

Programa DevOps





3. Contenedores y Orquestación

En este manual veremos como los Contenedores revolucionan la escalabilidad, la performance y los tiempos hacia el mercado. El uso de Kubernetes, Docker y otras herramientas ha revolucionado la manera en la cual se despliega el software. Las réplicas automáticas y la configuración basada en código aceleran los ciclos de despliegue y mejoran la disponibilidad evitando a la vez errores humanos. En este viaje descubriremos las bondades de trabajar con contenedores.

¡Manos a la obra! 

Objetivos de aprendizaje

- Ser capaz de manejar contenedores e imágenes, comprender las arquitecturas de los diferentes sistemas de administración.
- Conocer los principios acerca de Orquestadores
- Administrar clústeres y escalabilidad de contenedores.



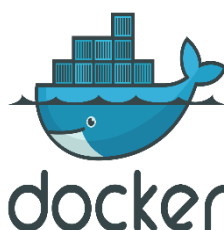
Índice

3. Contenedores y Orquestación.....	2
3.1 Docker.....	4
● 3.1.1. ¿Cómo funciona Docker?.....	4
● 3.1.2. Tecnología de Docker.....	5
● 3.1.3. ¿Por qué Docker?.....	6
● 3.1.4. Herramientas y Vocabulario de Docker.....	7
● 3.1.5. Comandos Básicos.....	9
● 3.1.6. Dockerfile.....	11
● 3.1.7. Instalación de Docker.....	12
3.2 Imágenes y Contenedores en Docker.....	16
● 3.2.1. Implementación y orquestación de Contenedores.....	16
● 3.2.2. Docker Swarn.....	17
● 3.2.3. Docker Compose.....	18
3.3 DockerFile en profundidad.....	20
● 3.3.1. Dockerfiles.....	20
3.4 Pipelines CI CD y Docker.....	30
● 3.4.1. Pasos en la fase de Build (ejemplo).....	30
● 3.4.2. Pasos en la fase de Deploy (ejemplo).....	31

3.5 Orquestación de Contenedores y Configuración.....	31
• 3.5.1. Puppet.....	31
• 3.5.2. Progress Chef.....	42
3.6 Kubernetes (K8s).....	50
• 3.6.1. Arquitectura de K8s.....	53
• 3.6.2. Componentes de K8s.....	56
• 3.6.3. Configuración de Contenedores.....	64
• 3.6.4. Políticas de Networking.....	68
• 3.6.5. Escalabilidad K8s.....	76
• 3.6.6. Actualización de Imágenes, Clusters y Nodos.....	79
• 3.6.7. Rollback de Contenedores.....	81
• 3.6.8. Almacenamiento Persistente.....	82
Resumen.....	87
Referencias.....	87

≡ 3.1 Docker

Es una plataforma de contenerización de código abierto. Permite a los desarrolladores empaquetar aplicaciones en contenedores: componentes ejecutables estandarizados que combinan el código fuente de la aplicación con las bibliotecas del sistema operativo (SO) y las dependencias necesarias para ejecutar dicho código en cualquier entorno. Los contenedores simplifican la entrega de aplicaciones distribuidas y se han vuelto cada vez más populares a medida que las organizaciones cambian al desarrollo nativo de la nube y entornos híbridos multi nube.



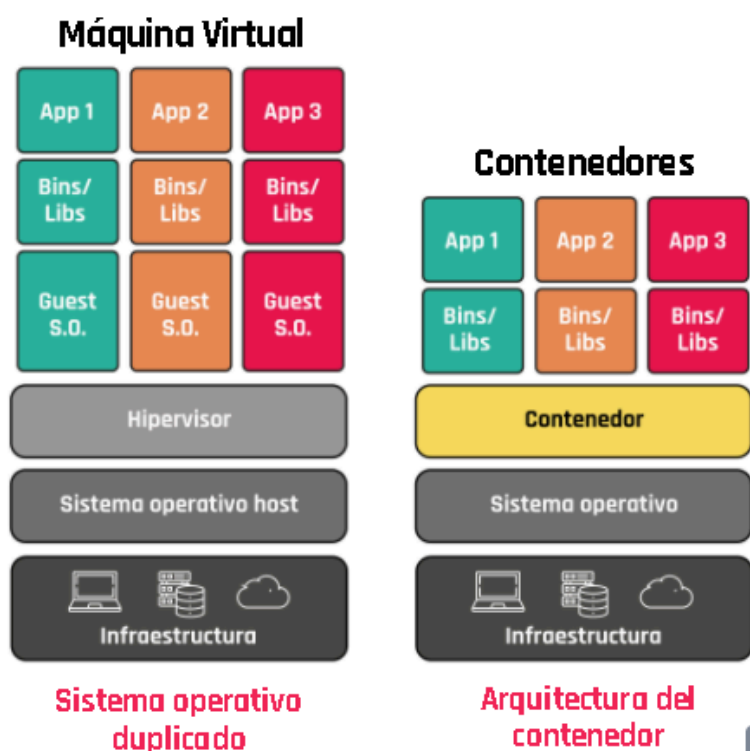
Los desarrolladores pueden crear contenedores sin Docker, pero la plataforma permite crear, implementar y gestionar contenedores de forma más fácil, sencilla y segura. Docker es esencialmente un kit de herramientas que permite a los desarrolladores crear, implementar, ejecutar, actualizar y detener contenedores utilizando comandos simples y automatización que ahorra trabajo a través de una única API.

• 3.1.1. ¿Cómo funciona Docker?

Docker proporciona una manera estándar de ejecutar código. Docker es un sistema operativo para contenedores. De manera similar a cómo una máquina virtual virtualiza (elimina la necesidad de administrar directamente) el hardware del servidor, los contenedores virtualizan el sistema operativo de un servidor. Docker se instala en cada servidor y proporciona comandos sencillos que puede utilizar para crear, iniciar o detener contenedores.

Un paso evolutivo importante para la distribución de aplicaciones ha sido la tecnología de contenerización, que permite distribuir aplicaciones de modo ágil sin tener que compartir una máquina virtual completa (es decir todo el disco, sistema operativo, configuraciones de red y aplicaciones).

Máquina Virtual vs. Contenedores



• 3.1.2. Tecnología de Docker

La tecnología Docker usa el kernel de Linux y las funciones de éste, como grupos de control (**Cgroups**) y espacios de nombres (**Namespaces**), para segregar los procesos, de modo que puedan ejecutarse de manera independiente. El propósito de los contenedores es esta independencia: la capacidad de ejecutar varios procesos y aplicaciones por separado para hacer un mejor uso de su infraestructura y, al mismo tiempo, conservar la seguridad que se tendría con sistemas separados.

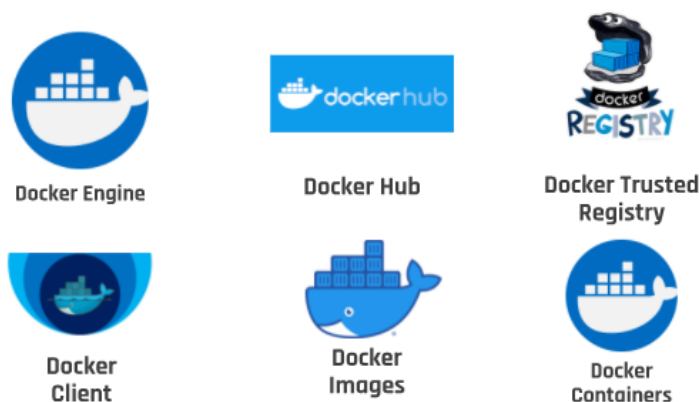
Las herramientas del contenedor, como Docker, ofrecen un modelo de implementación basado en imágenes. Esto permite compartir una aplicación, o un conjunto de servicios, con todas sus dependencias en varios entornos. Docker también automatiza la implementación de la aplicación (o conjuntos combinados de procesos que constituyen una aplicación) en este entorno de contenedores.

Estas herramientas desarrolladas a partir de los contenedores de Linux, lo que hace a Docker fácil de usar y único, otorgan a los usuarios un acceso sin precedentes a las aplicaciones, la capacidad de implementar rápidamente y control sobre las versiones y su distribución.

¿La tecnología Docker es la misma que la de los contenedores de Linux tradicionales?

No. Al principio, la tecnología Docker se desarrolló a partir de la tecnología LXC, lo que la mayoría de las personas asocia con contenedores de Linux "tradicionales", aunque desde entonces se ha alejado de esa dependencia. LXC era útil como virtualización ligera, pero no ofrecía una buena experiencia al desarrollador ni al usuario. La tecnología Docker no solo aporta la capacidad de ejecutar contenedores; también facilita el proceso de creación y diseño de contenedores, de envío de imágenes y de creación de versionado de imágenes (entre otras cosas).

Plataforma de Docker



● 3.1.3. ¿Por qué Docker?

Docker es tan popular hoy que "Docker" y "contenedores" se utilizan indistintamente. Pero las primeras tecnologías relacionadas con los contenedores estuvieron disponibles durante años, antes de que Docker fuera lanzado al público en 2013.

Sobre todo, en 2008, Linux Containers (LXC) se implementó en el kernel de Linux, lo que habilitó completamente la virtualización para una sola instancia de Linux. Mientras que LXC todavía se utiliza hoy, las nuevas tecnologías que utilizan el kernel de Linux están disponibles. Ubuntu, un sistema operativo Linux moderno y de código abierto, también proporciona esta capacidad.

Docker ha mejorado las capacidades nativas de contenerización de Linux con tecnologías que permiten:

- **Facilidad de uso:** Docker es fácil de usar y no requiere conocimientos profundos de virtualización o sistemas operativos.
- **Portabilidad mejorada y continua:** mientras que los contenedores LXC suelen hacer referencia a configuraciones específicas de la máquina, los contenedores Docker se ejecutan sin modificaciones en cualquier entorno de desktop, centro de datos y nube, se pueden ejecutar en cualquier plataforma que tenga Docker Engine instalado.
- **Peso aún más ligero y actualizaciones más granulares:** Con LXC, se pueden combinar varios procesos dentro de un único contenedor. Con los contenedores Docker, solo se puede ejecutar un proceso en cada contenedor. Esto permite crear una aplicación que puede continuar ejecutándose mientras una de sus partes se desactiva para realizar una actualización o reparación.
- **Creación automática de contenedores:** Docker puede crear automáticamente un contenedor basado en el código de origen de la aplicación.
- **Automatización:** Docker permite automatizar todo el proceso de creación, implementación y ejecución de aplicaciones, lo que reduce el tiempo y los costos de desarrollo e implementación.
- **Control de versiones de contenedor:** Docker puede rastrear versiones de una imagen de contenedor, retrotraer a versiones anteriores y rastrear quién creó una versión y cómo. Puede incluso cargar sólo los deltas entre una versión existente y una nueva.
- **Reutilización de contenedores:** los contenedores existentes se pueden utilizar como imágenes base, esencialmente como plantillas para crear nuevos contenedores.
- **Bibliotecas de contenedores compartidos:** los desarrolladores pueden acceder a un registro de código abierto que contiene miles de contenedores aportados por los usuarios.

Hoy en día, la containerización de Docker también funciona con el servidor Microsoft Windows. Y la mayoría de los proveedores de nube ofrecen servicios específicos para ayudar a los desarrolladores a crear, enviar y ejecutar aplicaciones en contenedores con Docker.

● 3.1.4. Herramientas y Vocabulario de Docker

Vocabulario

- **Host:** una máquina virtual que ejecuta Docker daemon para alojar una colección de contenedores Docker.
- **Cliente:** aquí se ejecutan los comandos que están siendo ejecutados. (cliente-servidor).
- **Imagen:** una colección ordenada de sistemas de archivos (capas) que se utilizarán al crear una instancia de un contenedor.
- **Contenedor:** una instancia en tiempo de ejecución de una imagen.
- **Registry:** una colección de imágenes de Docker.

Arquitectura y Herramientas de Docker

👉 **Docker Engine**, también conocido como **Docker Daemon**, es el programa que permite construir, enviar y ejecutar contenedores. Utiliza espacios de nombres y grupos de control del kernel de Linux para proporcionar un entorno de tiempo de ejecución aislado para cada aplicación.

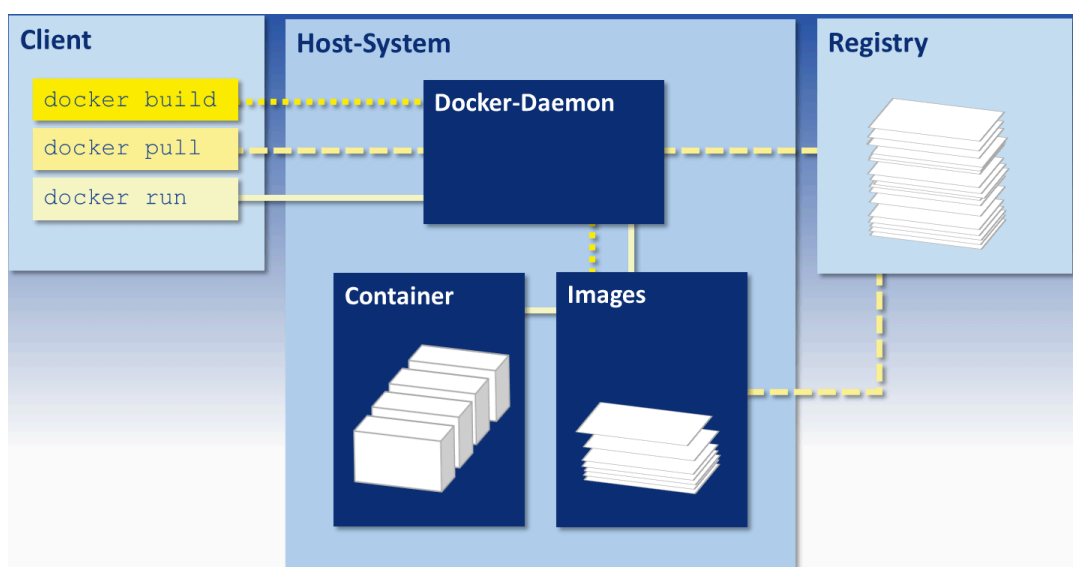
👉 **Docker Hub**, es un registro en línea de imágenes de Docker.

👉 **Docker Trusted Registry** es un registro privado en el sitio para imágenes de Docker.

👉 **Docker Client** es el que toma las entradas del usuario y las envía al daemon. El cliente y el daemon pueden ejecutarse en el mismo host o en diferentes hosts.

👉 **Docker Images** es una plantilla de solo lectura utilizada para crear contenedores. Contiene un conjunto de instrucciones para crear los contenedores.

👉 Por último, **Docker Containers** es una plataforma de aplicación aislada basada en una o más imágenes que contiene todo lo necesario para ejecutar una aplicación. En la siguiente imagen se puede ver la forma en que los componentes individuales de Docker se combinan tomando como ejemplo los comandos **docker build**, **docker pull** y **docker run**:



La arquitectura Docker se basa en la combinación de cliente (terminal), servidor (Docker daemon) y registro (Docker Hub)

Aunque en un principio Docker solo se usaba en las distribuciones de Linux, la versión actual del motor de contenedores se caracteriza por una gran independencia de plataforma. Hoy existen paquetes de instalación para Microsoft Windows y macOS, así como para servicios en la nube como Amazon Web Services (AWS), Microsoft Azure y Google Cloud Platform, entre otros proveedores.

● 3.1.5. Comandos Básicos

Para enumerar los comandos disponibles, ejecute Docker sin parámetros o ejecute:

docker help

Nombre	Descripción
attach	Adjuntar la entrada, salida y errores estándar locales a un contenedor en ejecución
build	Construir una imagen a partir de un Dockerfile
builder	Administrar compilaciones
checkpoint	Administrar checkpoints
commit	Crear una nueva imagen a partir de los cambios de un contenedor
config	Administrar configuraciones de Swarm
container	Administrar contenedores
context	Administrar contextos
cp	Copiar archivos/carpetas entre un contenedor y el sistema de archivos local
create	Crear un nuevo contenedor

Nombre	Descripción
diff	Inspeccionar cambios en archivos o directorios en el sistema de archivos de un contenedor
events	Obtener eventos en tiempo real del servidor
exec	Ejecutar un comando en un contenedor en ejecución
export	Exportar el sistema de archivos de un contenedor como un archivo tar
history	Mostrar el historial de una imagen
image	Administrar imágenes
images	Listar imágenes
import	Importar el contenido de un archivo tar para crear una imagen del sistema de archivos
info	Mostrar información a nivel de sistema
inspect	Obtener información de bajo nivel sobre objetos Docker
kill	Detener uno o varios contenedores en ejecución
load	Cargar una imagen desde un archivo tar o STDIN
login	Iniciar sesión en un registro
logout	Cerrar sesión en un registro
logs	Obtener los registros de un contenedor
manifest	Administrar manifiestos e listas de manifiestos de imágenes Docker
network	Administrar redes
node	Administrar nodos Swarm
pause	Pausar todos los procesos dentro de uno o varios contenedores
plugin	Administrar plugins
port	Listar mapeos de puertos o un mapeo específico para el contenedor
ps	Listar contenedores
pull	Descargar una imagen desde un registro
push	Subir una imagen a un registro
rename	Renombrar un contenedor
restart	Reiniciar uno o varios contenedores
rm	Eliminar uno o varios contenedores
rmi	Eliminar una o varias imágenes
run	Crear y ejecutar un nuevo contenedor a partir de una imagen
save	Guardar una o varias imágenes en un archivo tar (transmitido a STDOUT por defecto)
search	Buscar en Docker Hub imágenes
secret	Administrar secretos Swarm
service	Administrar servicios Swarm
stack	Administrar stacks Swarm

Nombre	Descripción
start	Iniciar uno o varios contenedores detenidos
stats	Mostrar una transmisión en vivo de estadísticas de uso de recursos de uno o varios contenedores
stop	Detener uno o varios contenedores en ejecución
swarm	Administrar Swarm
system	Administrar Docker
tag	Crear una etiqueta TARGET_IMAGE que hace referencia a SOURCE_IMAGE
top	Mostrar los procesos en ejecución de un contenedor
trust	Administrar la confianza en imágenes Docker
unpause	Reanudar todos los procesos dentro de uno o varios contenedores
update	Actualizar la configuración de uno o varios contenedores
version	Mostrar información sobre la versión de Docker
volume	Administrar volúmenes
wait	Bloquear hasta que uno o varios contenedores se detengan y luego mostrar sus códigos de salida

● 3.1.6. Dockerfile

Contiene todas las instrucciones necesarias para crear la imagen de una aplicación. Es un archivo de texto que contiene comandos para combinar imágenes.

Puede usar cualquier comando para llamar en la línea de comandos. Docker genera automáticamente una imagen al leer las instrucciones en el Dockerfile.

El comando **docker build** se usa para crear una imagen a partir de un Dockerfile. Puede usar la marca -f en el comando docker build para apuntar a un Dockerfile en cualquier parte del sistema de archivos.

Ejemplo:

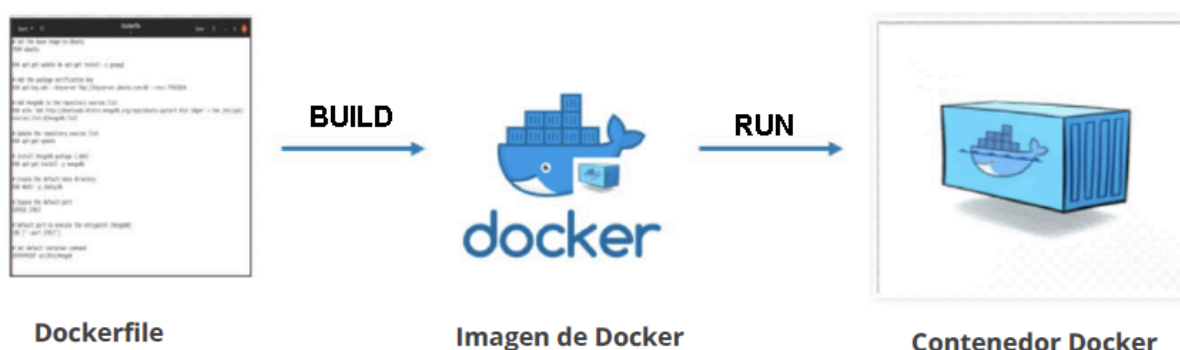
```
docker build -f /path/to/a/Dockerfile
```

Estructura básica de Dockerfile

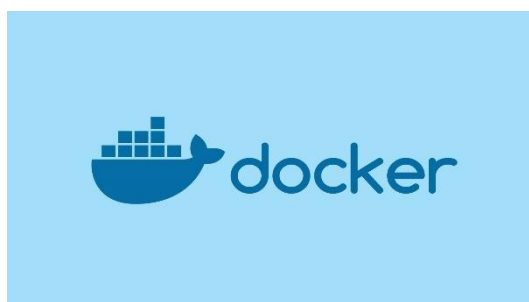
Dockerfile generalmente se divide en cuatro partes: información básica de la imagen, información del mantenedor, instrucciones de operación de la imagen e instrucciones para ejecutar cuando se inicia el contenedor. '#' es un comentario en el Dockerfile.

