INFRAESTRUCTURA II – RESUMEN PRIMER PARCIAL
- Infraestructura como código
- Pipelines de CI/CD
- Monitoreo

Infraestructura como código:

Es la gestión de la infraestructura en un modelo descriptivo, utilizando las mismas herramientas de versionado que un equipo utiliza para su código fuente. Así como el mismo código fuente genera el mismo código binario, un modelo de infraestructura como código debe generar el mismo entorno cada vez que se aplica. La infraestructura como código, en conjunto con los pipelines de despliegue continuo, permite automatizar los despliegues de infraestructura, haciéndolos mas rápidos y menos propensos a errores, ademas nos evita depender de un equipo de infraestructura.

Pipelines de CI/CD:

La integración **continua (CI)** es una practica de desarrollo que consiste en integrar el código a un repositorio compartido de manera frecuente, idealmente varias veces al día. Cada integración es verificada por un proceso automatizado, permitiendo a los equipos detectar problemas rápidamente.
El **despliegue continuo (CD)** es la capacidad de poner en producción, en manos de los usuarios, cambios de cualquier tipo (nuevas funcionalidades, cambios de configuración, soluciones de errores y experimentos) de manera segura y sostenible. Esto se logra al asegurarnos que el código de encuentra en un estado desplegable, incuso al hacer cambios constantes.
Estas dos practicas se llevan a cabo mediante **pipelines de CI o CD** respectivamente, que son procesos automatizados por los que pasa el código (fuente o binario) hasta llegar a su destino final, que puede ser un entorno de pruebas o de producción.

Monitoreo:

Se divide en dos grandes ramas:
- **Monitoreo de aplicaciones:** que es el proceso de medir la performance, disponibilidad y experiencia del usuario de una aplicación. Estas métricas se utilizan para identificar y resolver problemas en la aplicación antes de que impacten en los usuarios.
- **Monitoreo de servidores:** es el proceso de ganar visibilidad respecto a la actividad de nuestros servidores, sean físicos o virtuales. Se puede enfocar en distintas métricas de los servidores , pero las principales son la disponibilidad y la carga.

DEVOPS:

Herramientas que nos ayudan a automatizar procesos y a monitorear recursos y así lograr agilidad:

- *Control de versiones:* práctica de llevar un registro y gestión de los cambios que se hacen en el código fuente del software. Herramientas: sistema de control de versiones como git, bitbucket, gitlab, github. Si se comete un error, los desarrolladores pueden deshacer los cambios y volver a una versión anterior del código.

- *Contenedores:* unidad estándar de software que empaqueta el código y todas sus dependencias para que la aplicación pueda funcionar de forma confiable en distintos entornos. Herramientas: docker, aws ecs, kubernetes.

- *Orquestadores:* de contenedores. Se ocupan del despliegue, gestión, escalamiento, conectividad y disponibilidad de las aplicaciones basadas en contenedores.

- *Monitoreo de las aplicaciones:* proceso de medir performance, disponibilidad y UX de una aplicación. Herramientas: dynatrace, prometheus.
- *Monitoreo de servidores:* proceso de ganar visibilidad respecto a la actividad de nuestros servidores. Herramientas: datadog, splunk. Disponibilidad y carga son las principales.

- *Gestión de configuración:* proceso que lleva registro de las distintas configuraciones que un sistema adopta a lo largo de su ciclo de vida. Herramientas: puppet, chef, ansible.

- Integración *continua CI:* practica de desarrollo que consiste en integrar el código a un repositorio

compartido lo mas frecuentemente posible, idealmente varias veces al día. Cada integración es verificada por un proceso automatizado permitiéndole a los equipos detectar problemas rápidamente. Herramientas: bamboo, jenkins, github actions, cicd.

- *Despliegue continuo:* habilidad de poner en producción (usuarios) cambios de cualquier tipo de manera segura y sostenible (nuevas funcionalidades, cambios de configuración, solución de errores, experimentos). Se logra asegurándonos que el código se encuentra en un estado desplegable (Deployed).

- *Automatizacion de pruebas:* software que hace uso de herramientas de automatización para controlar la ejecución de las pruebas. Luego, los resultados son comparados con los resultados esperados de cada prueba. Reduce el tiempo de ejecución y minimiza la tasa de error humano. Herramientas: ranorex, selenium, kobiton.

- *Infraestructura como código:* es la gestión de infraestructura como un modelo descriptivo. En conjunto con pipelines, permiten automatizar los despliegues, haciéndolos mas rápidos y minimizando errores. Herramientas: pulumi, cloudFormation, terraform.

- Computación *en la nube:* uso de servicios de cómputos (servidores, almacenamiento, bases de datos, redes, software, analítica, inteligencia artificial) a través de internet. Herramientas: aws, google cloud plataform. Recursos flexibles + economía de escala = eficiente.

Perfiles que podemos encontrar de manera típica en un ecosistema DevOps:

- Desarrolladores de aplicaciones: Son quienes desarrollan la aplicación, los programadores front-end, back-end, mobile, full stack o especializados en una tecnología particular —como Solidity— o plataforma — por ejemplo, Internet de las cosas (IoT)—. En un entorno DevOps es importante que se comuniquen constantemente con los demás roles.

- Analistas de calidad (QA): Son quienes verifican y validan la aplicación. En un entorno DevOps es importante que también se concentren en automatizar pruebas para hacerlas repetibles y confiables.

- Analistas de infraestructura: Son quienes implementan la infraestructura sobre la cual se ejecutarán las aplicaciones y las bases de datos. También se ocupan del mantenimiento y la evolución de esta infraestructura. Buena parte de las prácticas de DevOps recaen sobre ellos, en especial la comunicación con quienes desarrollan la aplicación. Dado que muchas veces la infraestructura existe en la nube, también se los suele llamar analistas clouds o analistas de nube.

