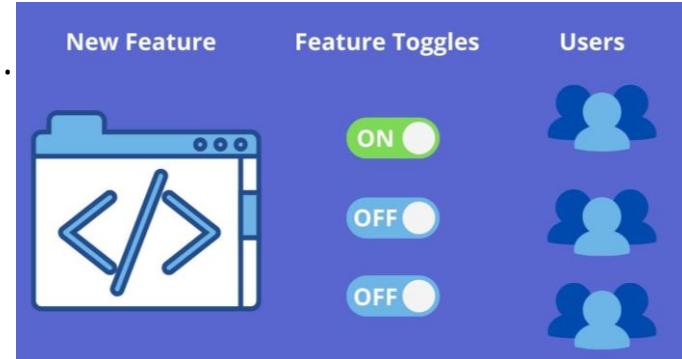
Continuous Deployment

- Continuous deployment is an **extension of CD**, but with a process that **automates the entire CI/CD pipeline** from the moment the developer commits their code to deployment in production through all of the verification steps.

- This practice is **rarely implemented** because it requires a wide coverage of tests (unit, functional, integration, performance, and so on).

- Successful execution of these tests is sufficient to validate the proper functioning of the application with all of these dependencies.
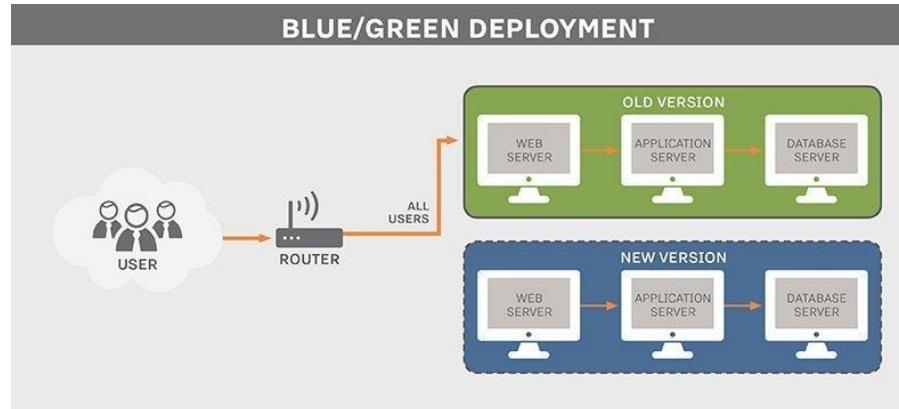
Continuous Deployment

- The continuous deployment process must also take into account all of the steps to restore the application in the event of a production problem.

- Continuous deployment can be implemented with the use and implementation of **Feature Toggle** techniques (or feature flags)

  - Encapsulating the application's functionalities in features.

  - Activating its features on demand directly in production.

  - Code of the application wont be redeployed.
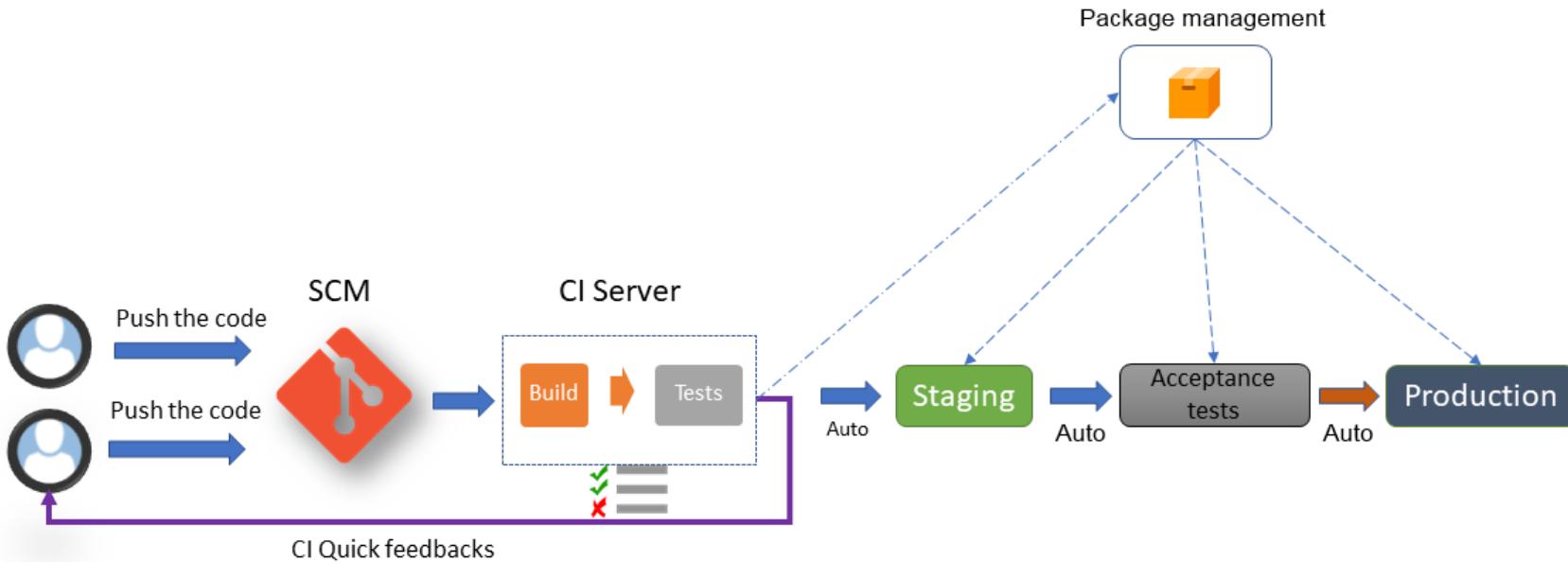
Continuous Deployment

- Another technique is to use a **Blue-Green production infrastructure.**

  - Consists of two production environments, one blue and one green.

  - First deploy to the blue environment, then to the green.

  - To ensure that there is no downtime required.

Continuous Deployment

The continuous deployment pipeline

Continuous Deployment

- CI/CD processes are an essential part of DevOps culture.

- CI allows teams to integrate and test the coherence of its code and to obtain quick feedback very regularly.

- CD automatically deploys on one or more staging environments and tests the entire application until it is deployed in production.

- Continuous Deployment automates the deployment of the application from commit to the production environment.

Understanding IaC Practices

- IaC is the process of **writing the code for provisioning and configuration** of infrastructure components to **automate its deployment** in a repeatable and consistent manner.

- Practice began due to the **rise of DevOps culture** and with the **modernization of cloud infrastructure**.

- Ops teams that **deploy infrastructures manually take time** to deliver infrastructure changes due to inconsistent handling and the risk of errors.

- With the modernization of the cloud and its scalability, the way an infrastructure is built requires a review of provisioning and change practices by adapting a more automated method.

Understanding IaC Practices

- The benefits of IaC

  ▪ The standardization of infrastructure configuration **reduces the risk of error**.

  ▪ The **code** that describes the infrastructure is **versioned and controlled** in a source code manager.

  ▪ The **code is integrated into CI/CD pipelines**.

  ▪ Deployments that make infrastructure changes are faster and more efficient.

  ▪ There's better management, control, and a reduction in infrastructure costs.

Understanding IaC Practices

- IaC Languages and Tools:

- The languages and tools used to code the infrastructure are **Scripting types** and **Declarative types**.

- **Scripting types:** These are scripts such as Bash, PowerShell, or any other languages that use the different clients (SDKs) provided by the cloud provider.

- Ex: Script the provisioning of an Azure infrastructure with the Azure CLI or Azure PowerShell.

Understanding IaC Practices

- Ex: Command that creates a resource group in Azure:

  - Using the Azure CLI:

  - az group create -location westeurope -name MyAppResourcegroup

  - Using Azure PowerShell:

  - New-AzResourceGroup -Name MyAppResourcegroup -Location westeurope

- Disadvantages: Lot of lines of code & necessary to write all of the steps.

- Advantages: Useful for tasks that automate repetitive actions, Complex processing with a certain logic

Understanding IaC Practices

- **Declarative types**: These are languages in which it is sufficient to write the state of the desired system or infrastructure in the form of configuration and properties.

- The user has to write the final state of the desired infrastructure and the tool takes care of applying it.

- Ex: Terraform code to define the desired configuration of an Azure resource group:

```
resource "azurerm_resource_group" "myrg" {
    name = "MyAppResourceGroup"
    location = "West Europe"

    tags = {
        environment = "Bookdemo"
    }
}
```

Understanding IaC Practices

● Ex: To install and restart nginx on a server using Ansible:

```
---
- hosts: all
  tasks:
  - name: install and check nginx latest version
    apt: name=nginx state=latest
  - name: start nginx
    service:
    name: nginx
    state: started
```

The IaC Topology

- In a cloud infrastructure, IaC is divided into several typologies:

  - The deployment and provisioning of the infrastructure

  - The server configuration and templating

  - The containerization

  - The configuration and deployment in Kubernetes

The IaC Topology

● The deployment and provisioning of the infrastructure

○ Provisioning is the act of **instantiating the resources** that make up the infrastructure.