Descripción del archivo Dockerfile

Docker ejecuta las instrucciones del Dockerfile en orden de arriba hacia abajo. Para especificar la imagen base, la primera instrucción debe ser FROM. Una declaración que comience con el carácter # se considera un comentario. Puede usar RUN, CMD, FROM, EXPOSE, ENV y otras instrucciones en los archivos Docker.



• 3.1.7. Instalación de Docker



Instalación en Windows de Docker Desktop

Requerimientos previos

Se requiere para poder instalar el Subsistema de Windows para Linux, y actualizar a WSL 2, además del siguiente hardware y software:

- Microprocesador de 64-bit con Second Level Address Translation (SLAT)
- Memoria RAM de 4GB ó superior.
- Soporte de virtualización a nivel de BIOS habilitado.
- Windows 11 64-bit: Versiones Home, Profesional, Educativa ó Enterprise la actualización 21H2 ó una superior,
- Windows 10 64-bit: Versiones Home, Profesional, Educativa ó Enterprise la actualización 21H1 ó una superior.

a. Abrir PowerShell como administrador y ejecutar:

dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart

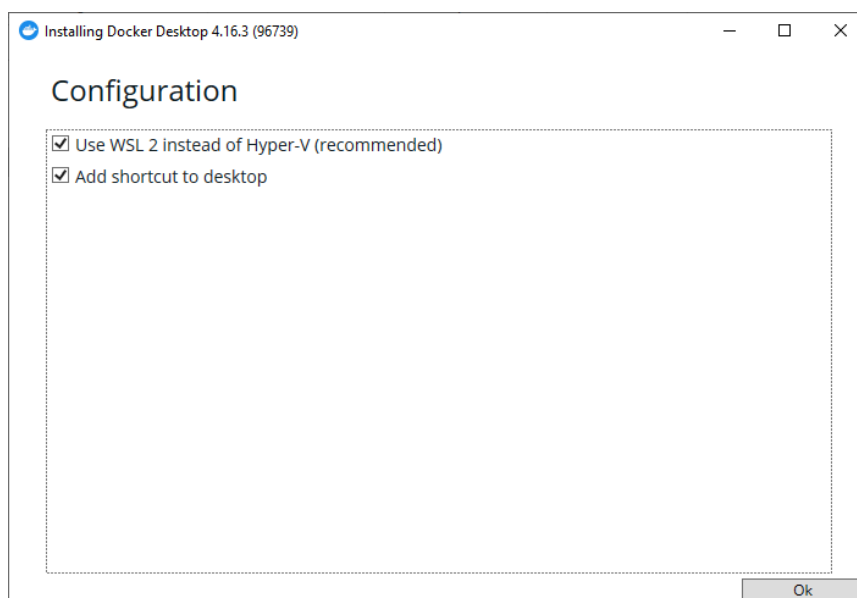
b. Reiniciar la máquina para completar la instalación de WSL y la actualización a WSL

- Descargar e instalar el Subsistema de Windows para Linux.
<https://docs.microsoft.com/windows/wsl/wsl2-kernel>

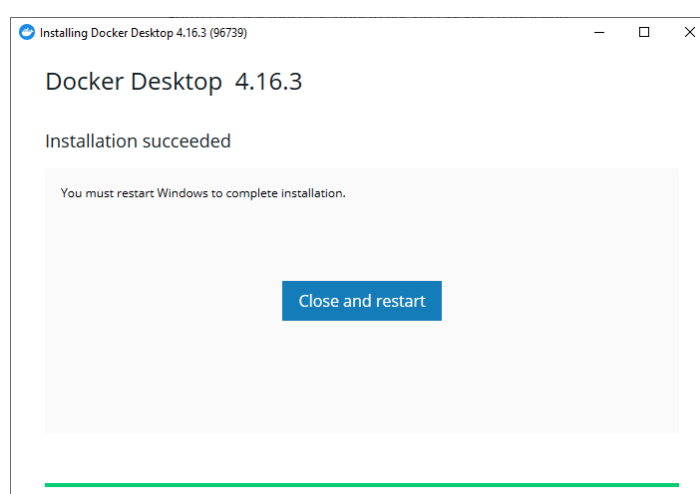
Instalación de Docker

Los pasos para instalar Docker en un computador con Sistema Operativo Windows son los siguientes:

1. Descargar el instalador desde el sitio oficial de Docker
<https://docs.docker.com/desktop/install/windows-install/>
2. Hacer doble clic en Docker Desktop Installer.exe para ejecutar el instalador.
3. Cuando se le solicite, asegurarse de que la opción **Usar WSL 2** en lugar de **Hyper-V**, en la página Configuración, esté seleccionada o no, según su elección de backend.

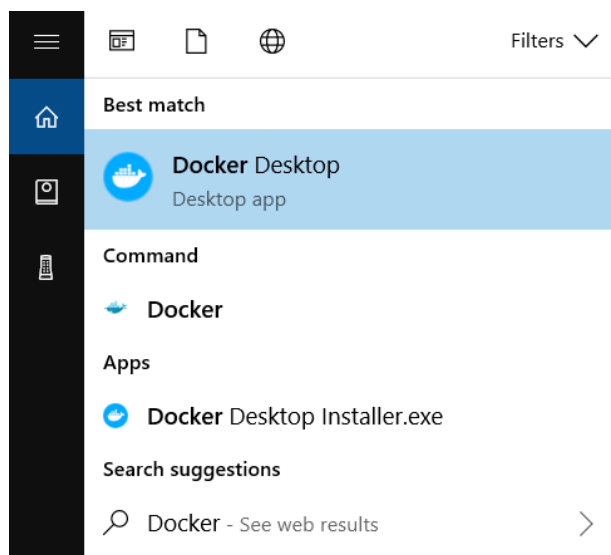


4. Si el sistema solo admite una de las dos opciones, no podrá seleccionar qué backend usar.
5. Seguir las instrucciones del asistente de instalación para autorizar al instalador y continuar con la instalación.
6. Cuando la instalación sea exitosa, deberás hacer clic en Cerrar para completar el proceso de instalación.
7. Si la cuenta de administrador es diferente a la cuenta de usuario, se debe agregar el usuario al grupo de usuarios de Docker. Ejecutar Administración de equipos como administrador y dirigirse a Usuarios y grupos locales > Grupos > usuarios de Docker. Hacer clic derecho para agregar el usuario al grupo. Cerrar sesión y reiniciar el computador para que los cambios surtan efecto.



Iniciar Docker Desktop

Buscar en la barra de búsqueda Docker y seleccionar Docker Desktop en la lista de resultados.



Instalación en Linux Ubuntu de Docker

Para efectuar la instalación en Linux por favor siga los pasos en la documentación oficial en:

<https://docs.docker.com/engine/install/ubuntu/>

Quick Learning Docker

Este apartado tiene el objetivo de ayudarte a seguir potenciando tus habilidades, por lo que a continuación encontrarás unos enlaces que podrás utilizar para aprender más rápidamente y a tu ritmo.

¡Manos a la obra! 🤖

Instalando y Usando Docker (Youtube)

<https://www.youtube.com/watch?v=6idFknRIOp4>

Duración: 14 minutos

≡ 3.2 Imágenes y Contenedores en Docker

● 3.2.1. Implementación y orquestación de Contenedores

Si se está ejecutando sólo unos pocos contenedores, como se ha visto previamente es bastante sencillo gestionar la aplicación dentro de Docker Engine, el runtime de facto de la industria. Si la implementación está formada por cientos o miles de contenedores y cientos de servicios, es casi imposible gestionar ese flujo de trabajo sin la ayuda de herramientas creadas específicamente, como los orquestadores de contenedores.

La orquestación de contenedores consiste en la automatización de la mayoría de las operaciones necesarias para ejecutar cargas de trabajo y servicios en contenedores. En sistemas a gran escala, las aplicaciones contenerizadas se vuelven difíciles de gestionar manualmente porque suelen incluir cientos e incluso miles de contenedores. Así que, la orquestación de contenedores es esencial para reducir la complejidad operacional a la hora de ejecutarlos.

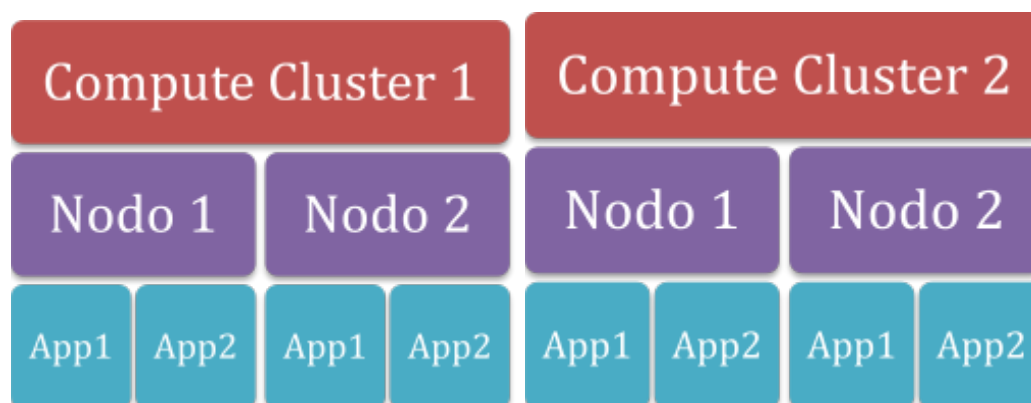
Ventajas de la orquestación de contenedores

En resumen, la orquestación de contenedores ofrece diversos beneficios:

- Reduce la complejidad operacional a la hora de gestionar contenedores.
- Mejora la seguridad al reducir las posibilidades de cometer errores humanos, gracias a la automatización.
- Permite escalar y reiniciar contenedores y clústeres automáticamente
- Ayuda a los equipos de IT a automatizar parte del trabajo y a aprovechar todos los beneficios de usar contenedores.

¿Qué es un Cluster?

Un clúster en la nube es una solución de computación en la nube que utiliza una arquitectura de clúster para ofrecer servicios de computación de alta disponibilidad. La arquitectura de clúster se compone de un conjunto de servidores interconectados que trabajan juntos para ofrecer un servicio de alta disponibilidad. Si un servidor se cae, otro servidor puede tomar el control y ofrecer el servicio.



• 3.2.2. Docker Swarn

Swarm es un software creado por los programadores de Docker -Hashicorp-, que permite agrupar una serie de hosts de Docker en un clúster y gestionar los clústeres de manera centralizada, así como también orquestar los contenedores.



Hasta la versión 1.11 de Docker había que implementar Swarm como una herramienta aparte, las versiones posteriores de la plataforma de contenedores soportan un modo nativo **swarm** (modo enjambre), por lo que cuando los usuarios instalan el Docker engine, también tienen a su disposición este gestor de clústeres.

Docker Swarm se basa en una arquitectura maestro-esclavo (master-slave). Cada clúster de Docker está formado al menos por un nodo maestro (también llamado administrador o manager) y tantos nodos esclavos (llamados de trabajo o workers) como sea necesario.

Mientras que el maestro de Swarm es responsable de la gestión del clúster y la delegación de tareas, el esclavo se encarga de ejecutar las unidades de trabajo (tasks o tareas). Además, las aplicaciones de contenedores se distribuyen en servicios en las cuentas de Docker elegidas.

En la jerga de Docker, “service” o servicio designa una estructura abstracta con la que se pueden definir aquellas tareas que deben ejecutarse en el clúster. Cada servicio está formado por un conjunto de tareas individuales que se procesan en contenedores independientes en uno de los nodos del clúster. Cuando se crea un servicio, el usuario determina la imagen del contenedor en la que se basa y los comandos que se ejecutan en el contenedor, operando sobre la base de la imagen.

Docker Swarm soporta dos modos de definir servicios swarm: servicios globales o replicados.

- **Servicios replicados**

Se trata de tareas que se ejecutan en un número de réplicas definido por el usuario. A su vez, cada réplica es una instancia del contenedor definido en el servicio. Los servicios replicados se pueden escalar, permitiendo a los usuarios crear réplicas adicionales. Si así se requiere, un servidor web como NGINX se puede escalar en 2, 4 o 100 instancias con una sola línea de comandos.

- **Servicios globales**

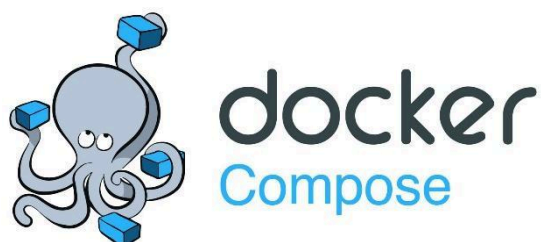
Si un servicio se ejecuta en modo global, cada nodo disponible en el clúster inicia una tarea para el servicio correspondiente. Si al clúster se le añade un nodo nuevo, el maestro de Swarm le atribuye una tarea para el servicio global de forma inmediata. Este tipo de servicios se recomienda para las aplicaciones de monitoreo o los programas antivirus.

Uno de los campos en los que se puede aplicar Docker Swarm es el reparto de cargas, pues con el modo enjambre Docker dispone de funciones integradas de balanceo de carga. Si se ejecuta, por ejemplo, un servidor web NGINX con cuatro instancias, Docker distribuye las consultas entrantes de forma inteligente entre las instancias del servidor web disponibles.

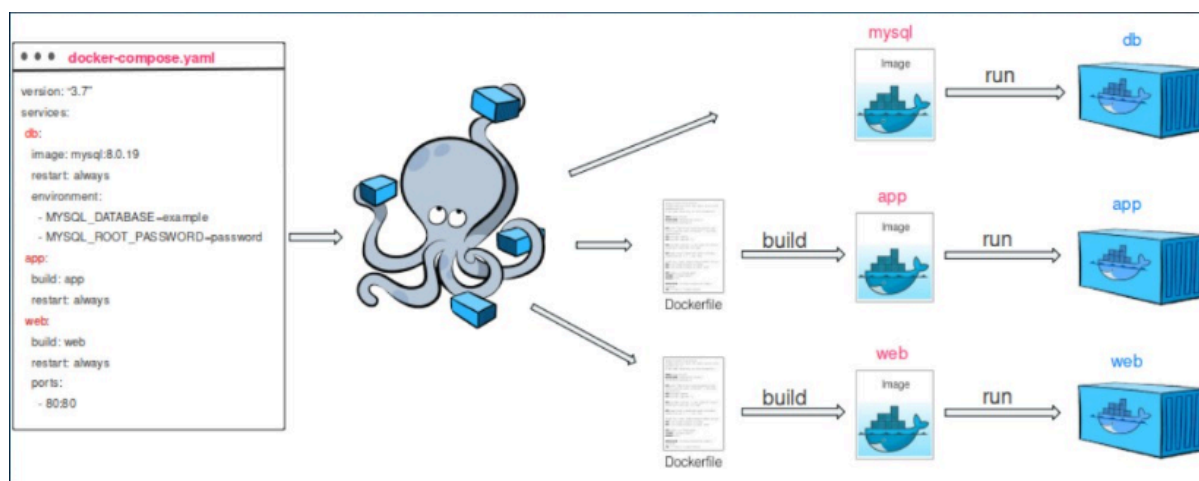
● 3.2.3. Docker Compose

Si estás creando una aplicación fuera de procesos en varios contenedores que residen en el mismo host, puedes utilizar Docker Compose para gestionar la arquitectura de la aplicación. Docker Compose crea un archivo YAML que especifica qué servicios se incluyen en la aplicación y puede implementar y ejecutar contenedores con un único comando.

Mediante Docker Compose, también puede definir volúmenes persistentes para el almacenamiento, especificar nodos base y documentar y configurar dependencias de servicio.



Docker Compose permite definir las aplicaciones multi-contenedor, conocidas como stacks, y ejecutarlas en un nodo individual o en un clúster. La herramienta recurre para ello a una consola de línea de comandos con el fin de gestionar el ciclo de vida completo de sus aplicaciones.



En el universo Docker los stacks o lotes son grupos de servicios interconectados que comparten las dependencias del software y que se escalan y orquestan de forma conjunta. Un stack de Docker permite definir las diversas funcionalidades de una aplicación en un archivo central, este es, docker-compose.yml, iniciarlo, ejecutarlo conjuntamente en un sistema de tiempo de ejecución y gestionarlo de forma central.

Dependiendo del sistema operativo en el que se ejecute Docker es posible que haya que instalar Compose por separado.

- Linux
- macOS
- iOS

Nos vamos a enfocar en los sistemas operativos que, como un ingeniero que utiliza DevSecOps podría llegar a interactuar en entornos corporativos. Los más comunes son solo dos, con sus respectivas distribuciones y versiones que pueden ser variadas. Cabe destacar que esto es solo una apreciación basada en cantidad de Sistemas Operativos instalados en ambientes corporativos. Esto no indica que no existan casos particulares de empresas que utilicen macOS, Android u otros sistemas operativos.

Quick Learning Compose y Swarm

Este apartado tiene el objetivo de ayudarte a seguir potenciando tus habilidades, por lo que a continuación encontrarás unos enlaces que podrás utilizar para aprender más rápidamente y a tu ritmo.

¡Manos a la obra! 🤖

Docker Compose (Youtube)

<https://www.youtube.com/watch?v=epcw4kiRCKw>

Duración: 15 minutos

Docker Swarn y su pronta desaparición (Youtube)

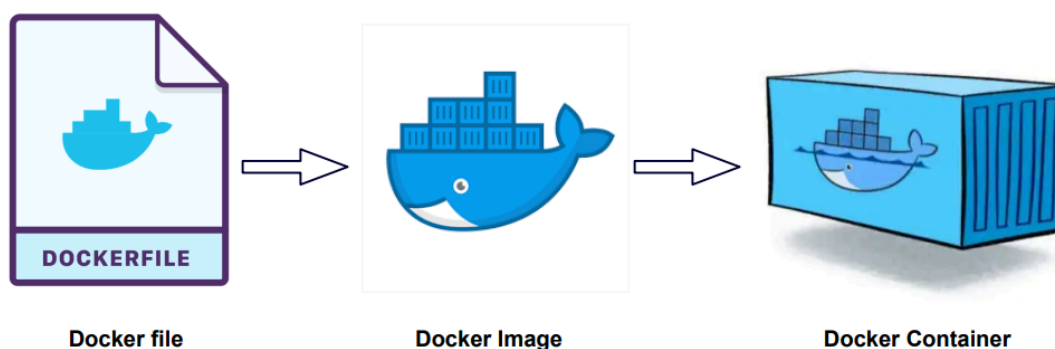
<https://www.youtube.com/watch?v=ePXwicznTCvQ>

Duración: 7 minutos

≡ 3.3 DockerFile en profundidad

● 3.3.1. Dockerfiles

El Dockerfile es la **unidad fundamental del ecosistema Docker**. Describe los pasos para crear una imagen de Docker. El flujo de información sigue este esquema central:



Un contenedor Docker tiene una vida limitada e interactúa con su entorno. Imagina que **contenedor es como un organismo vivo**. Piensa en un organismo unicelular, como una célula de levadura. Siguiendo esta analogía, una imagen Docker equivale, digamos, a la información genética: todos los contenedores creados a partir de una imagen son iguales, como todos los organismos unicelulares clonados a partir de una unidad de información genética.

Entonces, ¿cómo entra el Dockerfile en este esquema?

El Dockerfile define los pasos a seguir para **crear una nueva imagen**. Hay que entender que se empieza siempre con una imagen base existente. La nueva imagen nace de la imagen base. Además, hay ciertos cambios puntuales. En nuestro ejemplo de la célula de levadura, los cambios serían mutaciones. Un Dockerfile es determinante en dos aspectos de la nueva imagen de Docker:

- La imagen base de la que procede la nueva imagen. Esto ancla la nueva imagen al árbol genealógico del ecosistema Docker.
- Un conjunto de cambios específicos que diferencian la nueva imagen de la imagen base.

Funcionamiento

En el fondo, el Dockerfile es un archivo de texto totalmente normal. El Dockerfile contiene un conjunto de instrucciones, cada una en una línea distinta. Para crear una Docker Image, las instrucciones se ejecutan una tras otra. Quizás te suene este esquema de la ejecución de un script por lotes. Durante la ejecución, se añaden paso por paso más capas a la imagen. Te explicamos cómo funciona exactamente en nuestro artículo temático sobre Docker Image.

Una imagen Docker se crea ejecutando las instrucciones de un Dockerfile. Este paso se conoce como el proceso build y empieza con la ejecución del comando “docker build”. El contexto de construcción es un concepto crucial: define a qué archivos y directorios tiene acceso el proceso de construcción, donde un directorio local hace las veces de fuente. El contenido del directorio fuente se transfiere al Docker Daemon al accionar “docker build”. Las instrucciones contenidas en el Dockerfile reciben acceso a los archivos y directorios contenidos en el contexto de construcción.

A veces no queremos iniciar todos los archivos del directorio fuente local en el contexto build. Para estos casos existe el archivo “.dockerignore”, que sirve para excluir archivos y directorios del contexto de construcción y cuyo nombre se basa en el archivo “.gitignore” de Git. El punto antes del nombre del archivo indica que se trata de un archivo oculto.