- Analistas de redes: Se ocupan de la interconexión entre distintos sistemas, es decir, de las redes de computadoras —sean físicas o virtuales—. Es poco frecuente que se necesite una persona dedicada de forma exclusiva a las redes, es más común que este rol sea ocupado por el analista de infraestructura.
- Analistas de seguridad: Son personas que trabajan en la seguridad de la aplicación y de la infraestructura. A veces no se dispone de un empleado por equipo dedicado de forma exclusiva a este rol. En esos casos es importante que todo el equipo reciba entrenamiento en seguridad.

- Analistas de CI/CD: Son quienes mantienen los pipelines de integración y despliegue continuos. En aplicaciones simples es común que esta persona sea la misma que ocupa el rol de analista de infraestructura, pero en aplicaciones más complejas es necesario diferenciar roles.

- Arquitectos de nube: Definen la arquitectura del entorno en la nube: la estructura que tendrán los servidores, cómo se interconectan y varios aspectos de seguridad relacionados. También definen quiénes tendrán acceso a los distintos entornos. En organizaciones pequeñas no hay una persona dedicada de forma exclusiva a esto y la función recae sobre el analista de

infraestructura.

- Ingenieros de confiabilidad de sitio (SRE): Son los encargados de diseñar y monitorear el sistema para minimizar las suspensiones de servicio y el tiempo de recuperación de los servicios. Trabaja tanto de forma proactiva como reactiva, respondiendo a incidentes, pero también intentando que no ocurran o vuelvan a ocurrir.

- Gerentes de entregas: En algunos casos no es posible realizar despliegue continuo, por limitaciones del mercado o por la naturaleza del producto —por ejemplo, cada despliegue significa inevitablemente una suspensión temporal del servicio o cada cliente requiere una versión distinta del producto—. En estos casos, el gerente de entregas se ocupa de coordinar la entrega de nuevas versiones del producto a los clientes, llevar registro de qué cliente tiene qué versión del producto y orientar los esfuerzos del equipo hacia la satisfacción de los clientes.

INFRAESTRUCTURA COMO CÓDIGO: LA DISCIPLINA

Empleamos el término *Infraestructura como Código* (**IaC**, por sus siglas en inglés) para referirnos a la gestión de la infraestructura a través de templates que tienen la capacidad de ser versionados. De esta forma vamos a poder automatizar los procesos manuales que se requieren para lograr el objetivo final que buscamos.

- creación de un servidores

- aprovisionamiento de base de datos

- creación de un cluster para correr nuestros contenedores

Pasos para la infraestructura como código:

- Analizar qué infraestructura necesitamos según requisitos de la

aplicación. - Cuántas réplicas debe haber de nuestra infraestructura.

- Escribir nuestro template.

- Ejecutar nuestro template en nuestra herramienta IaC o se lo proveemos a otro equipo para que lo pueda ejecutar cunado lo necesite.

- Recibimos feedback de otro equipo o la herramienta de automatización nos avisa que termino de ejecutarse correctamente.
Se crean archivos de configuración que contienen las especificaciones que se necesitan, lo cual facilita la edición y distribución. A su vez, se garantiza que siempre se prepare el mismo entorno.

Procedimiento de IaC:

- Definir y describir las especificaciones de la infraestructura.

- Los archivos que se crean, se envían a un repositorio de código o de almacenamiento de archivos.

- La plataforma IaC toma las acciones necesarias para crear y configurar los recursos de infraestructura.

En AWS la **maquina virtual** como recurso se la conoce con el nombre de

EC2. BENEFICIOS DE LA INFRAESTRUCTURA COMO CÓDIGO:

- ➔ *Reducción del error humano:* Minimizamos el riesgo de equivocarnos cuando seguimos una serie de pasos. Mediante procedimientos claros y ordenados podemos evitar guardar una mala configuración o borrar algo que no debíamos. Esto va a aumentar la confianza que tengan en la infraestructura que brindemos.

➔ *Repetibilidad y predictibilidad:* Cuando sabemos que el contexto de nuestra aplicación funciona, vamos a poder repetir la cantidad de pasos que sean necesarios y ser capaces de predecir el resultado, ya que siempre será el mismo. Esto nos da —como resultado— una infraestructura más testeable y estable.

➔ *Tiempos y reducción de desperdicios:* El encargado de ejecutar nuestra infraestructura va a poder hacerlo en cuestión de minutos y sin necesidad de instalar algún componente extra.

➔ *Control de versiones:* Nuestra infraestructura se va a encontrar definida en archivos, por lo que vamos a poder versionar —al igual que el código fuente de nuestra aplicación— en templates o plantillas. Podemos utilizar parámetros para escribir nuestro código de la manera más genérica posible. Luego, al ejecutarlo, vamos a poder enviarle datos distintos en forma de parámetros para que nuestro código los reciba.

➔ *Reducción de costos:* Al automatizar procesos, podemos enfocarnos en otras tareas y mejorar lo ya hecho. Esto aporta a la flexibilidad de los equipos de infraestructura para abarcar más tareas.

➔ *Testeos:* La infraestructura como código permite que los equipos de infraestructura puedan realizar pruebas de las aplicaciones en cualquier entorno (incluso producción) al principio del ciclo de desarrollo.

➔ *Entornos estables y escalables:* Al evitar configuraciones manuales, la falta de dependencias y al obtener el estado final de infraestructura que necesitamos para nuestras aplicaciones, vamos a ofrecer entornos estables y escalables.

➔ *Estandarización de la configuración:* Estandarizar las configuraciones y el despliegue de la infraestructura nos permite evitar cualquier problema de incompatibilidad con nuestra infraestructura y que las aplicaciones se ejecuten con el mejor rendimiento posible.