○ Platform as a Service (PaaS) and serverless resource types, such as a web app.

○ Provisioning tools such as **Terraform**, the ARM template, **AWS Cloud training**, the **Azure CLI**, **Azure PowerShell**, and also Google Cloud Deployment Manager.

The IaC Topology

- Server Configuration

  - **Configuration of virtual machines**, such as the configuration of directories, disk mounting, network configuration (firewall, proxy, and so on), and middleware installation.

  - Different configuration tools: **Ansible**, PowerShell DSC, **Chef**, **Puppet**, and SaltStack.

  - To optimize server provisioning and configuration times, it is also possible to create and use server models, also called images, that contain all of the configuration of the servers.

The IaC Topology

- Immutable Infrastructure with Containers

  ▪ Containerization consists of **deploying applications in containers** instead of deploying them in VMs.

  ▪ Container technology to be used is **Docker** and that the configuration of a Docker image is also done in code in a Dockerfile.

  ▪ This file contains the **declaration of the base image**, **additional middleware to be installed on the image**, only the files and binaries necessary for the application, and the network configuration of the ports.

  ▪ Unlike VMs, containers are said to be immutable; the **configuration of a container cannot be modified during its execution.**

The IaC Topology

- Configuration and deployment in Kubernetes

  ▪ Kubernetes is a **container orchestrator.**

  ▪ It is the technology that most embodies IaC because the way it deploys containers, the network architecture (load balancer, ports, and so on), and the volume management, as well as the protection of sensitive information, are described completely in the YAML specification files.

IaC best practices

- Everything must be automated in the code

- The code must be in a source control manager

- The infrastructure code must be with the application code

- Separation of roles and directories

- Integration into a CI/CD process

- The code must be idempotent

- To be used as documentation

- The code must be modular

- Having a development environment

Using Ansible for Configuring IaaS Infrastructure

Introduction to Ansible

Ansible is a tool that provides:

IT automation

Instructions are written to automate the IT professional's work

Configuration management

Consistency of all systems in the infrastructure is maintained
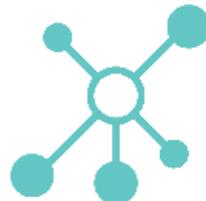
Automatic deployment

Applications are deployed automatically on a variety of environments

Introduction to Ansible
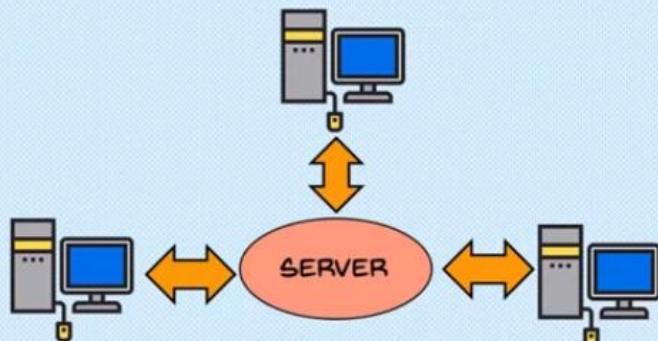
- Ansible is an open source **automation and orchestration tool** for **software provisioning**, **configuration management**, and **software deployment**.

- Can easily run and configure Unix-like systems as well as Windows systems to provide **infrastructure as code**.

- Contains its own declarative programming language for system configuration and management.
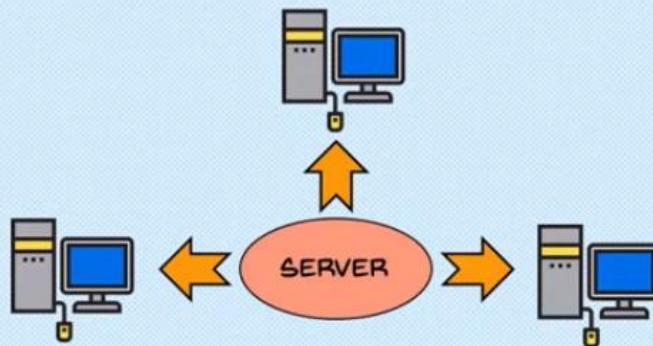
Why use Ansible?

- Free to use.

- No need for any special system administrator skills to install and use Ansible.

- Its modularity regarding plugins, modules, inventories, and playbooks make Ansible the useful for orchestrating large environments.

- Lightweight, consistent, and no constraints regarding the OS or underlying hardware.

- Secure due to its agentless capabilities and use of OpenSSH security features.

- It has a template engine and a vault to encrypt/decrypt sensitive data.

- Comprehensive documentation and easy to learn structure and configuration.

Pull & Push Configuration Tool

Pull configuration: Nodes check with the server periodically and fetch the configurations from it
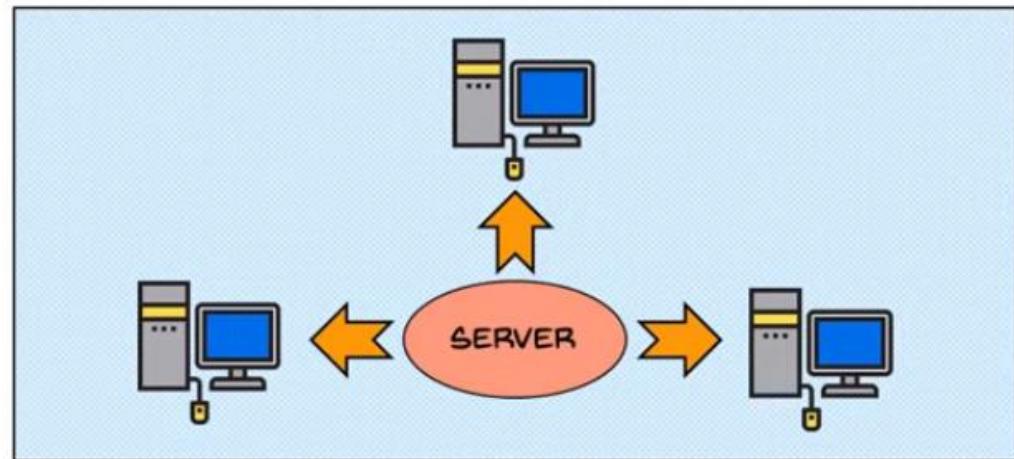
Push configuration: Server pushes configuration to the nodes

Ansible - Push Configuration Tool

Unlike Chef and Puppet, Ansible is push type configuration management tool

Push configuration: Server pushes configuration to the nodes

Important terms used in Ansible

- **Ansible server**: The machine where Ansible is installed and from which all tasks and playbooks will be run.

- **Module**: It is a command or set of similar Ansible commands meant to be executed on the client-side.

- **Task**: A task is a section that consists of a single procedure to be completed.

- **Role**: A way of organizing tasks and related files to be later called in a playbook.

- **Fact**: Information fetched from the client system from the global variables with the gather-facts operation.
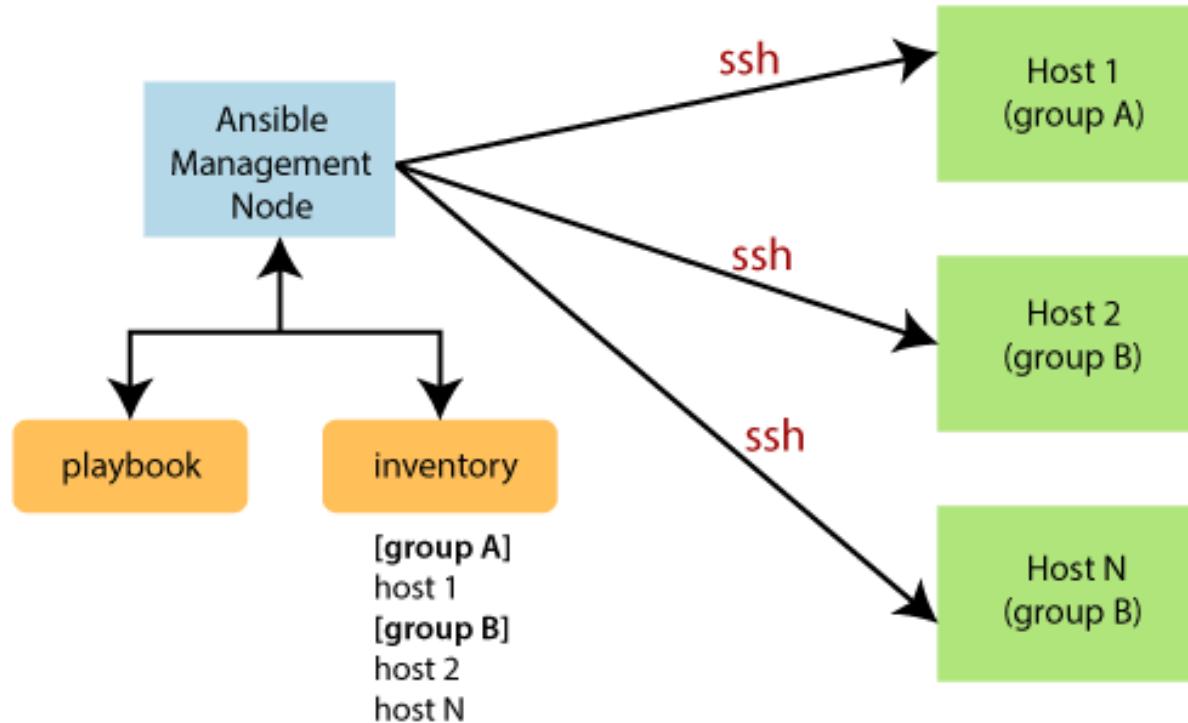
Important terms used in Ansible

- **Inventory:** File containing data about the ansible client servers. Defined in examples as hosts file.

