izertis

DEVOPS

GUÍA COMPLETA

Guía para directivos y técnicos

V.7

DEVOPS

Guía completa

El interés por DevOps (la combinación de **“desarrollo” y “operaciones”**) ha continuado creciendo de manera exponencial hasta convertirse en uno de los anhelos más profundos del negocio digital.

Pero ¿cuál es la motivación real que nos impulsa a incorporar este tipo de prácticas?, ¿cuáles son los detalles que tenemos que tener en cuenta para incorporarlas? y ¿cuáles son las mayores barreras que nos podemos encontrar en el camino?

Estas preguntas e inquietudes nos han impulsado a formular esta guía que nos permitirá, con un enfoque cercano y práctico, aterrizar todo lo que CEOs, CIOs, arquitectos y desarrolladores necesitan saber para **entender la importancia de la cultura DevOps y cómo con ella se puede contribuir** a la entrega de valor al cliente y negocio.

En un mundo cada vez más acelerado y centrado en el cliente, se hace necesario hacer **entregas continuas de software, testar la respuesta del usuario y tener capacidad de entender y adaptarse a sus necesidades.**

Trabajar bajo una filosofía DevOps facilita la **recuperación** en caso de escenarios de contingencia, al tener perfectamente identificados los activos de software y hardware y automatizadas las tareas rutinarias.

En el mercado podemos ver ofertas de perfiles DevOps polarizados en conocimientos de automatización de entornos y configuración en la nube. En la primera parte de esta guía veremos una perspectiva integral.

“Una de las claves para garantizar la calidad en las entregas es reducir la intervención humana a la hora de instalar entornos, empaquetar, testar, entregar o incluso volver al estado anterior en caso de problemas.”

En la segunda parte de esta guía, el objetivo es tratar de **describir las piezas fundamentales** que puede tener una infraestructura moderna, entender la complejidad que estas pueden añadir y cómo prácticas como infraestructura como código son imprescindibles, se pueden gestionar y escalar sistemas muy estables. Partiendo de este punto, tener una infraestructura inestable no solo daña intangibles, como la imagen de marca, sino que provoca pérdidas directas.

¿Cuánto cuestan nuestros proveedores?, ¿cuánto cuesta el hardware?, ¿y las licencias de software? Si no sabemos responder, es probable que estemos pagando plata a precio de oro. Los proveedores de infraestructura en la nube cuentan con **calculadoras** que nos van a servir para estimar cuánto nos va a costar todo. Son complicadas de utilizar y requieren conocer si todo o parte de nuestro sistema puede adaptarse a este nuevo paradigma.

No conocer esta realidad, no nos exime de vivir en ella, y aprovecharnos de sus ventajas puede suponer no sólo un **ahorro en costes**, sino una oportunidad para mejorar la imagen corporativa, la atracción de talento e incluso una ventaja competitiva frente a nuestra competencia.

En la tercera parte de esta guía, se profundiza en los conocimientos que cualquier desarrollador debería tener acerca de la **administración de sistemas**. Sin estos conocimientos, nuestro sendero hacia las grandes cimas de DevOps terminaría aquí. Aunque, normalmente, hay personal especializado a cargo de las infraestructuras, es necesario conocer lo indispensable para mantener nuestro **entorno de trabajo preparado** para la acción y estar al tanto de las características que los sistemas operativos nos ofrecen.

Uno de los temas más importantes a este respecto es la **virtualización de entornos**. Gracias a ella, podemos reproducir con fidelidad las condiciones para el desarrollo o el despliegue de una aplicación de forma sencilla. En los últimos años han surgido diferentes enfoques y tecnologías destinados a facilitarnos las cosas que es imprescindible conocer.

A partir de ese momento, una vez preparado el terreno, la guía irá profundizando en diferentes herramientas y tecnologías relacionadas con la filosofía DevOps, como contenedores, herramientas de provisión, configuración y despliegue, IaC, orquestadores de contenedores y más.

DEVOPS

Guía completa

Índice

Parte 1: Flujo de desarrollo

Parte 2: Piezas básicas de la infraestructura

- ¿En la nube o no? Ese es el dilema...
    - Ventajas e inconvenientes de los entornos on premise
    - Ventajas e inconvenientes de los entornos cloud e híbridos
- Servidor Web
    - HTTP
    - DNS
    - HTTPS
- Balanceadores
- Servidores de aplicaciones
- Sistemas de bases de datos
- Sistemas de caché
- CDN
- API Gateway
- APM
- Monitorización de infraestructura
- Monitorización funcional
- Perfil SRE

Parte 3: Administración de sistemas unix

- Objetivo
- Contexto
- Virtualización de sistemas
- Diferentes distribuciones
- Montando el entorno
- Gestión de usuarios, permisos y accesos
    - Usuarios
    - Home del usuario
    - Grupos
    - Permisos

- Particiones en Linux
    - Tipos de particiones
    - Sistemas de Ficheros
    - Estructura de Directorios en Linux
    - Recomendaciones y buenas prácticas
    - Añadir un nuevo disco
    - Crear una partición
    - Mover a la nueva partición
    - Borrar una partición
    - Redimensionar una partición
- Editor Vi
    - Modos
    - Moverse con el cursor
    - Cambiar de Modo
    - Borrar texto
    - Copiar, cortar y pegar
    - Buscar y reemplazar
    - Manipulación de ficheros
- Procesos, servicios y tareas programadas
    - Herramientas para el manejo de procesos
    - Matar procesos
    - Listar los fichero abiertos por procesos
    - Arrancando una aplicación como servicio
    - Tareas programadas
- Bash Scripting
    - Variables
    - Sentencias condicionales
    - Operadores
    - Bucles
    - Manejo de ficheros
    - Funciones
    - Control de Errores
- Utilidades esenciales en UNIX
    - Comandos de red

    - Comandos entrada y salida
    - Comandos de filtrado
    - Comandos para buscar
    - Miscelanea

Parte 4: Docker

- Introducción a Docker
    - ¿Qué es Docker?
    - ¿Por qué se usa?
    - ¿Qué es un contenedor?
    - ¿Cómo funciona Docker?
    - Anatomía de contenedores
    - Instalación de Docker
- Trasteando con Docker
    - Arrancar mi primer contenedor
    - Arrancar, parar, borrar contenedores
    - Monitorizar los logs de un contenedor
    - Ejecutar una shell dentro de Docker
    - Configurando contenedores
- Construyendo imágenes a medida
    - ¿Qué es una imagen?
    - Creando imágenes
    - Compartiendo imagenes: subida y etiquetado
- Volúmenes de datos
    - Gestión de los volúmenes
- Información del Sistema en Docker
    - Consumo de recursos
    - Liberar recursos no utilizados

- Redes
    - Modelo de red en un contenedor
    - Cortafuegos
    - Implementaciones de red
    - Gestión de puertos
- Docker Compose
    - Arrancando un App Multi-Servicio
    - Construir imágenes con docker compose
    - Arrancar una aplicación con docker compose
    - Escalar un servicio

Parte 5: Ansible

- Introducción
    - Algo de historia
- Gestión de la configuración
    - Infraestructura como Código
    - Otras herramientas
- Arquitectura de Ansible
    - Preparando el entorno de pruebas
- Inventario
    - ¿Qué es?
    - Opciones de configuración
    - Hosts
    - Grupos
    - Variables
    - Múltiples inventarios
    - Inventario dinámico
- Playbooks
    - Variables y hechos
    - Plantillas con Jinja
- Roles
    - Estructura

        - Dependencias entre roles
        - Roles para diferentes plataformas
- Modulos
        - Custom modules
        - Depurando los módulos
        - AWS con Ansible
        - Docker con Ansible
- Protegiendo secretos en Ansible
        - Cifrar ficheros
        - Modificar fichero cifrado
        - Descifrar fichero
        - Cambiar la clave de cifrado
- Control de las tareas
        - Handlers
- Testing
- Debugging con Ansible
        - Comprobar sintaxis
        - Depuración de tareas
        - Logs
- Buenas prácticas
        - Organización de los archivos
        - Diferencias entre entornos
        - Lanzando tareas por lotes
        - Mencionar el estado
        - Uso de los roles
        - Diferentes sistemas operativos
        - Algunos consejos generales

Parte 6: Kubernetes

- Introducción
        - ¿Qué es Kubernetes?
        - Algo de historia
- Clúster de Kubernetes

        - Componentes
        - API de Kubernetes
        - Pasos para crear un clúster
- Contenedores
    - Imágenes
- Cargas de trabajo
    - Pods
    - Controladores
- Nodos
    - Estado de un nodo
    - Controlador de nodos
    - Registro de nodos
    - Comunicación Nodo-Maestro
- Almacenamiento
    - Volúmenes
    - Configuración de volúmenes
    - Límites
- Servicios, balanceo de carga y redes
    - Modelo de red de Kubernetes
    - Servicios de red
    - Enrutamiento
    - DNS
    - Ingress
    - Política de tráfico interno
    - Políticas de red
- Configuración
    - Consejos de configuración y buenas prácticas
    - Kubectl
    - ConfigMaps
    - Secrets

Parte 7: Kubernetes Avanzado

- Configuración

    - Gestión de recursos de pod y contenedores
    - Acceso al clúster con archivos kubeconfig

- Seguridad
    - Las 4C’s de Cloud Native Security
    - Seguridad de un pod
    - Seguridad y control de la API
    - Configuraciones por defecto
    - Multitenancy
- Istio
    - ¿Qué es y para qué sirve?
    - Instalación del panel de control de Istio en Kubernetes
    - Desarrollo de una aplicación
- Despliegue
    - Helm
    - Kustomize
    - GitOps

Bibliografía

Lecciones aprendidas con esta guía

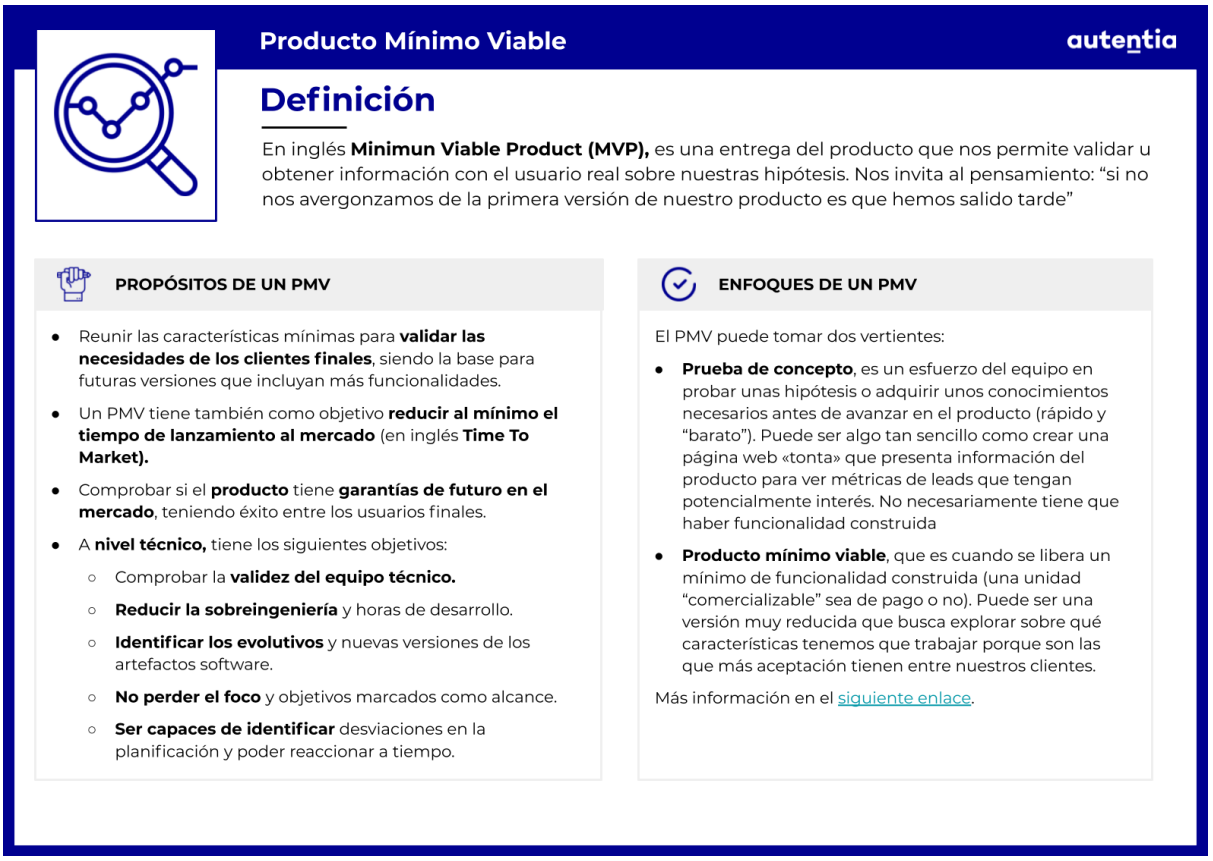
Parte 1

Flujo de desarrollo

Ideando el producto

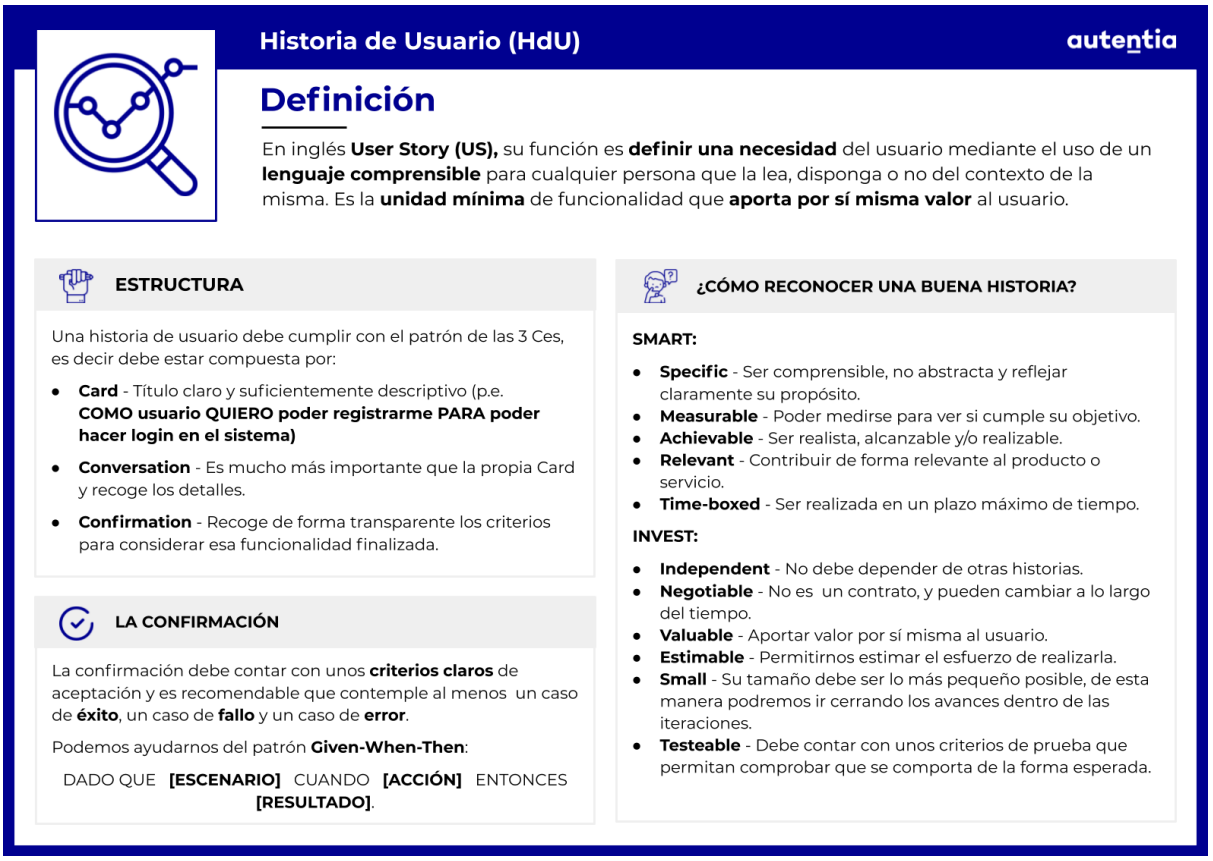
Los negocios están continuamente buscando innovar para optimizar sus procesos y lograr la eficiencia operativa. A día de hoy la industria tiende a seguir un modelo ágil de desarrollo de software, donde se consiguen probar los requisitos de forma fácil y rápida. El objetivo es validarlos o desecharlos rápidamente, siguiendo el principio **“falla rápido, falla barato”**.

En nuevas iniciativas, se pretende construir un producto mínimo viable, más comúnmente conocido como MVP (Minimum Viable Product) para validar el conjunto de requisitos de negocio que se pretenden implementar. Este es un proceso iterativo e incremental, en el cual se entregan productos potencialmente utilizables aunque incompletos, construidos en ciclos cortos, en el que pueden surgir cambios o nuevos requisitos que se podrán aplicar durante las siguientes iteraciones (versiones del producto).



No siempre se tienen claros los detalles de las ideas o la profundidad de las mismas. En muchas ocasiones se parte de una semilla de idea, que va creciendo y cambiando de formas en la medida en que es validada con el cliente. Esta información no siempre puede preverse al principio.

Es importante que todo el equipo entienda que un cambio de prioridad o de la historia de usuario es posible y es necesario poder adaptarse a estos cambios siempre teniendo en cuenta el impacto que pueda ocasionar. Hay que evitar los extremos, ni poner trabas ni frustrarse ante dichos cambios, ni tampoco que éstos sean continuados.

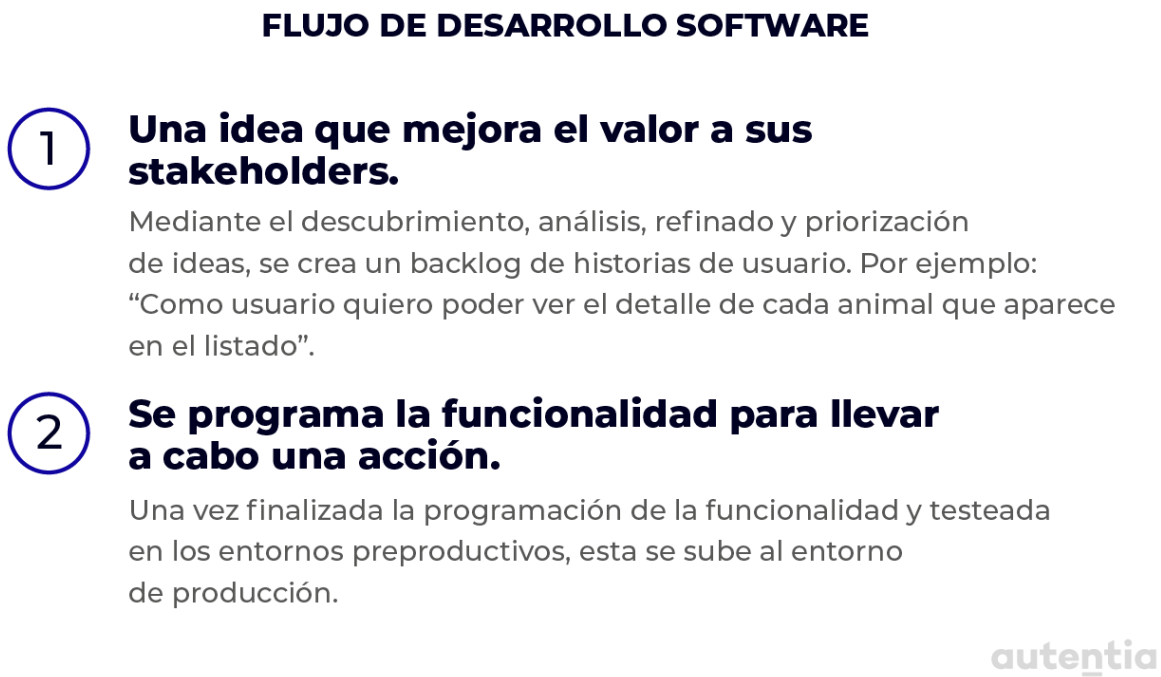


En este sentido, DevOps promueve un conjunto de prácticas que combina el **desarrollo de software (dev)** con **operaciones y gestión** de la infraestructura sobre la que va a ir instalado el software **(ops)**. Todas estas partes participan juntas en el ciclo de vida de un producto o aplicación, y **cuanto más involucradas estén, más rápida y productiva será la entrega de valor**.

El flujo de desarrollo software

A continuación, se hace una aproximación de lo que puede ser el flujo de desarrollo de software de un equipo que no posee una cultura Devops y luego se introduce la implantación de prácticas que ayudan a mejorar en cada punto.