Construcción

Un Dockerfile es un **archivo de texto simple que lleva el nombre de archivo “Dockerfile”**. Ten en cuenta que es obligatorio que la primera letra sea mayúscula. El archivo contiene una entrada por fila. Te mostramos a continuación cómo se construye generalmente un Dockerfile:

```
# Comentario
INSTRUCCIONES Argumentos
```

Además de los comentarios, el Dockerfile contiene **instrucciones y argumentos**, que describen la construcción de las imágenes.

Comentarios e instrucciones del analizador

Los **comentarios** contienen información pensada principalmente para las personas. Conocido por los lenguajes de programación Python o Perl, los comentarios en un Dockerfile empiezan con una **almohadilla (#)**.

Durante el proceso de construcción, las filas de comentarios se eliminan antes del procesamiento, por lo que es importante tener en cuenta que solo se consideran como filas de comentarios las que empiezan con la almohadilla.

Este sería un comentario válido:

```
# nuestra Base Image
FROM busybox
```

En cambio, aquí se generaría un fallo, ya que la almohadilla no está al principio de la línea:

```
FROM busybox # nuestra Base Image
```

Como variante de los comentarios, están las **instrucciones del analizador**. Van dentro de las líneas de comentario y deben aparecer siempre al principio del Dockerfile. Si no, se tratarán como comentarios y se eliminarán durante el build. También hay que tener en cuenta que **Dockerfile solo puede utilizar una vez** una determinada instrucción del analizador.

En el momento de escribir este manual, solo existen dos tipos de instrucciones de análisis: “syntax” y “escape”. La instrucción de análisis “escape” define el símbolo de escape que debe utilizarse. Se utiliza para escribir instrucciones en varias filas y para expresar caracteres especiales. La instrucción de análisis “syntax” define las reglas según las cuales el analizador tiene que procesar las instrucciones del Dockerfile. He aquí un ejemplo:

```
# syntax=docker/dockerfile:1
# escape=\
```

Instrucciones, argumentos y variables

Las instrucciones conforman la parte principal del contenido del Dockerfile. Ejecutadas una tras otra, **las instrucciones describen la estructura específica de una imagen Docker**.

Al igual que los comandos de la línea de comandos, las instrucciones tienen argumentos. Algunas son directamente comparables a comandos específicos de la línea de comandos. De esta manera, existe una instrucción COPY, que copia archivos y directorios y corresponde aproximadamente al comando “cp” de la fila de comandos. A diferencia de la línea de comandos, para algunas instrucciones del Dockerfile hay reglas específicas para su secuencia. Además, ciertas instrucciones sólo pueden ocurrir una vez por Dockerfile.

 **NOTA:** NO ES OBLIGATORIO ESCRIBIR EN MAYÚSCULAS LAS INSTRUCCIONES, PERO HAS DE SEGUIR LA CONVENCION PARA CREAR UN DOCKERFILE.

En el caso de los argumentos, hay que **distinguir entre las partes de código fijo y variable**. Según la metodología de la app de 12 factores, Docker utiliza variables de entorno para la configuración de los contenedores.

Dentro de un Dockerfile, las variables de entorno se definen con la instrucción ENV. Así se asigna un valor a la variable de entorno:

Los valores almacenados en las variables de entorno pueden leerse y utilizarse como **partes variables de los argumentos**. Además, se utiliza una sintaxis especial que recuerda al script de Shell. El nombre de la variable de entorno se indica con un signo de dólar: \$env_var.

También existe una escritura alternativa para delimitar explícitamente el nombre de la variable. En este caso, el nombre de la variable va entre corchetes: \${env_var}.

Veamos un ejemplo concreto:

```
# Establecer variable 'user' como valor 'admin'
ENV user="admin"

# Establecer nombre de usuario como 'admin_user'
USER ${user}_user
```

Las instrucciones Dockerfile más importantes

A continuación, te presentamos las instrucciones de Dockerfile más importantes. Tradicionalmente, algunas instrucciones, sobre todo FROM, solo podían utilizarse una vez por Dockerfile. Con el paso del tiempo, ha surgido la construcción en múltiples etapas, o multi-stage builds, que describe múltiples imágenes en un Dockerfile, por lo que la limitación se refiere a cada etapa individual.


Instrucción	Descripción	Comentario
FROM	Establecer imagen base	Debe presentarse como primera instrucción; solo una entrada por etapa
ENV	Variable de entorno para el proceso de construcción y establecer vida del contenedor	—
ARG	Declarar parámetros de la línea de comandos para el proceso de construcción	Debe aparecer antes que la instrucción FROM
WORKDIR	Cambiar de directorio actual	—
USER	Cambiar usuario y pertenencia al grupo	—

Instrucción	Descripción	Comentario
COPY	Copiar los archivos y directorios de una Image	Crea una nueva capa
ADD	Copiar los archivos y directorios de una Image	Crea una nueva capa; uso no recomendable
RUN	Ejecutar el comando de Image durante el proceso de construcción	Crea una nueva capa
CMD	Establecer argumentos estándar para el inicio del contenedor	Solo una entrada por build stage
ENTRYPOINT	Establecer comando estándar para el inicio del contenedor	Solo una entrada por etapa de construcción
EXPOSE	Definir asignación de puerto para contenedores en ejecución	Los puertos deben estar activos al iniciar el contenedor
VOLUME	Integrar como volumen directorio de Image al iniciar el contenedor en el sistema anfitrión	—

Instrucción FROM

La instrucción FROM establece la imagen base sobre la que operan las instrucciones posteriores. Esta directiva solo puede incluirse una vez por etapa de construcción y debe ser la primera instrucción. Con una restricción: la instrucción ARG puede presentarse antes que FROM. Esto permite especificar exactamente qué imagen se utiliza como imagen base a través de un argumento de línea de comandos al iniciar el proceso build.

Cada imagen Docker debe basarse en una imagen base. En otras palabras, todas las imágenes Docker tienen exactamente una imagen predecesora. Esto da lugar al clásico dilema “del huevo y la gallina”: la cadena debe empezar en algún sitio. En el universo Docker, el linaje comienza con la imagen “scratch”. Esta imagen mínima sirve como origen de toda imagen Docker.

 **NOTA:** EN INGLÉS, “FROM SCRATCH” SIGNIFICA QUE ALGO ESTÁ HECHO CON INGREDIENTES BÁSICOS. EL TÉRMINO SE UTILIZA EN REPOSTERÍA Y COCINA. SI UN DOCKERFILE COMIENZA CON LA LÍNEA “FROM SCRATCH”, ALUDE A QUE LA IMAGEN SE COMPILA DESDE CERO.

Instrucciones ENV y ARG

Estas dos instrucciones asignan un valor a una variable. La distinción entre ambas afirmaciones radica principalmente en el origen de los valores y en el contexto en el que están disponibles las variables. Veamos primero la instrucción ARG.

Con la instrucción ARG, se declara dentro del Dockerfile una variable que solo está disponible mientras dure el proceso de construcción. El valor de una variable declarada con ARG se pasa como argumento de la línea de comandos cuando se inicia el proceso build. Veamos un ejemplo declarando la variable de build “user”:

```
ARG user
```

Al iniciar el proceso build pasamos el valor real de la variable:

```
docker build --build-arg user=admin
```

Al declarar la variable, existe la opción de especificar un valor por defecto. Si no se pasa ningún argumento adecuado al iniciar el proceso build, la variable recibe el valor por defecto:

```
ARG user=tester
```

Si no se utiliza “--build-arg”, la variable “user” contiene por defecto el valor “tester”:

```
docker build
```

Mediante la instrucción ENV, definimos una variable de entorno. A diferencia de la instrucción ARG, una variable definida con ENV existe tanto durante el proceso de construcción como durante la ejecución del contenedor. Hay dos escrituras posibles para la instrucción ENV.

Escritura recomendada:

```
ENV version="1.0";
```

 **NOTA:** LA FUNCIONALIDAD DE LA INSTRUCCIÓN ENV SE CORRESPONDE MÁS O MENOS A LA DEL COMANDO “EXPORT” DE LA LÍNEA DE COMANDOS.

Instrucciones WORKDIR y USER

La instrucción WORKDIR sirve para cambiar los directorios durante el proceso de construcción, así como al iniciar el contenedor. Al activar WORKDIR, esta se aplica a todas las instrucciones posteriores. Durante el proceso de construcción, se ven afectadas

las instrucciones RUN, COPY y ADD; durante la ejecución del contenedor, las instrucciones CMD y ENTRYPOINT.

 **NOTA:** LA INSTRUCCIÓN **WORKDIR** SE CORRESPONDE MÁS O MENOS CON EL COMANDO CD DE LA LÍNEA DE COMANDOS.

Similarmente al cambio de directorio, la instrucción USER permite cambiar el usuario actual (de Linux). Existe la opción de especificar el grupo al que pertenece el usuario. La llamada a USER se aplica a todas las instrucciones posteriores. Durante el proceso de construcción, las instrucciones RUN se ven influenciadas por su pertenencia al usuario y al grupo; durante el tiempo de ejecución del contenedor, esto se aplica a las instrucciones CMD y ENTRYPOINT.

 **NOTA:** LA INSTRUCCIÓN **USER** CORRESPONDE APROXIMADAMENTE AL COMANDO SU DE LA LÍNEA DE COMANDOS.

Instrucciones COPY y ADD

Las instrucciones COPY y ADD sirven ambas para añadir archivos y directorios a la Docker Image. Ambas instrucciones crean una nueva capa que se apila a la imagen existente. En la instrucción COPY, la fuente siempre es el contexto de construcción. En el siguiente ejemplo, copiamos un archivo readme del subdirectorio “doc” del contexto build en el directorio de nivel superior “app” de la imagen:

```
COPY ./doc/readme.md /app/
```

 **NOTA:** LA INSTRUCCIÓN **COPY** CORRESPONDE APROXIMADAMENTE AL COMANDO CP DE LA LÍNEA DE COMANDOS.

La instrucción ADD se comporta de forma casi idéntica, pero puede recuperar recursos URL fuera del contexto de construcción y descomprimir archivos comprimidos. En la práctica, esto puede conllevar efectos secundarios inesperados, por lo tanto, se desaconseja totalmente el uso de la instrucción ADD. En la mayoría de los casos debe utilizarse exclusivamente la instrucción COPY.

Instrucción RUN


La instrucción RUN es una de las más comunes de Dockerfile. Con ella, indicamos a Docker que ejecute un comando de la línea de comandos durante el proceso de construcción. Los cambios resultantes se apilan como una nueva capa sobre la imagen existente. Hay dos escrituras para la instrucción RUN:

Escritura Shell: los argumentos pasados a RUN se ejecutan en el Shell estándar de la Image. Los símbolos especiales y las variables de entorno se sustituyen según las reglas de Shell. He aquí un ejemplo de una llamada que saluda al usuario actual y utilizar una subshell “\$()”:

```
RUN echo &quot;Hello $(whoami) &quot;;
```

2. Escritura “Exec”: en vez de pasar un comando a la Shell, se activa directamente un archivo ejecutable. En el proceso, pueden pasarse argumentos adicionales. Este es un ejemplo de una llamada a la herramienta de desarrollo “npm” indicándole que ejecute el script “build”:

```
CMD [&quot;npm&quot;;, &quot;run&quot;;, &quot; build&quot;;]
```

 **NOTA:** EN PRINCIPIO, LA INSTRUCCIÓN **RUN** PUEDE SUSTITUIR A ALGUNAS DE LAS OTRAS INSTRUCCIONES DE **DOCKER**. POR EJEMPLO, LA LLAMADA “**RUN CD SRC**” ES MÁS O MENOS EQUIVALENTE A “**WORKDIR SRC**”. **NO** OBSTANTE, ESTE ENFOQUE CREA **DOCKERFILES**, QUE CONFORME VAN AUMENTANDO DE TAMAÑO SE VUELVEN MÁS DIFÍCILES DE LEER Y GESTIONAR, POR LO QUE ES MEJOR UTILIZAR INSTRUCCIONES ESPECIALIZADAS SI ES POSIBLE.

Instrucciones **CMD** y **ENTRYPOINT**

La instrucción **RUN** ejecuta un comando durante el proceso build y crea una nueva capa en la Docker Image. En cambio, las instrucciones **CMD** o **ENTRYPOINT** ejecutan un comando cuando se inicia el contenedor. La diferencia entre ambas afirmaciones es sutil:

ENTRYPOINT se utiliza para crear un contenedor que siempre realiza la misma acción cuando se inicia, por lo que el contenedor se comporta como un archivo ejecutable. **CMD** se utiliza para crear un contenedor que ejecuta una acción definida cuando se inicia sin más parámetros. La acción preestablecida se puede sobrescribir fácilmente mediante parámetros adecuados.

Lo que ambas instrucciones tienen en común es que solo pueden darse una vez por Dockerfile. Sin embargo, es posible combinar ambas instrucciones. En este caso, **ENTRYPOINT** define la acción estándar que se realizará al iniciar el contenedor, mientras que **CMD** define parámetros fácilmente anulables para la acción.

Nuestra entrada en Dockerfile:

```
ENTRYPOINT ["echo", "Hello"]  
CMD ["World"]
```

Los comandos correspondientes de la línea de comandos:

```
# Salida "Hello World"
docker run my_image
# Salida "Hello Moon"
docker run my_image Moon
```

Instrucción EXPOSE

Los contenedores Docker se comunican a través de la red. Los servicios que se ejecutan en el contenedor se dirigen a través de los puertos especificados. La instrucción EXPOSE documenta la asignación de puertos y soporta los protocolos TCP y UDP. Si un contenedor se inicia con “docker run -P”, el contenedor escucha en los puertos definidos a través de EXPOSE. De manera alternativa, los puertos asignados pueden ser sobrescritos con “docker run -p”.

He aquí un ejemplo. Nuestro Dockerfile contiene la siguiente instrucción EXPOSE:

```
EXPOSE 80/tcp
EXPOSE 80/udp
```

Luego, existen las siguientes vías para hacer que los puertos se activen al iniciar el contenedor:

```
# Container escucha para TCP / UDP Traffic en Port 80
docker run -P
# Container escucha para TCP Traffic en Port 81
docker run -p 81:81/tcp
```

Instrucción VOLUME

Un Dockerfile define una imagen Docker que consiste en capas apiladas las unas sobre las otras. Las capas son solo de lectura para que se garantice siempre el mismo estado cuando se inicie un contenedor. Necesitamos un mecanismo para intercambiar datos entre el contenedor en ejecución y el sistema anfitrión. La instrucción VOLUME define un “mount point” dentro del contenedor.


Consideremos el siguiente fragmento de un archivo Docker. Creamos un directorio “shared” en el directorio de nivel superior de la imagen. También especificamos que este directorio se incluya en el sistema anfitrión cuando se inicie el contenedor:

```
RUN mkdir /shared
VOLUME /shared
```

Ten en cuenta que dentro del Dockerfile no podemos definir la ruta real en el host system. De manera estándar, los directorios definidos mediante la instrucción VOLUME se incluyen en el sistema anfitrión bajo "/var/lib/docker/volumes/".

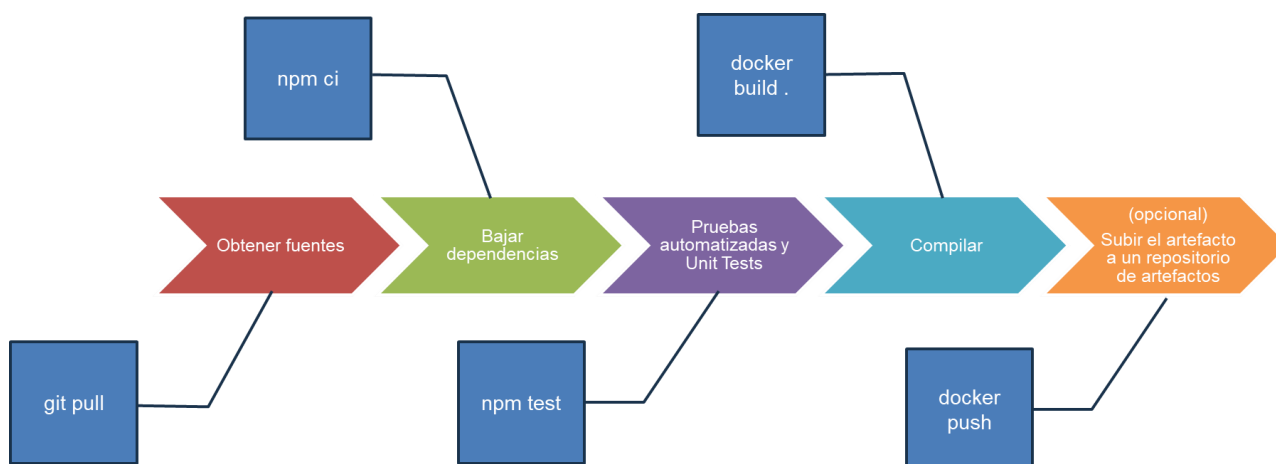
¿Cómo se modifica un Dockerfile?

Recuerda: un Dockerfile es un archivo de texto (plano). Se puede editar con los métodos típicos; un editor de texto plano es probablemente el más común. Puede ser un editor con una interfaz gráfica de usuario. ¡Será por opciones! Los editores más populares son VSCode, Sublime Text, Atom y Notepad++. Como alternativa, hay varios editores disponibles en la línea de comandos. Además de los originales Vim o Vi, los editores mínimos Pico y Nano también se usan mucho.

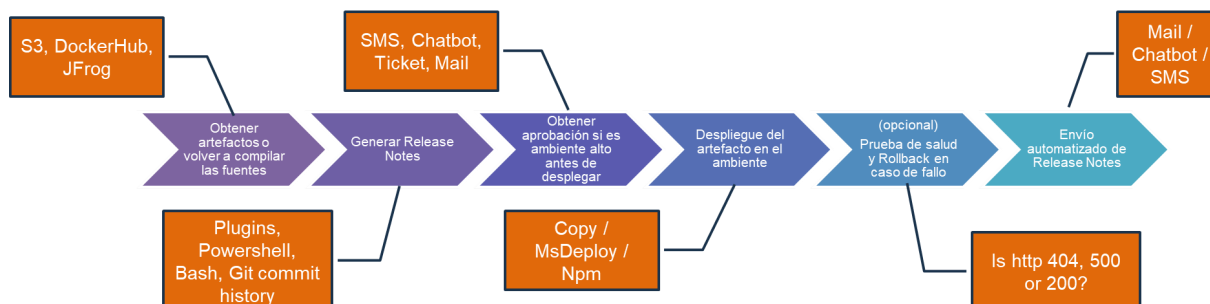
 **NOTA: EDITA ARCHIVOS DE TEXTO PLANO ÚNICAMENTE CON EDITORES ADECUADOS. NO USES BAJO NINGÚN CONCEPTO PROCESADORES DE TEXTOS COMO MICROSOFT WORD, APPLE PAGES O LIBRE U OPENOFFICE PARA EDITAR UN DOCKERFILE.**

3.4 Pipelines CI CD y Docker

• 3.4.1. Pasos en la fase de Build (ejemplo)

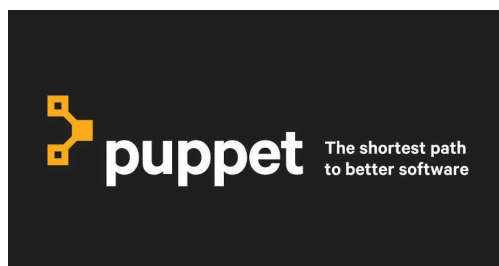


● 3.4.2. Pasos en la fase de Deploy (ejemplo)



≡ 3.5 Orquestación de Contenedores y Configuración

● 3.5.1. Puppet



A medida que los administradores de sistemas adquieren cada vez más sistemas para gestionar, la automatización de tareas rutinarias se vuelve cada vez más importante. En lugar de desarrollar scripts internos, es deseable compartir un sistema que todos puedan utilizar e invertir en herramientas que se puedan utilizar independientemente del empleador de cada uno. Ciertamente, realizar tareas de forma manual no es escalable.