- **Play:** Execution of a playbook.

- **Handler:** Task which is called only if a notifier is present.

- **Notifier:** Section attributed to a task which calls a handler if the output is changed.

- **Tag:** Name set to a task which can be used later on to issue just that specific task or group of tasks.
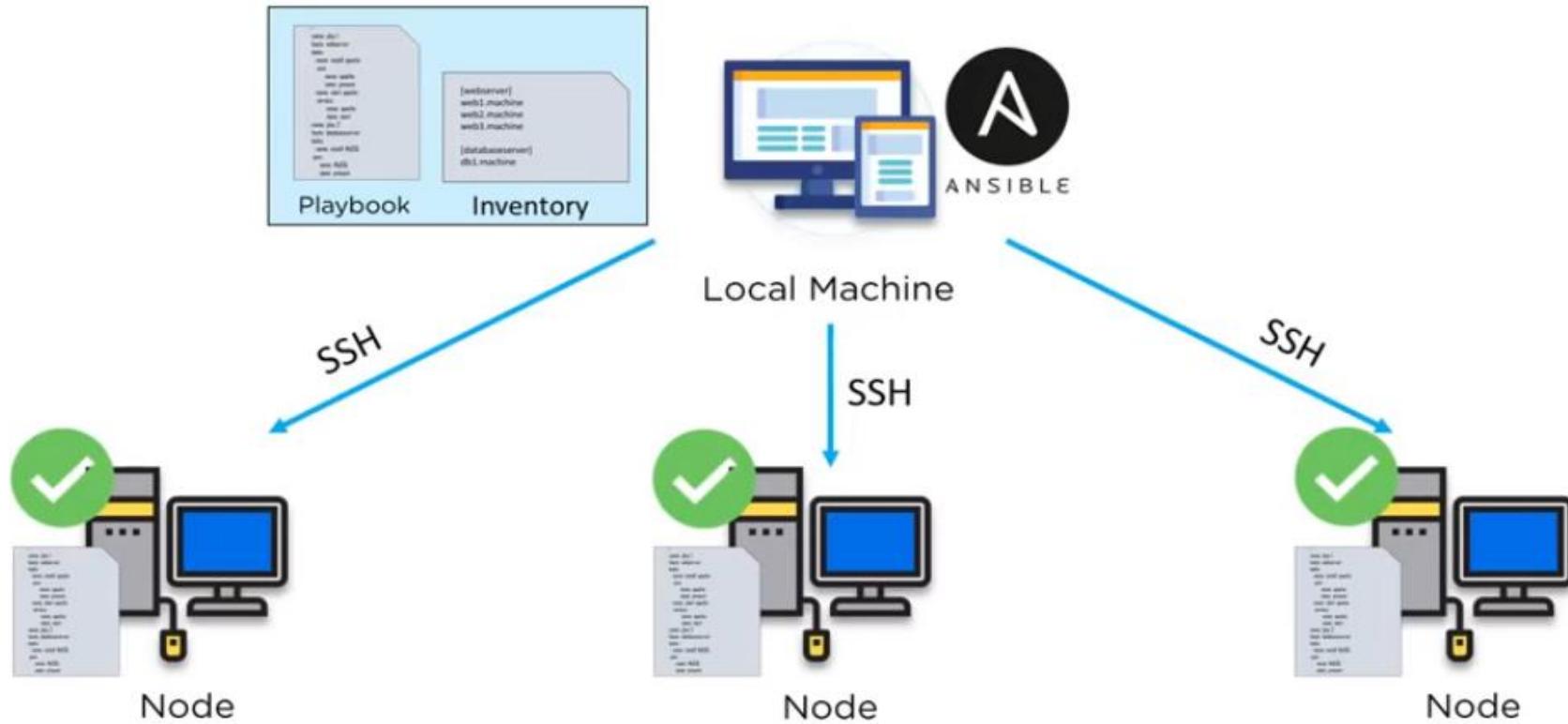
Ansible Workflow

- Ansible works by connecting to your nodes and pushing out a small program called Ansible modules to them.

- Then Ansible executes these modules and removes them after it is finished.

- The library of modules can reside on any machine, and there are no daemons, servers, or databases required.

Ansible Workflow

Working of Ansible
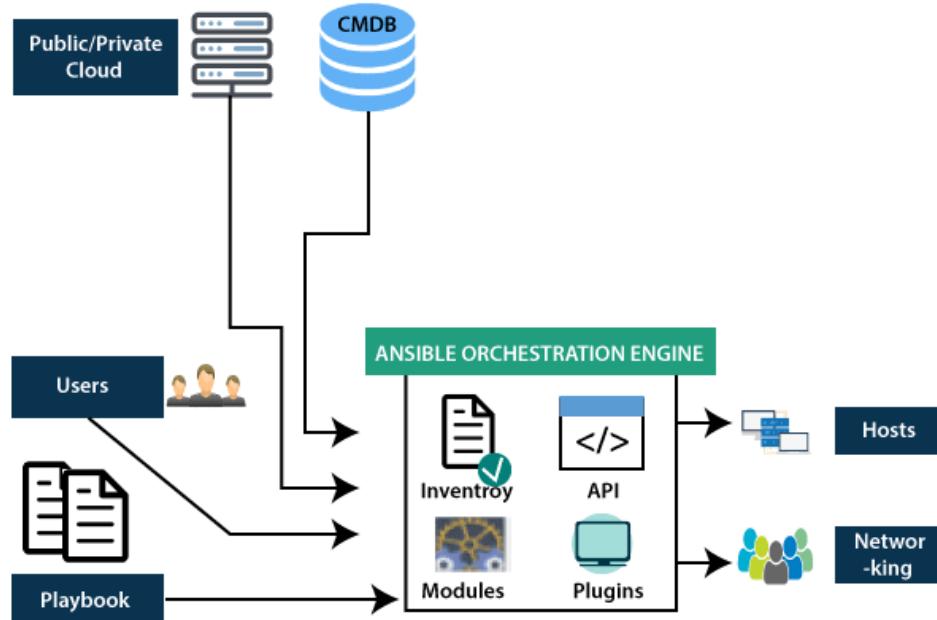
Ansible Workflow

- The Management Node is the controlling node that controls the entire execution of the playbook.

- The inventory file provides the list of hosts where the Ansible modules need to be run.

- The Management Node makes an SSH connection and executes the small modules on the host's machine and install the software.

- Ansible removes the modules once those are installed.

- It connects to the host machine executes the instructions, and if it is successfully installed, then remove that code.

Ansible Architecture

- The Ansible orchestration engine

  - Interacts with a user who is writing the Ansible playbook to execute the Ansible orchestration

  - Interact with the services of private or public cloud and configuration management database.

Ansible Architecture

- **Inventory**: Lists of nodes or hosts having their IP addresses, databases, servers, etc. which are need to be managed.

- **API's**: The Ansible API's works as the transport for the public or private cloud services.

- **Modules**:

  - Ansible connected the nodes and spread out the Ansible modules programs.

  - Ansible executes the modules and removed after finished. These modules can reside on any machine; no database or servers are required here.

Ansible Architecture

- **Plugins**: Plugins is a piece of code that expands the core functionality of Ansible. There are many useful plugins, and you also can write your own.

- **Playbooks**: Playbooks consist of code written in YAML format, which describes the tasks and executes through the Ansible. Can launch the tasks synchronously and asynchronously with playbooks.

- **Hosts**: These are the node systems, which are automated by Ansible, and any machine such as RedHat, Linux, Windows, etc.

Ansible Architecture

- **Networking**:

  - Ansible is used to automate different networks, and it uses the simple, secure, and powerful agentless automation framework for IT operations and development.

  - It uses a type of data model which separated from the Ansible automation engine that spans the different hardware quite easily.

- **Cloud**: A cloud is a network of remote servers on which you can store, manage, and process the data.

- **Configuration Management Database (CMDB)**: Type of repository which acts as a data warehouse for the IT installations.