Suele constar a grandes rasgos de las siguientes fases:

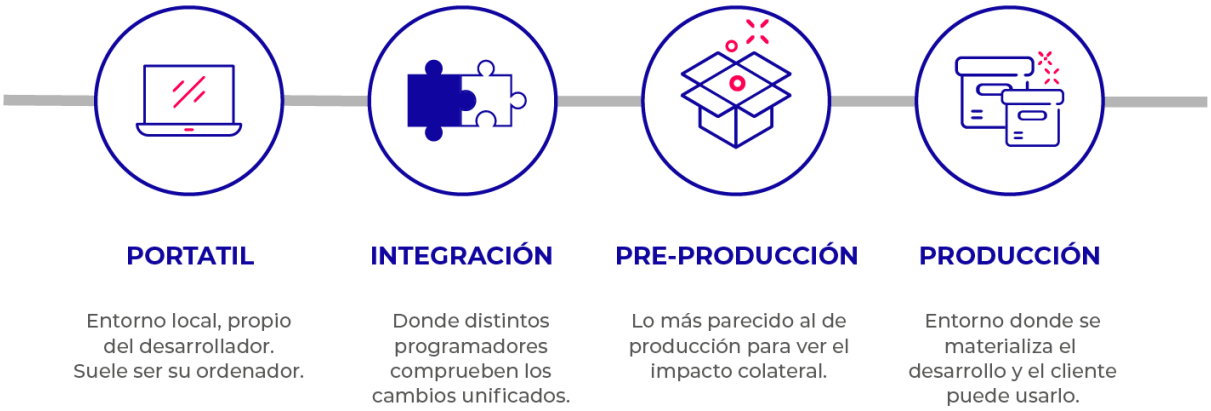


Al negocio se le ocurre una idea para mejorar la entrega de valor a sus stakeholders o clientes.

Mediante un proceso de descubrimiento, análisis, refinamiento y priorización, que puede ser *ágil*, **al desarrollador le llega una necesidad o requisito, que podría estar formulada como una historia de usuario**. Esta puede estar escrita del siguiente modo: "*Como usuario quiero poder ver el detalle de cada animal que aparece en el listado*".

Ahora mismo el desarrollador se encuentra con la responsabilidad de **programar la funcionalidad** necesaria para que los usuarios finales puedan llevar a cabo esa acción.

Una vez **finalizada la programación** de la funcionalidad y testeada contra el entorno local del desarrollador **hay que subir los cambios a los distintos entornos**. Existirán al menos los siguientes entornos:



Esta subida suele hacerse de forma manual con riesgo a cometer errores humanos, resulta ser lenta y para nada es óptima. Imagina tener que hacer el proceso manualmente cada vez que haya un cambio, es decir, ejecutar los comandos necesarios de todas las herramientas en cada despliegue.

Estos comandos suelen requerir de un orden determinado y, al ser tan frecuente el proceso, lo mejor es automatizarlos. Así además evitamos errores por el factor humano. Por ello, como parte de la cultura DevOps, los cambios que se hagan en el desarrollo, al subir al repositorio de código, van a lanzar un proceso de **integración contínua** CI (continuous integration), que generalmente incluye la compilación del código, la ejecución de tests automáticos, la generación de artefactos y algunas otras fases como el análisis de calidad de código automático. **Si todo eso fuera correcto se podría ejecutar un proceso de despliegue continuo, (Continuous Deployment)** de tal forma que la nueva versión de la aplicación con los nuevos cambios incluídos estaría disponible para ser probada en un **entorno de integración**.

Integración continua

autentia

¿Qué es?

En inglés **Continuous Integration (CI)** es la práctica que tiene como objetivo integrar los cambios en el repositorio central de forma periódica a través de varios procesos automatizados donde cada versión generada se comprueba mediante pruebas y tests para detectar posibles errores de forma temprana.

¿EN QUÉ CONSISTE?

Durante el proceso de integración continua se pasan ciertas fases o tareas a ejecutar, como por ejemplo, instalar las dependencias necesarias, lanzar los tests, entre otras. En caso de que alguien del equipo decida hacer un Pull Request/Merge Request, comprobamos que dicho pipeline ha pasado correctamente y procedemos a integrar dichos cambios ya sea en la rama principal o en nuestra propia rama local en caso de necesitarlos.

¿Qué beneficios nos aporta?

- **Detección rápida de fallos** de forma continua.
- **Aumento de la productividad del equipo**.
- **Automatización** y ejecución inmediata de procesos.
- **Monitorización** continua de las métricas de calidad del proyecto.

Debemos tener en cuenta:

- **integrar los cambios** varias veces al día.
- La programación en equipo es un problema de **"divide, vencerás e integrarás".**
- **La integración es un paso no predecible** que puede costar más que el propio desarrollo.
- **Integración síncrona:** cada desarrollador una vez completada la funcionalidad, sube sus cambios, espera a que se complete la construcción y se hayan pasado todas las pruebas sin ningún problema de regresión.
- **Integración asíncrona**: cada noche se hace un build diario en el que se construye la nueva versión del sistema. Si se producen errores se notifica con alertas de emails.
- **El sistema resultante debe ser un sistema listo para lanzarse**.

Despliegue continuo

autentia

¿Qué es?

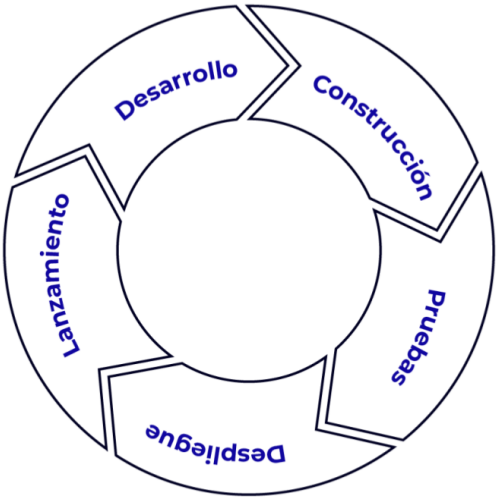
En inglés **Continuous Deployment (CD),** es una práctica que tiene como objetivo proporcionar una manera ágil, fiable y **automática** de poder entregar o desplegar los nuevos cambios en el entorno específico, normalmente producción. Suele utilizarse junto con la integración contínua.

¿EN QUÉ CONSISTE?

El despliegue continuo se basa en automatizar todo el proceso de despliegue de la aplicación en cualquiera de los entornos disponibles, ya sea sólo en algunos de ellos o en todos, todo ello, sin que haya **ninguna intervención humana** en el procedimiento.

El objetivo es hacer **despliegues predecibles, automáticos y rutinarios** en cualquier momento, ya sea de un sistema distribuido a gran escala, un entorno de producción complejo, un sistema integrado o una aplicación.

Dependiendo de cada proyecto o propósito de negocio de la empresa, el despliegue estará configurado para hacerse de forma semanal, quincenal o cada 2 o 3 días, aunque el objetivo es hacerlo de manera constante y en periodos cortos para obtener feedback rápido del cliente.

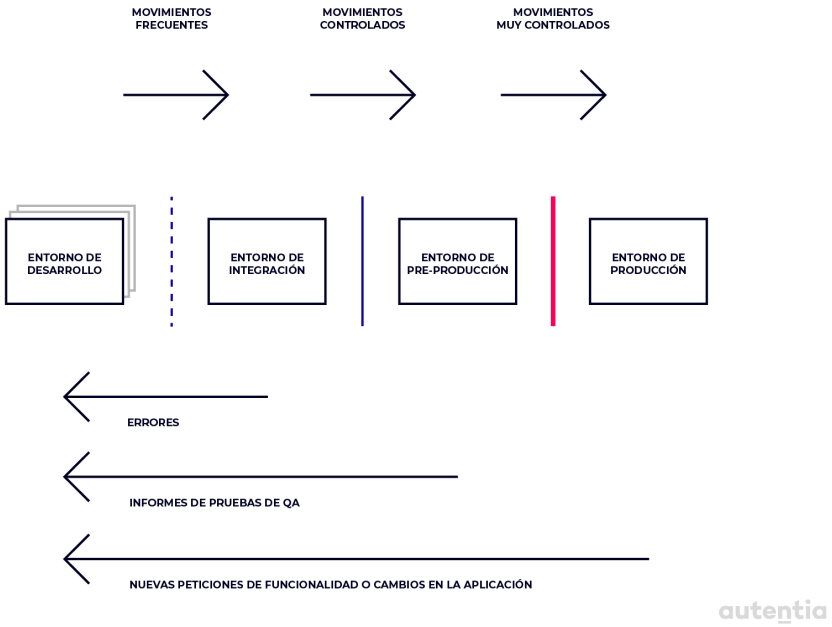


El **entorno de integración** es usado en su mayoría por los desarrolladores para verificar que la funcionalidad entregada cumple con los requisitos y no tiene errores. Es tan importante aportar nueva funcionalidad como no romper las que ya se habían entregado.

Tenemos que ser conscientes de que en las empresas suele haber muchos equipos de desarrollo que necesitan usar el entorno de integración. **No dejar al libre albedrío el acceso y modificaciones de este entorno permite reducir errores humanos** (como la ejecución de comandos incorrectos, la subida de versiones sin aprobación de los tests o, incluso, el olvido de la ejecución de alguno de los pasos del proceso) que, bien por inexperiencia o falta de conocimiento, pueden ocurrir. En este sentido, **lo ideal es que los cambios sean automatizados**.

Si las pruebas han ido bien y hemos verificado que la nueva funcionalidad es correcta en el entorno de integración, el siguiente paso sería promocionar al siguiente entorno: el **entorno de preproducción**. Este debe ser aún más estable que el anterior, ya que puede haber terceros utilizándolo para integrarse con nosotros, QAs probando historias...

Si todo es correcto terminaría el ciclo de desarrollo poniendo esta funcionalidad en el **entorno de producción**. Si preproducción tenía que ser un entorno estable, el entorno de producción muchísimo más. Aquí es vital la automatización de procesos.



Empezando con el desarrollo

Una vez tenemos definido lo que tenemos que hacer, es el turno del desarrollador de software. Este va a hacer uso de una serie de herramientas para hacer su trabajo.

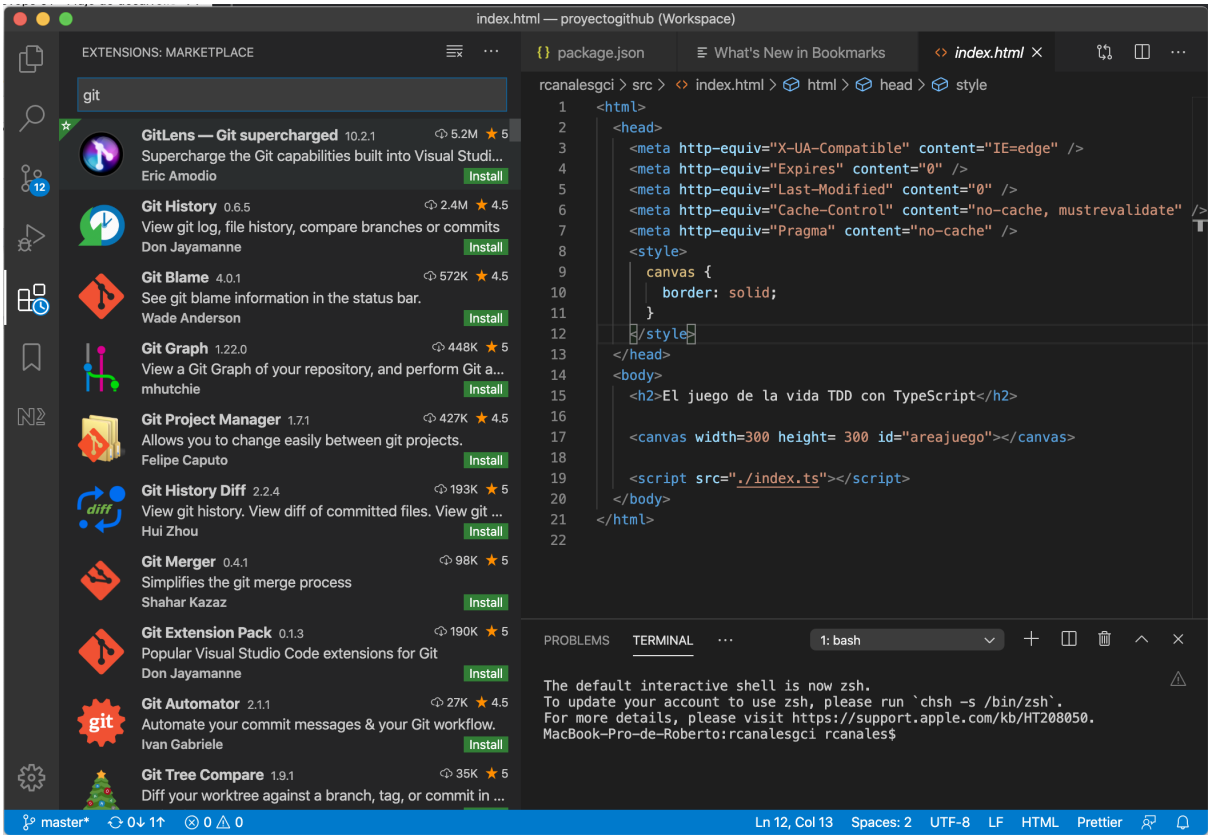
En un mundo cada vez más interconectado, los profesionales requieren autonomía, por lo que tendrán estas herramientas instaladas en su ordenador (probablemente un portátil con grandes capacidades), configuradas para poder trabajar con los proyectos. Al conjunto de estas herramientas se las conoce como "**entorno local de un desarrollador**". Vamos a ver algunas de las más comunes.

El uso del IDE

Un **IDE** (Integrated Development Environment) es una interfaz de desarrollo, una herramienta que nos permite realizar desde las tareas más básicas, como codificar, hasta otras más avanzadas o accesorias, como hacer cambios en múltiples ficheros al mismo tiempo.

A menudo se pueden extender instalando plugins, que son extensiones que han desarrollado terceros y que añaden funcionalidad extra o utilidades que van a simplificar nuestras tareas diarias. Estos plugins nos permiten, por ejemplo, ver el historial de cambios que ha tenido un fichero y quienes lo han modificado. Para ello harán uso de un sistema distribuido de control de versiones como Git (que veremos más adelante). Hay muchos IDEs distintos que ofrecen características y opciones diferentes y pueden estar centrados en distintos lenguajes de programación. Entre los más populares, encontramos Eclipse, IntelliJ IDEA o Visual Studio Code.

Hemos visto que podemos llegar a trabajar junto con decenas o incluso centenas de programadores, pero ¿cómo se coordina todo esto?



Ejemplo de IDE: Visual Studio Code con plugins disponibles.

Git

Git es, como dice su propia página web, un **sistema de control de versiones distribuido**, de código abierto y gratuito, es decir, donde vamos a guardar el código fuente de nuestras aplicaciones y todos los cambios que se hagan sobre el mismo. La palabra clave es "distribuido". En el caso de Subversion, CVS o similares hay un repositorio central con el cual se sincroniza todo el mundo. Este repositorio central está situado en una máquina concreta y es el repositorio que contiene el histórico, etiquetas, ramas, etc. En los sistemas de control de versiones distribuidos, como es el caso de Git, esta idea de repositorio central no existe, está distribuido entre los participantes; cada participante tiene en su local el histórico, etiquetas y ramas. La gran ventaja de esto es que no necesitas estar conectado a la red para hacer cualquier operación contra el repositorio, por lo que el trabajo es mucho más rápido y tiene menos dependencias. Ante esta idea hay algunas preguntas comunes que suelen asaltar nuestra cabeza:

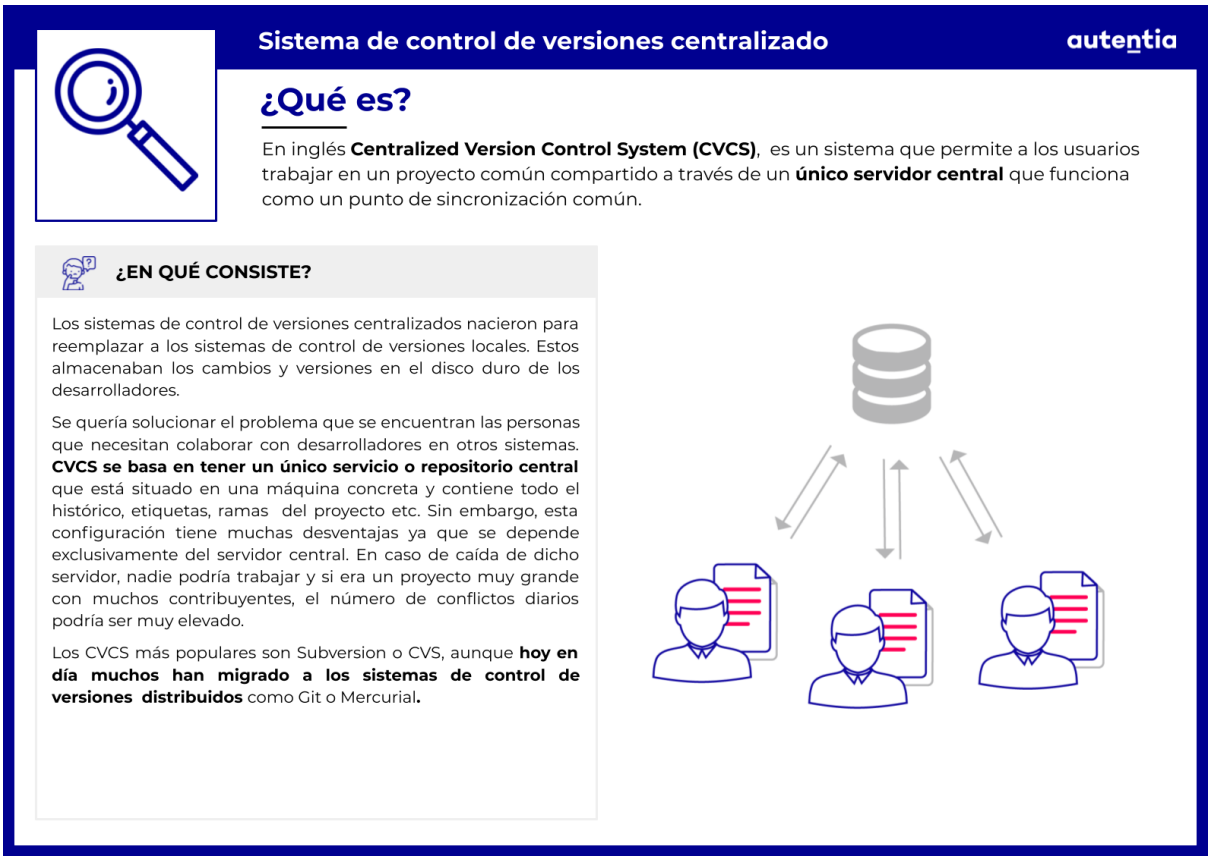
- **¿Si tenemos todo el repositorio en local, no ocupará mucho espacio?**

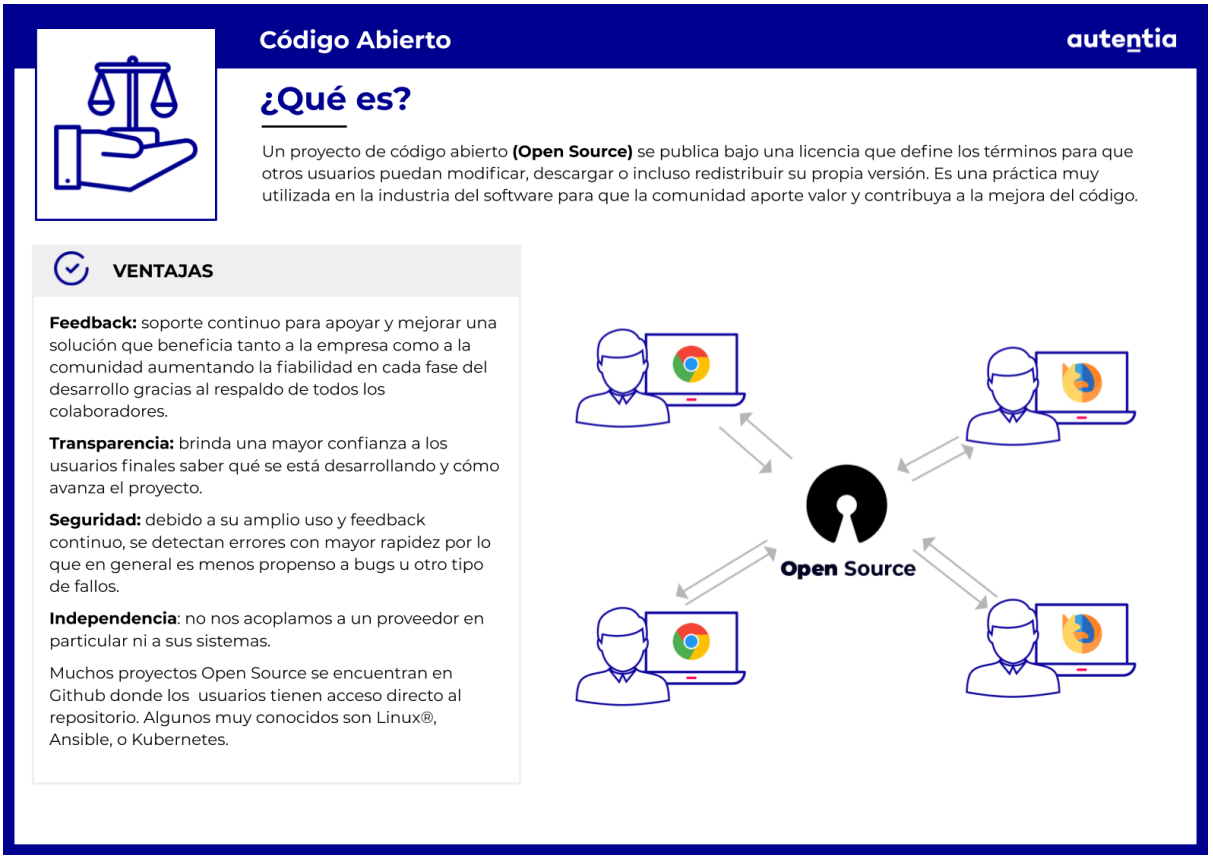
  Lo normal es que no, porque al ser un repositorio distribuido, solamente tendremos las partes que nos interesan.
  Habitualmente, si tenemos un repositorio central, tendremos muchos proyectos, ramas, etiquetas, etc. Al tratarse de un repositorio distribuido, solo tendremos en local la parte con la que estamos trabajando (la rama de la que hemos partido).