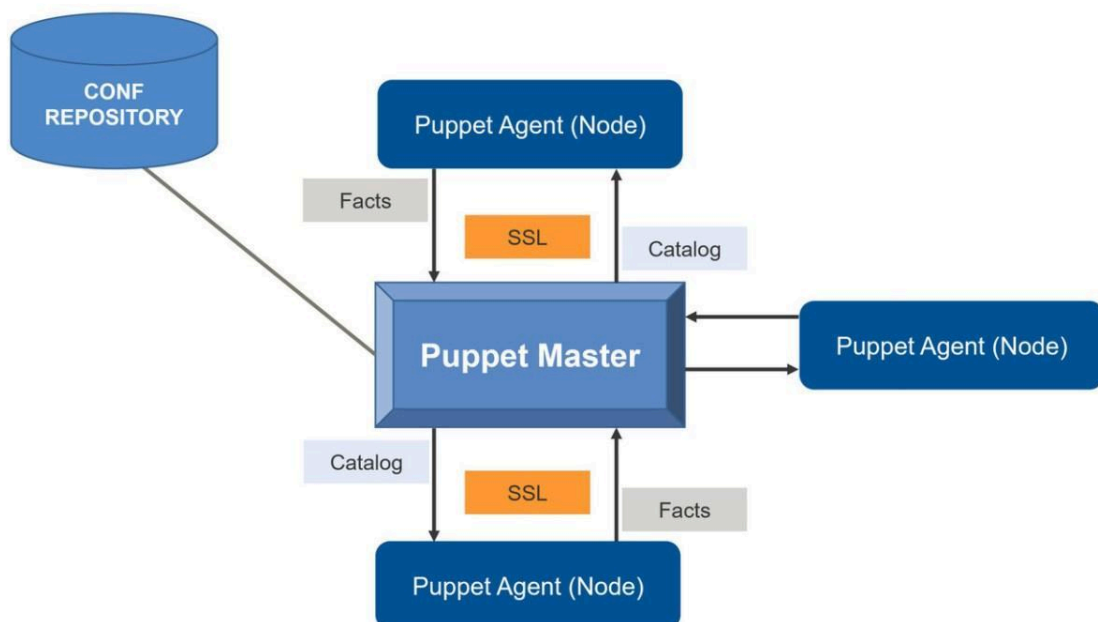
Puppet ha sido desarrollado para ayudar a la comunidad de administradores de sistemas a pasar a la creación y compartición de herramientas maduras que eviten la duplicación de resolver el mismo problema por parte de todos. Lo hace de dos maneras:

Proporciona un marco poderoso para simplificar la mayoría de las tareas técnicas que los administradores de sistemas necesitan realizar.

El trabajo del administrador de sistemas se escribe como código en el lenguaje personalizado de Puppet, que es compatible de la misma manera que cualquier otro código.

Esto significa que tu trabajo como administrador de sistemas puede realizarse mucho más rápido, porque Puppet puede encargarse de la mayoría o de todos los detalles, y puedes descargar código de otros administradores de sistemas para ayudarte a terminar aún más rápido. La mayoría de las implementaciones de Puppet utilizan al menos uno o dos módulos desarrollados por otra persona, y ya hay cientos de módulos desarrollados y compartidos por la comunidad.

Arquitectura Puppet



La funcionalidad de Puppet se construye como una pila de capas separadas, cada una responsable de un aspecto específico del sistema, con controles rigurosos sobre cómo se transmite la información entre las capas.

Características de Puppet



Versiones de Puppet

Puppet Open Source (OSP)

- Es la versión gratuita y de código abierto de Puppet.
- Proporciona las funcionalidades esenciales para la automatización de configuraciones y gestión de sistemas.
- Es adecuado para pequeñas y medianas empresas que buscan aprovechar las capacidades de automatización de Puppet sin incurrir en costos de licencia.

Puppet Enterprise

- Es la versión comercial de Puppet y ofrece funcionalidades adicionales y herramientas de gestión avanzadas.
- Incluye características como un panel de control visual, informes detallados, capacidades de orquestación y soporte técnico dedicado.
- Está diseñado para empresas a gran escala y entornos empresariales que requieren funcionalidades adicionales y un soporte más completo.

Componentes de Puppet

Puppet Master

- Es el servidor central en el modelo cliente-servidor de Puppet.
- Almacena la configuración y las políticas definidas por el usuario.
- Los nodos gestionados (clientes) se conectan al maestro para obtener y aplicar su configuración.

Puppet Agent

- Es el software que se instala en los nodos gestionados (servidores o máquinas clientes).
- Se ejecuta en segundo plano y se comunica con el Puppet Master para obtener y aplicar las configuraciones.

Facter

- Es una herramienta que Puppet utiliza para recopilar información sobre el sistema en el que se ejecuta.
- Proporciona datos factuales sobre el sistema, como el sistema operativo, la arquitectura del hardware, las interfaces de red, etc.
- Esta información se utiliza en las manifestaciones de Puppet para tomar decisiones basadas en el estado del sistema.

Puppet DSL (Domain-Specific Language)

- Es un lenguaje específico de dominio desarrollado por Puppet para definir la configuración de los sistemas.
- Los archivos de configuración de Puppet, llamados manifiestos, están escritos en este lenguaje declarativo.

Manifiestos

- Son archivos escritos en Puppet DSL que contienen las declaraciones de configuración y políticas que se aplicarán a los nodos gestionados.
- En un manifiesto, definen la configuración del sistema declarando el estado deseado de los recursos. Un "recurso" puede ser cualquier entidad gestionada por Puppet, como archivos, servicios, usuarios, paquetes, etc.

```
# Ejemplo de un manifiesto simple que gestiona un
archivo
```

```
file { ['/ruta/al/archivo']:
  ensure => present,
```

```
content => 'Contenido del archivo',
owner   => 'usuario',
group   => 'grupo',
mode    => '0644',
}
```

Recursos y Tipos de Recursos

Los recursos son instancias concretas de tipos de recursos. Los tipos de recursos son categorías generales, como archivos, servicios, usuarios, etc. En el ejemplo anterior, file es un tipo de recurso, y el bloque { ... } es una instancia específica de ese recurso.

Relaciones

Los manifiestos también pueden incluir relaciones entre recursos. Por ejemplo, puedes establecer que la creación de un archivo dependa de la instalación de un paquete.

```
package { 'nombre_del_paquete':
  ensure => installed,
}

file { '/ruta/al/archivo':
  ensure => present,
  content => 'Contenido del archivo',
  require => Package['nombre_del_paquete'],
}
```

Clases y Herencia

Los manifiestos pueden contener clases que agrupan recursos y definiciones de recursos relacionados. La herencia de clases permite la reutilización y extensión de configuraciones.

```
# Definición de una clase
class mi_clase {
  file { '/ruta/al/archivo':
    ensure => present,
    content => 'Contenido del archivo',
  }
}

# Herencia de clase
```

```
class otra_clase inherits mi_clase {
  # Configuraciones adicionales van aquí
}
```

Plantillas

Puppet permite el uso de plantillas para generar dinámicamente contenido de archivos. Esto es útil cuando el contenido del archivo puede depender de variables o factores del nodo.

```
file { '/ruta/al/archivo':
  ensure => present,
  content => template('mi_modulo/mi_plantilla.erb'),
}
```

Parámetros

Los manifiestos pueden aceptar parámetros, lo que los hace más flexibles y reutilizables. Los parámetros se definen en las clases y se pueden pasar al instanciar la clase.

```
# Definición de una clase con parámetros
class mi_clase($parametro) {
  file { '/ruta/al/archivo':
    ensure => present,
    content => $parametro,
  }
}
```

Módulos

- Los módulos son unidades organizativas que contienen manifiestos, archivos y datos necesarios para gestionar un conjunto específico de recursos.
- Facilitan la reutilización y la compartición de código, permitiendo a los administradores utilizar configuraciones predefinidas para tareas comunes.
- Un módulo en Puppet sigue una estructura de directorios específica. Los directorios comunes incluyen manifest para los archivos de configuración (manifiestos), files para archivos estáticos, templates para plantillas de archivos y lib para módulos escritos en Ruby.

```
mimodulo/
```

```
|-- manifests/
|   |-- init.pp
|-- files/
|-- templates/
|-- lib/
```

Hiera

- Es una herramienta de Puppet que facilita la gestión de datos externos y variables en los manifiestos.
- Permite separar los datos de la lógica de configuración, facilitando la modularidad y la reutilización.

PuppetDB

- Es una base de datos que almacena información sobre los nodos gestionados y las relaciones entre recursos.
- Proporciona una interfaz que permite consultar y visualizar datos sobre los nodos y su configuración.

Puppet Bolt

- Es una herramienta adicional que permite la ejecución de comandos y tareas de forma ad hoc en nodos gestionados sin necesidad de una infraestructura de Puppet completa.

Recursos

En Puppet, los "recursos" son unidades fundamentales de configuración que representan componentes individuales del sistema. Los recursos describen el estado deseado de esos componentes.

File (Archivo):

Representa un archivo o directorio en el sistema de archivos.

Puede especificar propiedades como el contenido del archivo, los permisos, el propietario y el grupo.

puppet

```
file { '/path/to/file':
  ensure => present,
  content => 'Contenido del archivo',
  mode   => '0644',
  owner  => 'usuario',
  group  => 'grupo',
```

```
}
```

Package (Paquete):

Gestiona la instalación, actualización o desinstalación de paquetes de software. Puede especificar la versión del paquete que se debe instalar.

```
package { 'nombre_del_paquete':  
  ensure => installed,  
  version => '1.2.3',  
}
```

Service (Servicio):

Gestiona el estado de los servicios del sistema, como iniciar, detener o reiniciar. Puede especificar el estado deseado del servicio.

```
service { 'nombre_del_servicio':  
  ensure    => running,  
  enable    => true,  
}
```

User (Usuario) y Group (Grupo):

Representan la creación y gestión de usuarios y grupos en el sistema. Pueden especificar propiedades como la contraseña, el directorio de inicio, etc.

```
user { 'nombre_del_usuario':  
  ensure    => present,  
  home      => '/ruta/del/directorio',  
  password  => 'contraseña_encriptada',  
}
```

```
group { 'nombre_del_grupo':  
  ensure => present,  
}
```

Exec (Ejecutable):

Ejecuta comandos en el sistema y gestiona su estado. Puede ser útil para realizar acciones personalizadas que no están cubiertas por otros recursos.

```
exec { 'comando_personalizado':  
  command => '/ruta/al/comando',  
  creates => '/ruta/al/resultado',  
  cwd     => '/directorio/de/trabajo',  
}
```

```
}
```

Clases

En Puppet, las "clases" son una forma de organizar y estructurar la configuración. Proporcionan un mecanismo para agrupar recursos y definiciones de recursos relacionados en un solo lugar. La idea es modularizar y reutilizar configuraciones de manera eficiente. Aquí hay algunos conceptos clave relacionados con las clases en Puppet:

Una clase se declara usando la palabra clave `class` seguida del nombre de la clase y un bloque de código que contiene los recursos y definiciones de recursos asociados.

```
class nombre_de_la_clase {
  # Recursos y definiciones de recursos van aquí
}
```

Inclusión de Clases

Puedes incluir una clase en un nodo específico para que se aplique la configuración asociada a esa clase en particular.

```
node 'nombre_del_nodo' {
  include nombre_de_la_clase
}
```

Parámetros de Clases

Las clases pueden aceptar parámetros que les permiten ser más flexibles y reutilizables. Los parámetros se definen dentro de la clase y se pueden pasar al incluir la clase en un nodo.

```
class nombre_de_la_clase($parametro1, $parametro2) {
  # Uso de parámetros en la clase
}
```

Espacios de Nombres (Namespaces)

Las clases pueden organizarse en espacios de nombres para evitar conflictos de nombres y facilitar la modularización. Por ejemplo, puedes tener una clase llamada `webserver::apache` dentro del espacio de nombres `webserver`.

```
class webserver::apache {
```

```
# Configuración de Apache va aquí
}
```

Herencia de Clases

Puppet permite la herencia de clases, lo que significa que una clase puede heredar recursos y configuraciones de otra clase. Esto facilita la reutilización de configuraciones comunes.

```
class base_config {
  # Configuración común va aquí
}
```

```
class app_config inherits base_config {
  # Configuración específica de la aplicación va aquí
}
```

Clases de Nodo Predeterminadas

Puedes definir clases de nodo predeterminadas para aplicar configuraciones comunes a todos los nodos que no tengan una configuración específica asignada.

```
node default {
  include base_config
}
```

Clases en Manifiestos y Estructura de Directorios

Las clases suelen definirse en archivos llamados manifiestos. La estructura de directorios y archivos es importante para organizar eficientemente tu código de Puppet.

Estructura de Carpetas de Puppet

La estructura de carpetas y archivos en Puppet es esencial para organizar y gestionar eficientemente la configuración de tu infraestructura.

site.pp:

Este archivo, a menudo ubicado en el directorio manifests, es el archivo de entrada principal para las configuraciones en Puppet. Contiene definiciones de nodos y las inclusiones de clases correspondientes. Las configuraciones específicas del nodo se pueden declarar aquí.

```
# site.pp en manifests/
node 'mi_servidor' {
  include mi_clase
}
```

manifests/:

El directorio manifests alberga los manifiestos principales que describen la configuración de los nodos. En este directorio, generalmente encuentras el archivo site.pp y otros manifiestos organizados en carpetas según su función.

```
manifests/
|-- site.pp
|-- mi_clase.pp
|-- otra_clase/
|   |-- init.pp
```

modules/:

El directorio modules es donde suelen almacenarse los módulos de Puppet. Cada módulo es una unidad independiente que organiza archivos y configuraciones específicas de una tarea o servicio particular. Los módulos pueden tener subdirectorios como manifests, files, templates, etc.

```
modules/
|-- mi_modulo/
|   |-- manifests/
|   |   |-- init.pp
|   |-- files/
|   |-- templates/
|   |-- lib/
|-- otro_modulo/
|   |-- manifests/
|   |   |-- init.pp
|   |-- files/
|   |-- templates/
|   |-- lib/
```

files/ y templates/:

En el directorio files, puedes colocar archivos estáticos que serán copiados directamente a los nodos gestionados. En templates, se almacenan plantillas utilizadas por el recurso template para generar archivos configurables dinámicamente.

```
mi_modulo/
|-- files/
|   |-- archivo_estatico.txt
|-- templates/
|   |-- plantilla.erb
```

lib/:

El directorio lib es utilizado para almacenar módulos escritos en Ruby (llamados "librerías") que pueden ser utilizados en tus manifestos de Puppet.

```
mi_modulo/
|-- lib/
|   |-- mi_libreria.rb
```

facts.d/:

Este directorio permite agregar scripts ejecutables que proporcionan información adicional como hechos (facts) del nodo. Estos scripts pueden agregar hechos personalizados que luego pueden utilizarse en tus manifestos.

```
facts.d/
|-- mi_fact_script.sh
```

● 3.5.2. Progress Chef



Permite describir y gestionar la infraestructura en forma de código. Está pensado para conocer mediante código nuestra infraestructura y poder recrearla, reconfigurarla o

replicarla. Chef configura y despliega aplicaciones y configuraciones, orquestando la configuración de ambientes compuestos por más de un nodo.

Es una herramienta escrita en Ruby que automatiza el procedimiento de aprovisionamiento. Trabaja con un modelo Maestro – Cliente en el que se requiere un equipo independiente desde el que controlar el nodo maestro.

Se integra perfectamente con plataformas en nube (Amazon, GoogleCloud, Azure, OpenStack)

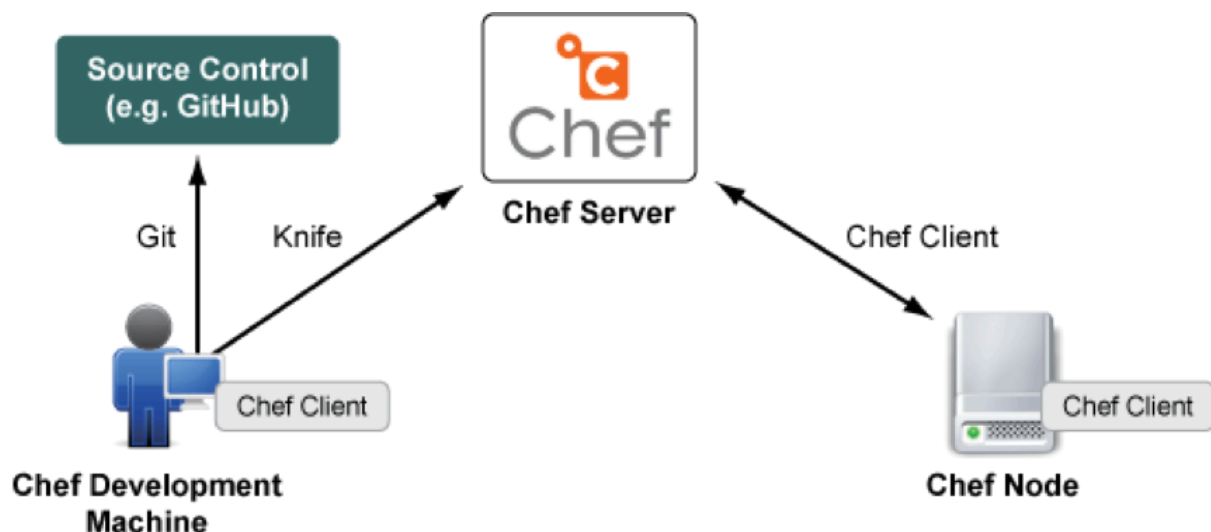
No es agent-less. Requiere de la instalación de un agente en cada nodo gestionado. Los agentes pueden ser instalados desde el cliente usando knife que usa SSH para desplegar. Una vez hecho eso, todos los nodos gestionados se comunican con el servidor principal a través del uso de certificados y reciben el despliegue.

Chef Software fue adquirida en septiembre de 2020 por Progress, así que los lineamientos de la empresa cambiaron, aunque su filosofía se mantiene.

Productos que ofrecen:



Arquitectura Chef



Características de Chef

1. Infraestructura como Código (IaC):

- Chef sigue el paradigma de IaC, lo que significa que la configuración de la infraestructura se define mediante código, permitiendo una gestión consistente y versionada.

2. Reusabilidad con Recetas y Cookbooks

3. Modelo Cliente-Servidor:

- Chef utiliza una arquitectura cliente-servidor. Los nodos gestionados se conectan a un servidor de Chef para obtener y aplicar configuraciones.

4. Soporte Multiplataforma:

- Chef es compatible con una amplia variedad de sistemas operativos, lo que lo hace versátil en entornos heterogéneos.

5. Orquestación y Escalabilidad:

- Chef proporciona capacidades de orquestación para coordinar y ejecutar tareas en múltiples nodos.
- Es escalable y puede gestionar eficientemente desde entornos pequeños hasta infraestructuras masivas.

6. Auditoría y Control:

- Chef proporciona capacidades de auditoría y control, lo que facilita el seguimiento de cambios y la implementación de políticas de seguridad.

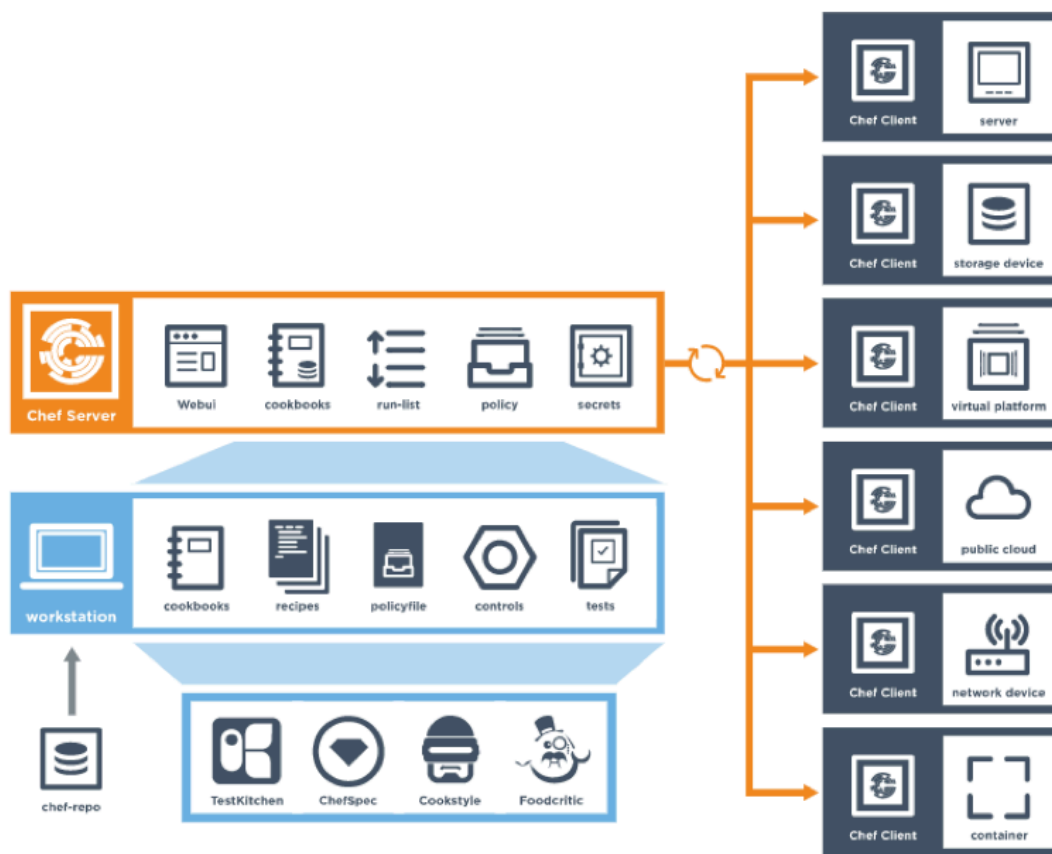
7. Soporte Comunitario y Empresarial:

- Chef cuenta con una comunidad activa y ofrece una versión de código abierto (Chef Infra Open Source) y una versión empresarial (Chef Infra Client) con características y soporte adicionales.