Ansible Tower
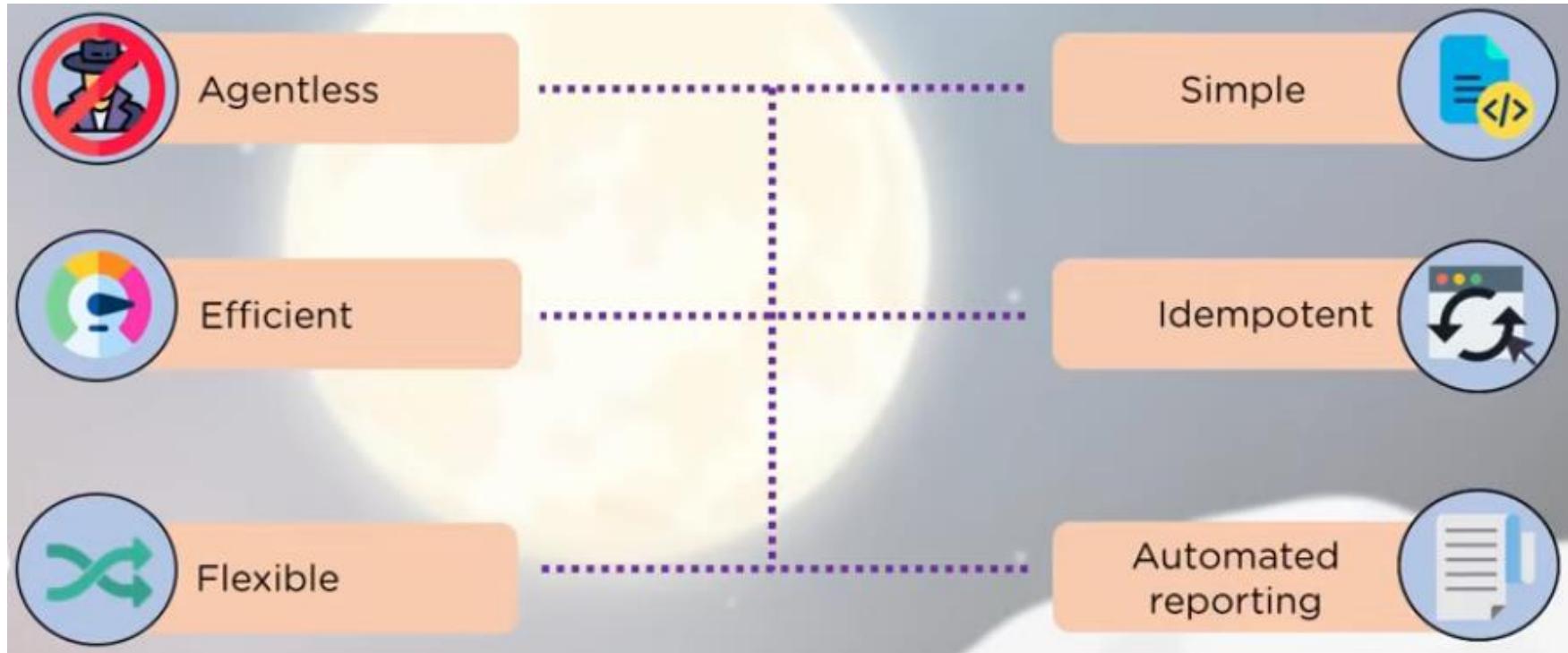
Ansible Tower is a framework for Ansible

ANSIBLE TOWER by Red Hat®

It provides a GUI. Thus, reducing the dependency on the command prompt window

Instead of typing long commands, tasks can now be performed in a single click

Benefits of Ansible

Agentless

Efficient

Flexible

Simple

Idempotent

Automated reporting

Installing Ansible with a script

- Ansible is not multiplatform and can only be installed on the following OSes:

    ▪ Red Hat, Debian, CentOS, macOS, or any of the BSDs

▪ Its **installation is done by a script** that differs according to your OS.

- To install its latest version on Ubuntu:

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

Installing Ansible with a script

● To install Ansible on a Windows OS machine, there is no native solution, but it can be installed on the **Windows Subsystem for Linux** (WSL).

● WSL allows **developers** who are **on a Windows OS to test their scripts** and applications directly on their workstation **without having to install a virtual machine**.

● To use Ansible in Azure Cloud, we must do the following:

  ▪ 1. Connect to the Azure portal at **https:/ /portal. azure.com**

  ▪ 2. Open Cloud Shell

  ▪ 3. Choose Bash mode

  ▪ 4. In the Terminal that opens, we now have access to all Ansible commands

Creating an inventory for targeting Ansible hosts

- The inventory contains the list of hosts on which Ansible will perform administration and configuration actions.

- There are two types of inventories:

- **1 Static inventory:**

    - Hosts are listed in a text file in INI (or YAML) format.

    - This is the basic mode of Ansible inventory.

    - The static inventory is used in cases where we know the host addresses (IP or FQDN).

Creating an inventory for targeting Ansible hosts

- There are two types of inventories:

- 2 Dynamic inventory:

  - The list of hosts is dynamically generated by an external script (Ex: With a Python script).

  - The dynamic inventory is used the addresses of the hosts are not available.

  - Ex: Infrastructure that is composed of on-demand environments.

The inventory file

- To configure hosts when running the playbook, a file that contains the list of hosts with the **list of IP** or **Fully Qualified Domain Name (FQDN)** addresses of the **target machines**.

- This list of hosts is noted in a **static file called the inventory file**.

- By default, Ansible contains an inventory file created during its installation; this file is **/etc/ansible/hosts**, and it contains several inventory configuration examples.

Configuring hosts in the inventory

- The entire Ansible configuration is in the **ansible.cfg** file.

- This **configuration is generic** and applies to all Ansible executions as well as connectivity to hosts.

- When using Ansible to configure VMs from **different environments or roles** with **different permissions**, it is important to have **different connectivity configurations**, such as different admin users and **SSL keys** per environment.

- **Override the default Ansible configuration** in the inventory file by configuring specific parameters per host defined in this inventory.

Configuring hosts in the inventory

- The main configuration parameters that can be overridden are as follows:

- **ansible_user**: This is the user who connects to the remote host.

- **ansible_port**: It is possible to change the default value of the SSH port.

- **ansible_host**: This is an alias for the host.

- **ansible_connection**: This is the type of connection to the remote host and can be Paramiko, SSH, or local.

- **ansible_private_key_file**: This is the private key used to connect to the remote host.

Writing the first Playbook

- Playbook is one of the essential elements of Ansible.

- It contains the **code of the actions or tasks** that need to be performed to configure or administer a VM.

- **Once the VM is provisioned, it must be configured**, with the installation of all of the middleware needed to run the applications that will be hosted on this VM.

- Necessary to perform administrative tasks concerning the configuration of directories and their access.

Writing a basic Playbook

- The code of a playbook is **written in YAML**, which allows us to easily visualize the configuration steps.

- Simple example of installation of an nginx server on an Ubuntu VM.

- Create a working **devopsansible** directory & in that folder create a **playbook.yml** file and insert the following content code:

```
---
- hosts: all
  tasks:
  - name: install and check nginx latest version
      apt: name=nginx state=latest
  - name: start nginx
    service:
    name: nginx
    state: started
---
```

Writing a basic Playbook

- The YAML file starts and ends with the optional **---** characters.

- The **- hosts** property contains the **list of hosts to configure**.

- In the example this property is set as **all to install nginx** on all of the VMs listed in the inventory.

- To install it only on a particular group, write it as follows:

  - hosts: webserver

Writing a basic Playbook
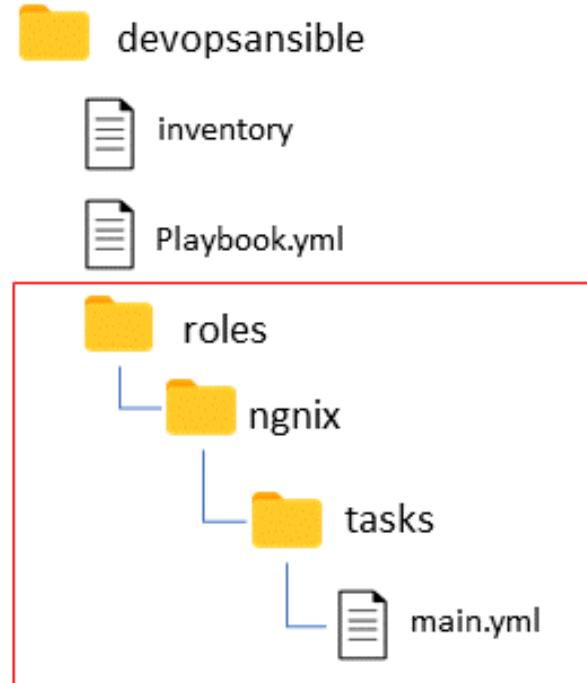
- Indicate the list of tasks or actions to be performed on these VMs, with the property of the **list of tasks**.