- **¿Si todo el mundo trabaja en su local, no puede resultar en que distintas ramas diverjan?**

  Efectivamente, esto puede ocurrir y es natural. En un desarrollo tipo open-source no hay mucho problema, es lo habitual y constantemente están saliendo nuevos "forks" (desarrollos paralelos y distintos del original) de los productos. En un desarrollo "interno" tampoco hay problema si ocurre esto, porque luego vamos a acabar haciendo "merge" (fusionando los cambios que tenemos en local con los que hay en el repositorio, para que tanto la versión local como la del repositorio sean iguales). Al final, todo depende de nuestro proceso de desarrollo, no tanto de la herramienta que usemos; es normal que los desarrolladores abran nuevas ramas constantemente para desarrollar partes del sistema y que luego hagan merge para unir estos cambios sobre una rama principal.

Si quieres saber más sobre esto, en el artículo **Git y cómo trabajar con un repositorio de código distribuido** podrás encontrar más información.

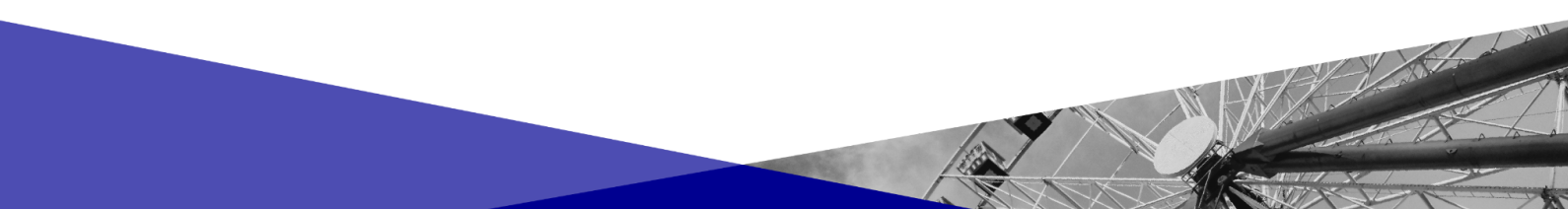


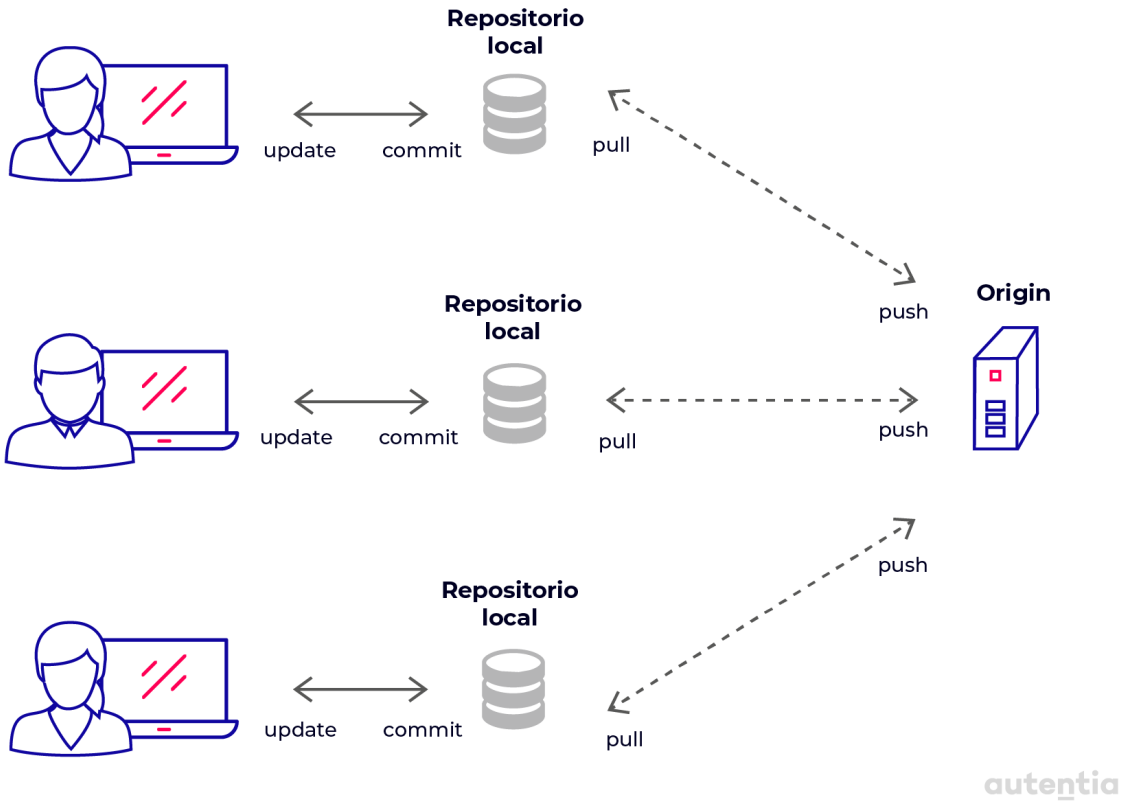


Git Flow

Git Flow surge de un [post de Vincent Driessen en 2010](#) donde explica un modelo de desarrollo basado en las ramas de git utilizando una serie de convenciones para el nombrado de las ramas y definiendo un uso específico de las mismas.

Aunque el enfoque de Git es descentralizado, se necesita un **repositorio central** que contenga los cambios hechos por los componentes del equipo. A este repositorio se le llama “origin”. Cada desarrollador hace “pull" (traerse los cambios del origen a tu carpeta local) y “push” (subir los cambios de la carpeta local al repositorio origen) al origen. Además de las relaciones centralizadas, cada desarrollador puede hacer pull de cambios de otros compañeros para formar subequipos. Esto es útil cuando trabajas con dos o más desarrolladores en una gran característica nueva, antes de llevar el trabajo en progreso al “origin” prematuramente y causar que haya un desarrollo incompleto.





El autor del post coloca una nota que aclara que este modelo no aplica a todos los equipos y que **no se debe forzar su adopción**. No es un dogma que haya que seguir siempre en todo tipo de proyectos. Este modelo tiene sentido en un proyecto donde haya que tener un fuerte control de las distintas versiones del proyecto, manteniendo y dando soporte a varias versiones a la vez. En cambio, si el proyecto no requiere de mantenimiento de distintas versiones a la vez, de tener que volver atrás a una versión antigua, puede que no tenga tanto sentido ceñirse a git flow.

Ramas principales

El repositorio central tiene dos ramas principales, paralelas, que nunca se borran y sobre las que todo el sistema de ramas de soporte se va a apoyar:

- **Master**: rama que contiene el código listo para subir a producción o hacer una nueva versión; este código será el del commit al que apunte HEAD (el último commit de la rama).
- **Develop**: rama que siempre refleja el estado del desarrollo con los últimos cambios listos para ser entregados en la próxima versión; este código será el del commit al que apunte HEAD (el último commit de la rama). Esta rama puede tener otros nombres como, por

ejemplo, rama de integración.

Cuando el código fuente en la rama de desarrollo alcanza un punto estable y está listo para ser lanzado en forma de nueva versión, todos los cambios deben fusionarse nuevamente en la rama master y luego etiquetarse con un número de versión.

Por lo tanto, cada vez que los cambios se fusionan en la rama master, por definición, se genera una nueva versión de producción. Podríamos compilar y desplegar automáticamente el software en producción cada vez que haya un commit en master.

Además de las ramas master y develop, el modelo de desarrollo utiliza una variedad de ramas de soporte para ayudar al desarrollo paralelo entre los miembros del equipo, facilitar el seguimiento de las características, prepararse para los lanzamientos de producción y ayudar a solucionar rápidamente los problemas de producción en vivo. A diferencia de las ramas principales, estas ramas siempre tienen un tiempo de vida limitado, ya que una vez cumplido su propósito, se eliminan.

Los diferentes tipos de ramas que podemos usar son: feature, release y hotfix.

Cada una de estas ramas tiene un propósito específico y están sujetas a reglas estrictas sobre qué ramas pueden ser su rama de origen y a qué ramas pueden hacer merge.

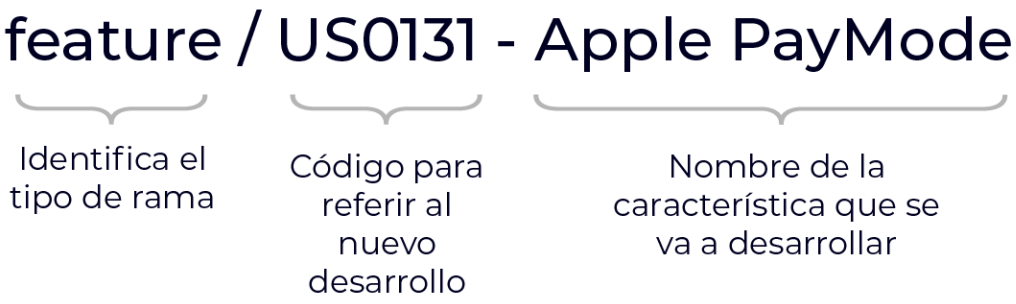
Estas ramas no son especiales desde una perspectiva técnica, son ramas normales de git y esta clasificación se hace por el tipo de uso que pretendemos darle al código dentro de la rama:

Ramas Feature:

Utilizadas **para desarrollar nuevas funcionalidades** para las próximas versiones. Al comenzar el desarrollo de una funcionalidad, la versión de destino en la que se incorporará puede ser desconocida en ese momento. La esencia de una rama feature es que existe mientras la funcionalidad esté en desarrollo, pero **finalmente se fusionará de nuevo en la rama develop** (para agregar definitivamente la nueva funcionalidad a la próxima versión) o se descartará (en caso de que la característica resulte ser un experimento decepcionante) borrando la rama.

Esta rama **se va a crear partiendo de la rama develop y, al terminar, se va a fusionar (merge) con la rama develop**. La convención de nombres de estas

ramas suele ser feature/(nombre de la característica a desarrollar). Si se está usando un sistema para manejar las tareas del desarrollo y estas tareas tienen un número o identificador, el nombre de estas ramas suele ser feature/(identificador)-(nombre de la característica a desarrollar).



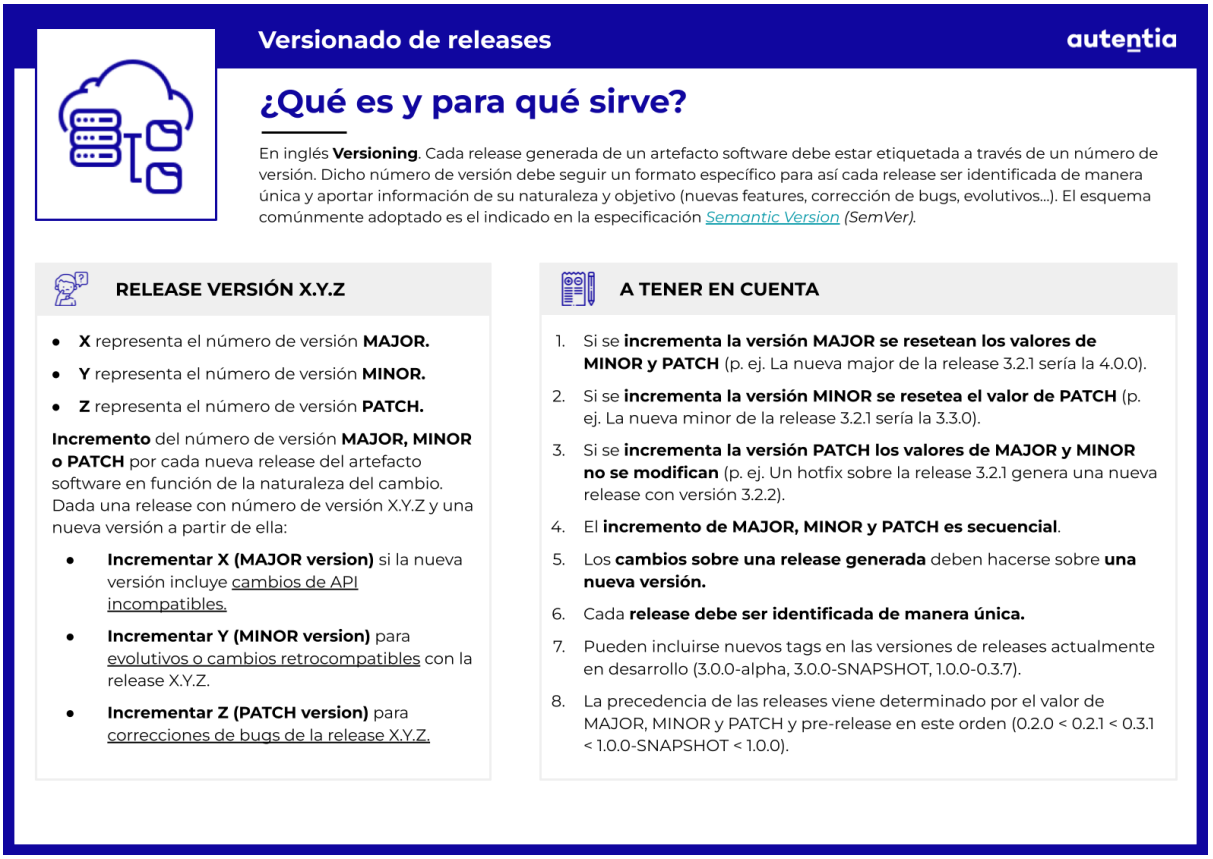
Ramas Release:

Apoyan la preparación de una nueva versión de producción. Además, **permiten pequeñas correcciones de errores** y preparan metadatos para un lanzamiento (número de versión, fechas de compilación, etc.). Al hacer este trabajo en una rama release, la rama de desarrollo puede ser usada para recibir características para la próxima gran versión sin crear problemas al cruzar trabajo entre dos versiones.

El momento clave para crear una nueva rama release desde la rama de desarrollo es cuando la rama desarrollo (casi) refleja el estado deseado de la nueva versión. Antes de crear la rama release desde la de develop, todas las funcionalidades que tienen que ir en la nueva versión tienen que ser primero fusionadas con la rama develop; luego ya se puede crear la rama release. Las ramas feature que contengan funcionalidades no pertenecientes a la versión que se va a lanzar a la rama release, no deben ser fusionadas con la rama develop.

Es exactamente en la creación de una rama release cuando se le asigna un número de versión, no antes (de acuerdo a las reglas de versionado del proyecto). Hasta ese momento, la rama de desarrollo contiene cambios para la "próxima versión", pero no está claro si esa "próxima versión" se convertirá finalmente en la 0.3 o 1.0 hasta que se cree la rama. Aún así, no se va a sacar versión del proyecto hasta que se haga el merge con master y, por lo tanto, los eventuales bugs que pueda tener esta rama estén

resueltos.



Esta rama se va a crear partiendo de la rama develop y al terminar se va a fusionar (merge) con las ramas develop y master. La convención de nombre de estas ramas suele ser release/(número de versión).

Ramas Hotfix:

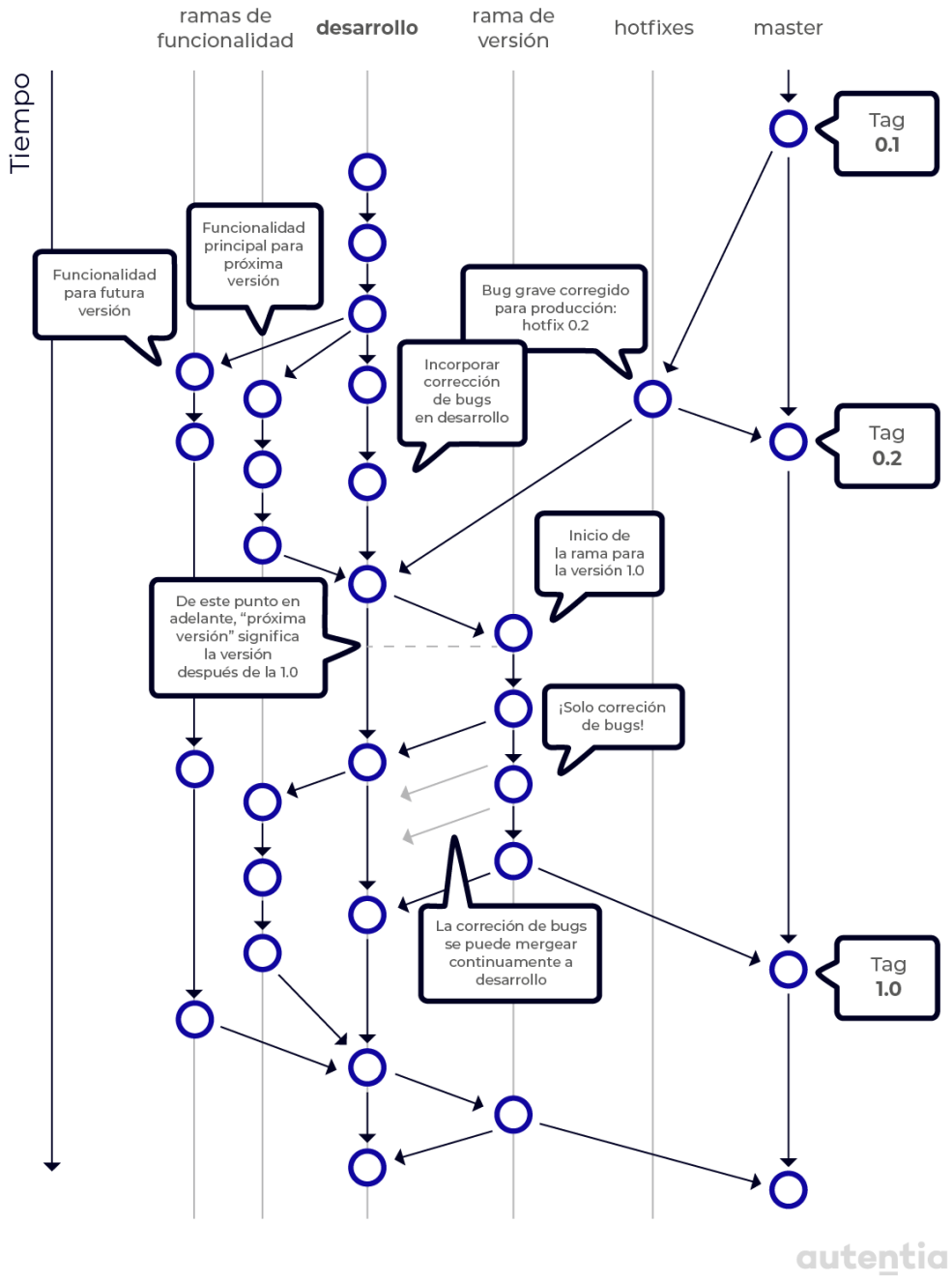
Muy parecidas a las ramas release, ya que también están destinadas a prepararse para un nuevo despliegue en producción. Surgen de la necesidad de actuar inmediatamente sobre un estado no deseado (un bug o error) de una versión que se está usando en producción. **Cuando se descubre un error crítico** en una versión de producción **debe resolverse de inmediato**, una rama hotfix se puede crear partiendo de la rama master que esté marcada (con un tag) con el número de la versión desplegada en producción.