8. Integración con Herramientas Externas:

- Chef se integra fácilmente con otras herramientas y servicios, lo que facilita la incorporación en entornos existentes.

Componentes

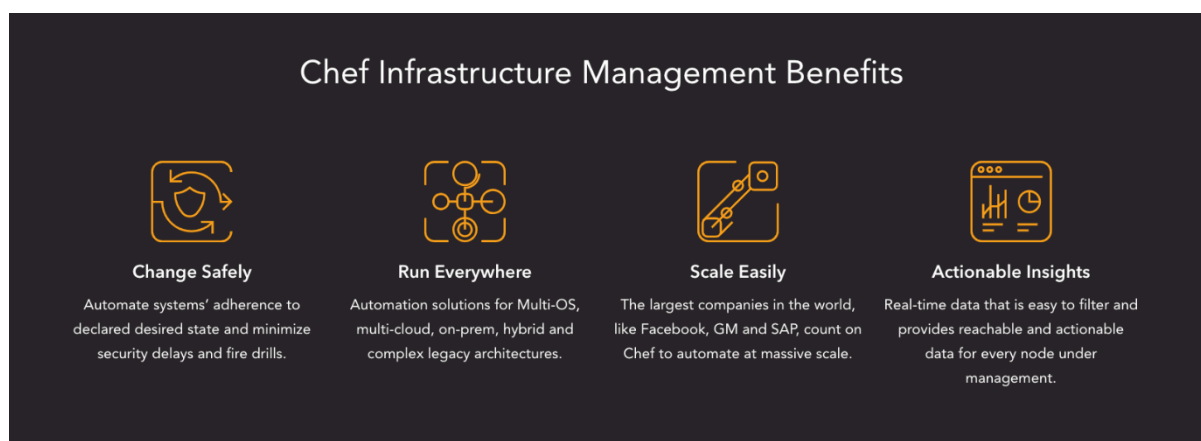


- **Workstation:** Uno o más workstations son configurados para crear, probar y mantener los cookbook. A menudo un Workstation se configura para usar un Chef Workstation como kit de herramientas de desarrollo.
- **Recipes (recetas):** Las recetas son unidades de configuración en Chef. Cada receta describe cómo configurar un componente específico del sistema.
- **Cookbook (conjunto de recetas):** Los cookbook se cargan en el Chef Infra Server desde un Workstation. Algunos son personalizados para la organización y otros se basan en la comunidad de cookbooks disponibles desde el Chef Supermarket.
- **Ruby:** Lenguaje de programación que se utiliza en la sintaxis de los CookBook.
- **Node:** Un nodo es una máquina que está bajo la administración de Chef.
- **Chef Client (Agente):** Está instalado en cada nodo que está bajo la gestión de Chef. Realiza todas las tareas de configuración que están especificadas bajo una lista de ejecución que desplegará cualquier dato de configuración requerido del Chef Server.

- **Chef Server:** Actúa como centro de información. Cookbooks y Policy Settings se cargan en el Chef Server por los usuarios de los Workstation. El Chef Client accede al Chef Server desde el nodo desde el que está instalado para obtener los datos de configuración, realiza búsquedas de datos históricos de ejecución de Chef Client, y descarga los datos de configuración necesarios. Cuando el Chef Client termina su ejecución, sube los datos de ejecución actualizados al Chef Server.
- **Chef Supermarket:** Es la ubicación en la que se comparten y gestionan los Cookbooks de la comunidad.
- **Ohai:** Similar a Facter en Puppet, Ohai es una herramienta en Chef que recopila información sobre el sistema y la expone para su uso en recetas.
- **ChefDK (Chef Development Kit):** Proporciona un conjunto de herramientas y bibliotecas para el desarrollo y prueba de recetas y cookbooks.

Chef Infra Management

Cuando nos acercamos al sitio web de la empresa veremos una gran cantidad de productos. No es mas que un ecosistema que utilizan para solucionar requerimientos de clientes específicos. Lo importante es comprender que cuando nos referimos a la herramienta Chef, en opensource OSS es Chef Infra Management y es la combinación de Chef Infra y Chef Automation.



Chef infra

Automatiza el proceso de gestionar configuraciones, asegurando que cada sistema esté configurado correctamente y de manera consistente. Aplica actualizaciones de manera dinámica, realizando cambios condicionales basados en el entorno en ejecución o el hardware. Garantiza que el mismo código que configura el desarrollo pueda utilizarse

hasta la producción. Hace que las configuraciones de la infraestructura sean testables, portátiles y auditables. Chef Automate proporciona visibilidad operativa sobre el estado de tu infraestructura en un momento dado y a lo largo del tiempo.

Chef Automate

Información en Tiempo Real Se recopilan detalles de configuración y cumplimiento para cada centro de datos, proveedor de servicios en la nube y entorno gestionado por Chef Infra, y los datos se presentan en paneles agregados que pueden filtrarse.

Colaboración Sencilla Entre Equipos Los equipos de desarrollo, operaciones, seguridad y cumplimiento, responsables de entregar software de manera segura y eficiente, comparten una vista consistente de cómo se construyen y validan los entornos.

Potentes Capacidades de Auditoría Crea escaneos de cumplimiento sin agente directamente en la interfaz web. Escanea servidores tradicionales, máquinas virtuales, entornos en la nube y soluciones SaaS en un solo lugar.

Controles de Acceso Inteligentes Aseguran que los equipos adecuados tengan el acceso correcto utilizando soluciones de control de acceso existentes (LDAP/SAML).

Activos de Cumplimiento Incorporados Aprovecha perfiles precreados para validar parches de software, seguridad del sistema y evaluar conforme a marcos estándar de la industria como CIS Benchmarks y DISA STIGs.

Información Accionable a Cualquier Escala Una arquitectura modernizada para una experiencia receptiva e intuitiva.

Recetas y Cookbooks en Chef

Una receta de Chef se compone de recursos y declaraciones en el lenguaje de dominio específico (DSL) de Chef. La receta define el estado deseado de la configuración en un nodo gestionado. Aquí hay una estructura básica de cómo se compone una receta de Chef:

1. Nombre de la Receta:

- Una receta comienza con la definición de su nombre. Esto se hace utilizando la directiva **recipe** seguida del nombre de la receta. Por ejemplo:

```
# En el archivo example_recipe.rb
recipe 'my_cookbook::my_recipe'
```

2. Declaraciones de Recurso:

- El corazón de una receta son las declaraciones de recursos. Los recursos son los componentes individuales del sistema que deseas configurar. Algunos ejemplos comunes de recursos incluyen archivos, servicios, paquetes, usuarios, etc. Aquí tienes un ejemplo de declaración de recurso de archivo:

```
file '/path/to/my/file.txt' do
  content 'Contenido del archivo'
  owner 'usuario'
  group 'grupo'
  mode '0644'
  action :create
end
```

En este ejemplo, se está utilizando el recurso **file** para especificar la ruta del archivo, el contenido, el propietario, el grupo, el modo y la acción (en este caso, **:create** para crear el archivo).

3. Atributos y Variables:

- Puedes utilizar atributos y variables para hacer tus recetas más dinámicas y reutilizables. Los atributos se pueden definir a nivel de receta o de cookbook. Aquí hay un ejemplo simple:

```
4.
5. file '/path/to/my/file.txt' do
6.   content node['my_cookbook']['file_content']
7.   owner 'usuario'
8.   group 'grupo'
9.   mode '0644'
10.  action :create
end
```

En este caso, el contenido del archivo proviene de un atributo llamado **'file_content'** definido en el nodo.

11. Acciones:

- Las acciones especifican qué debe hacer Chef con respecto al recurso.

Variables en Chef

En Chef, las variables se utilizan para almacenar y referenciar información que puede cambiar según el entorno, el nodo o las necesidades específicas de la configuración. Existen varias formas de trabajar con variables en Chef, y aquí te proporcionaré información sobre algunas de ellas:

1. Atributos:

- Los atributos son una forma común de manejar variables en Chef. Pueden ser específicos de nodos, roles o entornos, y se utilizan para almacenar información sobre la configuración del sistema. Los atributos pueden ser definidos en el nodo directamente o a través de roles y entornos.

Ejemplo de un atributo en un nodo:

```
node.default['mi_cookbook']['variable'] = 'valor'
```

Este atributo puede luego ser referenciado en recetas o templates.

2. Variables de Entorno:

- Puedes utilizar variables de entorno para pasar información dinámica a tus recetas durante la ejecución. Estas variables se establecen antes de ejecutar Chef y pueden ser referenciadas en recetas.

Ejemplo de definición de variable de entorno:

```
ENV['MI_VARIABLE'] = 'valor'
```

Puedes acceder a esta variable en tus recetas usando `ENV['MI_VARIABLE']`.

3. Variables Dentro de Recetas:

- Dentro de una receta, puedes utilizar variables locales para almacenar información temporal. Estas variables solo son válidas dentro del alcance de la receta en la que se definen.

Ejemplo de variable local en una receta:

```
mi_variable = 'valor'
```

4. Ohai:

- Ohai, la herramienta de Chef que recopila información sobre el sistema, también proporciona una gran cantidad de datos que se pueden utilizar como variables en recetas. Estos datos se almacenan en el atributo **node**.

Ejemplo de uso de datos de Ohai en una receta:

```
plataforma = node['platform']
```

Aquí, **plataforma** contendría el nombre de la plataforma del sistema (por ejemplo, "ubuntu", "centos", etc.).

5. Data Bags:

- Los Data Bags son otro mecanismo para almacenar datos en Chef. Puedes crear Data Bags para almacenar información estructurada y referenciarla desde tus recetas.

Ejemplo de creación de un Data Bag:

```
{
  "id": "mi_data_bag",
  "variable": "valor"
}
```

Luego puedes acceder a este Data Bag desde tus recetas.

Acciones en Chef

Las acciones son comandos específicos que le indican a Chef qué hacer con un recurso particular. Cada recurso en una receta puede tener una acción asociada.

1. **:create:**
 - Esta acción indica que Chef debe crear el recurso si no existe.
2. **:delete:**
 - La acción **:delete** indica que Chef debe eliminar el recurso si existe.
3. **:modify, :edit u otras acciones específicas:**
 - Algunos recursos permiten acciones relacionadas con la modificación, como **:create**, **:delete**, **:touch**, etc.
4. **:nothing:**
 - La acción **:nothing** indica que Chef solo debe realizar la acción si es necesario.
5. **Acciones específicas del recurso:**
 - Algunos recursos pueden tener acciones específicas relacionadas con su función, como **:start**, **:stop**, **:restart**, etc.

≡ 3.6 Kubernetes (K8s)

Para supervisar y gestionar ciclos de vida de contenedor en entornos más complejos, deberá recurrir a una herramienta de orquestación más potente, la mayoría de los cargas de trabajo complejas utilizan Kubernetes, que es una plataforma de orquestación de

contenedores de código abierto descendiente de un proyecto desarrollado para uso interno en Google.



kubernetes

Kubernetes planifica y automatiza las tareas integradas en la gestión de arquitecturas basadas en contenedores, incluida la implementación de contenedores, las actualizaciones, el descubrimiento de servicios, el suministro de almacenamiento, el equilibrio de carga, la supervisión del estado y más.

Además, el ecosistema de código abierto de herramientas para Kubernetes, incluyendo Istio y Knative, permite a las organizaciones implementar una Plataforma como servicio (PaaS) de alta productividad para aplicaciones en múltiples contenedores y una incorporación de acceso más rápida a la computación sin servidor.

Para más información sobre esta plataforma, ver <https://kubernetes.io/es/docs/>



Comparación entre K8s y Docker

Docker es un tiempo de ejecución de contenedores, mientras que Kubernetes es una plataforma para ejecutar y gestionar contenedores a partir de muchos tiempos de ejecución de contenedores. Kubernetes admite numerosos tiempos de ejecución de contenedores, como Docker, containerd, CRI-O y cualquier implementación de Kubernetes CRI (Container Runtime Interface).

Kubernetes podría entenderse como un "sistema operativo" y, los contenedores de Docker, como las "aplicaciones" que se instalan en él.

Por sí solo, Docker es muy beneficioso para el desarrollo de aplicaciones modernas y resuelve el clásico problema de "funciona en mi máquina", pero no en otras. La herramienta de orquestación de contenedores Docker Swarm puede gestionar la implementación de una carga de trabajo basada en contenedores de producción formada por varios contenedores. Cuando un sistema crece y tiene que añadir muchos

contenedores conectados en red entre sí, Docker puede, de forma independiente, hacer frente a algunos problemas crecientes que Kubernetes ayuda a resolver.

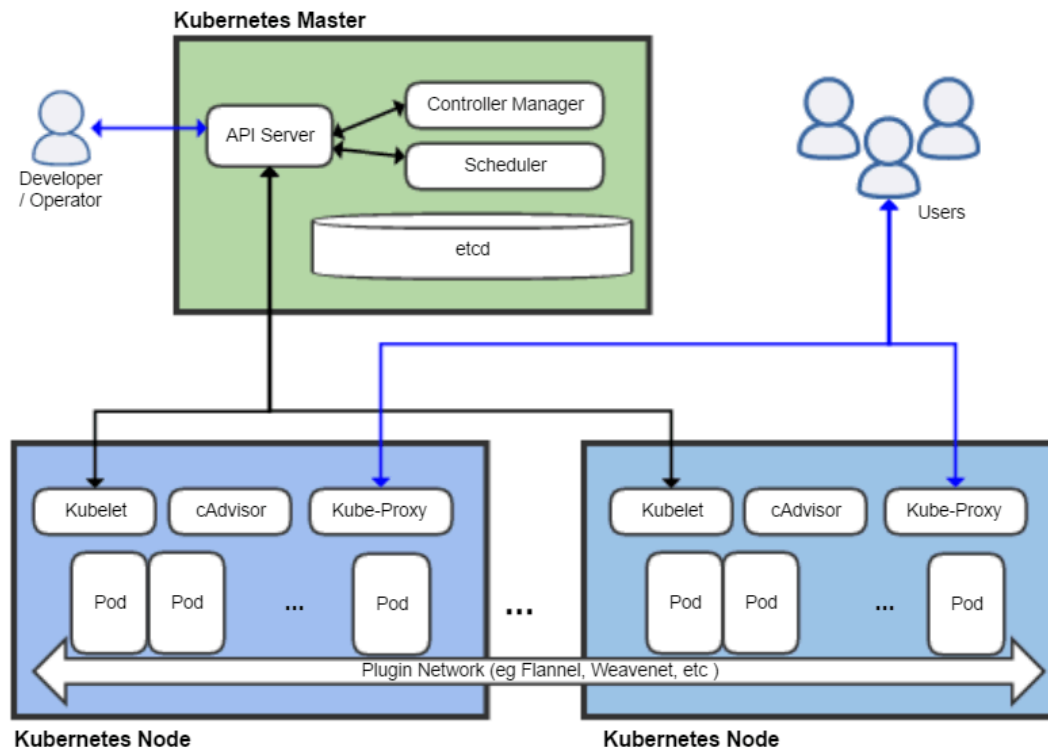
 <p>kubernetes</p> <ul style="list-style-type: none"> • Gran ecosistema de código abierto • Ofertas gestionadas por proveedores de nube • Puede manejar necesidades de aplicaciones complejas 	 <ul style="list-style-type: none"> • Estrechamente integrado con el ecosistema docker • Menos configuración y alternativas • Mas fácil de aprender
--	---

Al comparar ambas herramientas, es mejor comparar Kubernetes con Docker Swarm. Docker Swarm, o el modo swarm de Docker, es una herramienta de orquestación de contenedores como Kubernetes, lo que significa que permite gestionar varios contenedores implementados en distintos hosts que ejecutan el servidor de Docker. El modo swarm está deshabilitado de forma predeterminada y un equipo de DevOps debe instalarlo y configurarlo.

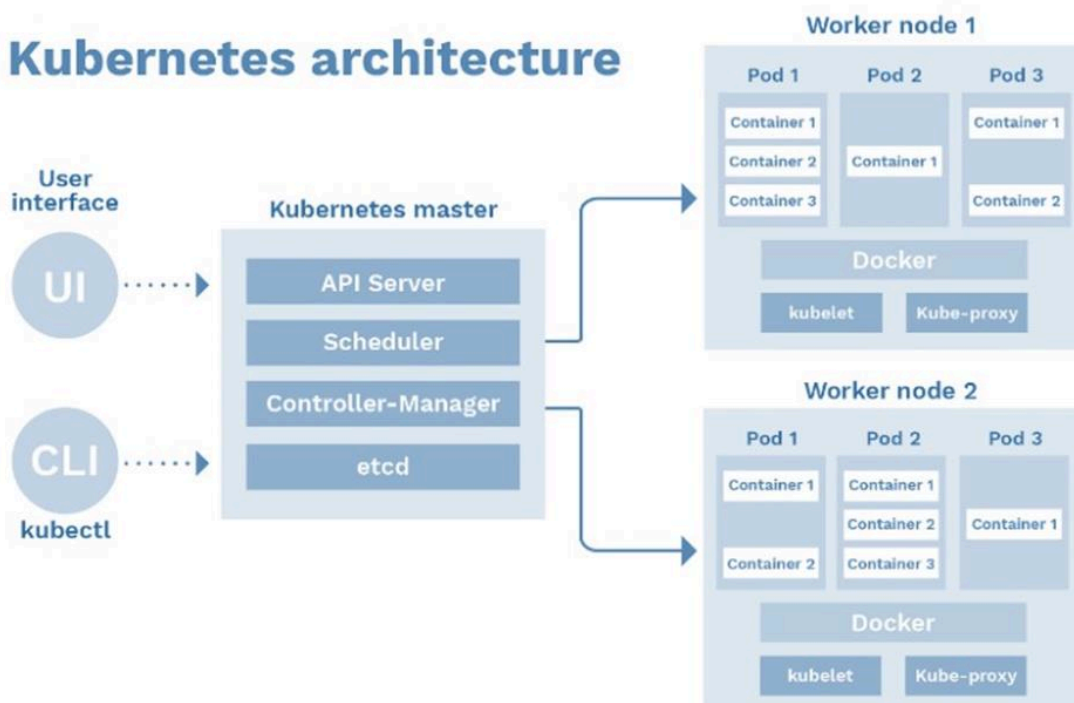
Kubernetes orquesta clústeres de máquinas para que funcionen conjuntamente y programa contenedores para que se ejecuten en esas máquinas en función de los recursos que tienen disponibles. Los contenedores se agrupan a través de una definición declarativa en pods, la unidad básica de Kubernetes. Kubernetes gestiona automáticamente tareas como el descubrimiento de servicios, el equilibrio de carga, la asignación de recursos, el aislamiento y el escalado vertical u horizontal de los pods.

Ha sido adoptado por la comunidad de código abierto y ahora forma parte de la Cloud Native Computing Foundation. Amazon, Microsoft y Google ofrecen servicios de Kubernetes gestionados en sus plataformas de computación en la nube, lo que reduce significativamente la carga de trabajo que supone ejecutar y mantener los clústeres de Kubernetes y sus cargas de trabajo containerizadas.

• 3.6.1. Arquitectura de K8s



Kubernetes architecture



● Cluster (Cúmulo o Agrupación):

Definición: Un cluster en Kubernetes es un conjunto de nodos (máquinas físicas o virtuales) que se agrupan para ejecutar aplicaciones y servicios de manera distribuida. El cluster proporciona la capacidad de escalar y gestionar eficientemente contenedores en un entorno de producción.

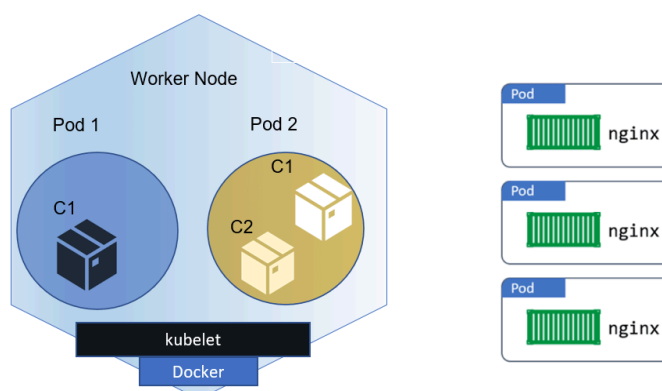
● Nodo (Worker Node):

Definición: Un nodo en Kubernetes es una máquina individual que forma parte de un cluster. Puede ser una máquina física o virtual que ejecuta los servicios de Kubernetes y es responsable de ejecutar contenedores. Cada nodo tiene su propia dirección IP y recursos (CPU, memoria, almacenamiento) que se comparten con los pods.

● Workloads (Cargas de Trabajo)

Una carga de trabajo (workload) es una aplicación que se ejecuta en Kubernetes. Ya sea que tu carga de trabajo sea un único componente o varios que trabajen juntos, en Kubernetes la ejecutas dentro de un conjunto de pods. En Kubernetes, un Pod representa un conjunto de contenedores en ejecución en tu clúster.