- Under the tasks element, **describe the list of tasks** and, for each of them, a **name that serves as a label**, in the name property.

- Under the name, call the function to be executed using the Ansible modules and their properties. Here, we have used two modules:

  - **apt**: To retrieve a package (the apt-get command) to get the latest version of the nginx package.

  - **service**: To start or stop a service—in this example, to start the nginx service.

Improving your Playbooks with Roles

- When configuring a VM, there will be a repetition of tasks for each application.

- Ex: Several applications require the identical installation of nginx.

- With Ansible, this repetition will require duplicating the playbook code.

- To **avoid this duplication** to save time, avoid errors, and homogenize installation and configuration actions, encapsulate the **playbook code in a directory** called **role** that can be used by several playbooks.

Improving your Playbooks with Roles

- To create the nginx role, create the following directory and file tree within our devopsansible directory:

Improving your Playbooks with Roles

● Then, in the main.yml file, located in tasks, copy and paste the following code from playbook in the file that is created:

```
- name: install and check nginx latest version
    apt: name=nginx state=latest
- name: start nginx
    service:
    name: nginx
    state: started
```

Improving your Playbooks with Roles

- Then, modify playbook to use this role with the following content:

```
---

- hosts: webserver
  roles:
    - nginx
```

- Following the node roles, provide a list of roles (names of the role directories) to be used.

- So, this nginx role is now centralized and can be used in several playbooks simply without having to rewrite its code.

Improving your Playbooks with Roles

- The following is the code of a playbook that configures a VM web server with Apache and another VM that contains a MySQL database:

```
---
- hosts: webserver
  roles:
    - php
    - apache
- hosts: database
  roles:
    - mysql
```

- **Ansible Galaxy** : A large public repository of Ansible roles and Roles ship with READMEs detailing the role's use and available variables.

- Within an enterprise, **create custom roles** and **publish them in a private galaxy** within the company.

Executing Ansible

- So far we have seen

    - Installation of Ansible

    - Listed the hosts in the inventory

    - Set up our Ansible playbook.

- Now Run Ansible to configure our VMs.

- Run the Ansible tool with the **ansible-playbook** command like:

    - *ansible-playbook -i inventory playbook.yml*

Executing Ansible

- The basic options of this command are as follows:

  ▪ The -i argument with the inventory file path

  ▪ The path of the playbook file

```
/devopsansible# ansible-playbook -i inventory playbook.yml --check

PLAY [webserver] ***********************************************************

TASK [Gathering Facts] *****************************************************
ok: [webserver1]
ok: [webserver2]

TASK [ngnix : install and check nginx latest version] **********************
changed: [webserver1]
changed: [webserver2]

TASK [ngnix : start nginx] *************************************************
changed: [webserver2]
changed: [webserver1]

PLAY RECAP ****************************************************************
webserver1                 : ok=3    changed=2    unreachable=0    failed=0
webserver2                 : ok=3    changed=2    unreachable=0    failed=0
```

Protecting data with Ansible Vault
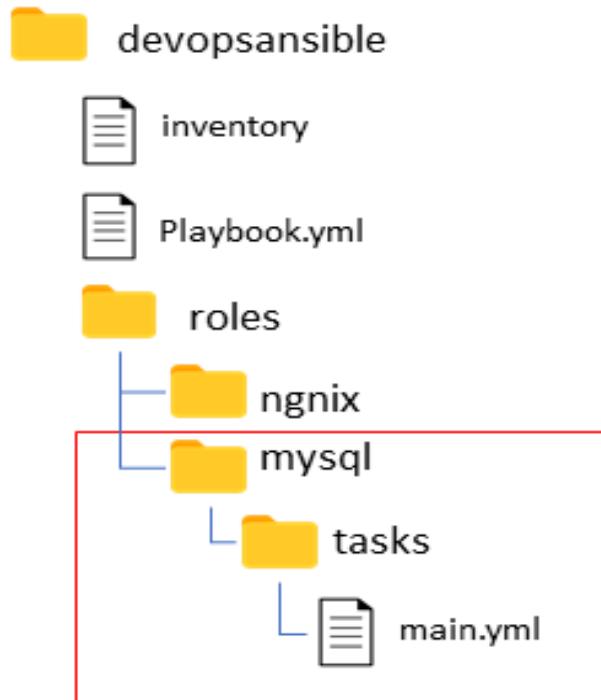
- Using Ansible with an inventory file that contains the list of hosts to configure, and with a playbook that contains the code of the host's configuration actions.

- In all IaC tools, it will be **necessary to extract** some **data** that is **specific to a context or environment inside variables**.

  - How to use variables in Ansible.

  - How to protect sensitive data with Ansible Vault.

Protecting data with Ansible Vault

- Using variables in Ansible for better configuration

- When deploying infrastructure with IaC, the code used is often composed of two parts:

  - A part that describes the elements or resources that make up the infrastructure.

  - Another part that differentiates the properties of this infrastructure from one environment to another.

- The second part of differentiation for each environment is done with the use of variables, and Ansible has a whole system that allows us to inject variables into playbooks.

Protecting data with Ansible Vault

- Using variables in Ansible for better configuration

- To see the use of variables in Ansible, add a role called mysql in the roles directory with the following tree structure:

Protecting data with Ansible Vault

- Using variables in Ansible for better configuration

- In the main.yml file of this role, we write the following code:

```
---
- name: Update apt cache
  apt: update_cache=yes cache_valid_time=3600
- name: Install required software
  apt: name="{{ packages }}" state=present
  vars:
  packages:
  - python-mysqldb
  - mysql-server
- name: Create mysql user
  mysql_user:
    name={{ mysql_user }}
    password={{ mysql_password }}
    priv=*.*:ALL
    state=present
```

Protecting data with Ansible Vault

- Using variables in Ansible for better configuration

- In this code, some static information has been replaced by variables. They are as follows:

- **packages**: This contains a list of packages to install and this list is defined in the following code.

- **mysql_user**: This contains the user admin of the MySQL database.

- **mysql_password**: This contains the admin password of the MySQL database.

Protecting data with Ansible Vault

- Using variables in Ansible for better configuration

- The different tasks of this role are as follows:

1. Updating packages

2. The installation of the MySQL server and Python MySQL packages

3. The creation of a MySQL user

Protecting data with Ansible Vault

- Using variables in Ansible for better configuration

- To define the values of these variables, **create a group_vars directory**, which will **contain all of the values of variables for each group** defined in inventory.

- In this group_vars folder, **create a database subdirectory** corresponding to the database group defined in the inventory and a main.yml subfile.

- In this main.yml file, we put the desired values of these variables as follows:

```
---
mysql_user: mydbuserdef
mysql_password: mydbpassworddef
```

Protecting data with Ansible Vault

- Using variables in Ansible for better configuration

- Finally, complete the playbook with a call to the mysql role by adding the following code:

```
- hosts: database
  become: true
  roles:
    - mysql
```

- Execute Ansible with the following command

  - *ansible-playbook -i inventory playbook.yml*

Protecting sensitive data with Ansible Vault

- In the Ansible tool, there is a **sub-tool called Ansible Vault** that **protects the data transmitted** to Ansible through playbooks.

- How to manipulate Ansible Vault to encrypt and decrypt the information of the MySQL user.

- First **encrypt** the **group_vars/database/main.yml** file that contains the values of the variables by executing the following command:

  - *ansible-vault encrypt group_vars/database/main.yml*

Protecting sensitive data with Ansible Vault

- Ansible Vault requests the inclusion of a password that will be required to decrypt the file and then shows the execution of this command to encrypt the content of a file:

```
███████████████/devopsansible# ansible-vault encrypt group_vars/database/main.yml
New Vault password:
Confirm New Vault password:
Encryption successful
```

Protecting sensitive data with Ansible Vault

- After this command, the content of the file is encrypted.

- The values are no longer clear.

- The following is a sample from it:

```
$ANSIBLE_VAULT;1.1;AES256
62313136303036362323534353031313939396232646461666613731336335
32363138323237363730643534653532666533336383231660a6134653265
34343466396330663434838613338393038303866393433663332323465
33366330336437353310a6261343131643239333631393234373737303161
35633163303562383162646266653430383138363431393466632663303137
33356336303730393766666656635383639366563383663339643261363033
62626330323362633032306336313136653962
```

Protecting sensitive data with Ansible Vault

- To decrypt the file to modify it, it will be necessary to execute the decrypt command:

  - *ansible-vault decrypt group_vars/database/main.yml*

- Ansible Vault requests the password that was used to encrypt the file, and the file becomes readable again.

Protecting sensitive data with Ansible Vault

- It is preferable to **store the password in a file** in a protected location, for example, in the **~/.vault_pass.txt** file.