La esencia de esta rama es que el trabajo de los miembros del equipo (en la rama de desarrollo) puede continuar, mientras que otra persona está preparando una solución rápida para producción.

Esta rama se va a crear partiendo de la rama master y al terminar se va a

fusionar (merge) con las ramas develop y master. La convención de nombre de estas ramas suele ser hotfix/(número de versión, incrementado sobre la versión que hay en producción).

Lo importante en la convención de nombres de las ramas auxiliares es indicar el tipo de rama, poner la barra como separador y darle un nombre claro para que cualquiera que vea la rama sepa el trabajo que se está realizando en ella.



Documentación con más explicaciones y comandos para poner en práctica Git Flow.

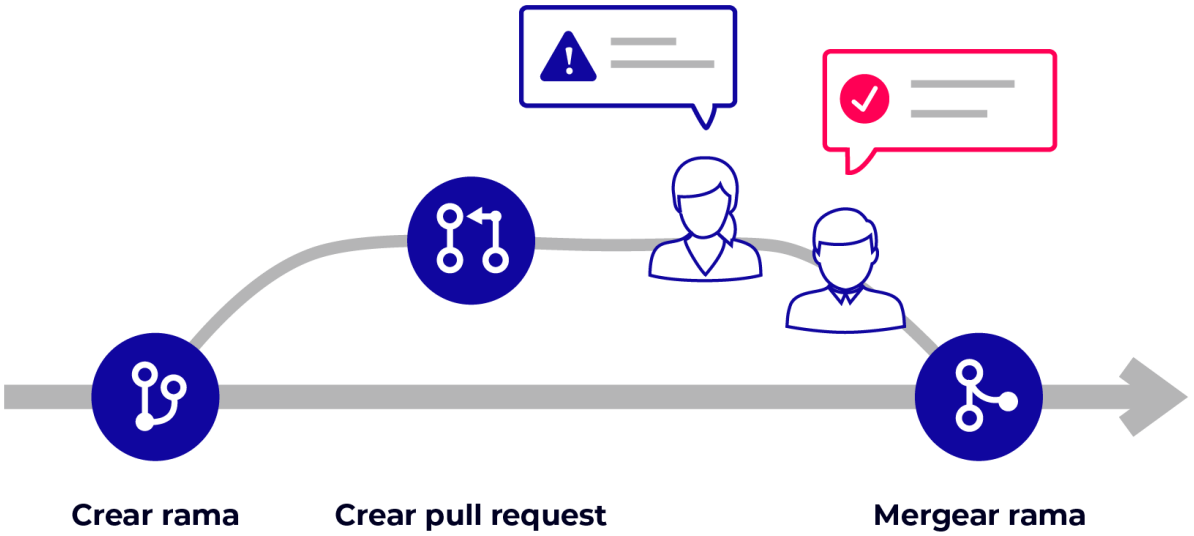
GitHub Flow

Es un flujo definido por GitHub para **estandarizar** la forma en la que todas las **contribuciones** en los proyectos comunitarios de GitHub tienen que hacerse. Está pensado para equipos y proyectos en los que se realizan **despliegues regularmente**. Este flujo también puede aplicarse fuera de GitHub como sucede con GitFlow, ya que sigue el mismo patrón de centralizado y distribuido.

En este modelo solo hay una regla: todo **el código que hay en la rama master tiene que poder desplegarse en cualquier momento.** Por este motivo, siempre que se quiera hacer un cambio, hay que crear una rama a partir de la rama master y darle un nombre descriptivo. En esta nueva rama creada se pueden hacer tantos commits como se quiera y experimentar, sabiendo que la rama no será fusionada (merge) con master hasta que alguien la revise y la apruebe. El mensaje incluido en cada commit tiene que ser descriptivo y detallado para que la persona o personas que revisen los cambios hechos en la rama puedan entender qué se van a encontrar al leer el histórico de commits.

Una vez terminado el trabajo en la rama y pasados todos los tests, se tiene que crear una **pull request** (petición de fusión) de tu rama con la rama master. Una pull request sirve para hacer "**code review**" (revisión de código), que es una técnica de mejora de calidad de código y transmisión efectiva del conocimiento, en el que cada funcionalidad o pieza de código que tenga cierta entidad pasa por un proceso de revisión donde las personas encargadas, que tendrán que ser añadidas para revisar esa pull request, comprueban el código y sugieren mejoras en el mismo a través de comentarios en la pull request. Una vez se resuelvan los comentarios o puntos pendientes y aprueben la pull request, esta se podrá fusionar con master.

Cuando los cambios han sido fusionados, se harán tests de integración para comprobar que funciona correctamente con el código que ya había y se desplegará una nueva versión en producción. Si en algún momento hay algún problema, se puede hacer "**rollback**" (volver a una versión anterior que funciona) usando la versión anterior de la rama master.



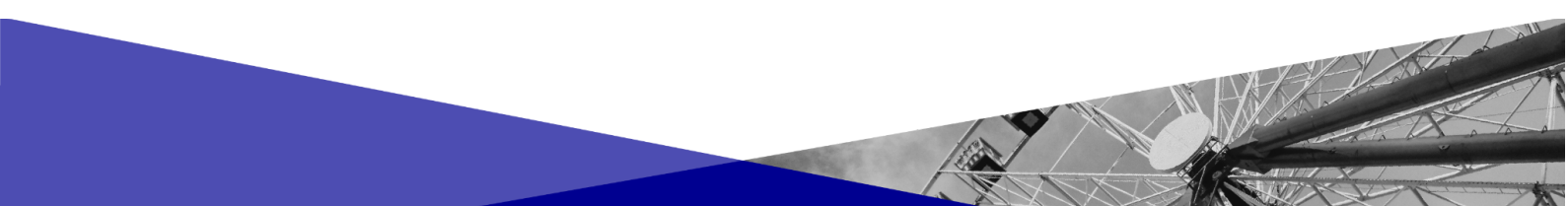
autentia

Trunk based development

Esta forma de trabajar apuesta **por no usar ramas**, para así evitar los merges grandes que pueden surgir de no hacer merge durante días o semanas. Para que esta técnica funcione los equipos deben ser maduros, ya que no están haciendo commits en una rama de código que nadie ve, sino en la rama principal que usan todos los desarrolladores. **Romper la build** y subir algo que no pase los tests, o que no compile, **puede parar a decenas de personas**, por ello el equipo debe ser experimentado. Es posible, además, que se suban cambios a la rama principal que no están acabados; es código que va a ir a producción y, sin embargo, no debe ejecutarse. Para esto **se usan las feature toggles**, de las que puedes leer más en el siguiente enlace.

La **ventaja principal de este enfoque es que las integraciones de los cambios en la rama principal aunque se hacen más frecuentemente, éstas son más sencillas de resolver ya que existen menos posibilidad de conflicto al ser más acotadas y por lo tanto se reduce el número de errores que estos conllevan**. Se puede encontrar más información en el artículo Trunk-Based development: pusheando a master.

Otro artículo que se podría considerar de obligada lectura para la profundización en los mecanismos o patrones de integración de ramas es: Patterns for Managing Source Code Branches



Entorno local

No todos los lenguajes son compilados, ciertos lenguajes como Java, Kotlin, C y otros, requieren el uso de un compilador. En el caso de algunos lenguajes, este viene incluído en el **IDE**. Pero normalmente queremos tener el compilador instalado y configurado por fuera del IDE para poder compilar o ejecutar la batería de tests y verificar que los cambios realizados recientemente no hayan introducido nuevos errores o bugs al código ya existente.

Otras herramientas que encontramos en un **entorno local de desarrollo** son aquellas que nos permiten ejecutar el código en un entorno similar al productivo con la principal finalidad de validar que la aplicación cumple con los requisitos funcionales mínimos y que no tiene fallos antes de publicar dichos cambios en los repositorios y/o promoverlos a otros ambientes. Entre estas herramientas podemos encontrar **servidores web** como Apache o Nginx, **servidores de aplicaciones** como Apache Tomcat, JBoss Wildfly, Jetty, Liberty, que incluso pueden ser ejecutados localmente dentro de contenedores Docker, o servidores de desarrollo incluidos en la tecnología que estemos usando como, por ejemplo, Angular CLI. Otras herramientas que se suelen utilizar para estas pruebas en el desarrollo mobile son los **emuladores de sistemas operativos móviles**, o incluso **un móvil conectado al ordenador** y al entorno de desarrollo, para ejecutar y verificar la aplicación en un entorno lo más cercano al real. Hablaremos de la función de estas herramientas más adelante.

Muchas aplicaciones utilizan **bases de datos** para el almacenamiento de información o datos del negocio. Para poder desarrollar en un entorno similar al productivo, muchas veces es necesario utilizar bases de datos de forma local, ya sean ejecutadas en memoria dentro de la aplicación cómo podrían ser H2 o SQLite, o instaladas localmente (o bien usando algún mecanismo de virtualización ligero como Docker), como PostgreSQL, MySQL o MariaDB entre otras. Además, estas bases de datos se suelen utilizar para tener algunos **datos de prueba**, y probar modificaciones sobre el esquema o los datos **sin impactar al resto de entornos que están siendo usados por muchas personas.**

Base de Datos

autentia

¿Qué es?

Una base de datos es una especie de "almacén" que nos permite guardar grandes cantidades de información para su posterior **recuperación, análisis y/o transmisión**. Podemos encontrar diversos tipos de bases de datos y se clasifican de acuerdo a las necesidades que busquen solucionar en dos grandes bloques.

TIPOS

Concepto	Definición	Ejemplos
Relacionales	Los datos se relacionan entre ellos a través de identificadores en tablas. El lenguaje predominante es SQL (Structured Query Language). Aportan a los sistemas mayor robustez y ser menos vulnerables ante fallos gracias a su **atomicidad, consistencia, aislamiento y durabilidad (ACID)**.	MySQL, PostgreSQL, Oracle, SQLite.
No relacionales	No siguen un patrón fijo como estructura de almacenamiento por lo que **su uso es muy común cuando no se tiene un esquema exacto de lo que se va a almacenar.** Se pueden encontrar bases de datos de gráficos, orientada a documentos, de columnas (Wide column store) o de claves-valor. Algunas desventajas es que no todas contemplan la atomicidad ni la integridad de los datos.	MongoDB, Redis, Elasticsearch, Cassandra, DynamoDB.

No Relacionales	Definición	Ejemplos
Clave-Valor	Cada elemento en la base de datos se almacena como un par (clave-valor) donde la clave sirve como un identificador único. Tanto la clave como valor pueden ser cualquier tipo de primitivo, como objetos compuestos.	DynamoDB, Redis, Riak, Voldemort
Columnas (Wide column stores)	También conocidas como familia de columnas o base de datos columnar. **Permiten manejar un gran volumen de datos** y mezclan conceptos de las bases de datos relaciones con una base de datos clave-valor. **Almacenan tablas de datos como secciones de columnas** en lugar de filas de datos y en cada sección se puede encontrar elementos con clave-valor	Cassandra ,HBase, Microsoft Azure Cosmos

Base de Datos

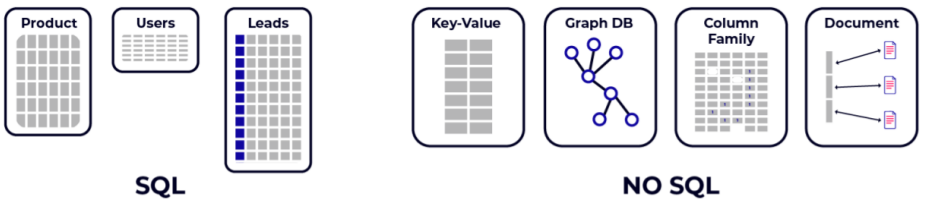
autentia

¿Qué es?

Una base de datos es una especie de "almacén" que nos permite guardar grandes cantidades de información para su posterior **recuperación, análisis y/o transmisión**. Podemos encontrar diversos tipos de bases de datos y se clasifican de acuerdo a las necesidades que busquen solucionar en dos grandes bloques.

TIPOS

No Relacionales	Definición	Ejemplo
Documentales	**Los datos se almacenan y se consultan como documentos tipo JSON**. Los documentos pueden contener muchos pares diferentes clave-valor, o incluso documentos anidados. Son más flexibles al cambio ya que si el modelo de datos necesita cambiar, solo se deben actualizar los documentos afectados y no todo el esquema.	MongoDB
De Gráfos	**Usan nodos para almacenar entidades de datos y aristas para almacenar las relaciones entre nodos**. El valor de estas bases de datos se obtiene de las relaciones entre nodos y no hay un límite para el número de relaciones que un nodo pueda tener. Un borde siempre tiene un nodo de inicio, un nodo final, un tipo y una dirección.	Neo4J, HyperGraphDB



Adicionalmente, podemos ejecutar en nuestro entorno local herramientas como **Sonar**, para realizar inspecciones preliminares del código y verificar su calidad. Estas verificaciones se pueden lanzar también mediante plugins integrados en el IDE.

En local funciona, y ahora qué...

Una vez que nuestro código compila y cumple los criterios de aceptación funcional solicitados ya es hora de subir nuestros cambios, dejarlos disponibles al resto de los desarrolladores y publicarlos en los entornos de prueba.

Para esto hay que realizar varios pasos, comenzando por hacer un push al repositorio GIT. Según la forma de trabajo del proyecto, esto puede requerir hacer un Pull Request para poder integrar nuestros cambios en la rama principal del proyecto.

Luego, habrá que compilar nuestro código y generar el entregable correspondiente, ya sea una librería o una aplicación. Dichos entregables se recomienda sean publicados en un repositorio de software, como Nexus o NPM para desplegarlos posteriormente en un ambiente de pruebas, ya sea subiendo mediante FTP o mediante herramientas de despliegue como las incluidas en los servidores de aplicaciones (aplicación web o despliegue remoto). Si hemos usado la opción del FTP, seguramente también tendremos que reiniciar algún servidor para que se apliquen las actualizaciones.

Por lo general, este ambiente de pruebas es conocido como **entorno de integración**.

Entorno de integración

Este entorno seguramente será **distinto** al nuestro de **desarrollo local**, ya que lo **ideal** es que sea lo más **parecido** posible al **entorno de producción**. Es deseable que a nivel infraestructura, este entorno utilice la misma versión de base de datos, el mismo sistema operativo, la misma versión de servidor web y / o de aplicaciones, sistemas de virtualización y orquestación, etc. Este entorno y el entorno de producción no van a usar la misma base de datos, cada uno tendrá la suya. Pero será el mismo software y en la misma versión para asegurarnos de que lo que funciona en un entorno, funcione en el otro. Seguramente, y por un tema de costes, este entorno no disponga de los mismos recursos de hardware que el entorno productivo, y por un tema de seguridad y privacidad disponga de datos de prueba distintos a los existentes en producción. La configuración y parametrización de este entorno será muy similar a la de los demás entornos pre-productivos e incluso al de producción, pero siempre pueden existir particularidades.

Todas estas tareas manuales, además de suponer mucho más trabajo, son un posible punto de generación o introducción de problemas o errores. Aquí es donde la filosofía DevOps viene al rescate, haciendo uso de conceptos como la integración y la entrega contínua (CI/CD), la infraestructura y la configuración cómo código (IaC, Infrastructure as Code, y CaC, Configuration as Code), y otros conceptos. Todo va encaminado a lo mismo: que el desarrollador se preocupe solamente de desarrollar, y tareas como la configuración de entornos, subida de artefactos, el despliegue de los mismos se realicen de forma automática y predecible.

Integración Continua (CI)

Supongamos que el equipo finalizó el desarrollo de un conjunto de funcionalidades que deben ser incluidas en la próxima release del producto. Se acerca la fecha de liberación de esta release, por lo que debemos fusionar (o mergear) toda esta nueva funcionalidad en la rama master del proyecto. Hacer merge de todos estos cambios en la rama principal no solo es una tarea ardua y complicada, sino que puede hacer que surjan problemas o errores inesperados y, lo que es peor, que algunos de estos errores o bugs se "filtren" a la rama master y, de esta forma, al entorno productivo. Cuanta más gente hay trabajando en el proyecto, mayor es la probabilidad de que esto ocurra.

Esto es lo que se suele conocer como "**Merge Day**", un día entero o gran parte del día donde se paran todos los desarrollos y todo se centra en conseguir que la fusión de todos los cambios en la rama master salga bien. Siempre hay conflictos en los merges y si no se tiene un buen conocimiento de git y del trabajo hecho, seguro que alguien pisa el cambio de otro y cuando se prueba la integración todo falla y nadie sabe el porqué. En ese momento comienza una búsqueda para encontrar el fallo y al culpable, lo que crea muchas tensiones en el equipo. Además de esto, imagina que esto ocurre un viernes y te tienes que quedar horas extras, pues no puedes irte de fin de semana teniendo fallos en el sistema.

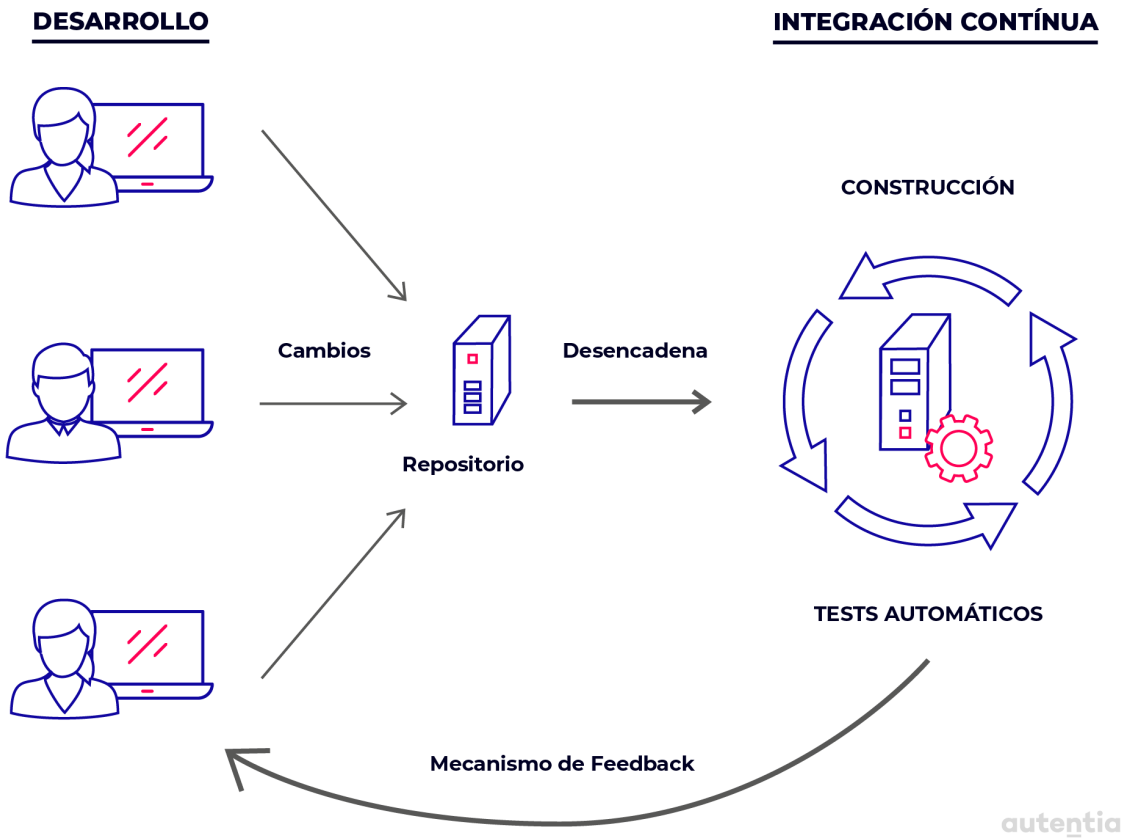
La Integración Continua (CI) surge para ayudar con esta parte del proceso. Tal cómo lo explica Martin Fowler en su artículo, ***"La integración contínua es una práctica del desarrollo de software en la que los miembros de un equipo integran su trabajo con frecuencia. Usualmente cada persona integra sus cambios al menos diariamente - lo que lleva a múltiples integraciones por día. Cada integración se verifica mediante una compilación automatizada, que incluye las pruebas y tests, para detectar errores de integración lo más rápido posible."***

Esta práctica permite, entre otras ventajas, **mejorar la productividad del desarrollo**, **reducir la cantidad de problemas** de integración, **detectar errores** y fallos **de manera temprana**, realizar actualizaciones y **entregar valor con mayor rapidez**.