⚠ A no confundir con Worker node. Que sería el equivalente a un agente en otros softwares

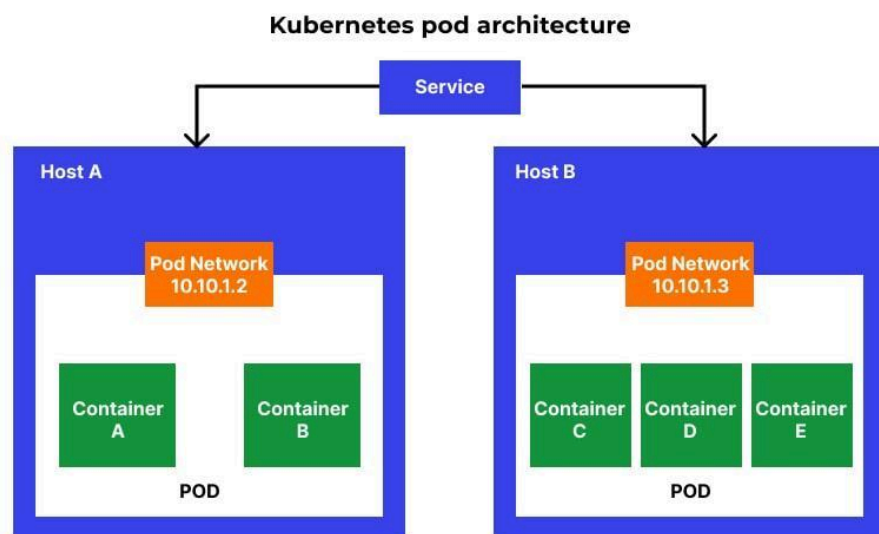
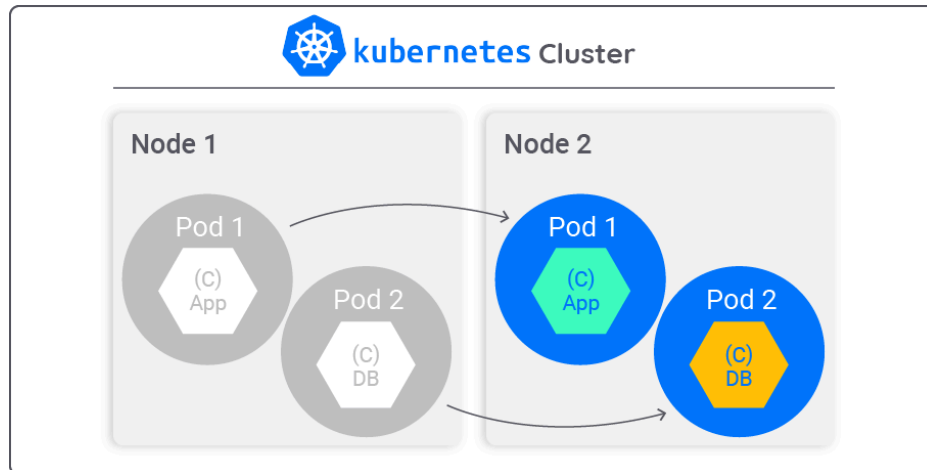


POD

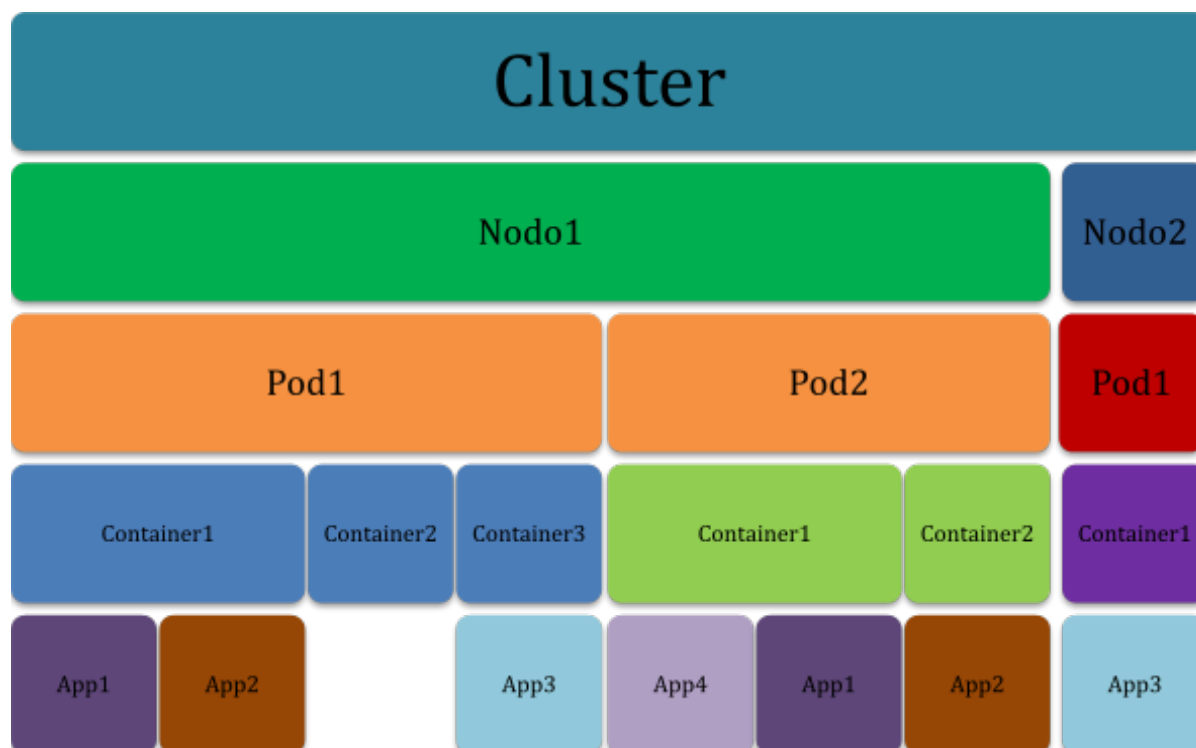
Un Pod es un conjunto de uno o más contenedores, con almacenamiento y recursos de red compartidos, y una especificación sobre cómo ejecutar los contenedores.

El contenido de un Pod siempre está ubicado y programado de manera conjunta, ejecutándose en un contexto compartido. Un Pod modela un "host lógico" específico de

la aplicación: contiene uno o más contenedores de aplicación que están relativamente acoplados. En contextos no basados en la nube, las aplicaciones ejecutadas en la misma máquina física o virtual son análogas a las aplicaciones en la nube ejecutadas en el mismo host lógico.



Resumiendo 📌



• 3.6.2. Componentes de K8s

Componentes del Master Server

Los componentes que forman el plano de control toman decisiones globales sobre el clúster (por ejemplo, la planificación) y detectan y responden a eventos del clúster, como la creación de un nuevo pod cuando la propiedad réplicas de un controlador de replicación no se cumple.

Estos componentes pueden ejecutarse en cualquier nodo del clúster. Sin embargo, para simplificar, los scripts de instalación típicamente se inician en el mismo nodo de forma exclusiva, sin que se ejecuten contenedores de los usuarios en esos nodos. El plano de control se ejecuta en varios nodos para garantizar la alta disponibilidad.

♦ kube-apiserver

El servidor de la API es el componente del plano de control de Kubernetes que expone la API de Kubernetes. Se trata del frontend de Kubernetes, recibe las peticiones y actualiza acordeamente el estado en etcd.

La principal implementación de un servidor de la API de Kubernetes es kube-apiserver. Es una implementación preparada para ejecutarse en alta

disponibilidad y que puede escalar horizontalmente para balancear la carga entre varias instancias.

♦ etcd

Almacén de datos persistente, consistente y distribuido de clave-valor utilizado para almacenar toda a la información del clúster de Kubernetes.

♦ kube-scheduler

Componente del plano de control que está pendiente de los Pods que no tienen ningún nodo asignado y seleccione uno donde ejecutarlo.

Para decidir en qué nodo se ejecutará el pod, se tienen en cuenta diversos factores: requisitos de recursos, restricciones de hardware/software/políticas, afinidad y anti-afinidad, localización de datos dependientes, entre otros.

♦ kube-controller-manager

Componente del plano de control que ejecuta los controladores de Kubernetes.

Lógicamente cada controlador es un proceso independiente, pero para reducir la complejidad, todos se compilan en un único binario y se ejecuta en un mismo proceso.

Estos controladores incluyen:

- Controlador de nodos: es el responsable de detectar y responder cuándo un nodo deja de funcionar
- Controlador de replicación: es el responsable de mantener el número correcto de pods para cada controlador de replicación del sistema
- Controlador de endpoints: construye el objeto Endpoints, es decir, hace una unión entre los Services y los Pods
- Controladores de tokens y cuentas de servicio: crean cuentas y tokens de acceso a la API por defecto para los nuevos Namespaces.
- cloud-controller-manager

Cloud-controller-manager ejecuta controladores que interactúan con proveedores de la nube. El binario cloud-controller-manager es una característica alpha que se introdujo en la versión 1.6 de Kubernetes.

Cloud-controller-manager sólo ejecuta ciclos de control específicos para cada proveedor de la nube. Es posible desactivar estos ciclos en kube-controller-manager pasando la opción `--cloud-provider=external` cuando se arranque el kube-controller-manager.

Cloud-controller-manager permite que el código de Kubernetes y el del proveedor de la nube evolucionen de manera independiente. Anteriormente, el código de Kubernetes dependía de la funcionalidad específica de cada proveedor de la

nube. En el futuro, el código que sea específico a una plataforma debería ser mantenido por el proveedor de la nube y enlazado a cloud-controller-manager al correr Kubernetes.

Los siguientes controladores dependen de alguna forma de un proveedor de la nube:

- Controlador de nodos: es el responsable de detectar y actuar cuándo un nodo deja de responder
- Controlador de rutas: para configurar rutas en la infraestructura de nube subyacente
- Controlador de servicios: para crear, actualizar y eliminar balanceadores de carga en la nube
- Controlador de volúmenes: para crear, conectar y montar volúmenes e interactuar con el proveedor de la nube para orquestarlos

Componentes de nodo

Los componentes de nodo corren en cada nodo, manteniendo a los pods en funcionamiento y proporcionando el entorno de ejecución de Kubernetes.

♦ kubelet

Agente que se ejecuta en cada nodo de un clúster. Se asegura de que los contenedores estén corriendo en un pod.

El agente kubelet toma un conjunto de especificaciones de [Pod](#), llamados PodSpecs, que han sido creados por Kubernetes y garantiza que los contenedores descritos en ellos estén funcionando y en buen estado.

♦ kube-proxy

kube-proxy permite abstraer un servicio en Kubernetes manteniendo las reglas de red en el anfitrión y haciendo reenvío de conexiones.

♦ Runtime de contenedores

El runtime de los contenedores es el software responsable de ejecutar los contenedores. Kubernetes soporta varios de ellos: Docker, containerd, cri-o, rktlet y cualquier implementación de la interfaz de runtime de contenedores de Kubernetes, o Kubernetes CRI.

Addons

Los addons son pods y servicios que implementan funcionalidades del clúster. Estos pueden ser administrados por Deployments, ReplicationControllers y otros. Los addons asignados a un espacio de nombres se crean en el espacio kube-system.

Más abajo se describen algunos addons. Para una lista más completa de los addons disponibles, por favor visite Addons.

♦ DNS

Si bien los otros addons no son estrictamente necesarios, todos los clústers de Kubernetes deberían tener un DNS interno del clúster ya que la mayoría de los ejemplos lo requieren.

El DNS interno del clúster es un servidor DNS, adicional a los que ya podrías tener en tu red, que sirve registros DNS a los servicios de Kubernetes.

Los contenedores que son iniciados por Kubernetes incluyen automáticamente este servidor en sus búsquedas DNS.

♦ Interfaz Web (Dashboard)

El Dashboard es una interfaz Web de propósito general para clústeres de Kubernetes. Les permite a los usuarios administrar y resolver problemas que puedan presentar tanto las aplicaciones como el clúster.

♦ Monitor de recursos de contenedores

El Monitor de recursos de contenedores almacena de forma centralizada series de tiempo con métricas sobre los contenedores, y provee una interfaz para navegar estos datos.

♦ Registros del clúster

El mecanismo de registros del clúster está a cargo de almacenar los registros de los contenedores de forma centralizada, proporcionando una interfaz de búsqueda y navegación.

Objetos

Los Objetos de Kubernetes son entidades persistentes dentro del sistema de Kubernetes. Kubernetes utiliza estas entidades para representar el estado de tu clúster. Específicamente, pueden describir:

👉 **Qué aplicaciones corren en contenedores (y en qué nodos)**

👉 **Los recursos disponibles para dichas aplicaciones**

👉 **Las políticas acerca de cómo dichas aplicaciones se comportan, como las políticas de reinicio, actualización, y tolerancia a fallos**

Un objeto de Kubernetes es un "registro de intención" -- una vez que has creado el objeto, el sistema de Kubernetes se pondrá en marcha para asegurar que el objeto existe. Al crear un objeto, en realidad le estás diciendo al sistema de Kubernetes cómo

quieres que sea la carga de trabajo de tu clúster; esto es, el **estado deseado** de tu clúster.

Para trabajar con los objetos de Kubernetes -- sea para crearlos, modificarlos, o borrarlos -- necesitarás usar la API de Kubernetes. Cuando utilizas el interfaz de línea de comandos **kubectl**, por ejemplo, este realiza las llamadas necesarias a la API de Kubernetes en tu lugar. También puedes usar la API de Kubernetes directamente desde tus programas utilizando alguna de las Librerías de Cliente.

Alcance y Estado de un Objeto

Cada objeto de Kubernetes incluye dos campos como objetos anidados que determinan la configuración del objeto: el campo de objeto **spec** y el campo de objeto **status**. El campo **spec**, que es obligatorio, describe el estado deseado del objeto -- las características que quieres que tenga el objeto. El campo **status** describe el estado actual del objeto, y se suministra y actualiza directamente por el sistema de Kubernetes. En cualquier momento, el Plano de Control de Kubernetes gestiona de forma activa el estado actual del objeto para que coincida con el estado deseado requerido.

Por ejemplo, un Deployment de Kubernetes es un objeto que puede representar una aplicación de tu clúster. Cuando creas el Deployment, puedes especificar en el **spec** del Deployment que quieres correr tres réplicas de la aplicación. El sistema de Kubernetes lee el **spec** del Deployment y comienza a instanciar réplicas de tu aplicación -- actualizando el estado para conciliarlo con tu **spec**. Si cualquiera de las instancias falla (un cambio de estado), el sistema de Kubernetes soluciona la diferencia entre la **spec** y el estado llevando a cabo una corrección -- en este caso, iniciando otra instancia de reemplazo.

Describir un Objeto de Kubernetes

Cuando creas un objeto en Kubernetes, debes especificar la **spec** del objeto que describe su estado deseado, así como información básica del mismo (como el nombre). Cuando usas la API de Kubernetes para crear el objeto (bien de forma directa o usando **kubectl**), dicha petición a la API debe incluir toda la información en formato JSON en el cuerpo de la petición. **A menudo, le proporcionas la información a kubectl como un archivo .yaml.** **kubectl** convierte esa información a JSON cuando realiza la llamada a la API.

Aquí hay un ejemplo de un archivo **.yaml** que muestra los campos requeridos y la **spec** del objeto Deployment de Kubernetes:

```
apiVersion: apps/v1 # Usa apps/v1beta2 para versiones anteriores a
1.9.0
kind: Deployment
metadata:
  name: nginx-deployment
spec:
```

```
selector:
  matchLabels:
    app: nginx
replicas: 2 # indica al controlador que ejecute 2 pods
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
          - containerPort: 80
```

Una forma de crear un Deployment utilizando un archivo .yaml como el indicado arriba sería ejecutar el comando `kubectl apply` en el interfaz de línea de comandos, pasándole el archivo .yaml como argumento. Aquí tienes un ejemplo de cómo hacerlo:

```
kubectl apply -f
https://k8s.io/examples/application/deployment.yaml --record
```

La salida del comando sería algo parecido a esto:

```
deployment.apps/nginx-deployment created
```

Campos requeridos

En el archivo .yaml del objeto de Kubernetes que quieras crear, obligatoriamente tendrás que indicar los valores de los siguientes campos (como mínimo):

- `apiVersion` - Qué versión de la API de Kubernetes estás usando para crear este objeto
- `kind` - Qué clase de objeto quieres crear
- `metadata` - Datos que permiten identificar unívocamente al objeto, incluyendo una cadena de texto para el `name`, `UID`, y opcionalmente el `namespace`

También deberás indicar el campo `spec` del objeto. El formato del campo `spec` es diferente según el tipo de objeto de Kubernetes, y contiene campos anidados específicos de cada objeto.

Un objeto de Kubernetes es un "registro de intención" -- una vez que has creado el objeto, el sistema de Kubernetes se pondrá en marcha para asegurar que el objeto

existe. Al crear un objeto, en realidad le estás diciendo al sistema de Kubernetes cómo quieres que sea la carga de trabajo de tu clúster; esto es, el estado deseado de tu clúster.



K8s Object Management

La herramienta de línea de comandos kubectl admite varias formas diferentes de crear y gestionar objetos en Kubernetes.

Advertencia: Un objeto de Kubernetes debe gestionarse utilizando únicamente una técnica. Mezclar y combinar técnicas para el mismo objeto resulta en un comportamiento no definido.

Técnica de Administración	Opera en	Entorno recomendado	Escritores soportados	Curva de aprendizaje
Comandos Imperativos	Objetos vivos	Desarrollo	1+	baja
Configuración de objetos imperativa	Archivos individuales	Producción	1	moderada
Configuración de objetos declarativa	Archivos de un directorio	Producción	1+	alta

Comandos Imperativos

Cuando se utilizan comandos imperativos, un usuario opera directamente sobre objetos activos en un clúster. El usuario proporciona operaciones al comando kubectl como argumentos o banderas.

Esta es la forma recomendada de comenzar o ejecutar una tarea puntual en un clúster. Debido a que esta técnica opera directamente sobre objetos activos, no proporciona un historial de configuraciones anteriores.

```
kubectl create deployment nginx --image nginx
```

✓ Pros:

- Se expresan como una única palabra de acción.
- Requieren un paso simple para hacer cambios en el clúster

✗ Contrás:

- No se integran con proceso de revisión de cambios.
- No proveen un registro de auditoría asociado con el cambio.
- No proveen un registro de orígenes excepto por lo que esta live.
- No proveen un template para crear nuevos objetos.

Objetos imperativos

En la configuración imperativa de objetos, el comando kubectl especifica la operación (crear, reemplazar, etc.), banderas opcionales y al menos un nombre de archivo. El archivo especificado debe contener una definición completa del objeto en formato YAML o JSON.

```
kubectl create -f nginx.yaml
```

✓ Pros:

- La configuración de objetos puede almacenarse en un sistema de control de origen como Git.
- La configuración de objetos puede integrarse con procesos, como revisar cambios antes de la confirmación y mantener un registro de auditoría.
- La configuración de objetos proporciona una plantilla para la creación de nuevos objetos.
- La configuración imperativa de objetos tiene un comportamiento más simple y es más fácil de entender que la declarativa.
- A partir de la versión 1.5 de Kubernetes, la configuración imperativa de objetos es más madura.

✗ Contrás:

- La configuración de objetos requiere un entendimiento básico del esquema del objeto.
- Además, implica el paso adicional de escribir un archivo YAML.
- La configuración imperativa de objetos funciona mejor en archivos, no en directorios.
- Las actualizaciones en objetos activos deben reflejarse en los archivos de configuración, o se perderán durante la próxima sustitución.

Objetos Declarativos

Cuando se utiliza la configuración declarativa de objetos, un usuario opera en archivos de configuración de objetos almacenados localmente; sin embargo, el usuario no define las operaciones a realizar en los archivos. Las operaciones de creación, actualización y eliminación son detectadas automáticamente por objeto a través de kubectl. Esto

permite trabajar en directorios, donde diferentes operaciones pueden ser necesarias para diferentes objetos.

```
kubectl diff -f configs/
kubectl apply -f configs/
```

✓ Pros:

- Los cambios realizados directamente en objetos activos se conservan, incluso si no se fusionan de nuevo en los archivos de configuración.
- La configuración declarativa de objetos tiene un mejor soporte para operar en directorios y detectar automáticamente los tipos de operación (crear, parchear, eliminar) por objeto.

✗ Contras:

- La configuración declarativa de objetos puede ser más difícil de depurar, y entender los resultados puede volverse complicado cuando son inesperados.
- Las actualizaciones parciales mediante diferencias generan operaciones de fusión y parcheo complejas.

• 3.6.3. Configuración de Contenedores

ConfigMaps

Un ConfigMap es un objeto de API utilizado para almacenar datos no confidenciales en pares de clave-valor. Los pods pueden consumir ConfigMaps como variables de entorno, argumentos de línea de comandos o como archivos de configuración en un volumen.