➔ *Documentación:* Al aportar a la documentación de los procesos internos de nuestros equipos vamos a mejorar tiempos y costos. Como ya vimos, podemos versionar nuestras automatizaciones. Esta característica nos permite que cada cambio se encuentre documentado, registrado por usuario y con una vuelta atrás (rollback) rápida si encontramos errores en los despliegues, al igual que el código fuente.

➔ *Mas rapidez sin descuidar la seguridad:* Al momento de mejorar nuestra infraestructura, nunca hay que dejar de pensar en la seguridad que la compone. Al momento de estandarizar la ejecución de la infraestructura, también podemos estandarizar los grupos de seguridad con los permisos mínimos, pero necesarios para que todos los equipos puedan trabajar y evitar tareas manuales por parte de los equipos de seguridad.

Existen dos paradigmas de programación aplicados a la Infraestructura como código. Al escribir nuestro IaC podemos optar por el **paradigma imperativo**, que nos posibilita controlar el flujo de trabajo de nuestro código, o bien enfocarnos en el resultado final y en el cambio de nuestra infraestructura, el **paradigma declarativo**. Es el "cómo" versus el "qué".

Utilizamos el **paradigma imperativo** cuando al escribir nuestro código nos enfocamos en **cómo** se va a ejecutar a través de diversas operaciones y el flujo de trabajo que va a realizar. Vamos a definir variables constantes y definir decisiones. Las más conocidas son:

- IF
- ELSE
- ELIF (en otros lenguajes se conoce como ELSE IF)
- FOR y FOREACH
- WHILE y DO WHILE

- SWITCH (no existe en todos los lenguajes)
- Manejo de errores con excepciones (TRY/CATCH/FINALLY)

Consideramos que la utilización de este tipo de controles son imperativos porque estamos controlando de manera explícita nuestro flujo de trabajo dentro del código y qué decisiones se ejecutan según las condiciones que definamos.

Se utilizan estructuras de control o loops para controlar el proceso de nuestro código.

Al utilizar el **paradigma declarativo**, vamos a trabajar sobre la lógica de **qué** se va a ejecutar, sin indicar los detalles de cómo lo va a hacer. Al utilizar este método, nuestro código va a estar compuesto por un conjunto de funciones que van a realizar la tarea que definamos. Es muy importante tener test automatizados para probar nuestro código. Al ejecutarlos y ver los resultados vamos a tener la posibilidad de identificar errores en la lógica de nuestro código. Podemos decir que el enfoque declarativo define el estado final de nuestra infraestructura.

Se utilizan métodos para el proceso de nuestro código y luego se ejecutan pruebas (o tests).

El principio de idempotencia

La idempotencia es un principio matemático utilizado en infraestructura. Propiedad de ejecutar nuestro código la cantidad de veces que sea necesario, siempre obteniendo el mismo resultado.

Herramientas que ayudan con la idempotencia: Terraform, Ansible y

CloudFormation. IaC en etapas:

- *Origen* -> Archivo de configuración (JSON o YML)
- *Proceso* -> Operaciones que realizan las herramientas de IaC en base al archivo de origen
- *Destino* -> Estado final de la infraestructura, tal como se necesite.

Beneficios de la idempotencia en estas tres etapas:

- Modificando el archivo de origen, se modifica la infraestructura
- Fácilmente documentable
- Solo se debe agregar documentación complementaria cuando se necesite. -

Implementar practicas de desarrollo de software (EJ: versionar los archivos de config).

- Automatizable.

Al automatizar nuestra infraestructura, es probable que utilicemos distintos proveedores o que usemos una parte cloud y otra parte on-premise (un datacenter propio). Existen herramientas que poseen su propia sintaxis (en general, JSON y YAML) para poder administrar la infraestructura en múltiples proveedores o en uno solo, pero de una manera más eficiente.

TERRAFORM:

Terraform es un software de código libre desarrollado por HashiCrop. Es una herramienta declarativa de aprovisionamiento y orquestación de infraestructura que permite automatizar el aprovisionamiento de todos los aspectos de la infraestructura, tanto para la nube como la infraestructura on-premise (en los mismos datacenter). Tiene algunas características interesantes, como comprobar el estado de la infraestructura antes de aplicar los cambios. Es la herramienta más popular porque es compatible con todos los

proveedores de nube sin realizar modificaciones en nuestros templates.

AWS CLOUDFORMATION:

AWS CloudFormation es la solución nativa de AWS para aprovisionar recursos en esta nube. En este caso se pueden definir templates en formato JSON o YAML. Se pueden utilizar para crear, actualizar y eliminar recursos las veces que sea necesario. Una ventaja de CloudFormation es que, al ser un servicio propio de Amazon, tiene una integración completa con los demás servicios de AWS, por lo que es nuestra mejor opción si solo utilizamos este proveedor de nube.

AZURE RESOURCE MANAGER:

ARM es la herramienta nativa en Azure para implementar infraestructura como código, Azure Resource Manager (ARM Templates). Estas plantillas llevan una sintaxis declarativa en formato JSON, que nos permiten definir los recursos y las propiedades que conforman la infraestructura.

GOOGLE CLOUD DEPLOYMENT MANAGER:

Google Cloud Deployment Manager es la herramienta IaC para la plataforma Google Cloud —lo mismo que CloudFormation es para AWS—. Con esta herramienta los usuarios de Google pueden administrar fácilmente mediante archivos de configuración YAML.

ANSIBLE:

Ansible es una herramienta de automatización de infraestructuras creada por Red Hat. Ansible modela nuestra infraestructura describiendo cómo se relacionan sus componentes y el sistema entre sí, en lugar de gestionar los sistemas de forma independiente.