Pero no hay que olvidarse de la otra gran ventaja que aporta esta práctica: **la transparencia del proceso**. Como todos trabajamos utilizando el mismo repositorio de código y la información del servidor de integración es

pública, para todo el equipo e incluso a veces para el cliente, se conoce con precisión el estado real del proyecto. No hay que esperar durante meses para saber cómo van las cosas y las reuniones de seguimiento pueden ser cortas y precisas.

Todos estos cambios entonces son integrados en un entorno en particular, comúnmente conocido como entorno de integración, donde cada cierto período de tiempo se ejecutan pruebas de regresión o de integración. Este entorno contiene siempre la última versión del código (no la última versión productiva) y puede utilizarse a su vez como entorno de pruebas y QA.



Despliegue continuo (CD)

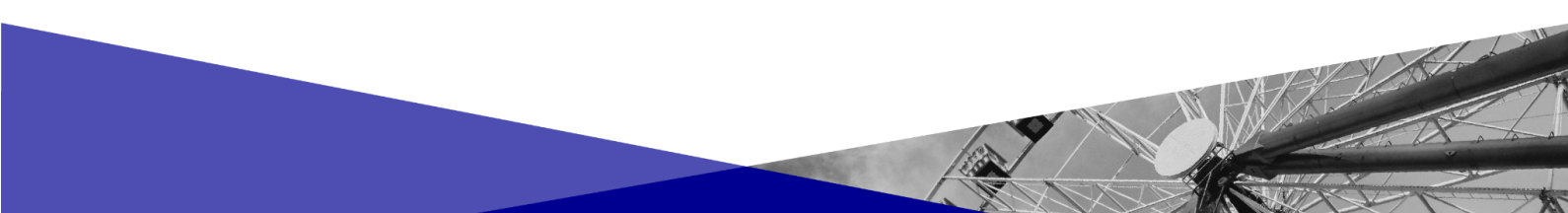
El despliegue continuo o entrega continua es **la capacidad de llevar cambios** de todo tipo (incluyendo nuevas funcionalidades, cambios de configuración, corrección de errores...) **al entorno de producción** o a manos de los usuarios finales de forma **segura**, lo más **automática** posible, **rápida** y **continua**.

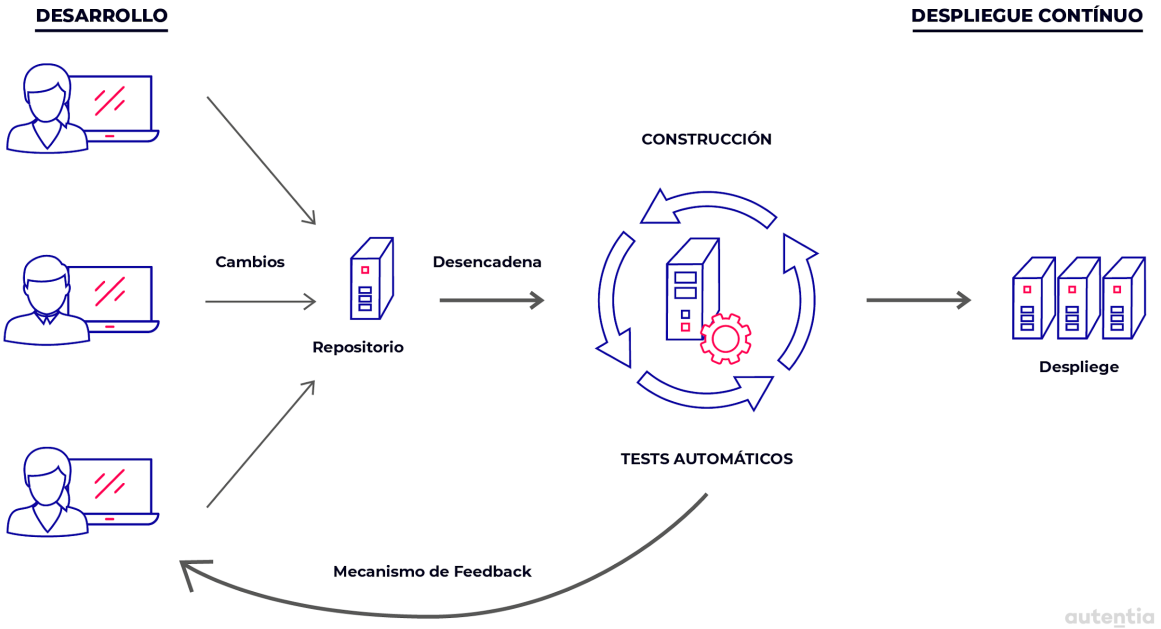
El objetivo es hacer **despliegues predecibles, automáticos y rutinarios** en cualquier momento, ya sea de un sistema distribuido a gran escala, un entorno de producción complejo, un sistema integrado o una aplicación.

Logramos todo esto asegurando que nuestro código esté siempre en un estado desplegable, incluso cuando estamos trabajando con equipos de miles de desarrolladores que realizan cambios a diario. Es decir, cualquier cambio que subamos ni rompe la funcionalidad ya existente ni introduce algún tipo de error que pueda llegar a causarle problemas a un usuario final.

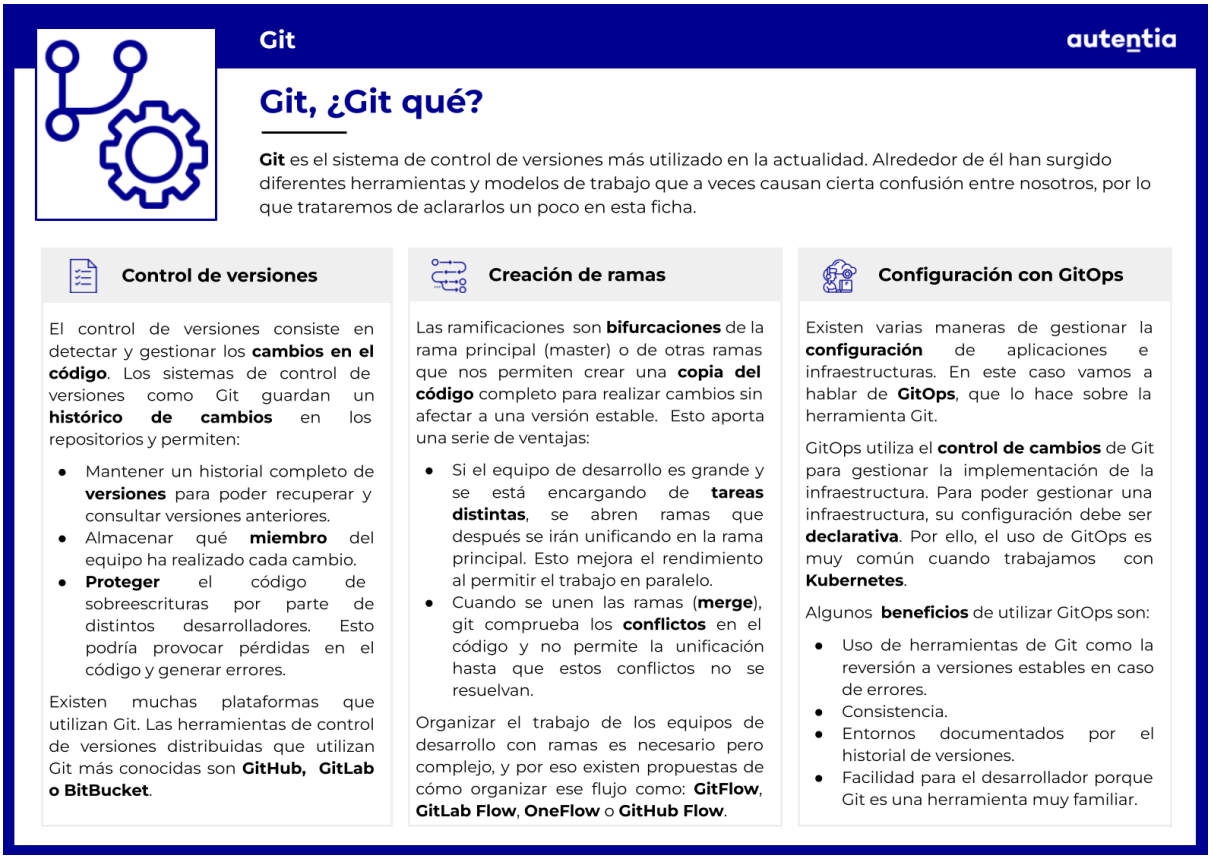
¿Por qué es importante la entrega continua?

La gente asume que si queremos desplegar software con frecuencia, debemos aceptar niveles más bajos de estabilidad y calidad en nuestros sistemas. **No tenemos por qué hacer despliegues renunciando a la calidad y estabilidad** del proyecto si tenemos un sistema bien configurado, con automatización de los procesos de despliegue, minimizando la intervención humana, mediante integración continua por ejemplo. Al automatizar todos esos procesos conseguiremos un sistema de despliegue que **siempre funcionará igual** y que **mantendrá la calidad** a la vez que **mejora los tiempos**, pues las máquinas son más rápidas que el hombre. ¿Te imaginas subir por FTP una aplicación para instalarla en 200 servidores? Esta capacidad de hacer entrega continua proporciona una ventaja competitiva increíble para los clientes, puesto que cualquier cambio que se requiera se podrá desplegar en minutos una vez esté implementado y probado.




DESARROLLO
DESPLIEGUE CONTÍNUO
CONSTRUCCIÓN
Cambios
Repositorio
Desencadena
Despliege
TESTS AUTOMÁTICOS
Mecanismo de Feedback
autentia

El planteamiento de GitOps



La idea detrás de **GitOps** fue en su inicio desarrollada y compartida por la compañía WeaveWorks. Se define como un sistema de despliegue basado en infraestructura como código que tiene como pieza central la herramienta Git. Es una **evolución de las mejores prácticas de DevOps e infraestructura como código (IaC)**. Por ponerlo de forma simple, GitOps es un 90% de buenas prácticas y un 10% de nuevos conceptos. Sus principales características son:

- Entorno definido mediante lenguaje declarativo.
- Git como fuente de la verdad.
- Despliegues automáticos bajo aprobación.
- Uso de agentes para asegurar que el despliegue ha sido correcto.

Con GitOps todos los despliegues los tenemos versionados en un repositorio de configuración de Git. El repositorio se modifica a través de pull requests, cuyos cambios tenemos que aprobar para luego unirse a la rama principal y de manera automática se configurará la infraestructura reflejando el estado del repositorio. Este **flujo contínuo de pull requests es**

la esencia de GitOps.

Los beneficios de este planteamiento son muchos. Nos proporciona transparencia y claridad gracias a trabajar con un repositorio central. El control de versiones nos da un histórico de la configuración de la infraestructura y podemos dar marcha atrás a cambios que rompen la funcionalidad existente o introducen un error. Mejora la productividad de nuestro equipo al poder experimentar sin miedo con nuevas configuraciones. Y finalmente la facilidad de adoptar como metodología al usar herramientas que nos son familiares.

En un pipeline CI/CD tradicional se realiza el proceso de construcción y a continuación el proceso de despliegue en el que llevamos los cambios al entorno de producción. En cambio, el pipeline de GitOps **separa el proceso de construcción del proceso de despliegue**. El proceso de construcción es idéntico, pero a la hora del despliegue existe un operador que monitoriza si hay cambios en el repositorio de configuración y cuando existan los va a llevar al entorno de producción. Este operador se encuentra entre el pipeline y nuestro orquestador, que podría ser Kubernetes u otra solución. Una explicación más en profundidad la tenemos en este artículo.

Beneficios de la integración y la entrega continua (CI/CD)

Las prácticas de la integración continua y entrega continua nos ayudan a lograr importantes beneficios:

- **Releases (lanzamientos de incrementos o mejoras del código) de bajo riesgo.** Hacer que los despliegues de software sean indoloros, procesos de bajo riesgo que se pueden realizar en cualquier momento, bajo demanda o automáticamente, según sea necesario.
- **Time to market (tiempo que pasa entre que surge la necesidad de aplicar un cambio hasta que está desplegado en producción para que pueda ser usado por el cliente) más rápido.** No es raro que la fase de integración, prueba y solución de posibles errores en el ciclo de entrega de software consuma semanas o incluso un par de meses. Cuando los equipos trabajan juntos para automatizar los procesos de creación de los artefactos, el sistema de despliegue, la configuración de los entornos y la ejecución de tests, los desarrolladores pueden trabajar de forma más eficiente, ya que tienen una red de seguridad gracias a todo el trabajo realizado y se evitan tener que hacer con cada cambio el mismo tipo de pruebas manuales que consumen tiempo y pueden conllevar errores.
- **Mejor calidad**. Cuando los desarrolladores tienen herramientas automatizadas de tests regresivos (comprueban que un cambio nuevo no quite funcionalidad existente), los equipos tienen la libertad de centrar su esfuerzo en otras actividades como por ejemplo, mejorar la calidad del código haciendo ejercicios de refactorización (simplificar la funcionalidad quitándole complejidad al código mejorando su lectura), hacer pruebas de usabilidad, rendimiento o seguridad. Al construir un pipeline de integración continua, estas actividades se pueden realizar continuamente durante todo el proceso de entrega, asegurando que la calidad se incorpore desde el principio.
- **Costes más bajos**. Cualquier producto o servicio de software exitoso evolucionará significativamente a lo largo de su vida útil. Al invertir en la construcción, prueba, implementación y configuración automática del entorno, reducimos sustancialmente el coste de realizar y entregar cambios incrementales al software al eliminar muchos de

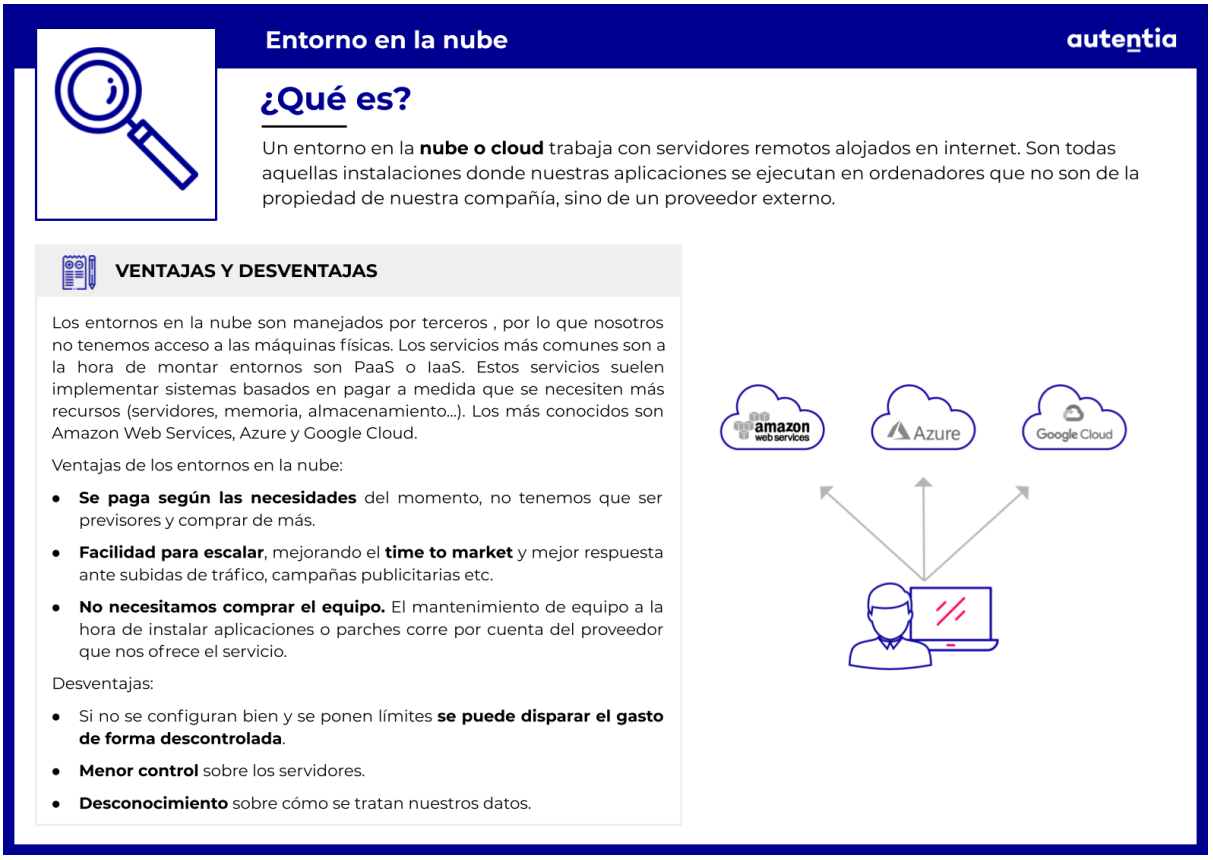
los costes fijos asociados al proceso de lanzamiento.

- **Mejores productos.** La entrega continua hace que sea económico trabajar en pequeños ciclos de semanas (normalmente de 2 o 3 semanas) llamados sprints, donde se define una funcionalidad a cubrir en ese tiempo. Al final de cada ciclo se produce un entregable y el cliente decide si lo quiere llevar al entorno de producción. De esta manera podemos obtener comentarios de los usuarios o el cliente a lo largo del ciclo de vida de entrega. Gracias a este enfoque podemos probar ideas con el cliente antes de desarrollar funciones completas. Esto significa que podemos evitar el problema recurrente de desarrollar una característica (feature) que creamos que ofrece un valor y que, una vez entregada al cliente, este compruebe que realmente la característica no aporta el valor esperado al negocio (otra vez la idea de “Falla rápido, falla barato”).
- **Equipos más felices.** Al ser todo el proceso más rápido, menos dado a errores y más automatizado, hacer cambios es menos doloroso, con lo que estaremos más atentos a las necesidades del negocio, habrá menos miedo al cambio y se reducirá el agotamiento del equipo. Además, cuando lanzamos con más frecuencia, los equipos de entrega de software pueden interactuar más activamente con los usuarios, aprender qué ideas funcionan y cuáles no y ver de primera mano los resultados del trabajo que han realizado. Al eliminar las actividades dolorosas de bajo valor asociadas con la entrega de software, podemos centrarnos en lo que más nos importa: **entregar valor al cliente**.

Un contraejemplo de las consecuencias de no utilizar este tipo de prácticas podemos verlo en el siguiente enlace donde se muestran evidencias de que puede salir muy caro hacer los procesos de forma manual. Esta vez en concreto se perdieron **450 millones** de dólares en 45 minutos, **por subir cambios de forma manual** en tan solo 7 de los 8 servidores. Si hubiera estado automatizada la tarea, este error, a todas luces humano, no hubiera sucedido.