Un ConfigMap te permite desvincular la configuración específica del entorno de tus imágenes de contenedor, de modo que tus aplicaciones sean fácilmente transportables.

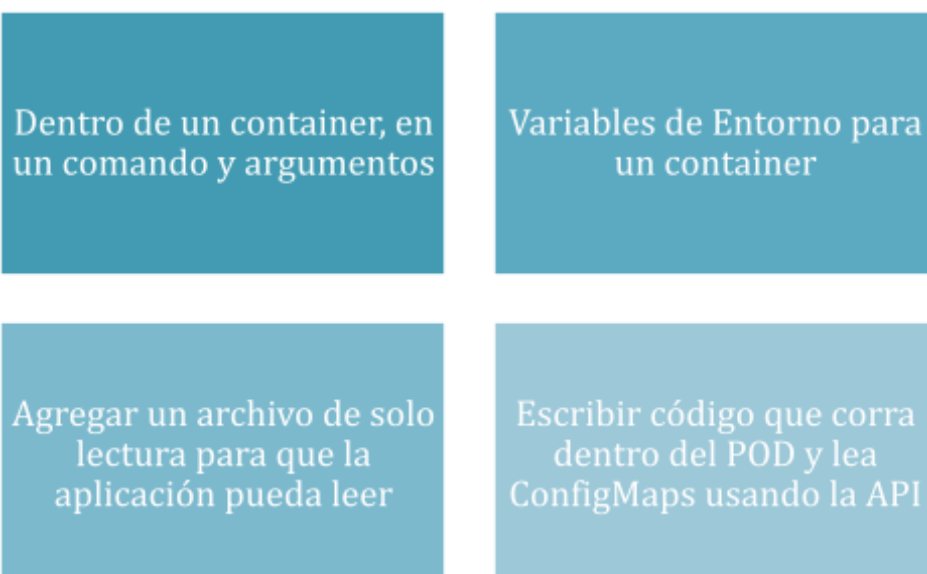
⚠ Cuidado: Un ConfigMap no provee encriptación ni datos secretos.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: "3"
  ui_properties_file_name: "user-interface.properties"

  # file-like keys
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
  user-interface.properties: |
```

```
color.good=purple
color.bad=yellow
allow.textmode=true
```

Utilización



```
volumes:
  # Configurar volúmenes a nivel de Pod, luego montar en los
  contenedores dentro de ese Pod.
  - name: config
    configMap:
      # Proporciona el nombre del ConfigMap que deseas montar.
      name: game-demo
      # Un conjunto de claves del ConfigMap para crear como
      archivos.
      items:
        - key: "game.properties"
          path: "game.properties"
        - key: "user-interface.properties"
          path: "user-interface.properties"
```

ConfigMap Inmutable

- Protege contra actualizaciones accidentales (o no deseadas) que podrían causar interrupciones en las aplicaciones

- Mejora el rendimiento de su clúster al reducir significativamente la carga en kube-apiserver al cerrar observadores para ConfigMaps marcados como inmutables.

```
apiVersion: v1
kind: ConfigMap
metadata:
  ...
data:
  ...
immutable: true
```

Secrets

Un Secret (secreto) es un objeto que contiene una pequeña cantidad de datos sensibles, como una contraseña, un token o una clave. Esta información de otro modo podría colocarse en una especificación de Pod o en una imagen de contenedor. El uso de un Secret significa que no necesitas incluir datos confidenciales en el código de tu aplicación.

Dado que los Secrets pueden crearse de manera independiente a los Pods que los utilizan, hay menos riesgo de que el Secret (y sus datos) se exponga durante el flujo de trabajo de creación, visualización y edición de Pods. Kubernetes y las aplicaciones que se ejecutan en tu clúster también pueden tomar precauciones adicionales con los Secrets, como evitar escribir datos sensibles en almacenamiento no volátil.

Los Secrets son similares a los ConfigMaps pero están específicamente destinados a contener datos confidenciales.

⚠ Advertencia: Por defecto, los Secrets de Kubernetes se almacenan sin cifrar en el almacén de datos subyacente del servidor API (etc). Cualquier persona con acceso a la API puede recuperar o modificar un Secret, y lo mismo puede hacer cualquier persona con acceso a etcd. Además, cualquier persona autorizada para crear un Pod en un espacio de nombres puede utilizar ese acceso para leer cualquier Secret en ese espacio de nombres; esto incluye acceso indirecto, como la capacidad de crear un Despliegue.

Para utilizar Secrets de manera segura, realiza al menos los siguientes pasos:

1. Habilita el cifrado en reposo para Secrets.
2. Habilita o configura reglas de RBAC con acceso de privilegio mínimo a Secrets.
3. Restringe el acceso a Secretos a contenedores específicos.
4. Considera el uso de proveedores externos de almacenamiento de Secrets.

Usos de Secrets

1. Establecer variables de entorno para un contenedor.
2. Proporcionar credenciales, como claves SSH o contraseñas, a Pods.
3. Permitir que el kubelet extraiga imágenes de contenedor desde registros privados.

Supongamos que deseas almacenar un nombre de usuario y una contraseña como Secret para utilizarlos en un Pod, los guardamos en base64 para este ejemplo de manera que no se puedan leer directamente por un humano sin un decoder.

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: RGFuaWVsIE0gU2FsYXphcg==
  password: ZXN0ZSB1cyBtaSBwYXNzd29yZA==
```

Para utilizar el secret en un pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mycontainer
      image: tu-imagen:version
      env:
        - name: MY_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
        - name: MY_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: password
```

Tipos de Secrets

Tipo integrado	Uso
Opaque	Datos de usuario
kubernetes.io/service-account-token	ServiceAccount token
kubernetes.io/dockercfg	serialized ~/.dockercfg file
kubernetes.io/dockerconfigjson	serialized ~/.docker/config.json file
kubernetes.io/basic-auth	credenciales para autenticación básica
kubernetes.io/ssh-auth	credenciales para autenticación SSH
kubernetes.io/tls	Datos para un cliente o servidor TLS
bootstrap.kubernetes.io/token	Datos de token de arranque

● 3.6.4. Políticas de Networking

La conectividad de las redes para contenedores de Windows se expone a través de complementos CNI (Container Network Interface). Los contenedores de Windows funcionan de manera similar a las máquinas virtuales en lo que respecta a la red. Cada contenedor tiene un adaptador de red virtual (vNIC) que está conectado a un conmutador virtual Hyper-V (vSwitch). El servicio de red del host (HNS) y el servicio de cómputo del host (HCS) trabajan juntos para crear contenedores y vincular vNICs de contenedor a redes. HCS se encarga de la gestión de contenedores, mientras que HNS se encarga de la gestión de recursos de red, tales como:

- Redes virtuales (incluida la creación de vSwitches).
- Puntos finales / vNICs.
- Espacios de nombres.
- Políticas que incluyen encapsulaciones de paquetes, reglas de equilibrio de carga, listas de control de acceso (ACL), y reglas de NAT.

HNS y vSwitch en Windows implementan la creación de espacios de nombres y pueden crear NIC virtuales según sea necesario para un pod o contenedor. Sin embargo, muchas configuraciones, como DNS, rutas y métricas, se almacenan en la base de datos del registro de Windows en lugar de en archivos dentro de /etc, que es la forma en que Linux almacena esas configuraciones. El registro de Windows para el contenedor es independiente del del host, por lo que conceptos como mapear /etc/resolv.conf desde el host a un contenedor no tienen el mismo efecto que tendrían en Linux. Estos deben configurarse utilizando las API de Windows ejecutadas en el contexto de ese

contenedor. Por lo tanto, las implementaciones de CNI deben llamar a HNS en lugar de depender de mapeos de archivos para transmitir detalles de red al pod o contenedor.

Windows admite cinco controladores/modes de red diferentes: L2bridge, L2tunnel, Overlay (Beta), Transparente y NAT. En un clúster heterogéneo con nodos de trabajo Windows y Linux, es necesario seleccionar una solución de red compatible con ambos sistemas operativos.

Tipos de red admitidos por K8s en Windows

Controlador de Red	Descripción	Modificaciones de Paquetes del Contenedor	Complementos de Red	Características del Complemento de Red
L2bridge	Los contenedores están conectados a un vSwitch externo. Los contenedores están conectados a la red subyacente, aunque la red física no necesita aprender las MAC de los contenedores porque se reescriben en entrada/salida.	MAC se reescribe a la MAC del host, la IP puede reescribirse a la IP del host usando la política HNS OutboundNAT.	win-bridge, Azure-CNI, Flannel host-gateway utiliza win-bridge	win-bridge utiliza el modo de red L2bridge, conecta contenedores a la infraestructura subyacente de los hosts, ofreciendo el mejor rendimiento. Requiere rutas definidas por el usuario (UDR) para la conectividad entre nodos.
L2Tunnel	Este es un caso especial de L2bridge, pero solo se usa en Azure. Todos los paquetes se envían al host de virtualización donde se aplica la política SDN.	MAC reescrita, IP visible en la red subyacente	Azure-CNI	Azure-CNI permite la integración de contenedores con Azure vNET y les permite aprovechar el conjunto de capacidades que proporciona Azure Virtual Network. Por ejemplo, conectarse de manera segura a los servicios de Azure o utilizar los NSG de Azure. Ver azure-cni para algunos ejemplos.
Overlay	Los contenedores tienen un vNIC conectado a un vSwitch externo. Cada red de superposición obtiene su propia subred IP, definida por un prefijo IP personalizado. El controlador de red de superposición utiliza la encapsulación VXLAN.	Encapsulado con un encabezado externo.	win-overlay, Flannel VXLAN (utiliza win-overlay)	win-overlay debe usarse cuando se deseen redes de contenedores virtuales aisladas de la infraestructura subyacente de los hosts (por ejemplo, por razones de seguridad). Permite reutilizar IPs para diferentes redes de superposición (que tienen diferentes etiquetas VNID) si hay restricciones de IPs en su centro de datos. Esta opción requiere KB4489899 en Windows Server 2019.
Transparente (caso especial para ovn-kubernetes)	Requiere un vSwitch externo. Los contenedores están conectados a un vSwitch externo que permite la comunicación intra-pod a través de redes lógicas	El paquete se encapsula ya sea a través de túneles GENEVE o STT para llegar a los pods que no están en el mismo host. Los paquetes se reenvían o descartan mediante la	ovn-kubernetes	Implementar a través de Ansible. Se pueden aplicar ACL distribuidas mediante políticas de Kubernetes. Soporte para IPAM. Se puede lograr el equilibrio de carga sin kube-proxy. La traducción de

Controlador de Red	Descripción	Modificaciones de Paquetes del Contenedor	Complementos de Red	Características del Complemento de Red
	(interruptores y enrutadores lógicos).	información de metadatos del túnel suministrada por el controlador de red ovn.		direcciones se realiza sin usar iptables/netsh.
NAT (no utilizado en Kubernetes)	Los contenedores tienen un vNIC conectado a un vSwitch interno. DNS/DHCP se proporciona utilizando un componente interno llamado WinNAT	MAC e IP se reescriben a la MAC/IP del host.	nat	Incluido aquí por completitud

Cluster Networking

La red es una parte central de Kubernetes, pero puede resultar desafiante entender exactamente cómo se espera que funcione. Hay 4 problemas distintos de red que abordar:

Comunicaciones altamente acopladas de contenedor a contenedor: esto se resuelve mediante Pods y comunicaciones locales.

- Comunicaciones de Pod a Pod: este es el enfoque principal de este documento.
- Comunicaciones de Pod a Servicio: esto está cubierto por los Servicios.
- Comunicaciones externas a Servicio: esto también está cubierto por los Servicios.

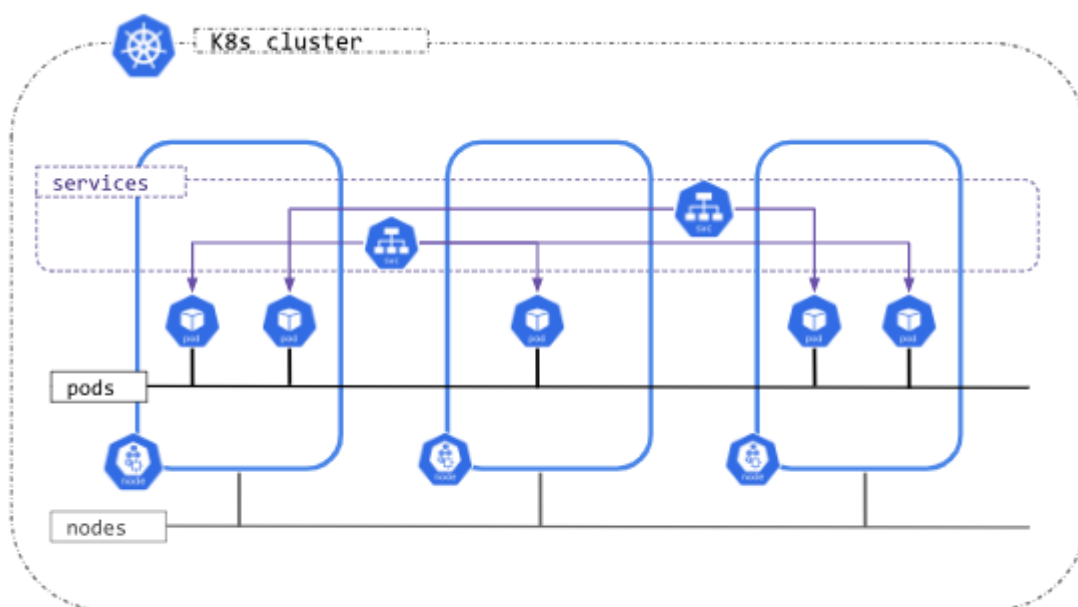
Kubernetes se trata de compartir máquinas entre aplicaciones. Normalmente, compartir máquinas requiere garantizar que dos aplicaciones no intenten usar los mismos puertos. Coordinar puertos entre múltiples desarrolladores es muy difícil de hacer a escala y expone a los usuarios a problemas a nivel de clúster fuera de su control.

La asignación dinámica de puertos trae consigo muchas complicaciones al sistema: cada aplicación tiene que tomar puertos como banderas, los servidores de API tienen que saber cómo insertar números de puerto dinámicos en bloques de configuración, los servicios tienen que saber cómo encontrarse, etc. En lugar de lidiar con esto, Kubernetes adopta un enfoque diferente.

Rangos de direcciones IP de Kubernetes

Los clústeres de Kubernetes requieren asignar direcciones IP no superpuestas para Pods, Servicios y Nodos, desde un rango de direcciones disponibles configurado en los siguientes componentes:

- El complemento de red está configurado para asignar direcciones IP a los Pods.
- El kube-apiserver está configurado para asignar direcciones IP a los Servicios.
- El kubelet o el cloud-controller-manager están configurados para asignar direcciones IP a los Nodos.



Tipos de red del clúster

Los clústeres de Kubernetes, según las familias de IP configuradas, se pueden categorizar en:

- Solo IPv4: El complemento de red, kube-apiserver y kubelet/cloud-controller-manager están configurados para asignar solo direcciones IPv4.
- Solo IPv6: El complemento de red, kube-apiserver y kubelet/cloud-controller-manager están configurados para asignar solo direcciones IPv6.
- Doble pila IPv4/IPv6 o IPv6/IPv4:
 - El complemento de red está configurado para asignar direcciones IPv4 e IPv6.
 - El kube-apiserver está configurado para asignar direcciones IPv4 e IPv6.
 - El kubelet o cloud-controller-manager está configurado para asignar direcciones IPv4 e IPv6.
 - Todos los componentes deben estar de acuerdo con la familia de IP principal configurada.

Los clústeres de Kubernetes solo consideran las familias de IP presentes en los objetos Pods, Servicios y Nodos, independientemente de las IPs existentes de los objetos representados. Por ejemplo, un servidor o un pod puede tener múltiples direcciones IP en sus interfaces, pero solo se consideran las direcciones IP en `node.status.addresses` o `pod.status.ips` para implementar el modelo de red de Kubernetes y definir el tipo de clúster.

Cómo implementar el modelo de red de Kubernetes

El modelo de red se implementa mediante el tiempo de ejecución de contenedores en cada nodo. Los tiempos de ejecución de contenedores más comunes utilizan complementos de Interfaz de Red de Contenedor (CNI) para gestionar su red y capacidades de seguridad. Existen muchos complementos CNI diferentes de diversos proveedores. Algunos de ellos proporcionan solo funciones básicas de agregar y eliminar interfaces de red, mientras que otros ofrecen soluciones más sofisticadas, como la integración con otros sistemas de orquestación de contenedores, ejecución de múltiples complementos CNI, funciones avanzadas de IPAM, etc.

¿Qué son las Network Policies?

En Kubernetes, las Network Policies son un mecanismo que permite definir reglas para controlar el tráfico de red entre diferentes Pods en un clúster. Estas políticas proporcionan una forma de especificar qué Pods pueden comunicarse entre sí y qué tipo de tráfico está permitido o bloqueado. Las Network Policies permiten segmentar y restringir el tráfico de red según los requisitos de seguridad y la arquitectura de la aplicación.

♦ Definición De Reglas

Las Network Policies permiten definir reglas que especifican cómo los Pods seleccionados pueden comunicarse entre sí. Las reglas pueden basarse en criterios como las etiquetas de los Pods, los puertos y los protocolos de red.

♦ Selector de Pods

Las Network Policies utilizan un selector para especificar qué Pods están sujetos a las reglas de la política. Esto permite aplicar políticas a conjuntos específicos de Pods en función de las etiquetas asignadas a ellos.

♦ Ingress y Egress

Las Network Policies pueden definir reglas tanto para el tráfico de entrada (Ingress) como para el tráfico de salida (Egress). Esto significa que puedes controlar no solo quién puede comunicarse con un Pod, sino también a dónde puede ir el tráfico desde ese Pod.

♦ Puertos y Protocolos

Puedes especificar puertos y protocolos permitidos o bloqueados en las reglas de la Network Policy. Esto proporciona un control granular sobre qué tipo de tráfico es permitido entre los Pods.

♦ Default Deny

Si no se especifica ninguna Network Policy, Kubernetes sigue la política de "Default Deny", lo que significa que todo el tráfico está bloqueado por defecto. Las Network Policies permiten abrir selectivamente las comunicaciones según las reglas definidas.

♦ Implementación por el Plugin de Red

La implementación exacta de las Network Policies depende del proveedor del plugin de red utilizado en el clúster. Cada plugin de red puede tener su propia manera de implementar las Network Policies, aunque el modelo general de reglas y selectores es consistente.

♦ Compatibilidad con CNI

Las Network Policies están diseñadas para ser independientes del proveedor del plugin de red y son parte del estándar CNI (Container Network Interface), lo que significa que son compatibles con varios plugins de red.

♦ Herramientas y Visualización

Existen herramientas y comandos en la línea de comandos, como `kubectl`, que permiten visualizar y gestionar las Network Policies en un clúster de Kubernetes.

♦ Escenarios de Uso

Las Network Policies son útiles en escenarios donde se requiere un control preciso sobre el tráfico de red entre Pods, como aplicaciones sensibles a la seguridad,

segmentación de entornos de desarrollo y producción, o limitación del acceso a servicios específicos.

Para implementar y gestionar Network Policies, es importante revisar la documentación específica del proveedor del plugin de red utilizado en tu clúster, ya que las implementaciones pueden variar.

Ejemplo de recurso de networkpolicy.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 172.17.0.0/16
            except:
              - 172.17.1.0/24
        - namespaceSelector:
            matchLabels:
              project: myproject
        - podSelector:
            matchLabels:
              role: frontend
      ports:
        - protocol: TCP
          port: 6379
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/24
      ports:
        - protocol: TCP
          port: 5978
```

Selectores From y To

Hay cuatro tipos de selectores que se pueden especificar en una sección `from` de un `ingress` o en una sección `to` de un `egress`:

♦ **podSelector**

Esto selecciona Pods específicos en el mismo espacio de nombres (namespace) que la NetworkPolicy, los cuales deben permitirse como fuentes de entrada o destinos de salida.

♦ **namespaceSelector**

Esto selecciona espacios de nombres específicos para los cuales todos los Pods deben permitirse como fuentes de entrada o destinos de salida.

♦ **namespaceSelector y podSelector**

Una única entrada `to/from` que especifica tanto `namespaceSelector` como `podSelector` selecciona Pods específicos dentro de espacios de nombres particulares. Ten cuidado de usar la sintaxis YAML correcta. Por ejemplo:

```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: alice
    podSelector:
      matchLabels:
        role: client
...

```

Esta política contiene un único elemento `from` que permite conexiones desde Pods con la etiqueta `role=client` en espacios de nombres con la etiqueta `user=alice`. Pero la siguiente política es diferente:

```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: alice
  - podSelector:
      matchLabels:
        role: client
...

```

Contiene dos elementos en el array ``from`` y permite conexiones desde Pods en el Namespace local con la etiqueta ``role=client``, o desde cualquier Pod en cualquier Namespace con la etiqueta ``user=alice``.

Cuando tengas dudas, utiliza ``kubectl describe`` para ver cómo Kubernetes ha interpretado la política.