- Then, to encrypt the variable file with this file, execute the ansible-vault command and add the --vault-password-file option:

- *ansible-vault encrypt group_vars/database/main.yml --vault-password-file ~/.vault_pass.txt*

Protecting sensitive data with Ansible Vault

- Now that the file is encrypted and the data is protected, run Ansible with the following commands:

- In interactive mode:

  - *ansible-playbook -i inventory playbook.yml --ask-vault-pass*

- Ansible asks the user to enter the password shown in the following screenshot:

Protecting sensitive data with Ansible Vault

- **In automatic mode**, that is, in a CI/CD pipeline, add the **--vault-passwordfile** parameter with the path of the file that contains the password to decrypt the data:

    - *ansible-playbook -i inventory playbook.yml --vault-password-file ~/.vault_pass.txt*

Using a dynamic inventory for Azure infrastructure

- When configuring an infrastructure that is composed of several VMs, along with **ephemeral environments** that are **built on demand,** maintaining a **static inventory** can quickly **become complicated** and its **maintenance takes a lot of time**.

- To overcome this problem

  - Ansible allows inventories to be obtained dynamically by calling a script.

  - This script is either provided by cloud providers or a script that engineers can develop.

  - The script aims to return the contents of the inventory.

Using a dynamic inventory for Azure infrastructure

1. Configure Ansible to access Azure resources.

- For this, we will create an Azure Service Principal in Azure AD.

- Then, export the information of four service principal IDs to the following environment variables:

  - export AZURE_SUBSCRIPTION_ID=<subscription_id>

  - export AZURE_CLIENT_ID=<client ID>

  - export AZURE_SECRET=<client Secret>

  - export AZURE_TENANT=<tenant ID>

Using a dynamic inventory for Azure infrastructure

2. Generate an inventory with groups and to filter VMs, it is necessary to add Tags to the VMs. Tags can be added using Terraform, an az cli command line, or an Azure PowerShell script.

3. To use a dynamic inventory in Azure, we need to Install the Python Azure SDK on the machine that runs Ansible.

4. To ensure that our dynamic inventory can be used by playbook, we need to install nginx on all hosts in the webserver group and install MySQL on all hosts in the database group.

Using a dynamic inventory for Azure infrastructure

5. After having set up all of the artifacts for our Ansible dynamic inventory in Azure, it is good to test its proper functioning, which includes the following:

- That there are no execution errors.

- The connection and authentication to our Azure environment are done correctly.

- Its execution returns the Azure VMs from our infrastructure.

Using a dynamic inventory for Azure infrastructure

6. Once we have tested our dynamic inventory in Azure, run Ansible on it, using the tags applied on the VMs.

- For this, execute the first Ansible command with the --check option, which allows us to check that the playbook execution is based on the right VM groups:

- ansible-playbook -i inventories/ playbook.yml --check --vaultpassword- file ~/.vault_pass.txt

Optimizing Infrastructure Deployment with Packer

What is Packer?

- Packer is an **open-source VM image creation tool** from Hashicorp.

- It helps you **automate** the process of **Virtual machine image creation** on the cloud and on-premise virtualized environments.

- All the manual steps performed to create a Virtual machine image can be automated through a simple Packer config template.

- You declare the state of the VM image you need, and Packer builds it for you.

What is Packer?

- It **integrates natively** with a bunch of configuration management systems eg: Ansible, Puppet.

- Packer is **cross-platform** (Linux/Window)

- Packer uses a **JSON template file** and lets you define immutable infrastructure.

- It's written in the **GO language**.

Introduction

- Automated configuration of VMs with Ansible.

- This automation allows us to benefit from a real improvement in productivity and very visible time-saving.

- Despite this automation, few important things:

1. **Configuring a VM can be very time-consuming** because it depends on its **hardening** as well as the **middleware** that will be installed and configured on this VM.

Introduction

2. Between each environment or application, the **middleware versions are not identical** because their automation script is not necessarily identical or maintained over time.

3. **Configuration and security compliance** is not often applied or updated.

- To address these issues, all cloud providers have integrated a service that allows them to **create or generate custom VM images**.

- These images contain all of the **configurations of the VMs** with their **security administration and middleware configurations**.

- These **images can then be used as a basis to create VMs** for applications.

Introduction

- The benefits of using these images are as follows:

  ▪ The provisioning of a VM from an image is very fast.

  ▪ Each VM is uniform in configuration.

  ▪ It is safety compliant

- Among the Infrastructure as Code (IaC) tools, there is Packer from the HashiCorp tools, which allows us to create VM images from a file (or template).

Packer Use Cases

- **Golden Image Creation**: With packer, you can template the configurations required for a golden VM image that can be used **across organizations**.

- **Monthly VM Patching**: You can integrate Packer in your **monthly VM image patching pipeline**.

- **Immutable Infrastructure**: If you want to create an immutable infrastructure using VM images as a deployable artifact, use Packer in your CI/CD pipeline.

    - Immutable Infrastructure is an approach to managing services and software deployments on IT resources wherein components are replaced rather than changed.

Installing Packer

- Packer is a cross-platform tool and can be installed on Windows, Linux, or macOS.

- The installation of Packer can be done in two ways: either manually or via a script.

- **Installing Manually**

  1. Go to the official download page (https://www.packer.io/downloads.html) and download the package corresponding to your operating system.

  2. After downloading, unzip and copy the binary into an execution directory (c:/Packer).

  3. Then, the PATH environment variable must be set with the path to the binary directory.

Installing Packer

- **Installing by a Script**

- An automatic script that can be installed on a remote server and be used on a CI/CD process.

- Packer can be used locally.

- But its real goal is to be **integrated into a CI/CD pipeline**.

- This automatic DevOps pipeline will allow the construction and publication of uniform VM images that will guarantee the integrity of middleware and VM security based on these images.

Installing Packer

- Installing Packer by script on Linux:

```
PACKER_VERSION="1.4.3" #Update with your desired version

curl -Os
https://releases.hashicorp.com/packer/${PACKER_VERSION}/packer_${PACKER_VER
SION}_linux_amd64.zip \
&& curl -Os
https://releases.hashicorp.com/packer/${PACKER_VERSION}/packer_${PACKER_VER
SION}_SHA256SUMS \
&& curl https://keybase.io/hashicorp/pgp_keys.asc | gpg --import \
&& curl -Os
https://releases.hashicorp.com/packer/${PACKER_VERSION}/packer_${PACKER_VER
SION}_SHA256SUMS.sig \
&& gpg --verify packer_${PACKER_VERSION}_SHA256SUMS.sig
packer_${PACKER_VERSION}_SHA256SUMS \
&& shasum -a 256 -c packer_${PACKER_VERSION}_SHA256SUMS 2>&1 | grep
"${PACKER_VERSION}_linux_amd64.zip:\sOK" \
&& unzip -o packer_${PACKER_VERSION}_linux_amd64.zip -d /usr/local/bin
```

Installing Packer

- This script performs the following actions:

1. Download the Packer version 1.4.0 package and check the checksum.

2. Unzip and copy the package into a local directory, /usr/local/bin (by default, this folder is in the PATH environment variable).

```
                                    /CHAP04# sh install_packer.sh
                                    Speed   Time    Time    Time  Current
                             Dload  Upload  Total   Spent   Left  Speed
100  1696  100  1696    0     0    3981     0 --:--:-- --:--:-- --:--:--  3971
gpg: key 51852D87348FFC4C: "HashiCorp Security <security@hashicorp.com>" not changed
gpg: Total number processed: 1
gpg:               unchanged: 1
gpg: Signature made Thu Apr 11 20:30:03 2019 DST
gpg:               using RSA key 91A6E7F85D05C65630BEF18951852D87348FFC4C
gpg: Good signature from "HashiCorp Security <security@hashicorp.com>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 91A6 E7F8 5D05 C656 30BE  F189 5185 2D87 348F FC4C
packer_1.4.0_linux_amd64.zip: OK
Archive:  packer_1.4.0_linux_amd64.zip
  inflating: /usr/local/bin/packer
```

Installing Packer

- **Installing Packer by script on Windows:**

- On Windows, use **Chocolatey**, which is a **software package manager**.

- Chocolatey is a free public package manager, like NuGet or npm, but dedicated to software.

- It is widely used for the **automation of software on Windows servers** or even local machines.

- Once installed, run the following command in PowerShell or in the CMD tool:

  ▪ *choco install packer -y*

Installing Packer

- The following is a screenshot of the Packer installation for Windows with Chocolatey:

Installing Packer

- **Integrating Packer with Azure Cloud Shell:**

- Packer is integrated with Azure Cloud Shell

- Checking the Packer installation

  ▪ *packer –version*

- To see all of the Packer command-line options:

  ▪ *packer --help*

Creating Packer templates for Azure VMs with scripts

- To create a VM image, Packer is based on a file (template) that is in JSON format.