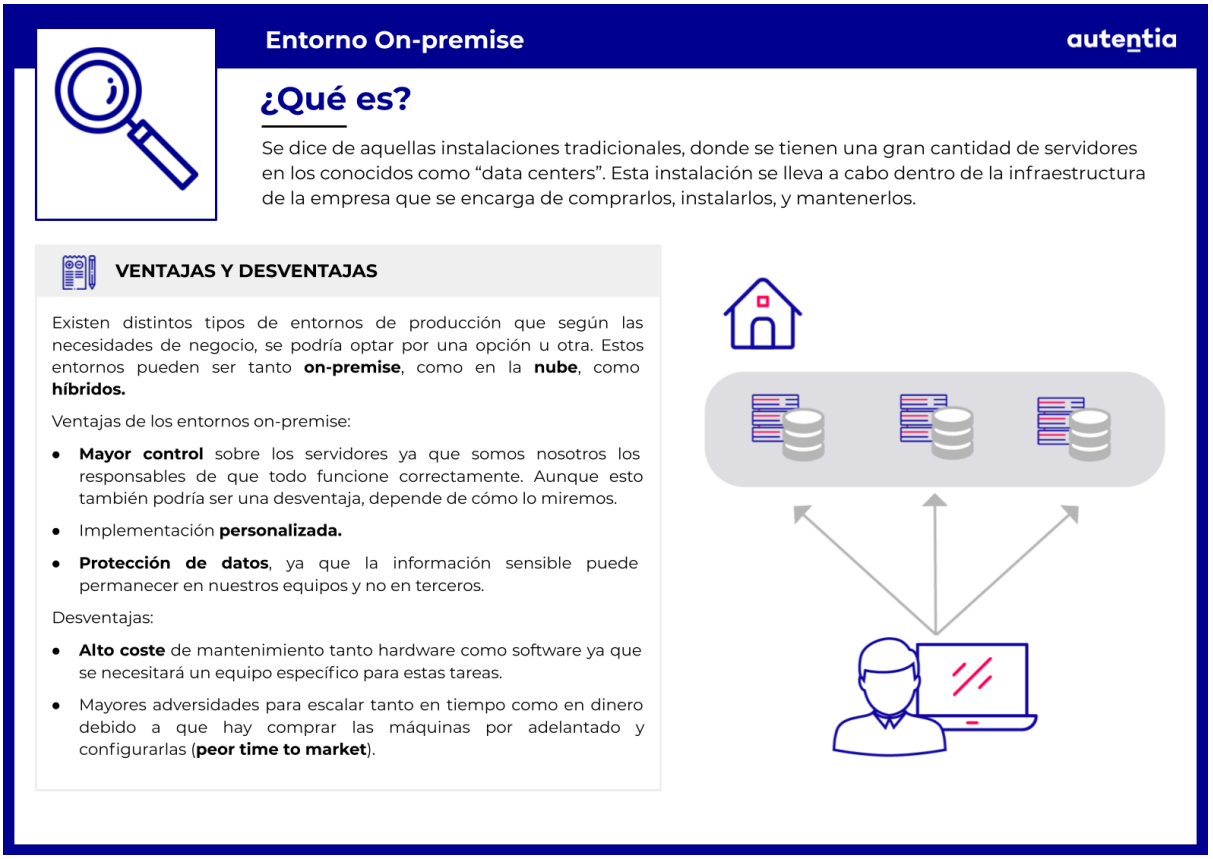
Infraestructura como código (IaC)

Ahora supongamos que nuestra empresa tiene una infraestructura montada, con bases de datos, servidores web y de aplicaciones, aplicaciones en distintos lenguajes, dominios, redes, firewalls, DNSs, etc., y se nos pide migrar toda esa infraestructura, por cambio de equipo físico (hardware) o proveedor de cloud, o incluso se nos solicita crear nuevas instancias de parte de esa infraestructura para poder atender la creciente cantidad de peticiones a un sitio web (escalado horizontal).



Esto nos supondría crear y configurar uno o varios servidores a mano, teniéndonos que apoyar en alguna guía de cómo hacerlo, y eso con suerte de que tal guía exista y no tengamos que crear nosotros dicha guía. Da igual que estos servidores sean virtuales (VPS - virtual private server) o físicos, situados en alguna instalación de la empresa (on premise). Además hay que agregar claves SSH, certificados, crear registros DNS, configurar redes, firewalls, dominios, balanceadores de carga, proxys, servidores web y de

aplicaciones. Cada uno con tecnologías distintas como pueden ser PHP, MySQL, Docker, Java, etc. Bastante trabajo, ¿verdad?



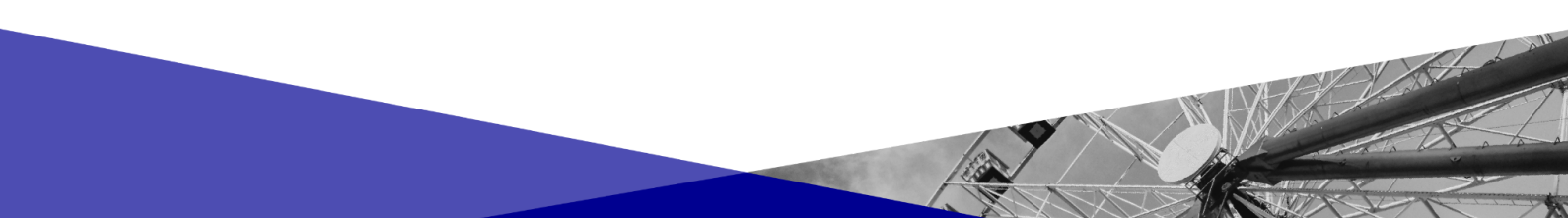
Ahora imaginemos que podemos tener toda esta infraestructura definida en ficheros, con todas las piezas (bases de datos, proxies, firewalls, servidores web y de aplicaciones, etc) que necesitamos para que nuestra aplicación funcione, y que dichos ficheros los tenemos versionados en nuestro repositorio de código, con un historial de cambios realizados, dándonos la posibilidad de automatizar el despliegue de toda o parte de la infraestructura y de volver fácilmente a una versión anterior en caso de un fallo. Esto es lo que se conoce por **Infraestructura como Código (IaC - Infrastructure as Code)**.

Una definición un poco más completa sería que la Infraestructura como Código es la administración de piezas de infraestructura (redes, máquinas virtuales, balanceadores de carga, topologías de red, servidores web y de aplicaciones, bases de datos, etc.) en un modelo descriptivo, el cual se puede versionar en el mismo sistema de versionado utilizado por el equipo de desarrollo. Es un método de aprovisionamiento y gestión de

infraestructura mediante código, que hace que la tarea de crear, modificar y mantener instancias de infraestructura sea más fácil y rápida ya que utilizando ese código y lanzando los mismos comandos conseguiremos el objetivo. Toda esta administración y configuración de infraestructura, o modelos, se termina reflejando en ficheros, comúnmente en formato Yaml o JSON, que son versionados para poder tener una historia con todos los cambios que se fueron realizando.

Siguiendo con el principio de que el mismo código fuente genera el mismo binario, un modelo de IaC genera el mismo entorno cada vez que es aplicado.

IaC nos permite reforzar la consistencia mediante la representación del estado deseado de un entorno mediante código, reduciendo la deriva de la configuración. Esto se refleja en un desarrollo más rápido y eficiente, con una minimización de riesgos, permitiendo la reusabilidad y la automatización. Como resultado, podremos ofrecer entornos estables rápidamente y a escala, mejorando el tiempo de salida a producción / mercado (time to production / market).



Configuración como código (CaC)

La Configuración como Código (CaC - Configuration as Code) puede parecernos, en primera instancia, algo igual o muy similar a la infraestructura como código, pero no lo es. IaC es la administración y aprovisionamiento de infraestructura mediante código, en lugar de manualmente, para crear y configurar sistemas y/o servicios.

La configuración como código es un proceso para administrar datos de configuración de una aplicación. Claramente, hay algunas áreas donde IaC y CaC se superponen, pero ambas administran dos aspectos distintos del entorno.

Una definición más precisa de CaC sería: *"**La configuración como código es la migración formal de la configuración entre entornos, respaldada por un sistema de control de versiones**"*.

Ahora imaginemos que un pequeño cambio en la configuración de nuestra aplicación, implica un redespliegue de la misma. Volver a desplegar una aplicación completa, en todas las instancias o entornos que tengamos dependiendo de nuestra infraestructura, por cambiar solo una propiedad en la configuración no tiene mucho sentido y puede llevar mucho tiempo. Es una práctica bastante frecuente (dejemos el debate de si es buena o mala para las charlas de café) mantener repositorios y pipelines separados para la configuración, para no tener que caer en el redespliegue de la aplicación completa si solo ha habido un cambio de configuración.

Aunque no nos hayamos percatado, ya ponemos en práctica ciertos conceptos de CaC en nuestro trabajo diario. El ejemplo tal vez más claro es en las aplicaciones Java con Maven, donde tenemos ficheros XML para el manejo de dependencias, ficheros de properties (o Yaml también en versiones más recientes) para el manejo de atributos de configuración, etc. Estos ficheros están versionados junto con el código, y se despliegan junto con el mismo. En aplicaciones con poca configuración o que no cambie con mucha frecuencia, este enfoque es más que suficiente.

Pero también podemos hacer esto por ejemplo con una aplicación desplegada en Kubernetes. Toda su configuración la podemos depositar en un repositorio de configuración por entorno independiente de la aplicación.

En este caso, puede suceder que, por ejemplo, debamos cambiar la dirección de acceso a proveedor externo en un determinado entorno. Resulta muy útil poder hacer un cambio únicamente de configuración en el entorno deseado, sin tener que compilar y desplegar toda la aplicación de nuevo.

Beneficios de la IaC y la CaC

Estos son algunos de los beneficios que se obtienen al implementar filosofías como IaC y CaC:

- **Trazabilidad**: Esto se desprende del uso de sistemas de versionado para almacenar los ficheros de modelado y configuración, pudiendo, por ejemplo, volver a una versión anterior mientras se rastrea cual fue el cambio que introdujo un error determinado, reduciendo el downtime (tiempo sin servicio) y facilitando el proceso de detección y corrección del error.
- **Reutilización**: Al igual que en el desarrollo, una misma pieza de código genera siempre el mismo binario, un mismo script de IaC genera siempre una misma infraestructura. Esto permite desde instalaciones rápidas, migraciones por cambio de equipos o proveedor, hasta replicado de entornos enteros.
- **Agilidad y eficiencia**: La infraestructura como código automatiza la mayor parte de la administración de recursos, lo que ayuda a optimizar el ciclo de vida del desarrollo del software.
- **Alta escalabilidad**: Al utilizar IaC y CaC y trabajar con recursos de nube virtualizados, se pueden aprovisionar instancias de aplicaciones adicionales con facilidad y, literalmente, en poco tiempo. También se pueden liberar tan pronto como no se necesiten, lo que garantiza una rentabilidad incomparable.
- **Reducción de riesgos e infraestructura inmutable**: Como todos los entornos se aprovisionan de manera automatizada mediante mecanismos de CI / CD, no hay lugar para errores humanos. Esto garantiza la uniformidad del sistema en todo el proceso de entrega y elimina el riesgo de deriva de configuración, una situación en la que diferentes servidores contienen diferentes configuraciones y/o versiones de software debido a actualizaciones manuales realizadas en varias ocasiones a lo largo del tiempo.
- **Mejora de seguridad**: Si todos los servicios, ya sean de almacenamiento, redes, etc., son provisionados mediante scripts,

entonces se implementan de la misma manera cada vez que se ejecutan. Esto significa que los estándares de seguridad pueden implementarse de manera fácil sin tener que hacer que un rol de seguridad revise y apruebe cada cambio.

- **Mejor documentación**: Al igual que pasa con las aplicaciones, la verdad está en el código. En el caso de IaC y CaC solo tenemos que mirar el repositorio para saber el estado y la configuración de cualquier máquina en cualquier entorno. No hace falta meterse en un servidor a revisar y tener acceso a él. El repositorio nos dice exactamente lo que tiene una máquina.

Entornos de preproducción

Una vez hemos **completado el desarrollo y pruebas en local y en integración** de la nueva funcionalidad que se va a entregar en la siguiente versión, **necesitamos probar** también en algún **entorno lo más parecido al de producción**. Este entorno es el de **preproducción**. Normalmente solo se puede utilizar por los desarrolladores y/o equipos de QA y quizás por un pequeño grupo de usuarios para poder probar y asegurarnos de que esté todo correcto y que cuando se exponga en producción no haya ningún fallo.

Aunque este entorno de preproducción debería ser la réplica más exacta posible de producción, lo más común en la mayoría de casos es que los recursos destinados a este entorno sean inferiores por un simple tema de costes.

Normalmente se tiene un entorno de producción, pero se puede llegar a tener varios entornos de integración y algún entorno de preproducción (usualmente se tiene uno). Esta situación de tener tantos entornos, y que solo uno de ellos lo vayan a usar los usuarios finales significa que **las empresas tienen que invertir mucho dinero en entornos que no son productivos**. Las empresas tratan de reducir costes, porque no tiene mucho sentido tener 40 servidores productivos y 200 no productivos. Es por esto que **los entornos que no son de producción tienden a tener menos servidores, menos licencias de pago y, en general, menos recursos**.

Asimismo, cuando nuestra empresa tiene que integrarse con servicios de terceros, estos suelen darnos acceso a sus entornos de preproducción para poder probar las funcionalidades antes de pasar a producción. **Estos entornos de terceros suelen**, al igual que los nuestros, **estar dimensionados de forma distinta a producción**. Lo normal es que tengan un 50% de capacidad o menos que el entorno de producción. Esto tiene implicaciones a la hora de hacer pruebas de rendimiento, pues tendremos que ajustar la volumetría de dichas pruebas a la capacidad de nuestros entornos y de los terceros con los que nos integramos, extrapolando y prediciendo cómo será su comportamiento en producción.

En estos costes que tiene un entorno entran muchas variables y motivos por los que tendremos que buscar alternativas más económicas que simulen lo más fielmente lo que hay en producción para poder tener entornos preproductivos que sirvan para probar funcionalidades destinadas

a producción.

Como mínimo, hay que tener 2 entornos de preproducción siempre disponibles que cumplan las siguientes funciones:

- **Entorno de integración**: Se utiliza para la **integración del código desarrollado por todos los equipos** y la **realización de pruebas** de todo tipo sin que ningún cliente o persona ajena al desarrollo pueda entrar en este entorno. Este entorno es un **entorno seguro de desarrollo**, que actúa de primer filtro y red de seguridad de los programadores y que permite momentos de cierta inestabilidad. No es lo mismo que se caiga este entorno durante una hora y tengamos a 100 desarrolladores con un retraso para entregar lo que han hecho, que el entorno de producción caído durante 1 hora, y dejar de vender nuestro producto, por ejemplo. Esto no quiere decir que si rompemos este entorno nos podemos ir a casa y no pasa nada. **Lo ideal es que esté estable y levantado casi el 100% del tiempo**. Todas las pruebas se pueden hacer en este entorno sin miedo a las repercusiones. Por dar una cifra de ejemplo, este entorno, o entornos (porque puede haber varios) suelen estar dimensionados de forma que tienen el 50% de la capacidad de un entorno de preproducción o incluso menos.
    - *Como este entorno no es de producción, se trata de un entorno "preproductivo", término que engloba tanto al entorno de integración como al de preproducción, pero para distinguirlos, a este se le llama de integración. Su estabilidad, seguridad, propósito y recursos dedicados suelen ser diferentes.*
- **Entorno de preproducción**: Una vez superadas de manera exitosa las pruebas pertinentes, el código se desplegará en este entorno. Aquí podrían entrar equipos de QA (Quality Assurance) para **realizar tests extras**, gente de **negocio para poder comprobar la funcionalidad** de primera mano antes de que llegue a producción y **hasta un pequeño grupo de usuarios objetivo** para que den sus impresiones, ya que este entorno también puede servir de escaparate para captar potenciales nuevos usuarios. Por ello, **este entorno es el que va a tener más recursos destinados de entre todos los entornos "pre productivos" y en el que se va intentar tener todo los más parecido a producción,** pues después de pasar todos los filtros requeridos, el siguiente paso será el despliegue en producción. Dependiendo del proyecto, puede que sea importante que este entorno esté siempre disponible y no haya caídas, a diferencia del entorno anterior; por ello también es

posible que el cliente quiera que haya un equipo monitorizando el entorno 24/7 (siempre) y avisando si se detecta algún problema. Este entorno se suele llamar Staging, otros lo pueden llamar Sandbox y otros UAT (user acceptance testing). Este entorno suele tener un 50% de la capacidad del entorno de producción. Puede haber clientes en los que tenga un 100% de capacidad, porque los entornos sean prácticamente iguales, y otros casos en donde se tenga un 10% de capacidad.

El nombre de estos entornos no es lo más importante, pues la convención de nombres puede variar dependiendo del proyecto o del cliente. Lo realmente importante es tener entornos separados que cumplan estas funciones.

Además de estos dos entornos, se pueden tener más entornos si el cliente o el equipo de desarrollo lo cree necesario para poder realizar el trabajo; depende de las necesidades y presupuesto del proyecto.

La información almacenada en la base de datos de los distintos entornos de preproducción será distinta y estará separada por entornos. Esta separación se puede hacer teniendo instancias distintas por cada entorno o dentro de la misma instancia de base de datos, creando esquemas distintos o hasta incluso en el mismo esquema pero en tablas distintas por cada entorno. **La forma de separar los entornos en la base de datos dependerá del proyecto. Lo ideal y recomendable es que la base de datos de producción esté totalmente separada del resto de entornos, para que sean independientes a nivel de infraestructura**, y si tiramos una base de datos de pre, por pruebas de carga, que no afecte al entorno de producción. Es importante que el modelo de datos o estructura (DDL, Data Definition Language) sea igual en todos los entornos aunque los datos (DML, Data Manipulation Language) puedan ser distintos.

Con estos entornos hay que tener en cuenta que, al estar segregados y ser distintos, las personas que puedan usarlos también pueden ser distintas. No todo el equipo de desarrollo tiene por qué tener acceso a todos los entornos de preproducción; dependerá del proyecto. El único entorno preproductivo al que todos los desarrolladores siempre tendrán acceso será el primer entorno, que nosotros hemos dado a conocer como “entorno de integración”.

Entorno de producción

Haciendo un resumen de todo lo que hemos visto hasta ahora, hemos pasado de una idea de negocio a un desarrollo de un equipo, que se ha puesto en un entorno de integración junto con muchas otras ideas de negocio ya plasmadas en nuevas funcionalidades. Se han corregido ciertos comportamientos que no eran como se esperaban. Luego se ha promocionado la versión de la aplicación al entorno de preproducción, donde se han hecho más pruebas y un manager de negocio incluso ha probado como funciona, dando el visto bueno. Esto significa que en el siguiente “pase” a producción la versión llevará los cambios que hemos ido siguiendo desde el principio de este documento. Por fin hemos llegado al entorno de producción.

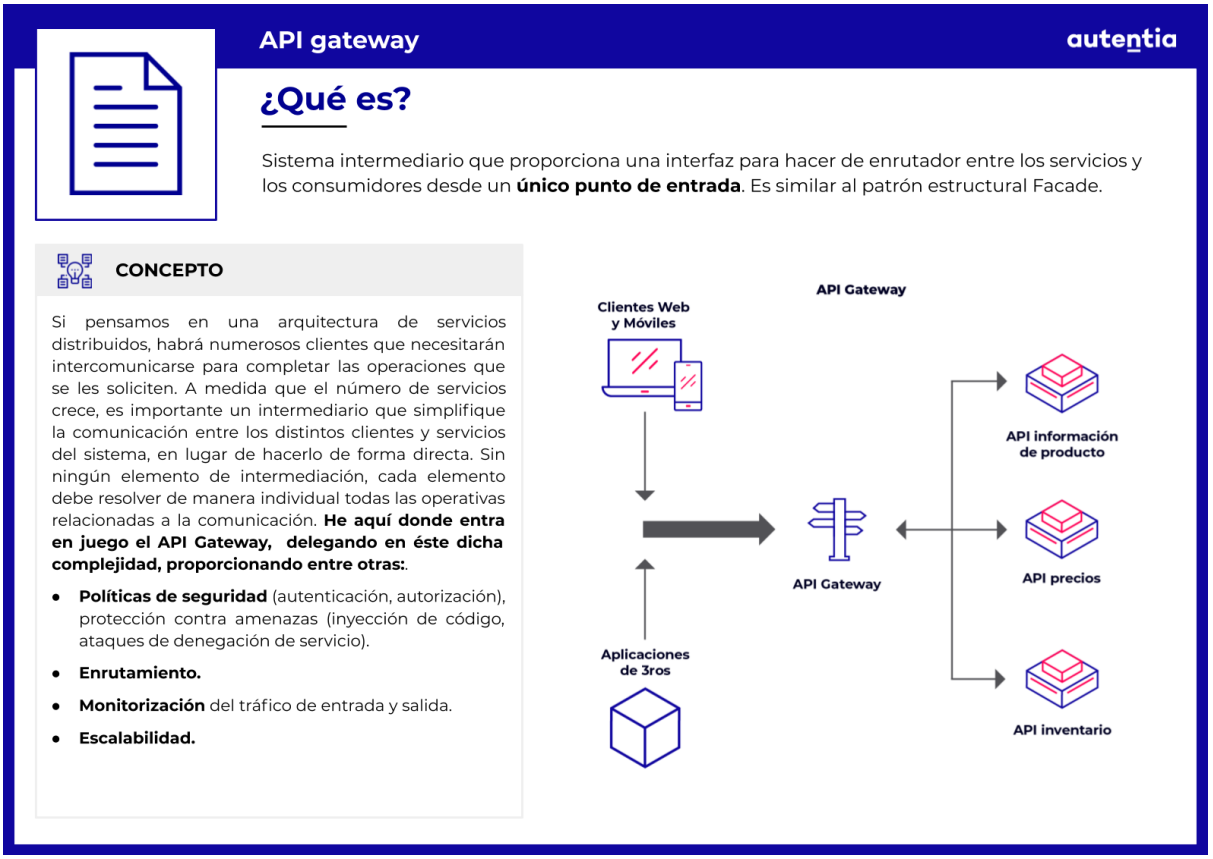
Infraestructura y servicios

El entorno de producción es toda la infraestructura y servicios relacionados con nuestras aplicaciones que usan los usuarios finales, y que nosotros controlamos de forma directa o indirecta. El entorno de producción no es el móvil del usuario donde se ejecuta nuestra aplicación, ni la página web que se le pinta al usuario final cuando accede a nuestra tienda. Para que se entienda mejor, tenemos los siguientes ejemplos de entornos de producción:

1. Los **servidores web** que sirven la página y exponen los servicios a los clientes.
2. La **CDN** externa que tenemos contratada, que cachea dicha página y elementos estáticos, como javascript, css e imágenes.
3. El **API gateway** que utilizamos para securizar nuestras APIs, monetizarlas y controlar su uso.
4. El sistema de **caché** distribuido que tenemos.
5. El servidor de **autenticación/autorización** que gestiona a los usuarios (login, registro, borrado de cuenta...)