♦ **IpBlock**

Esto selecciona rangos de direcciones IP CIDR específicos para permitir como fuentes de entrada o destinos de salida. Estas deben ser IPs externas al clúster, ya que las IPs de los Pods son efímeras e impredecibles.

Los mecanismos de entrada y salida del clúster a menudo requieren la reescritura de la IP de origen o destino de los paquetes. En casos donde esto sucede, no está definido si esto ocurre antes o después del procesamiento de NetworkPolicy, y el comportamiento puede ser diferente para diferentes combinaciones de plugins de red, proveedores en la nube, implementaciones de Servicio, etc.

En el caso de la entrada, esto significa que en algunos casos podrías filtrar paquetes entrantes basándote en la IP de origen original real, mientras que en otros casos, la "IP de origen" en la que actúa la NetworkPolicy podría ser la IP de un LoadBalancer o del nodo del Pod, etc.

Para la salida, esto significa que las conexiones desde los Pods a las IPs de Servicio que se reescriben a IPs externas al clúster pueden o no estar sujetas a políticas basadas en ``ipBlock``.

● **3.6.5. Escalabilidad K8s**

La escalabilidad en Kubernetes se refiere a la capacidad del sistema para manejar un crecimiento en la carga de trabajo y el tamaño del clúster de manera eficiente y efectiva. Kubernetes ha sido diseñado con un enfoque en la escalabilidad, permitiendo a los usuarios escalar tanto horizontal como verticalmente para satisfacer las demandas cambiantes de sus aplicaciones.

Escalabilidad Horizontal:

Kubernetes permite agregar o quitar nodos del clúster para adaptarse a la carga de trabajo. Esto se conoce como escalabilidad horizontal. Puedes agregar más nodos al clúster según sea necesario para distribuir la carga y mejorar el rendimiento.

Escalabilidad Vertical:

También es posible escalar verticalmente los recursos de un pod individual, aumentando la capacidad de CPU o memoria según sea necesario. Sin embargo, la escalabilidad vertical tiene límites prácticos y, en general, se prefiere la escalabilidad horizontal para garantizar una mayor flexibilidad y resistencia.

Autoescalado (Autoscaling):

Kubernetes ofrece funcionalidades de autoescalado que permiten que el clúster ajuste automáticamente el número de réplicas de los pods en función de la carga actual. Puedes configurar políticas de autoescalado basadas en métricas como el uso de CPU o memoria.

Escalabilidad de Servicios y Aplicaciones:

Los servicios y las aplicaciones desplegados en Kubernetes pueden escalar horizontalmente mediante la replicación de pods. Esto asegura la disponibilidad y la distribución de la carga entre los pods.

Escalabilidad del Panel de Control:

Kubernetes incluye un panel de control que facilita la administración y supervisión del clúster. La escalabilidad del panel de control es importante para garantizar un rendimiento óptimo, especialmente en clústeres grandes.

Escalabilidad de Almacenamiento:

La escalabilidad del almacenamiento es crucial para manejar grandes cantidades de datos. Kubernetes admite diferentes tipos de almacenamiento y provee opciones para escalar el almacenamiento según sea necesario.

Escalabilidad de Red:

La arquitectura de red en Kubernetes está diseñada para ser escalable. La capacidad de escalar la red es esencial para garantizar una comunicación eficiente entre los nodos del clúster y los pods.

Despliegue Distribuido:

Kubernetes permite distribuir aplicaciones y servicios en diferentes nodos del clúster, lo que mejora la disponibilidad y la capacidad de recuperación.

Cómo escalar un pod a N instancias:

Replicas

Una "réplica" se refiere a una instancia idéntica de un conjunto de contenedores. La idea fundamental detrás del uso de réplicas es la de proporcionar alta disponibilidad, escalabilidad y resiliencia a las aplicaciones.

Cuando despliegas una aplicación en Kubernetes, generalmente definen un conjunto de réplicas para garantizar que la aplicación esté siempre disponible y pueda manejar la carga de trabajo. Estas réplicas son instancias idénticas de la misma aplicación (o pod) que se ejecutan simultáneamente.

Deployment: Un objeto de Kubernetes que administra la creación y escalabilidad de un conjunto de réplicas de pods. Un Deployment permite declarativamente definir el estado deseado de la aplicación.

ReplicaSet: Un objeto de Kubernetes que garantiza que un número especificado de réplicas de un pod esté siempre en ejecución. Los Deployments utilizan ReplicaSets internamente para mantener el número deseado de réplicas.

Cómo escalar las réplicas de un POD

Puedes escalar el número de réplicas de un pod en Kubernetes usando el comando `kubectl`.

```
kubectl scale --replicas=3 deployment/nombre-del-deployment
```

Este comando escala el número de réplicas del deployment llamado "nombre-del-deployment" a 3. Asegúrate de reemplazar "nombre-del-deployment" con el nombre real de tu deployment.

Si en lugar de un Deployment, estás utilizando un conjunto de replicaset o directamente un pod, puedes ajustar el comando en consecuencia. Por ejemplo, si estás escalando un replicaset, el comando se vería así:

```
kubectl scale --replicas=3 replicaset/nombre-del-replicaset
```

O si estás trabajando directamente con un pod (aunque esto no es común para escalabilidad, ya que generalmente se gestiona mediante replicaset o deployments), podrías hacerlo así:

```
kubectl scale --replicas=3 pod/nombre-del-pod
```

● 3.6.6. Actualización de Imágenes, Clusters y Nodos

La actualización de imágenes en Kubernetes es un proceso crucial para garantizar que las aplicaciones se ejecuten con las últimas versiones de los contenedores. Aquí hay algunos conceptos clave relacionados con la actualización de imágenes en clústeres y nodos Kubernetes:

Imágenes de Contenedor

Una imagen de contenedor es un paquete ligero y ejecutable que incluye todo lo necesario para ejecutar una aplicación, incluidos el código, las bibliotecas y las dependencias. En Kubernetes, las aplicaciones se empaquetan y distribuyen en contenedores, y estas imágenes pueden actualizarse para incluir correcciones de errores, mejoras de seguridad o nuevas características.

Despliegue Declarativo

En Kubernetes, la actualización de imágenes se realiza típicamente de manera declarativa. Los recursos como Deployments o StatefulSets permiten definir el estado deseado de la aplicación, incluida la imagen del contenedor. Al actualizar la definición del recurso con la nueva versión de la imagen, Kubernetes se encarga de implementar los cambios necesarios para alcanzar ese estado.

Actualización de Deployment

Un Deployment es un recurso en Kubernetes que facilita la actualización de aplicaciones de manera controlada. Al modificar la definición del Deployment con una nueva versión de la imagen, Kubernetes realiza una actualización por lotes, reemplazando gradualmente las réplicas antiguas con las nuevas, lo que minimiza el impacto en la disponibilidad.

```
kubectl set image deployment/nombre-del-deployment
contenedor=nombre-de-la-nueva-imagen:version
```

Rolling Update (Actualización Gradual)

La estrategia predeterminada para las actualizaciones de Deployment es un "rolling update" o actualización gradual. Esto significa que las réplicas nuevas se crean y las antiguas se eliminan gradualmente, garantizando una transición suave sin interrupciones de servicio.

Actualización de StatefulSet:

Para aplicaciones que requieren identificadores persistentes, como bases de datos, se utiliza un StatefulSet en lugar de un Deployment. La actualización de un StatefulSet sigue un proceso similar al de un Deployment, pero garantiza la persistencia de los identificadores.

```
kubectl set image statefulset/nombre-del-statefulset
contenedor=nombre-de-la-nueva-imagen:version
```

Imágenes en Nodos:

Las imágenes de contenedor se almacenan en un registro de contenedor y se descargan en los nodos del clúster cuando se necesitan. Los nodos mantienen en caché las imágenes descargadas localmente para evitar descargarlas cada vez que se inicia un pod.

Gestión de Imágenes en Nodos:

Los nodos de Kubernetes pueden gestionar la limpieza de imágenes no utilizadas para evitar que el almacenamiento se llene con imágenes obsoletas. Sin embargo, es importante gestionar cuidadosamente el almacenamiento y la limpieza de imágenes, especialmente en entornos de producción.

Herramientas Adicionales:

Puedes utilizar herramientas como Helm para gestionar paquetes de Kubernetes, lo que facilita la actualización de imágenes y la gestión de versiones de aplicaciones.

```
helm upgrade nombre-del-release nombre-del-chart
```

● 3.6.7. Rollback de Contenedores

El rollback de contenedores en Kubernetes se refiere a revertir un despliegue o actualización de una aplicación a una versión anterior debido a problemas o errores detectados después de una actualización. Kubernetes proporciona mecanismos para realizar rollbacks de manera controlada y sin interrupciones significativas en el servicio. Aquí hay algunos conceptos clave relacionados con el rollback de contenedores en Kubernetes:

Historial de Revisiones:

Kubernetes mantiene un historial de revisiones para los recursos de despliegue, como Deployment o StatefulSet. Cada vez que realizas una actualización, se crea una nueva revisión, lo que permite revertir a versiones anteriores si es necesario.

Rollback Declarativo:

El rollback en Kubernetes se realiza de manera declarativa, lo que significa que defines el estado deseado de la aplicación y Kubernetes se encarga de realizar los cambios necesarios para alcanzar ese estado.

Comando Rollback:

Para realizar un rollback, puedes utilizar el comando `kubectl rollout undo` seguido del nombre del recurso y, opcionalmente, la revisión a la que deseas retroceder. Si no especificas una revisión, Kubernetes retrocede a la revisión anterior.

```
kubectl rollout undo deployment/nombre-del-deployment
```

Revisión Específica:

Si conoces la revisión específica a la que deseas retroceder, puedes especificarla en el comando de rollback. Puedes ver las revisiones disponibles utilizando `kubectl rollout history`.

```
kubectl rollout undo deployment/nombre-del-deployment
--to-revision=número-de-revisión
```


Rollback Gradual:

Similar a las actualizaciones graduales, los rollbacks también se realizan de manera gradual. Kubernetes reemplaza las réplicas actuales con las de la revisión anterior, garantizando una transición suave sin interrupciones de servicio.

Condiciones de Rollback:

Kubernetes permite establecer condiciones de rollback basadas en métricas o eventos. Por ejemplo, puedes configurar Kubernetes para realizar automáticamente un rollback si la nueva versión de la aplicación no cumple con ciertas métricas de salud.

Retroceso de StatefulSets:

Para los StatefulSets, el rollback garantiza la consistencia en la persistencia de los identificadores de los pods. Kubernetes maneja el retroceso de manera que los identificadores persistentes se mantengan.

Gestión de Rollback:

Es importante gestionar cuidadosamente los rollbacks para evitar situaciones no deseadas. Antes de realizar un rollback, es recomendable analizar las revisiones disponibles y entender las razones del retroceso.

```
kubectl rollout history deployment/nombre-del-deployment
```

Automatización con Helm:

Si estás utilizando Helm para gestionar tus despliegues, puedes aprovechar las capacidades de Helm para realizar rollbacks de manera más eficiente.

```
helm rollback nombre-del-release número-de-revisión
```

● 3.6.8. Almacenamiento Persistente

Introducción

La gestión del almacenamiento es un problema distinto de la gestión de las instancias de cómputo. El subsistema de PersistentVolume proporciona una API para usuarios y administradores que abstrae los detalles sobre cómo se proporciona el almacenamiento de cómo se consume. Para lograr esto, introducimos dos nuevos recursos de API: PersistentVolume y PersistentVolumeClaim.

Un **PersistentVolume (PV)** es una porción de almacenamiento en el clúster que ha sido provisionada por un administrador o provisionada de forma dinámica mediante Storage Classes. Es un recurso en el clúster al igual que un nodo es un recurso del clúster. Los PV son plugins de volumen al igual que los Volumes, pero tienen un ciclo de vida independiente de cualquier Pod individual que utilice el PV. Este objeto API captura los detalles de la implementación del almacenamiento, ya sea NFS, iSCSI o un sistema de almacenamiento específico del proveedor en la nube.

Un **PersistentVolumeClaim (PVC)** es una solicitud de almacenamiento realizada por un usuario. Es similar a un Pod. Los Pods consumen recursos del nodo y los PVC consumen recursos del PV. Los Pods pueden solicitar niveles específicos de recursos (CPU y memoria). Las Claims pueden solicitar un tamaño específico y modos de acceso (por ejemplo, pueden ser montados como ReadWriteOnce, ReadOnlyMany, ReadWriteMany, o ReadWriteOncePod, ver AccessModes).

Si bien PersistentVolumeClaims permiten a un usuario consumir recursos de almacenamiento de manera abstracta, es común que los usuarios necesiten PersistentVolumes con propiedades variables, como rendimiento, para diferentes problemas. Los administradores del clúster deben poder ofrecer una variedad de PersistentVolumes que difieran en más aspectos que el tamaño y los modos de acceso, sin exponer a los usuarios los detalles de cómo se implementan esos volúmenes. Para estas necesidades, existe el recurso StorageClass.

Ciclo de Vida de un Volumen Persistente

Los PersistentVolumes (PVs) son recursos en el clúster. Los PersistentVolumeClaims (PVCs) son solicitudes de esos recursos y también actúan como comprobantes de reclamación para el recurso. La interacción entre los PVs y los PVCs sigue este ciclo de vida:

1. Provisión

♦ Estático

Un administrador del clúster crea varios PVs. Llevan los detalles del almacenamiento real, que está disponible para su uso por parte de los usuarios del clúster. Están presentes en la API de Kubernetes y están disponibles para su consumo.

♦ Dinámico

Cuando ninguno de los PVs estáticos creados por el administrador coincide con un PersistentVolumeClaim del usuario, el clúster puede intentar aprovisionar dinámicamente un volumen especialmente para ese PVC. Esta aprovisionamiento se basa en StorageClasses: el PVC debe solicitar una clase de almacenamiento y el administrador debe haber creado y configurado esa clase para que se produzca la aprovisionamiento dinámica. Las reclamaciones que solicitan la clase "" desactivan efectivamente la aprovisionamiento dinámica para sí mismas.

Para habilitar la aprovisionamiento dinámica de almacenamiento basada en la clase de almacenamiento, el administrador del clúster debe habilitar el controlador de admisión DefaultStorageClass en el servidor API. Esto se puede hacer, por ejemplo, asegurándose de que DefaultStorageClass esté entre la lista de valores delimitados por comas y ordenados para la bandera --enable-admission-plugins del componente servidor API. Para obtener más información sobre las banderas de la línea de comandos del servidor API, consulta la documentación de kube-apiserver.

2. Enlace

Un usuario crea, o en el caso de la aprovisionamiento dinámica, ya ha creado, un PersistentVolumeClaim con una cantidad específica de almacenamiento solicitado y con ciertos modos de acceso. Un bucle de control en el plano de control observa los nuevos PVC, encuentra un PV coincidente (si es posible) y los enlaza juntos. Si un PV fue aprovisionado dinámicamente para un nuevo PVC, el bucle siempre vinculará ese PV al PVC. De lo contrario, el usuario siempre obtendrá al menos lo que solicitó, pero el volumen puede ser mayor de lo solicitado. Una vez enlazados, los enlaces de PersistentVolumeClaim son exclusivos, independientemente de cómo se hayan enlazado. Un enlace de PVC a PV es un mapeo uno a uno, utilizando un ClaimRef que es un enlace bidireccional entre el PersistentVolume y el PersistentVolumeClaim.

Las reclamaciones permanecerán sin enlazar indefinidamente si no existe un volumen coincidente. Las reclamaciones se enlazarán a medida que los volúmenes coincidentes estén disponibles. Por ejemplo, un clúster aprovisionado con muchos PVs de 50Gi no coincidiría con un PVC que solicita 100Gi. El PVC puede enlazarse cuando se agrega al clúster un PV de 100Gi.

3. Uso

Los Pods utilizan reclamaciones como volúmenes. El clúster inspecciona la reclamación para encontrar el volumen enlazado y monta ese volumen para un Pod. Para los

volúmenes que admiten múltiples modos de acceso, el usuario especifica qué modo se desea al usar su reclamación como un volumen en un Pod.

Una vez que un usuario tiene una reclamación y esa reclamación está enlazada, el PV enlazado pertenece al usuario siempre que lo necesite. Los usuarios programan Pods y acceden a sus PVs reclamados incluyendo una sección `persistentVolumeClaim` en el bloque de volúmenes de un Pod. Consulta "Reclamaciones como Volúmenes" para obtener más detalles sobre esto.

4. Protección de Objetos de Almacenamiento en Uso

El propósito de la función de Protección de Objetos de Almacenamiento en Uso es asegurar que las PersistentVolumeClaims (PVCs) en uso activo por un Pod y los PersistentVolume (PVs) que están vinculados a PVCs no se eliminen del sistema, ya que esto podría resultar en la pérdida de datos.

5. Reclamación

Cuando un usuario ha terminado con su volumen, puede eliminar los objetos PVC de la API que permite la reclamación del recurso. La política de reclamación para un PersistentVolume le indica al clúster qué hacer con el volumen después de que ha sido liberado de su reclamación. Actualmente, los volúmenes pueden ser Retenidos (Retain), Reciclados (Recycle) o Eliminados (Delete).

6. Retenidos (Retain)

La política de reclamación Retain permite la reclamación manual del recurso. Cuando se elimina el PersistentVolumeClaim, el PersistentVolume aún existe y el volumen se considera "liberado". Sin embargo, aún no está disponible para otra reclamación porque los datos del reclamante anterior permanecen en el volumen. Un administrador puede reclamar manualmente el volumen con los siguientes pasos:

1. Eliminar el PersistentVolume. El activo de almacenamiento asociado en la infraestructura externa aún existe después de que se elimina el PV.
2. Limpiar manualmente los datos en el activo de almacenamiento asociado según corresponda.
3. Eliminar manualmente el activo de almacenamiento asociado.
4. Si deseas reutilizar el mismo activo de almacenamiento, crea un nuevo PersistentVolume con la misma definición de activo de almacenamiento.

7. Eliminar (Delete)

Para los plugins de volumen que admiten la política de reclamación Delete, la eliminación elimina tanto el objeto PersistentVolume de Kubernetes como el activo de almacenamiento asociado en la infraestructura externa. Los volúmenes que se aprovisionaron dinámicamente heredan la política de reclamación de su StorageClass, que por defecto es Delete. El administrador debe configurar StorageClass según las expectativas de los usuarios; de lo contrario, el PV debe editarse o parcharse después de su creación.

Tipos de Persistent Volumes

Los tipos de Persistent Volumes se implementan como plugins. Actualmente, Kubernetes admite los siguientes plugins:

- **csi:** Interfaz de Almacenamiento de Contenedores (CSI, por sus siglas en inglés).
- **fc:** Almacenamiento de Canal de Fibra (Fibre Channel).
- **hostPath:** Volumen de HostPath (solo para pruebas en un solo nodo; NO FUNCIONARÁ en un clúster multinodo; considera usar un volumen local en su lugar).
- **iscsi:** Almacenamiento iSCSI (SCSI sobre IP).
- **local:** Dispositivos de almacenamiento local montados en los nodos.
- **nfs:** Almacenamiento del Sistema de Archivos en Red (Network File System, NFS)

Quick Learning K8

Este apartado tiene el objetivo de ayudarte a seguir potenciando tus habilidades, por lo que a continuación encontrarás unos enlaces que podrás utilizar para aprender más rápidamente y a tu ritmo.

Video de Youtube – K8s  de novato a pro

<https://www.youtube.com/watch?v=DcOBcpOA7W4>

Duración: 1:31

≡ Resumen

Este manual aborda la gestión de contenedores y orquestación, destacando Docker y sus fundamentos, desde su funcionamiento hasta la instalación y comandos básicos.

Explora la creación de imágenes y contenedores, así como Dockerfile.

Además, se profundiza en la integración de Docker con pipelines CI/CD. Se presenta la orquestación con herramientas como Puppet y Chef, y se detalla Kubernetes (K8s), incluyendo su arquitectura, componentes, configuración de contenedores, políticas de networking, escalabilidad y gestión de imágenes.

El manual proporciona una comprensión completa de las tecnologías esenciales para implementar y gestionar contenedores en entornos de desarrollo y producción.

≡ Referencias

Libros

1. "Docker: Up & Running" de Karl Matthias, Sean P. Kane, y Trent R. Hein
2. "The Docker Book: Containerization is the new virtualization" de James Turnbull
3. "Kubernetes: Up and Running" de Kelsey Hightower, Brendan Burns y Joe Beda
4. "Puppet 5 Beginner's Guide" de John Arundel y William Van Hevelingen
5. "The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win" de Gene Kim, Kevin Behr y George Spafford

Papers

1. "Docker: A Platform for Distributed Applications" (Paper de Solomon Hykes)
2. Sitio web de Docker (<https://www.docker.com/>)
3. Sitio web de Kubernetes (<https://kubernetes.io/>)
4. "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation" (Libro de Jez Humble y David Farley)
5. "The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations" (Libro de Gene Kim, Jez Humble, Patrick Debois y John Willis)