1. La gestión de Infraestructura como Código, es un concepto inspirado por la metodología [DevOps].
2. Dentro de los beneficiosde implementar IaC, está la mitigación de [riesgos] por error humano, ya que evitamos tareas [manuales].
3. Al momento de escribir el código de nuestra infraestructura, podemos elegir el paradigma imperativo o [declarativo].
4. [Terraform] es la herramienta más popular, dentro de las que son compatibles con múltiples nubes.

INFRAESTRUCTURA COMO CÓDIGO EN AWS: CLOUDFORMATION

CloudFormation ->Empresa: Amazon Web Services (AWS) ->AñoLanzamiento: 2018

CloudFormation es una herramienta nativa de Amazon Web Services (más conocido como AWS). Nos brinda la posibilidad de implementar prácticas de infraestructura como código (IaC) de forma nativa dentro de AWS.

CloudFormation crea y configura la infraestructura que definimos previamente en una plantilla (o template) de acuerdo a los requisitos que necesitamos. Esto nos ofrece algunas ventajas, como crear repositorios con nuestros templates para que sean accesibles o que se puedan realizar entregas rápidas de los recursos de infraestructura.

Caracteristicas mas notables:
- Templates reutilizables (Pueden ser JSON, YML, TXT o Template.)

- Templates parametrizables(Pueden ser JSON, YML, TXT o Template.)

- Automatizables.

- Rollbacks

Funcionamiento:

Ejemplo de template para la creación de EC2, base de datos en Amazon RDS, un balanceador de carga (load balancer), una VPC y subnets y servicios de grupos de seguridad.

```yaml
AWSTemplateFormatVersion: "2010-09-09"
Description: Automatizamos Bucket S3
#Se ingresa el bucket por parametro
Parameters:
  BucketName:
    Type: String
    Description: El nombre de mi Bucket S3
#se definde la info que muestra en la interfaz grafica
Metadata:
  AWS::CloudFormation::Interface:
    ParametersGroup:
    - Label:
        default: Bucket S3
      Parameters:
      - BucketName
    ParametersLabel:
      BucketName:
        default: 'Nombre de mi Bucket: '
#creamos el recurso
Resources:
  S3Bucket:
    Type: AWS::S3::Bucket
    Properties:
    #nombre => se recibe por parametro
      BucketName: !Ref BucketName
      #deshabilitamos visibilidad publica
      PublicAccessBlockConfiguration:
        BlockPublicAcls: true
        BlockPublicPolicy: true
        IgnorePublicAcls: true
        RestrictPublicBuckets: true
        #habilitamos versionado de archivos
      VersioningConfiguration:
        Status: Enabled
```

CloudFormation se puede utilizar en distintos ámbitos:

- Podemos hacerlo por línea de comando desde nuestros equipos.
- En scripts (como PowerShell).
- En pipelines, como parte de un conjunto de tareas automatizadas y encadenadas entre sí, formando una tubería con un inicio y un fin.

ANSIBLE:

Ansible -> Empresa: Red Hat -> Año lanzamiento: 2012

Ansible es un proyecto comunitario *open source* diseñado para ayudar a las organizaciones a automatizar el aprovisionamiento de infraestructura, la gestión de configuración y el despliegue de aplicaciones. Asimismo, es importante tener en cuenta que es fácil de aprender.

Con Ansible se crean archivos de configuración llamados *playbooks*, escritos en YAML, que se utilizan para especificar el estado requerido de la infraestructura. Al ejecutarlos, Ansible se ocupa de aprovisionar la infraestructura necesaria para alcanzar el estado descrito.

Esto quiere decir que se puede, por ejemplo, crear una máquina virtual en el proveedor de infraestructura -como una instancia EC2 dentro de AWS- aplicando metodologías de infraestructura como código.

Es agentless, y esto significa que no necesita instalar un agente para administrarlo. Utiliza SSH.

- Minimalismo por naturaleza : un sistema de gestión no deberia imponer dependencias adicionales al entorno.

- Consistencia : se pueden crear entornos consistentes.

- Seguridad : ansible no instala agentes en nodos, solo requiere que un nodo tenga instalado openssh y python.

- Confiabilidad : un playbook en ansible deberia ser idempotente para evitar efectos inesperados en los sistemas a gestionar.

- Minimo aprendizaje requerido : playbooks usan lenguaje facil y descriptivo

PARA EJECUTAR:

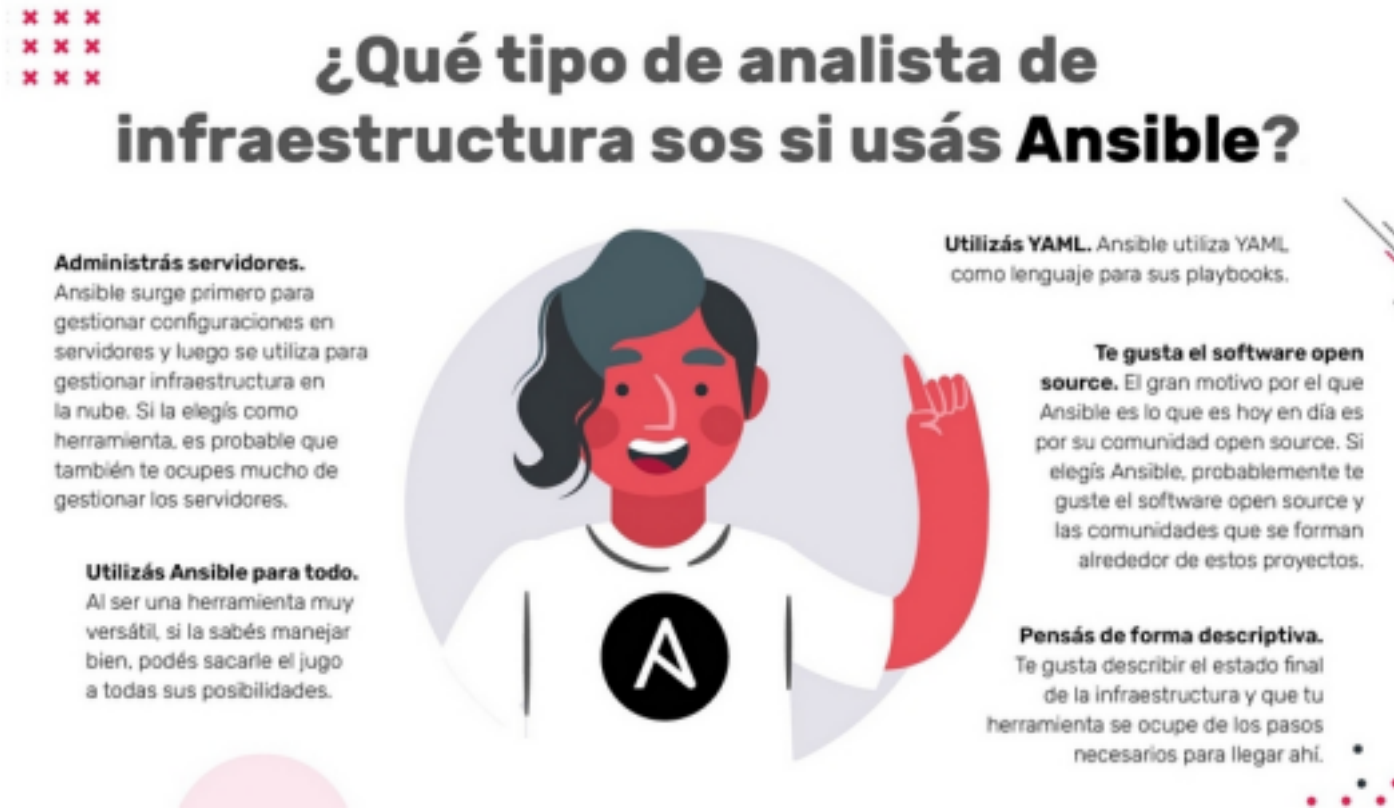
“ansible-playbook nombreArchivo.yml”

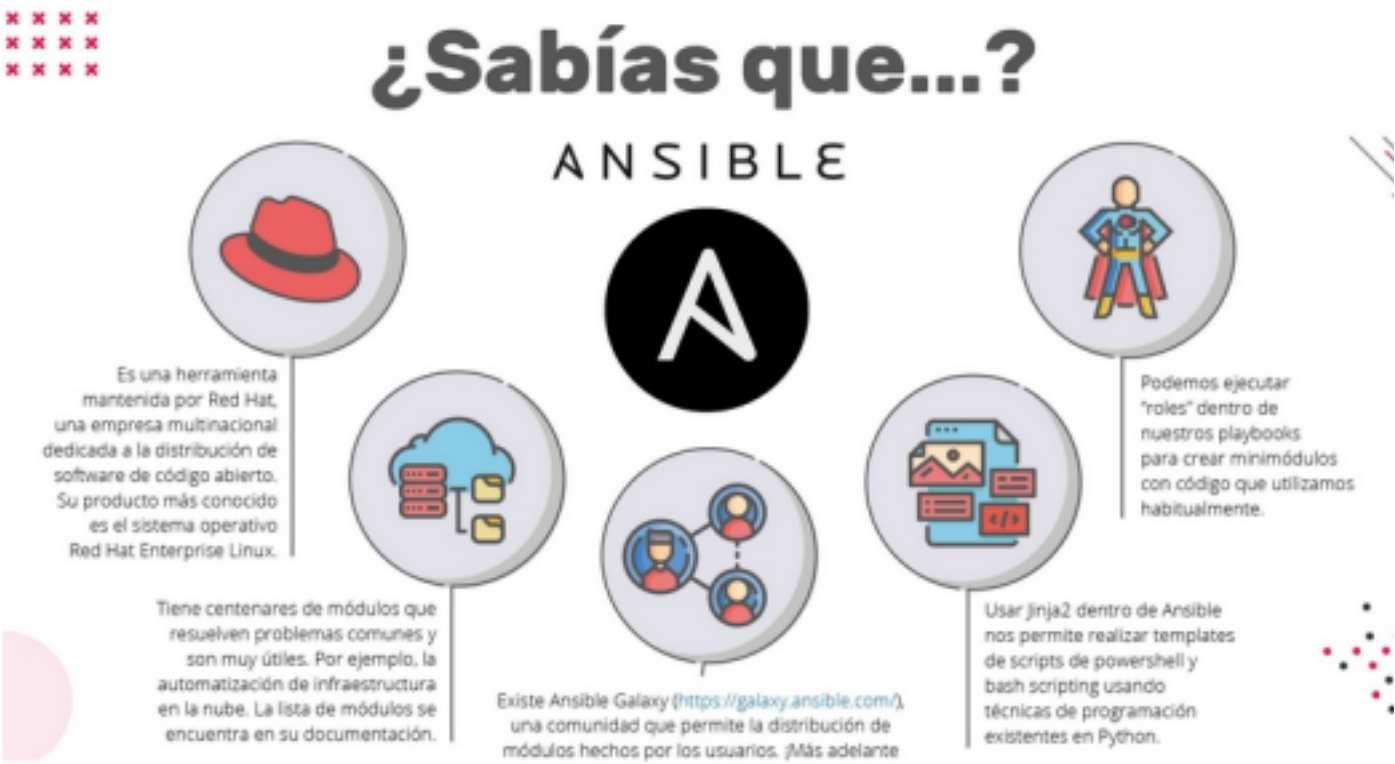
```
---
- hosts: localhost #en donde vamos a ejecutar el playbook
  tasks: #palabra reservada para indicar que vamos a ejecutar tareas

  - name: #aca indicamos lo que vamos a ejecutar (los mas comunes
    #para AWS son: aws_s3, ec2, ec2_ami, ec2_elb, ec2_tag, ecs_ecr)
    ec2:
      region: us-east-1 #declaramos la region en donde va a generarse nuestro ec2
      instance_type: t2.micro #la sacamos de aws
      image: ami-0c2b8ca1dad447f8a #ami-(unaSerieDeNumerosYLetras)  #aca va el ami
      que tambien obtenemos de aws
      wait: yes #es para esperar a que termine toda la ejecucion y luego nos de el
      okay
      wait_timeout: 500 #tiempo en segundos en que se espera si no se tiene
      respuesta de aws
      volumes: #se utiliza si se le asigna un disco a la instancia
        - device_name: /dev/xvda
          volume_type: gp2
          volume_size: 8
      vpc_subnet_id: subnet-08b7b906 #subnet-(serieDeNumYLetras) #parametro que
      debe estar creado previamente, se puede sacar de aws
      assign_public_id: yes #sirve para asignarle una ipPublica a la instancia
      key_name: prueba.pem #aca va el nombre del archivo que descargamos cuando
      generamos las claves
```