6. La aplicación con la que **monitorizamos** el estado de la infraestructura y lanzamos alertas.
7. La aplicación con la que recogemos las **trazas** de producción y recopilamos información del uso de nuestros servicios.
8. La aplicación que utiliza negocio para ver si se están cumpliendo los **KPIs (Key Performance Indicator)**.
9. Todos los servidores que ejecutan de forma directa nuestras aplicaciones, con toda su **infraestructura, procesadores, memoria, red, discos duros**. Tanto si son **virtuales**, como si son **físicos**. Tanto si son **on premise**, como si son **cloud**.
10. La **base de datos** que contiene la información de producción, con los usuarios registrados, nuestros productos...
11. El **firewall** que da acceso a toda nuestra infraestructura.
12. Los **balanceadores** que redirigen la carga a las máquinas.

Si, para ofrecer algunos de nuestros servicios, necesitamos integrarnos con un tercero, estaremos usando SU entorno de producción, pero como no está dentro de nuestra organización y no tenemos control sobre el mismo, eso no se considera parte de nuestro entorno de producción. En nuestra mano está la correcta configuración de los mismos. Pero si esta configuración es correcta, y esos entornos fallan, habrá casos en los que nosotros fallaremos también sin posibilidad de poder hacer nada por nuestra parte salvo esperar a que se arregle.



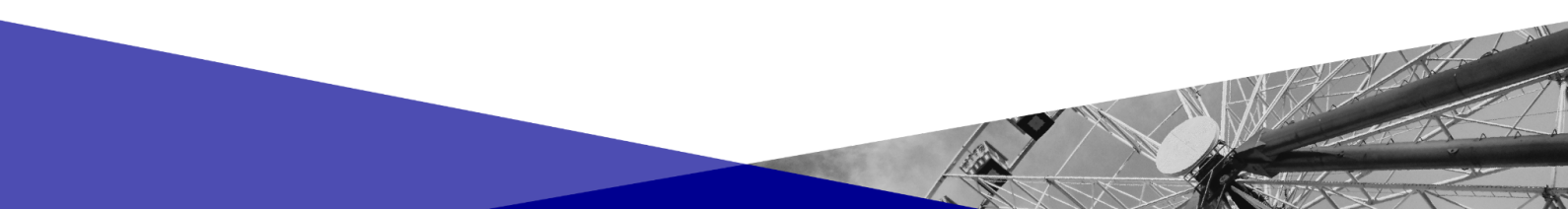
Características de un entorno de producción

Los entornos de producción, por lo general:

- Cuentan con **más recursos hardware** que los entornos previos.
- La **seguridad es mucho mayor**. Pueden tener servicios de terceros para controlar el acceso, y filtrar el mismo.
    - Suelen estar **monitorizados** 24/7 con personas dedicadas a ello.
    - Suelen contar con un equipo de **intervención rápida**, con disponibilidad 24/7.
    - Suelen tener **alta disponibilidad**. Esto significa que su infraestructura está dividida en varias localizaciones físicas, separadas entre ellas varios kilómetros, y ante un desastre natural (como inundación, incendio o una fuerte tormenta) puede seguir dando servicio.

    - El **número de personas** que pueden acceder a ellos a distancia o de forma física y operar en los mismos es **muy reducido** y está muy controlado.
    - En el caso de exponer un API se utilizará **comunicación segura HTTPs** y se hará uso de un certificado digital.

- Es un **entorno muy delicado**, cualquier metedura de pata, dependiendo del negocio, puede tener costes muy altos. Todos los procesos deberían estar automatizados, por lo tanto.
- Es crítico **conocer** exactamente las **versiones desplegadas** de nuestras aplicaciones.
- La **forma de desplegar esas aplicaciones puede variar** frente a entornos previos para garantizar en todo momento la alta disponibilidad.
    - Sería ideal que la mayor parte del **proceso de despliegue** esté **totalmente automatizado**.
    - También sería ideal que la mayor parte del **proceso de rollback** a una versión estable anterior esté **totalmente automatizado**.



Alta disponibilidad

autentia

¿Qué es?

En inglés **High Availability (HA)**, se aplica cuando queremos tener un plan de contingencia sobre cualquier componente en caso de que se presente alguna situación irregular que impida la continuidad de los servicios.

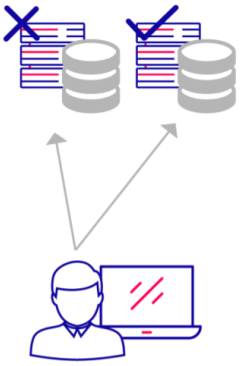
¿EN QUÉ CONSISTE?

Disponibilidad se refiere a la habilidad de los usuarios para acceder y usar el sistema. Se debe tener en cuenta que el tiempo de funcionamiento y disponibilidad no son sinónimos. Un sistema puede estar en funcionamiento y no disponible como en el caso de un fallo de red. **La criticidad del servicio es muy importante**, no podemos comparar la disponibilidad que debe tener un cajero automático (posiblemente 365 días 24x7) que otro servicio que solo esté disponible de Lunes a Viernes. Algunas **causas en la que podemos dejar de prestar servicio** ajenas a nuestra voluntad pueden ser:

- **Desastres naturales** (terremotos, inundaciones, incendios).
- **Cortes de suministro eléctrico.**
- **Errores operativos.**

Tener un sistema altamente disponible puede lograrse utilizando servicios cloud, los más conocidos pueden ser Amazon Web Services, Google Cloud o Azure. Necesitamos tener la capacidad de detectar un fallo en el servicio principal de una forma rápida y que a la vez, sea capaz de recuperarse del problema. La **monitorización** y **auditoría** de los componentes es fundamental para saber en tiempo real que está ocurriendo o como se comporta el sistema.

Si hablamos de un e-commerce se podría ofrecer el uso de **balanceadores de carga**, donde en vez de tener un único servidor, balanceamos la carga en varios servidores según la demanda que tengamos, de este modo, reduciremos las posibilidades de que el servidor principal colapse. Otras soluciones a tener en cuenta podría ser **dividir geográficamente la infraestructura en varias localizaciones físicas**, separadas entre ellas varios kilómetros, que ante cualquier desastre natural, pueda seguir prestando servicio.



Certificado

autentia

¿Qué es?

Documento virtual que contiene los datos identificativos de una persona física o de un sitio web y que está autenticado por un organismo oficial encargado de emitir y validar dicha información.

TIPOS

Cuando queremos identificar a una persona física, una entidad o un dominio de manera digital, hablamos de **certificados digitales**. Una **Autoridad Certificadora (AC)** será la encargada de firmar y emitir los certificados, verificando que quien lo solicite es quien dice ser.

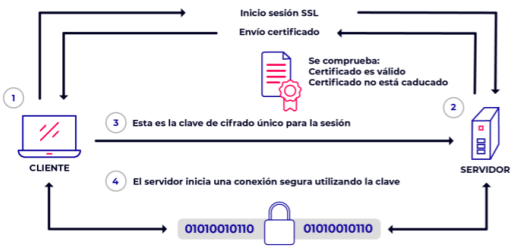
También podemos validar un sitio web a través de certificados. Se configura el servidor para que use dicho certificado y usando el protocolo SSL se cifra la información enviada al servidor. Esto es lo que comúnmente se conoce como **certificado SSL** debido a que este protocolo es el más extendido hoy en día. Los navegadores suelen incluir por defecto un repositorio de AC de confianza, pero si hay alguna AC en la que en principio no se confía (porque lleva poco tiempo en funcionamiento, por ejemplo), habrá que instalar manualmente el certificado en el repositorio del navegador. **Los certificados caducan** y esto se hace para asegurar que toda la información es precisa y demuestra una validez como propietario de confianza del dominio

¿Qué ventajas nos ofrecen los certificados?

- Ahorro de espacio, tiempo y dinero.
- Confidencialidad y seguridad en las comunicaciones.

¿CÓMO FUNCIONA?

Se hace uso de claves públicas y claves privadas. En el caso de un sitio web, la clave privada permanece en el servidor donde se ha configurado el certificado. Cuando un navegador web desea establecer una nueva conexión, el servidor comparte la clave pública con el cliente para establecer un método de cifrado y el cliente confirma que reconoce y confía en la entidad emisora del certificado. Este proceso inicia una sesión segura que protege la privacidad y la integridad del mensaje.



DevOps, DevOpsSec y demás siglas

DevOps es una combinación de las abreviaturas "**Dev**" (desarrollo) y "**Ops**" (operaciones). Podríamos definirlo como un marco de trabajo en el que tanto el departamento de desarrollo como el de operaciones colaboran aportando ideas, pruebas prácticas y el uso de nuevas tecnologías para conseguir mejorar los procesos de desarrollo del software. En ningún caso, se trata de aumentar la responsabilidad de los desarrolladores, haciéndoles responsables de funciones del departamento de operaciones.

- Últimamente también se están oyendo las siglas **DevOpsSec**, que implica la participación de un nuevo actor, el departamento de seguridad para forzar a que se piense en la seguridad desde el principio, evitando el enfoque tradicional de auditar la seguridad del software o la infraestructura únicamente cuando se sale a producción. Algunas de las formas que tiene el departamento de seguridad de incorporar la seguridad durante todo el ciclo de desarrollo son:

- Incorporar el escaneo de vulnerabilidades en el ciclo de desarrollo con herramientas como Anchore o SInk.

- Estandarizar y automatizar las políticas de seguridad concediendo a cada servicio el privilegios mínimos necesarios para poder ejecutarse.

- Centralizar la identificación de usuarios y control de accesos con servicios como IAM (Identity and Access Management) de AWS.

- Incorporar dentro del pipeline de CI, herramientas que realicen análisis estático de código centrado en la seguridad con herramientas como Kiuwan Code Security o el propio Gitlab.

- Utilizar entornos aislados y que usen lo imprescindible para realizar su función como los contenedores de docker, nos permite reducir la superficie de ataque.

- Automatizar las actualizaciones de seguridad y parches de vulnerabilidades conocidas.

- Hacer cumplir las políticas de seguridad, modificando los ficheros de

configuración de los entornos que se despliegan automáticamente, evitando errores manuales.

- Uso de cifrado punto a punto y APIs seguras.

Todas las técnicas o medidas expuestas son compatibles con la realización de Pentesting o pruebas de intrusión tanto a la aplicación como a la infraestructura.

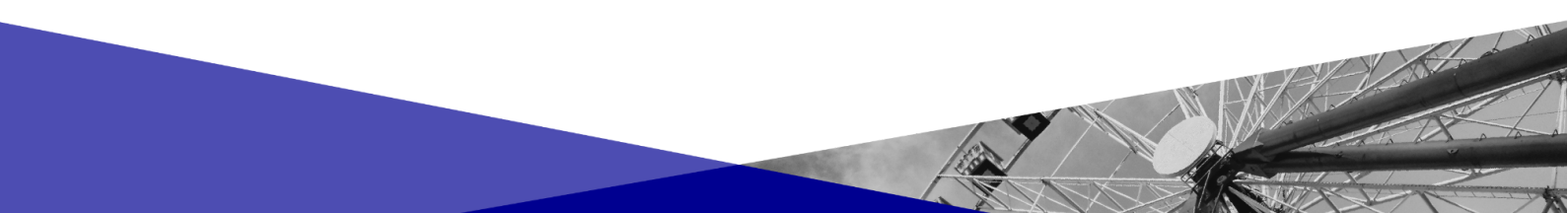
Parte 2

Piezas básicas de la infraestructura

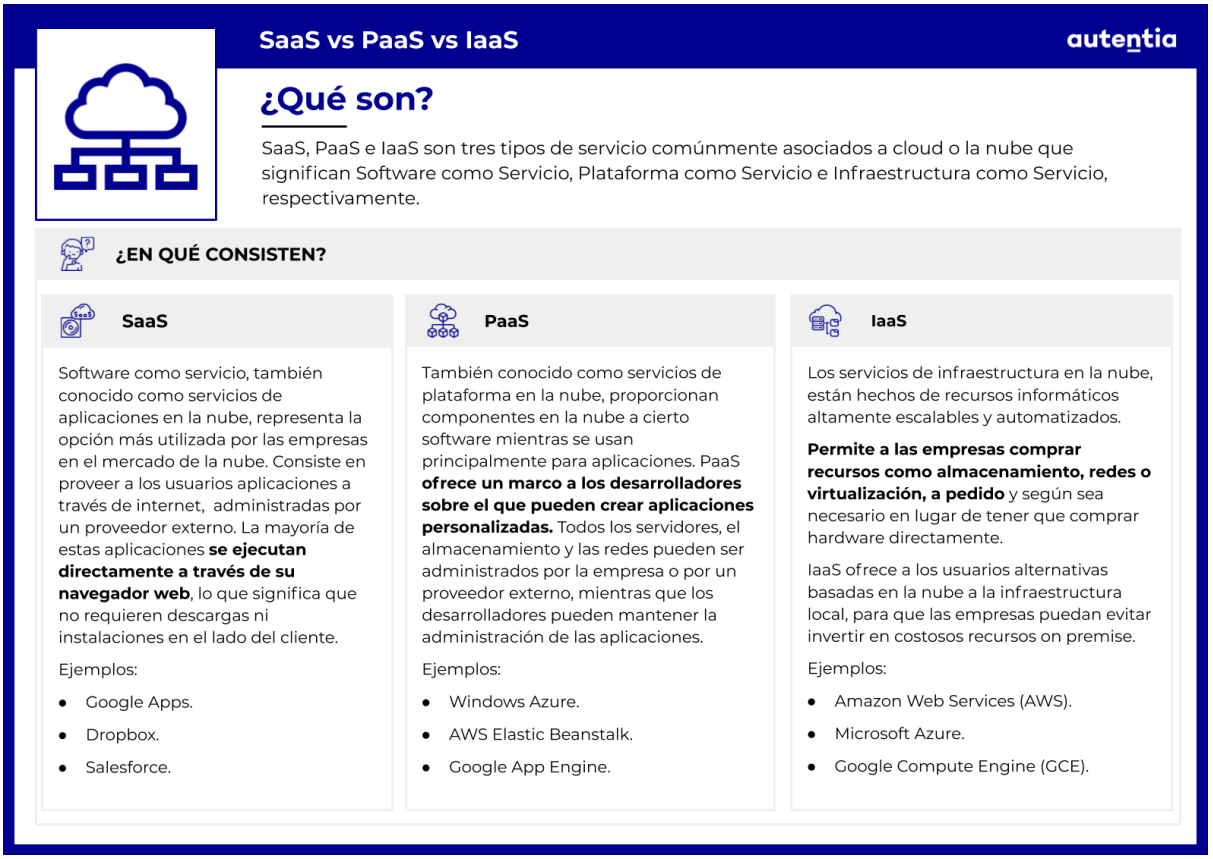
¿En la nube o no? Ese es el dilema...

En el capítulo anterior hemos visto todas las piezas que se utilizan a la hora de desarrollar software, así como el camino que éste sigue hasta alcanzar el entorno productivo, donde los usuarios finales pueden hacer uso de nuestra aplicación. Esto es así cuando las aplicaciones se llevan al entorno de producción. A esta acción de mover una aplicación de un entorno a otro (normalmente más cercano al usuario final y más lejano al desarrollador software) se le conoce también como **promocionar**. Cuando una aplicación es promocionada al entorno de producción, la versión anterior de dicha aplicación es sustituida por la nueva versión.

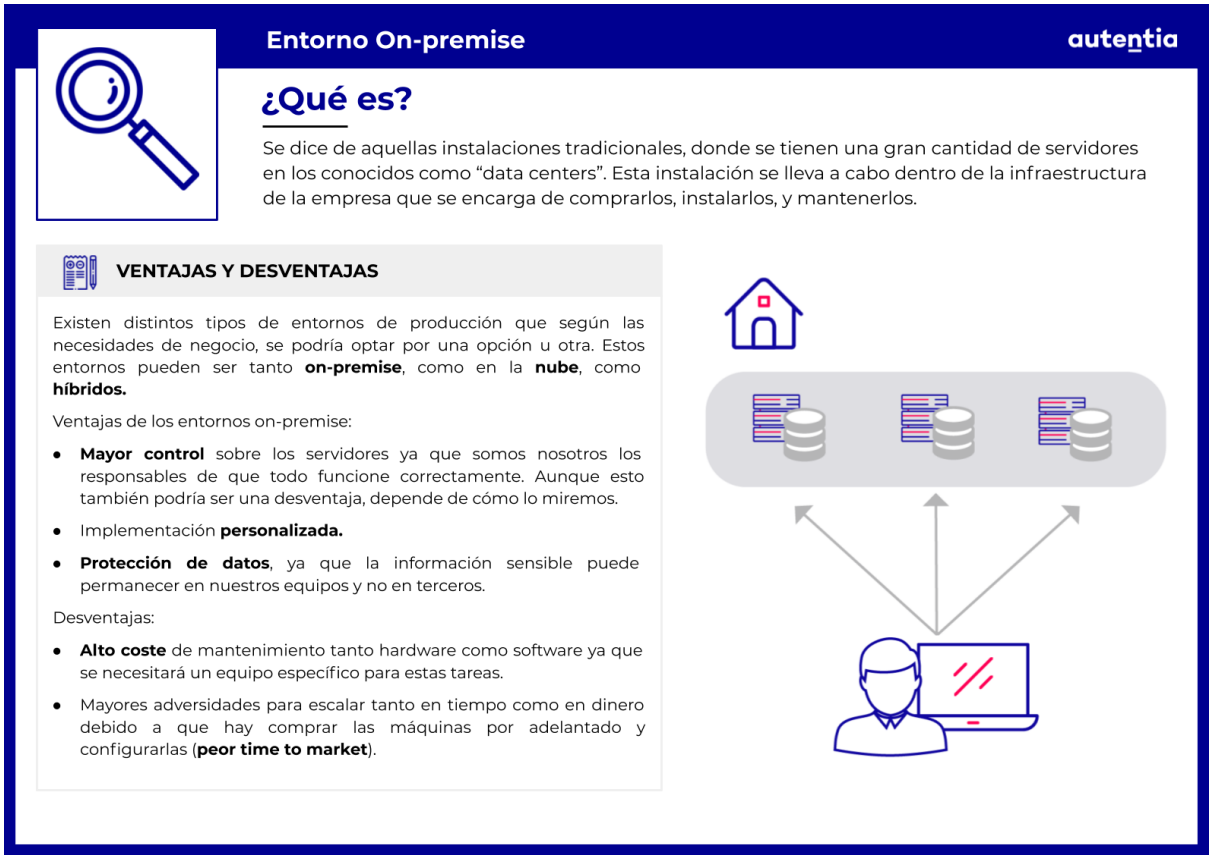
"Antes del advenimiento de la nube, los entornos eran todos on-premise con resultados en la gestión de grandes éxitos y grandes fracasos."