- **The structure of the Packer template:**

- The Packer JSON template is composed of several main sections such as **builders, provisioners, and variables.**

```
{
    "variables": {
    // list of variables
    ...
    },
    "builders": [

      {
      //builders properties
      ...
      }
    ],
    "provisioners": [
      {
      // list of scripts to execute
      ...
      }
    ]
}
```

Creating Packer templates for Azure VMs with scripts

- **The builders section:** The builders section is mandatory and contains all of the properties that define the image and its location, such as:

  - Name

  - Type of image

  - Cloud provider on which the image will be generated

  - Connection information to the cloud

  - Base image to use

  - And other properties that are specific to the image type.

Creating Packer templates for Azure VMs with scripts

- **The builders section:**

- In this, the builders section defines an image that will be stored in the Azure cloud and is based on the Linux Ubuntu OS.

- Also configure the authentication keys for the cloud.

```
{
"builders": [{
  "type": "azure-rm",
  "client_id": "xxxxxxxx",
  "client_secret": "xxxxxxxx",
  "subscription_id": "xxxxxxxxx",
  "tenant_id": "xxxxxx",
  "os_type": "Linux",
  "image_publisher": "Canonical",
  "image_offer": "UbuntuServer",
  "location": "westus"
.......
 }]
}
```

Creating Packer templates for Azure VMs with scripts

- If we want to **create the same image but on several providers**, indicate in the same template file multiple block builders which will contains the provider properties.

- In this Packer template consists of the information for an image of an AWS VM, and the information for a Docker image based on Alpine.

- Advantage: To standardize the scripts that will be detailed in the provisioning section of these two images.

```
{
"builders": [
    {
    "type": "azure-rm",
    "location": "westus",
    ....
    },
    {
    "type": "docker",
    "image": "alpine:latest",
    ...
    }
]
}
```

Creating Packer templates for Azure VMs with scripts

- **The provisioners section:**

- This is an **optional section** and contains a **list of scripts** that will be executed by Packer on a **temporary VM base image** in order to build custom VM image.

- If the Packer template does not contain a provisioners section, **no configuration will be made on the base images.**

- The actions defined in this section are for Windows as well as Linux images.

- The actions be of several types such as executing a local or remote script, executing a command, or copying a file.

Creating Packer templates for Azure VMs with scripts

- **The provisioners section:**

- Possible to extend Packer by creating custom provisioning types.

- Ex: Packer will **upload and execute** the **local script**, hardeningconfig.sh **on the remote temporary VM base** image, and **copy the content** of the **scripts/installers local folder** to the **remote folder, /tmp/scripts,** to configure the image.

```
{
...
    "provisioners": [
        {
            "type": "shell",
            "script": "hardening-config.sh"
        },
        {
            "type": "file",
            "source": "scripts/installers",
            "destination": "/tmp/scripts"
        }
    ]
...
}
```

Creating Packer templates for Azure VMs with scripts

- When **creating an image of a VM, it's necessary to generalize it**—in other words, **delete all of the personal user information** that was used to create this image.

- For a Windows VM image, use the **Sysprep** tool as the last step of provisioners with this code:

```
"provisioners": [
...
{
 "type": "powershell",
 "inline": ["& C:\\windows\\System32\\Sysprep\\Sysprep.exe /oobe
/generalize /shutdown /quiet"]}
]
```

Creating Packer templates for Azure VMs with scripts

- For deleting the personal user information on a Linux image, use the following code:

```
"provisioners": [
.....
{
  "type": "shell",
  "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'",
  "inline": [
      "/usr/sbin/waagent -force -deprovision+user && export HISTSIZE=0 &&
sync"
  ]
}
]
```

Creating Packer templates for Azure VMs with scripts

- **The variables section:**

- In the Packer template, need to **use values that are not static in the code.**

- This **section is optional.**

- Used to define variables that will be filled either

  - as command-line arguments

  - as environment variables.

- These variables will then be used in the builders or provisioners sections.

Creating Packer templates for Azure VMs with scripts

- Example of a variables section:

  - The access_key variable with the ACCESS_KEY environment variable

  - The image_folder variable with the /image value

  - The value of the VM image size, which is the vm_size variable

```
{
 "variables": {
     "access_key": "{{env `ACCESS_KEY`}}",
     "image_folder": "/image",
     "vm_size": "Standard_DS2_v2"
 },
 ....
}
```

Creating Packer templates for Azure VMs with scripts

- To use these user variables, use the {{user 'variablename'}} notation.

- Ex: Using these variables in the builders section:

```
"builders": [
    {
        "type": "azure-arm",
        "access_key": "{{user `access_key`}}",
        "vm_size": "{{user `vm_size`}}",
        ...
    }
],
```

Creating Packer templates for Azure VMs with scripts

- In the provisioners section, use the variables defined in the variables section:

```
"provisioners": [
    {
        "type": "shell",
        "inline": [
        "mkdir {{user `image_folder`}}",
        "chmod 777 {{user `image_folder`}}",

        ...
    ],
        "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'"
    },
  ...
  ]
```

Creating Packer templates for Azure VMs with scripts

- **Define the properties of the image with variables** that will be provided when executing the Packer template.

- **Use these variables in the provisioners section** for centralizing these properties and not have to redefine them.

- Ex: Here, the path of the images (/image) is repeated several times in the templates.

Building an Azure image with the Packer template

- To create a Packer template that will create a VM image in Azure.

- First **create an Azure AD (Active Directory) Service Principal (SP)** that will have the **permissions to create resources in our subscription**.

- Then, **on the local disk, create an azure_linux.json file**, which will be a **Packer template**.

Building an Azure image with the Packer template

```json
..."builders": [{
    "type": "azure-arm",
    "client_id": "{{user `clientid`}}",
    "client_secret": "{{user `clientsecret`}}",
    "subscription_id": "{{user `subscriptionid`}}",
    "tenant_id": "{{user `tenantid`}}",

   "os_type": "Linux",
    "image_publisher": "Canonical",
    "image_offer": "UbuntuServer",
    "image_sku": "18.04-LTS",
    "location": "West Europe",
    "vm_size": "Standard_DS2_v3",

    "managed_image_resource_group_name": "{{user `resource_group`}}",

    "managed_image_name": "{{user `image_name`}}-{{user `image_version`}}",

   "azure_tags": {
    "version": "{{user `image_version`}}",
    "role": "WebServer"
      }
}],....
```

Building an Azure image with the Packer template

- This section describes the following:

    - It describes the **azure_rm** type, which indicates the provider.

    - Describes the **client_id**, **secret_client**, **subscription_id**, and **tenant_id** properties, which contain information from the previously created SP.

        - For security reasons, these values are not written in plain text in the JSON template; they will be placed in variables.

Building an Azure image with the Packer template

- This section describes the following:

    - The managed_image_resource_group_name and managed_image_name properties indicate the resource group as well as the name of the image to be created.

        - The name of the image is also placed into a variable with a name and a version number.

    - The other properties correspond to the information of the OS type (Ubuntu 18), size (Standard_DS2_v3), region, and tag.

Building an Azure image with the Packer template

- Now, write the variables section that defines the elements that are not fixed:

```
..."variables": {
 "subscriptionid": "{{env `AZURE_SUBSCRIPTION_ID`}}",
 "clientid": "{{env `AZURE_CLIENT_ID`}}",
 "clientsecret": "{{env `AZURE_CLIENT_SECRET`}}",
 "tenantid": "{{env `AZURE_TENANT_ID`}}",

 "resource_group": "rg_images",
 "image_name": "linuxWeb",
 "image_version": "0.0.1"
},...
```

Building an Azure image with the Packer template

- Defined the variables and their default values with the following:

  - The four pieces of authentication information from the SP will be passed either in the Packer command line or as an environment variable.

  - The **resource group, name, size, and region of the image** to be generated are also in variables.

  - The **image_version** variable that contains the version of the image is defined.

- With these variables, it is possible to use the same JSON template file to generate several images with different names and sizes.

Building an Azure image with the Packer template

- Finally, the last action is to write the steps of the provisioners image with the following code:

```
"provisioners": [
{
    "type": "shell",
    "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'",
     "inline": [
        "apt-get update",
        "apt-get -y install nginx"
     ]
 },
 {
    "type": "shell",
    "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'",
    "inline": [
      "/usr/sbin/waagent -force -deprovision+user && export HISTSIZE=0 &&
sync"
    ]
  }
]
```

Building an Azure image with the Packer template

- Here is what the previous code block is doing:

    - It updates packages with **apt-get update** and **upgrade**.

    - It **installs nginx**.

- Then, in the last step before the image is created, the **VM is deprovisioned** to **delete the user information** that was used to install everything on the temporary VM using the following command:

/usr/sbin/waagent -force -deprovision+user && export HISTSIZE=0 && sync

Using Ansible in a Packer template

- To write a Packer template that uses command scripts (for example, apt-get).

- But it is also possible to use Ansible playbooks to create an image.

- Indeed, when we use IaC to configure VMs, we are often used to configuring the VMs directly using Ansible before thinking about making them into VM images.

- Reuse the same playbook scripts that was used to configure VMs to create VM images.

- It saves time as no need to rewrite the scripts.