Una vez ejecutado, puedo ir a buscar a aws la ip, y a traves del comando “ssh -i nombreCarpetaDondeEstanLlaves/nombreArchivo.pem ubuntu@laIP” podemos conectarnos y acceder

Más módulos de Ansible

Hacé clic acá para ver la lista completa de módulos para este proveedor cloud.





INFRAESTRUCTURA COMO CÓDIGO: TERRAFORM

Terraform es una una herramienta de código abierto desarrollada por HashiCorp y la última de las tres que vas a conocer en las clases de Infraestructura como código.

Esta herramienta te permite definir y aprovisionar la infraestructura completa utilizando un lenguaje declarativo que podés ejecutar como infraestructura como código.

¡Un dato a tener en cuenta! Si bien es similar a herramientas como CloudFormation, hay una gran diferencia: no es solo para AWS, sino que podés utilizarla con el resto de los proveedores de infraestructura cloud.

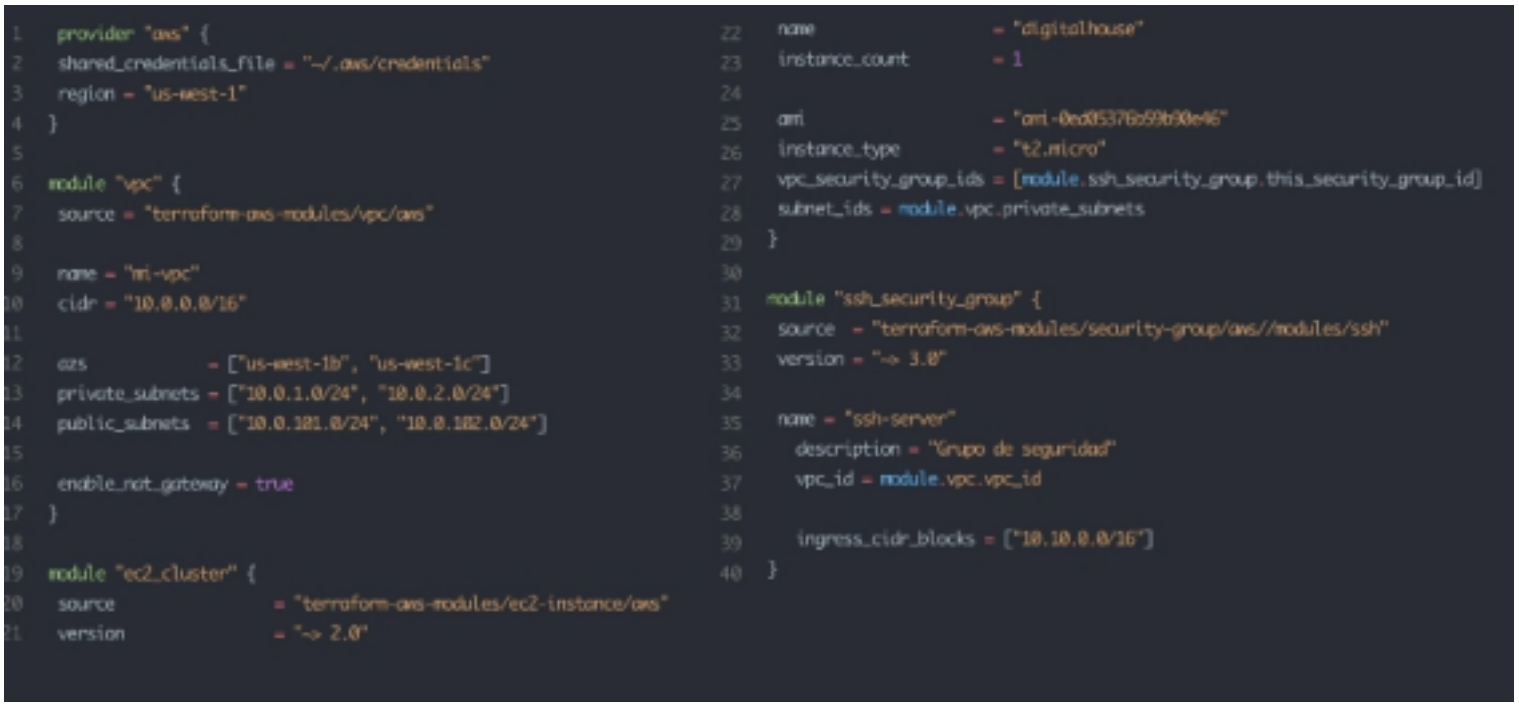
¡Otro dato más! Al ser declarativo te permite escribir tu código en el lenguaje de alto nivel HCL (HashiCorp Configuration Language) para describir el estado final que deseas de tu infraestructura.

Conozcamos juntos su historia, sus ventajas y por qué es la herramienta IaC más utilizada por los equipos de infraestructura.

Extension de los archivos => **.tf**

Es declarativo porque se utilizan modulos y solo nos enfocamos en el estado final de la infraestructura.

Ejemplo de código



```
provider "aws" {
 shared_credentials_file = "~/.aws/credentials"
 region = "us-west-1"
}

module "vpc" {
 source = "terraform-aws-modules/vpc/aws"

 name = "mi-vpc"
 cidr = "10.0.0.0/16"

 azs             = ["us-west-1b", "us-west-1c"]
 private_subnets = ["10.0.1.0/24", "10.0.2.0/24"]
 public_subnets  = ["10.0.101.0/24", "10.0.102.0/24"]

 enable_nat_gateway = true
}

module "ec2_cluster" {
 source                 = "terraform-aws-modules/ec2-instance/aws"
 version                = "~> 2.0"
 name                   = "digitalhouse"
 instance_count         = 1

 ami                    = "ami-0ed05376b59b90e46"
 instance_type          = "t2.micro"
 vpc_security_group_ids = [module.ssh_security_group.this_security_group_id]
 subnet_ids = module.vpc.private_subnets
}

module "ssh_security_group" {
 source  = "terraform-aws-modules/security-group/aws//modules/ssh"
 version = "~> 3.0"

 name = "ssh-server"
   description = "Grupo de seguridad"
   vpc_id = module.vpc.vpc_id

   ingress_cidr_blocks = ["10.10.0.0/16"]
}
```

Provider: indicamos que proovedor de infraestructura se va a utilizar, se señala donde estan las credentiales (las configuramos en la carpeta raiz) y la region

Module “vpc”: indicamos que modulo vamos a ejecutarlo. Source indica la direccion del modulo, en name el nombre que le queremos dar, en cidr las direcciones ip que va a utilizar. Podemos indicarle

entre dos zonas de disponibilidad y asignarles private subnets y public subnet respectivamente a cada una. Enable nat gateway en valor true significa que esta expuesto en internet nuestro servidor.

Module “ec2_cluster”: indicamos que modulo vamos a ejecutar. Source indica la direccion del modulo, la version del modulo tambien la indicamos, en name el nombre que le queremos dar, instance_count indica la cantidad de servidores q vamos a crear, se definde el ami que lo obtenemos de aws, al igual que el intance_type, y luego hacemos referencia al grupo de seguridad creado mas abajo y a la subnet private.

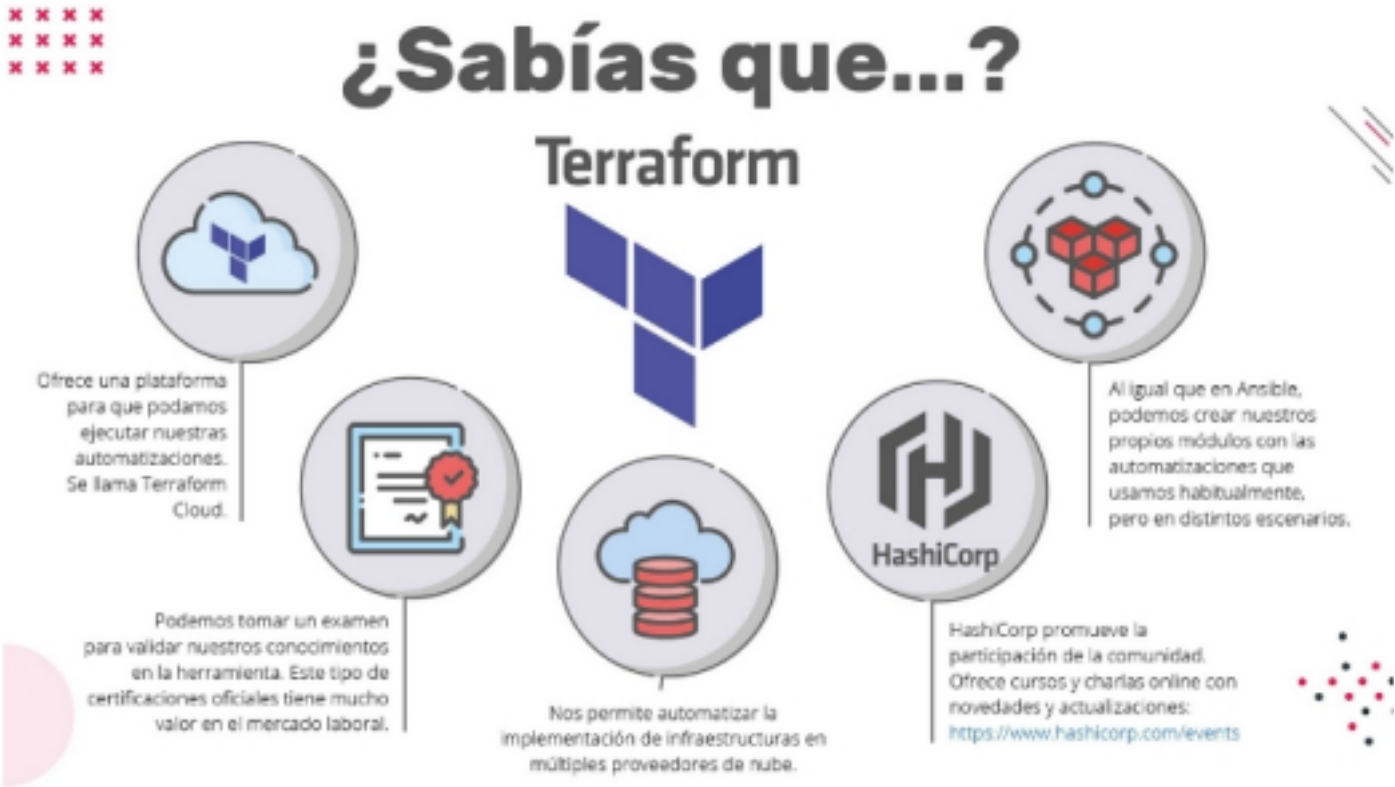
Module “ssh_security_group”: indicamos que modulo vamos a ejecutar. Source indica la direccion del modulo, la version del modulo tambien la indicamos, en name el nombre que le queremos dar, una breve descripcion, hacemos referencia a la vpc