Using Ansible in a Packer template

- To put this into practice, we will write the following:

  - An Ansible playbook that installs nginx

  - A Packer template that uses Ansible with our playbook

- **Writing the Ansible playbook**

- The playbook we are going to use is Ansible for Configuring IaaS Infrastructure with some changes.

Using Ansible in a Packer template

- Writing the Ansible playbook

- The changes made are as follows:

  - There is no inventory because it is Packer that manages the remote host, which is the temporary VM that will be used to create the image.

  - The value of hosts is the local IP address.

  - Keep only the installation of nginx in this playbook.

  - Deleted the task that installed the MySQL database.

```
---
- hosts: 127.0.0.1
  become: true
  connection: local
  tasks:
  - name: installing Ngnix latest version
    apt:
      name: nginx

    state: latest
- name: starting Nginx service
  service:
    name: nginx
    state: started
```

Using Ansible in a Packer template

- Integrating an Ansible playbook in a Packer template:

- In terms of the Packer template, the JSON builders and variables sections are identical to one of the templates that uses scripts as in the Using Ansible in a Packer template.

- What is different is the JSON provisioners section as follows:

Using Ansible in a Packer template

- Integrating an Ansible playbook in a Packer template:

```
provisioners": [
 {
     "type": "shell",
     "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'",
     "inline": [
        "add-apt-repository ppa:ansible/ansible", "apt-get update", "apt-get
install ansible -y"
     ]
 },
 {
    "type": "ansible-local",
    "playbook_file": "ansible/playbookdemo.yml"
 },
 {
    "type": "shell",
    "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'",
    "script": "clean.sh"
 },
 .....//Deprovision the VM
 ]
```

141

Using Ansible in a Packer template

- The actions described in this provisioners section, which Packer will execute using this template are as follows:

1. Install Ansible on the temporary VM.

2. On this temporary VM, the ansible-local provisioner runs the playbook playbookdemo.yaml that installs and starts nginx.

3. The clean.sh script deletes Ansible and its dependent packages that are no longer used.

4. Deprovision the VM to delete the local user information.

Executing Packer

- Created the Packer templates.

- The next step is to run Packer to generate a custom VM image.

- Custom VM image will be used to quickly provision VMs that are already configured and ready

- To generate VM image in Azure, follow these steps:

  1. Configure Packer to authenticate to Azure.

  2. Check Packer template.

  3. Run Packer to generate image.

Executing Packer

- **Configuring Packer to authenticate to Azure:**

- To allow Packer to create resources in Azure, use the Azure AD SP that was created.

- To execute Packer in Azure, use the four pieces of authentication information of this SP in the environment variables provided in our Packer template in the variables section

  - subscription_id, client_id, client_secret & tenant_id

Executing Packer

- In our following template, we have four variables (client_id, client_secret, subscription_id, and tenant_id), which take as their values four environment variables (ARM_CLIENT_ID, ARM_CLIENT_SECRET, ARM_SUBSCRIPTION_ID, and ARM_TENANT_ID):

```
"variables": {
    "client_id": "{{env `ARM_CLIENT_ID`}}",
    "client_secret": "{{env `ARM_CLIENT_SECRET`}}",
    "subscription_id": "{{env `ARM_SUBSCRIPTION_ID`}}",
    "tenant_id": "{{env `ARM_TENANT_ID`}}",
    "resource_group": "rg_images",
    "image_name": "linuxWeb",
    "image_version": "0.0.1",
    "location": "West Europe",
    "vm_size": "Standard_DS2_v2"
},
```

Executing Packer

- Set these environment variables as follows (Linux Example):

  - export ARM_SUBSCRIPTION_ID=<subscription_id>

  - export ARM_CLIENT_ID=<client ID>

  - export ARM_SECRET_SECRET=<client Secret>

  - export ARM_TENANT_ID=<tenant ID>

Executing Packer

- **Checking the validity of the Packer template:**

- Before executing Packer to generate the image, execute the packer validate command to check that our template is correct.

- So, inside the folder that contains the Packer template, execute the following command on the template:

  - *packer validate azure_linux.json*

- Output: Status of the check for whether the template is valid or not

Executing Packer

- Running Packer to generate our VM image:

- To generate image with Packer, execute Packer with the build command on the template file as follows:

  - *packer build azure_linux.json*

- In the output of the Packer execution, see the different actions being performed by Packer:

1. First is the creation of the temporary VM

Executing Packer

1. First is the creation of the temporary VM

Executing Packer

1. First is the creation of the temporary VM

- In the Azure portal, we see a temporary resource group and its resources created by Packer, as shown in the following screenshot:

Executing Packer

2. The execution time of Packer depends on the actions to be performed on the temporary VM.

- At the end of its execution, Packer indicates that it has generated the image and deletes the temporary resources.

3 The following screenshot is the end of the output of the Packer execution, which displays the deletion of the temporary resource group and the generation of the image

Executing Packer

3 The following screenshot is the end of the output of the Packer execution, which displays the deletion of the temporary resource group and the generation of the image:

Executing Packer

4 After the Packer execution, in the Azure portal,check if that the image is present.

● The following screenshot shows the generated image:

| NAME ↑↓	TYPE ↑↓	LOCATION ↑↓
linuxWeb-0.0.1	Image	West Europe
linuxWebAnsible-0.0.1	Image	West Europe
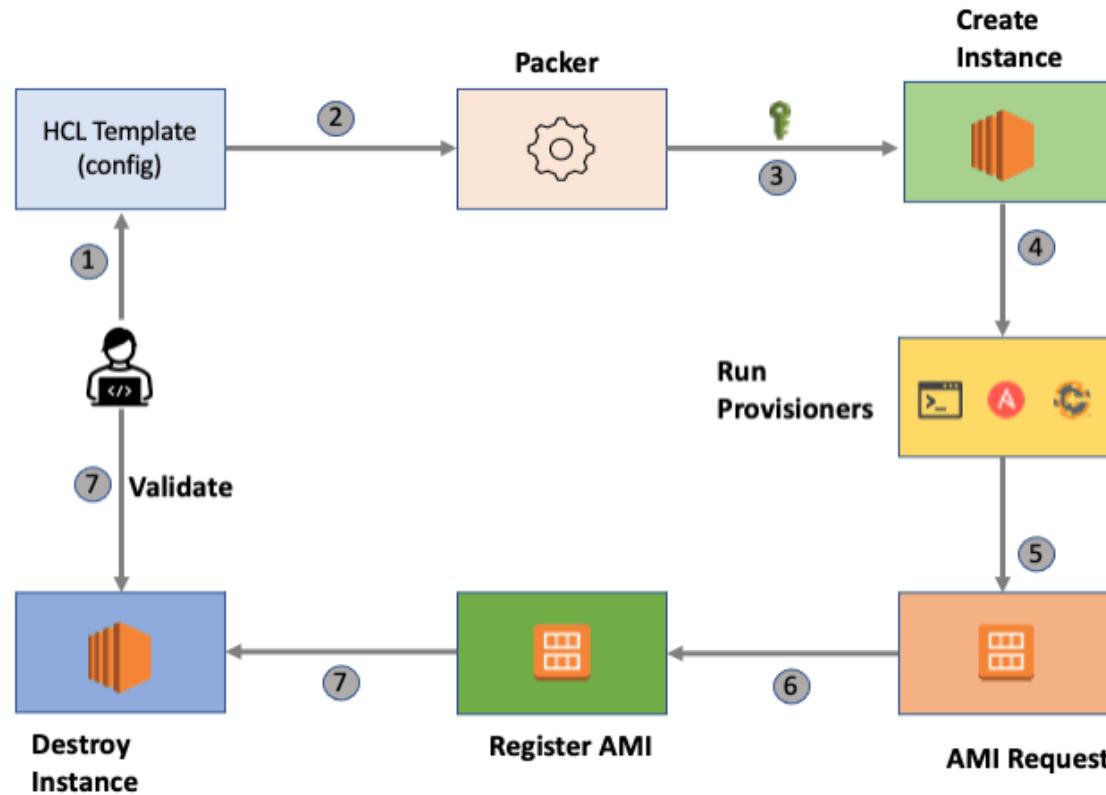
Building VM Image (AWS AMI) Using Packer

- Building a VM image using Packer is a simple process:

1. Declare all the required VM configurations in an HCL (Hashicorp configuration language) or a JSON file. Let's call it the Packer template.

2. To build the VM image, execute Packer with the Packer template.

3. Packer authenticates the remote cloud provider and launches a server. If you execute Packer from a cloud environment, it leverages the cloud service account for authentication.

Building VM Image (AWS AMI) Using Packer

- Building a VM image using Packer is a simple process:

4. Packer takes a remote connection to the server (SSH or Winrm).

5. Then it configures the server based on the provisioner you specified in the Packer template (Shell script, Ansible, Chef, etc).

6. Registers the AMI

7. Deletes the running instance.

Building VM Image (AWS AMI) Using Packer

Thank You