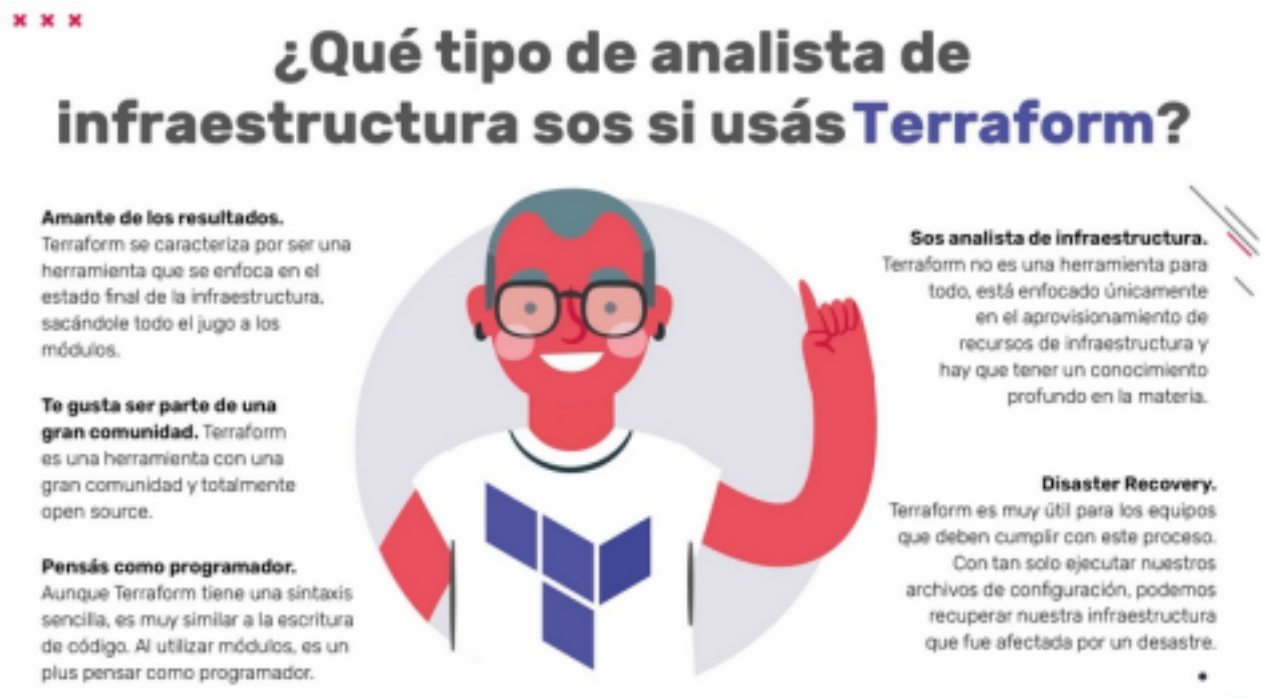
creada mas arriba y al final indicamos el rango de ips que va a utilizar el servidor, tiene que coincidir con el cidr de la vpc.

INICIAR TERRAFORM:

- actualizar credentials
- terraform init para baja provider y modulos
- terraform plan para ver lo que vamos a crearlas
- terraform apply para crear los recursos
- terraform destroy para destruir los recursos}

La documentación oficial de Terraform con todos los módulos para utilizar en AWS e imagines todo lo que podrías crear.





EJEMPLO DE CÓDIGO COMPLETO DE LA CLASE DE CIERRE (PARA VER SINTAXIS)

```
terraform {
      required_version = ">=0.12"
required_providers {
    aws = {
          source = "hashicorp/aws"
          version = "~> 3.20.0"
    }}}
provider "aws" {
    shared_credentials_file = "~/.aws/credentials"
    region = "us-east-1"
}
variable "aws_region_id" {
    description = "la region"
    type = string
    default = "us-east-1"
}
variable "main_vpc_cidr" {
    description = "Nuestro Security Group"
    type = string
    default = "10.0.0.0/24"
}
variable "public_subnets" {
    description = "subnet con acceso a internet"
    type = string
    default = "10.0.0.128/26"
}
variable "private_subnets" {
    description = "subnet sin acceso a internet"
    type = string
    default = "10.0.0.192/26"
}
variable "cidr_block" {
    description = "Private_RT-mesa6 variable"
    type = string
    default = "0.0.0.0/0"
}
# ------------------------------------------------------------------
```

Los outputs son lo que se vera por consola una vez finalizada la creación

Para los distintos recursos y parámetros, lo recomendable es mirar la documentación para no olvidarse de nada.