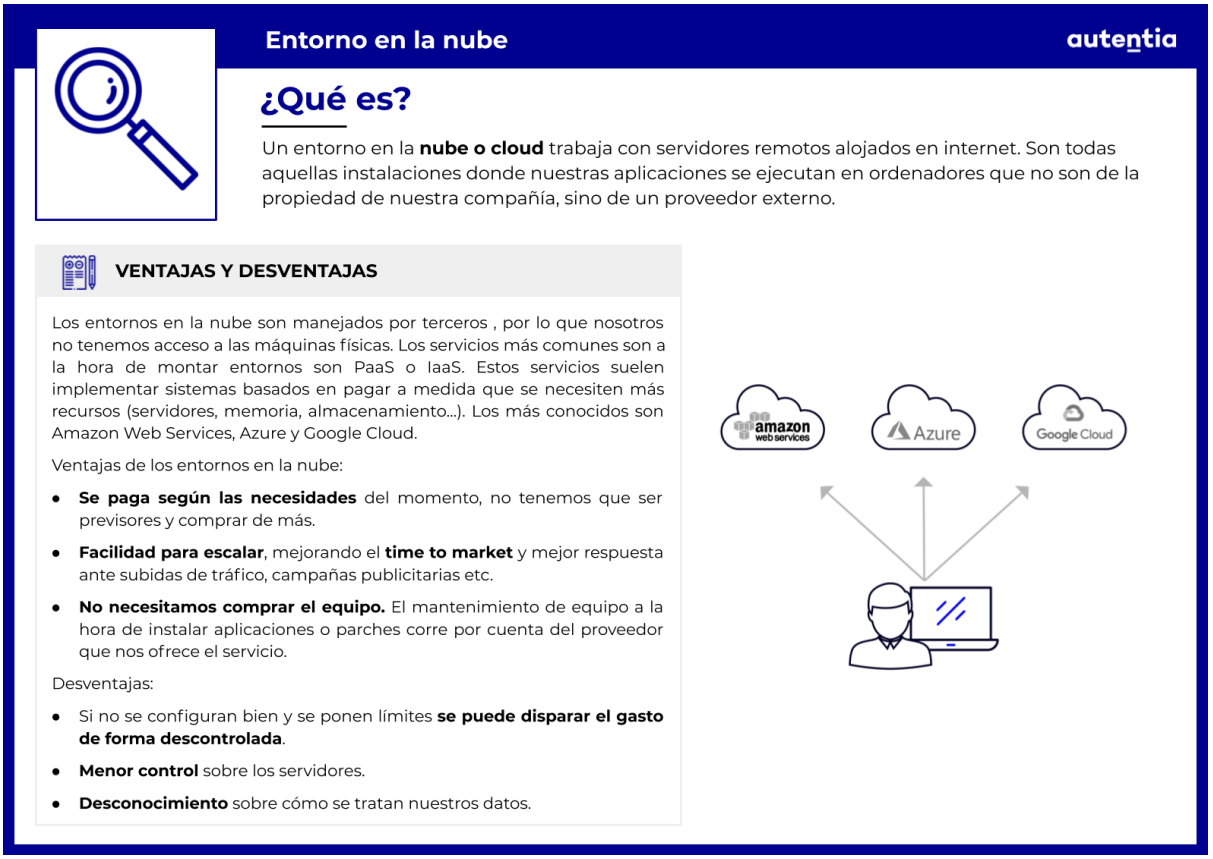
Estos entornos de producción pueden ser tanto on-premise, como en la nube. También existen entornos híbridos.



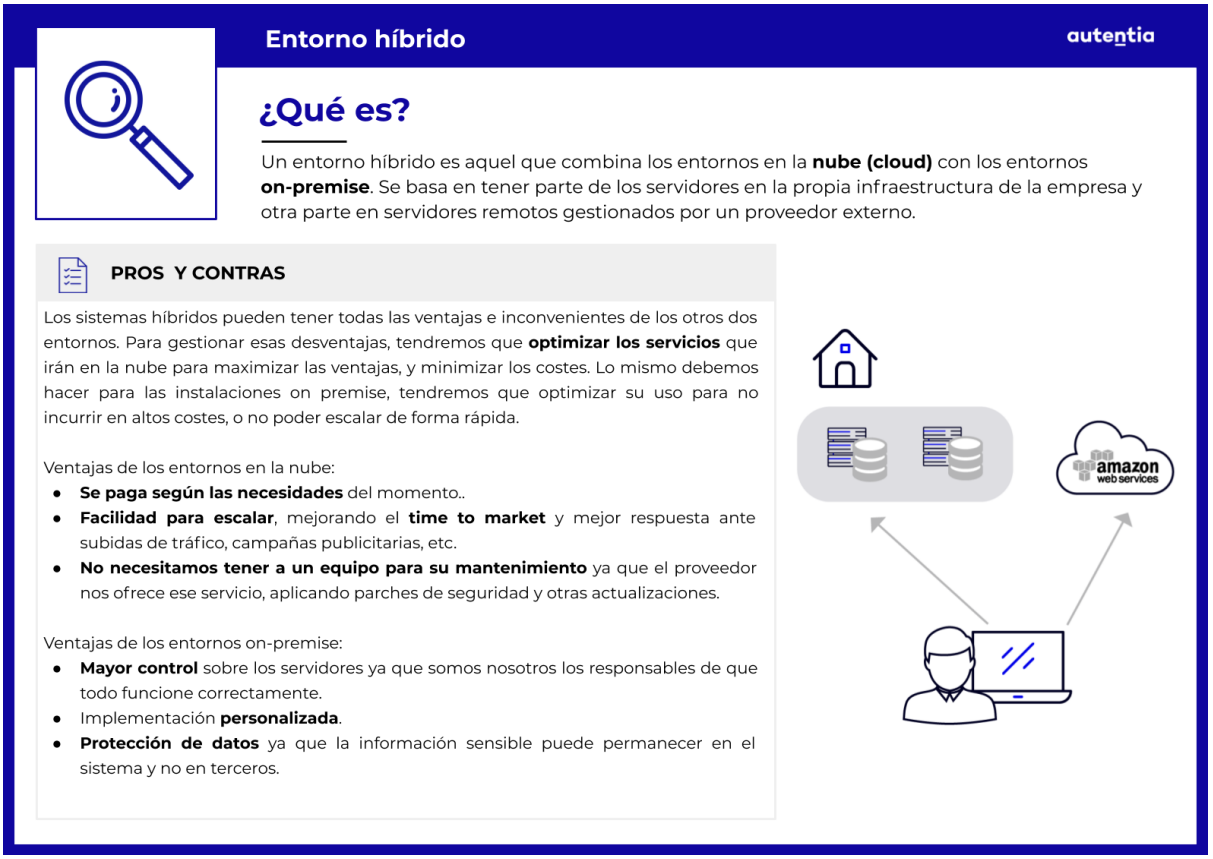
- Entornos **on-premise**: se dice de aquellas instalaciones tradicionales donde se dispone de un montón de servidores en los conocidos como “data centers”, que son salas llenas de servidores, con acceso restringido. Estos servidores son propiedad de la compañía que se encarga de comprarlos, instalarlos y mantenerlos. Es una opción que requiere un desembolso inicial grande.



- Entornos **cloud**: en este caso nuestras aplicaciones se ejecutan en ordenadores que no son de la propiedad de nuestra compañía sino de un proveedor de nube, que es quién compra esos ordenadores, los mantiene y se encarga de la distribución física de los mismos, de la conectividad de redes, del control de accesos, etc.



- Entornos **híbridos**: se trata de aquellos que **son una mezcla**. Parte de la infraestructura es de la empresa y parte está en la nube.



Ventajas e inconvenientes de los entornos on premise

"On Premise ofrece mayor control. Nosotros somos responsables directos de que todo funcione... A un mayor coste y peor time-to-market"

Ventajas:

- El **mayor control** que tenemos sobre los servidores, su conectividad, su seguridad, su mantenimiento. Somos nosotros los responsables de que todo funcione como debe.

Inconvenientes:

- Tener más control, que en principio es una ventaja, es muy costoso, ya que tenemos que disponer de departamentos dedicados a ello: mantenimiento y actualización de los sistemas, sustitución de piezas que se rompen, solucionar problemas de conectividad, etc. **La principal desventaja de esto es el gran coste que conlleva**.
- Estos sistemas no escalan de forma automática. Para escalar hay que comprar las máquinas por adelantado y configurarlas, lo que acarrea un **mayor coste y mucho más tiempo**, o lo que es lo mismo, **peor time to market**.

Ventajas e inconvenientes de los entornos cloud e híbridos

Ventajas:

- No tenemos que comprar máquinas, solo configurarlas y usarlas, lo que en principio supone un **menor coste**.
- Se paga por lo que se usa, no tenemos que ser previsores y comprar de más, volvemos a tener **menor coste** por lo tanto.
- Estas infraestructuras **escalan de forma automática**, tenemos mejor time to market, y mejor respuesta ante subidas de tráfico, campañas publicitarias etc.
- Los **servidores se actualizan solos** aplicando parches con actualizaciones y mantienen conectividad sin que tengamos que hacer nada.
- La configuración de acceso a los servidores se simplifica gracias a las facilidades que ofrece el proveedor de cloud. Eso implica lógicamente **una reducción de costes** en este sentido.

Inconvenientes:

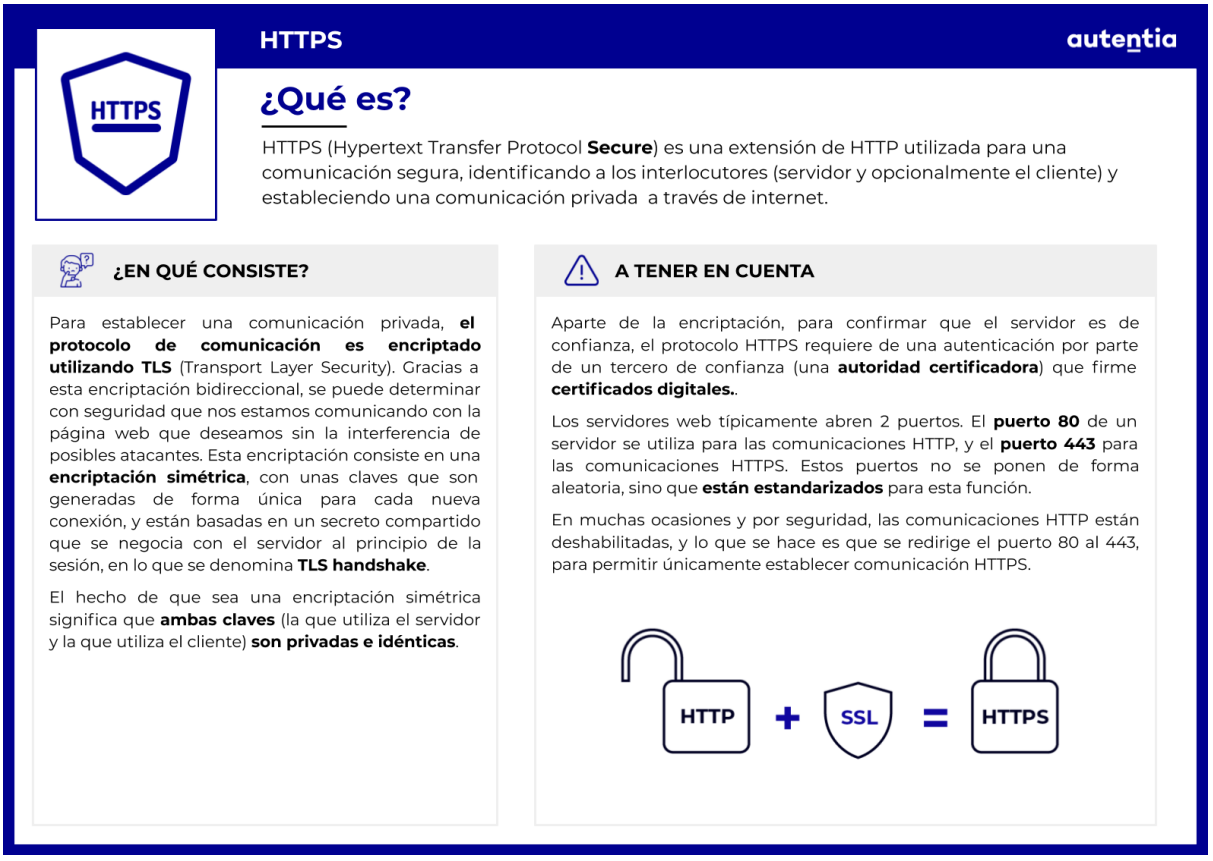
- Si no se configuran bien y se ponen límites **pueden disparar el gasto de forma descontrolada**.
- En la mayoría de los casos tenemos **menor control sobre los servidores**.
- Si ocurre un fallo, en muchas ocasiones no podremos hacer nada salvo esperar a que el proveedor de nube lo arregle.

“Cloud representa menor coste, pago por uso y escalado automático. Si no se controla, se dispara el uso de recursos y aumenta el gasto.”

Los sistemas híbridos pueden tener todas las ventajas e inconvenientes de las otras dos opciones. Tendremos que optimizar los servicios que irán en la nube para maximizar las ventajas, lo mismo que para las instalaciones on premise deberemos optimizar su uso para no incurrir en altos costes o no poder escalar de forma rápida.

Servidor Web

Un Servidor Web es un software que se utiliza para **procesar peticiones HTTP o HTTPS** y servir las páginas web o el contenido estático (ficheros Javascript, hojas de estilo, imágenes, fuentes, etc.) al dispositivo que le esté realizando esas requests (cliente). Los más utilizados actualmente son Apache, Nginx y Microsoft IIS. Es frecuente que a la máquina que tiene instalado este software, junto con los ficheros de la página web (documentos HTML, imágenes, CSS y ficheros JavaScript) se le llame también “Servidor web”.



HTTP

Ahora bien, ¿cómo funcionan las peticiones a estos servidores?, ¿qué es HTTP? HTTP o Hypertext Transfer Protocol es la base de la comunicación de datos para la World Wide Web. Es un protocolo de aplicación para sistemas de información hypermedia, lo que significa que es un medio de

transmisión de información que puede incluir gráficas, audio, vídeo, texto puro e hyperlinks (comúnmente llamados links).

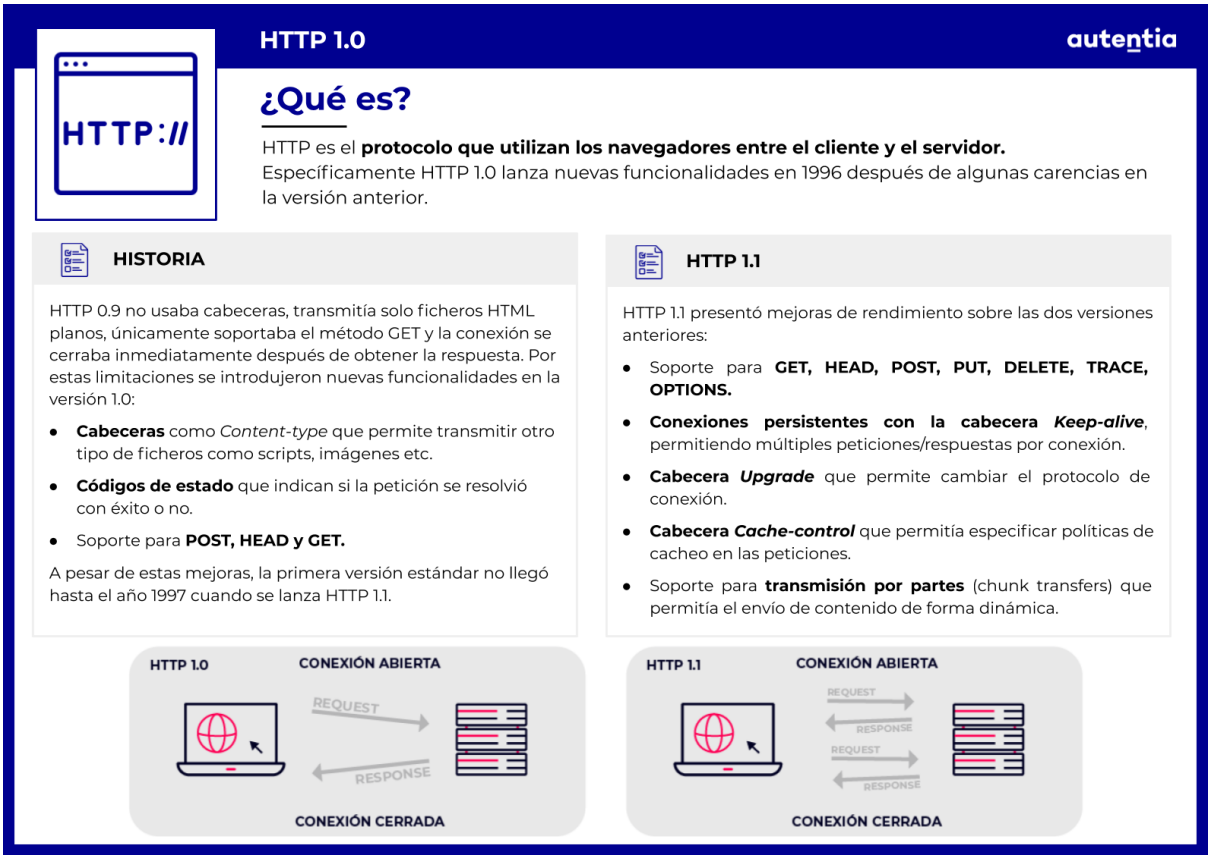
HTTP es un protocolo petición-respuesta (request-response), lo que quiere decir que será el cliente (dispositivo navegando por internet) el que realizará peticiones al servidor y este simplemente, responderá a estas peticiones. Este protocolo se basa en varios tipos de operaciones o **verbos HTTP**. Básicamente, cuando se hace una petición hay que especificar al servidor qué tipo de operación queremos realizar: ¿necesitamos visualizar una página o recoger información? Lo indicaremos usando el verbo **GET**.

¿Queremos mandar nueva información al servidor, como cuando se da de alta un usuario nuevo o un post en una red social? Usaremos el verbo **POST**. ¿Queremos modificar alguno de nuestros datos de usuario en alguna página o editar un comentario en una foto? Realizaremos la petición con el verbo **PATCH** o **PUT**. ¿Queremos borrar un comentario o una foto? Para ello utilizaremos **DELETE**. Hay más verbos además de los ya mencionados pero estos son los más utilizados.

Aparte de indicar el verbo tenemos que indicar cuál es el recurso al que queremos acceder. **HTTP expone recursos** que son localizados a través de

lo que se denomina URL (Uniform Resource Locator), por ejemplo, si hacemos un GET de *https://protectoranimales.com/pets*, estamos diciendo que queremos acceder al **recurso "pets"** y con el **verbo GET** indicamos que queremos recibir este recurso. De existir ese servicio, seguramente nos devolvería un listado de animales.

También **tenemos que indicar la versión del protocolo** (HTTP 1.0, HTTP 2.0) que estamos utilizando y **opcionalmente, una serie de cabeceras**. En las cabeceras se pueden indicar muchas cosas, como por ejemplo, si estamos solicitando un recurso que está securizado, podemos incluir en una cabecera nuestro token de identificación, que nos da acceso al recurso. También podemos indicar el **formato de respuesta** que vamos a aceptar, si es una página web o si es una serie de información en formato JSON o XML...



HTTP 2.0

autentia

Http/2

¿Qué es?

HTTP 2.0 es un protocolo binario que **busca aumentar la velocidad y reducir la complejidad de las aplicaciones**. Esto lo consigue cambiando el modo en el que se formatean los datos y se transportan.

CARACTERÍSTICAS

HTTP 2.0 conserva la semántica de HTTP 1.0, se mantienen sus conceptos básicos (verbos, códigos de estado, campos de cabecera, etc.) y se introducen mejoras con el objetivo de mejorar el rendimiento y optimizar nuestras aplicaciones.

De este modo **no es necesario cambiar cómo las aplicaciones existentes funcionan**, pero los nuevos desarrollos pueden aprovechar las funcionalidades que este protocolo ofrece.

HISTORIA

Se considera una evolución del **protocolo SPDY**, desarrollado por Google en 2009 y que también buscaba reducir los tiempos de carga de la web. Muchas de las mejoras que introducía SPDY fueron llevadas a HTTP 2.0.

En 2012 aparecen los primeros borradores de HTTP 2.0 y en los años siguientes ambos protocolos continuaron evolucionando en paralelo, hasta que en 2015 SPDY fué deprecado en favor de HTTP 2.0.

MEJORAS

- **Multiplexación**: permite enviar y recibir varias peticiones al mismo tiempo usando una misma conexión. Así se consigue mejorar la velocidad de carga de la página y se disminuye la carga en los servidores web.
- **Protocolo binario en lugar de texto**: ofrece mayor eficiencia en el transporte del mensaje y en su interpretación. Además son menos propensos a errores ya que con el texto hay problemas de espacios en blanco, capitalización, finales de línea, etc.
- **Servicio 'server push'**: el servidor envía información al cliente antes de que el cliente la pida. Así cuando el cliente necesite esos recursos ya los tiene, ahorrando tiempo.
- **Compresión de cabeceras**: en general las cabeceras cambian poco entre peticiones y además con el uso de cookies su tamaño puede incrementar mucho. Por eso con HTTP 2.0 se van a enviar comprimidas.
- **Priorización de transmisiones**: dentro de una misma conexión se da más peso a aquellas transmisiones que tengan más importancia para gestionar mejor los recursos.

Cabeceras HTTP más utilizadas

autentia

¿Qué es?

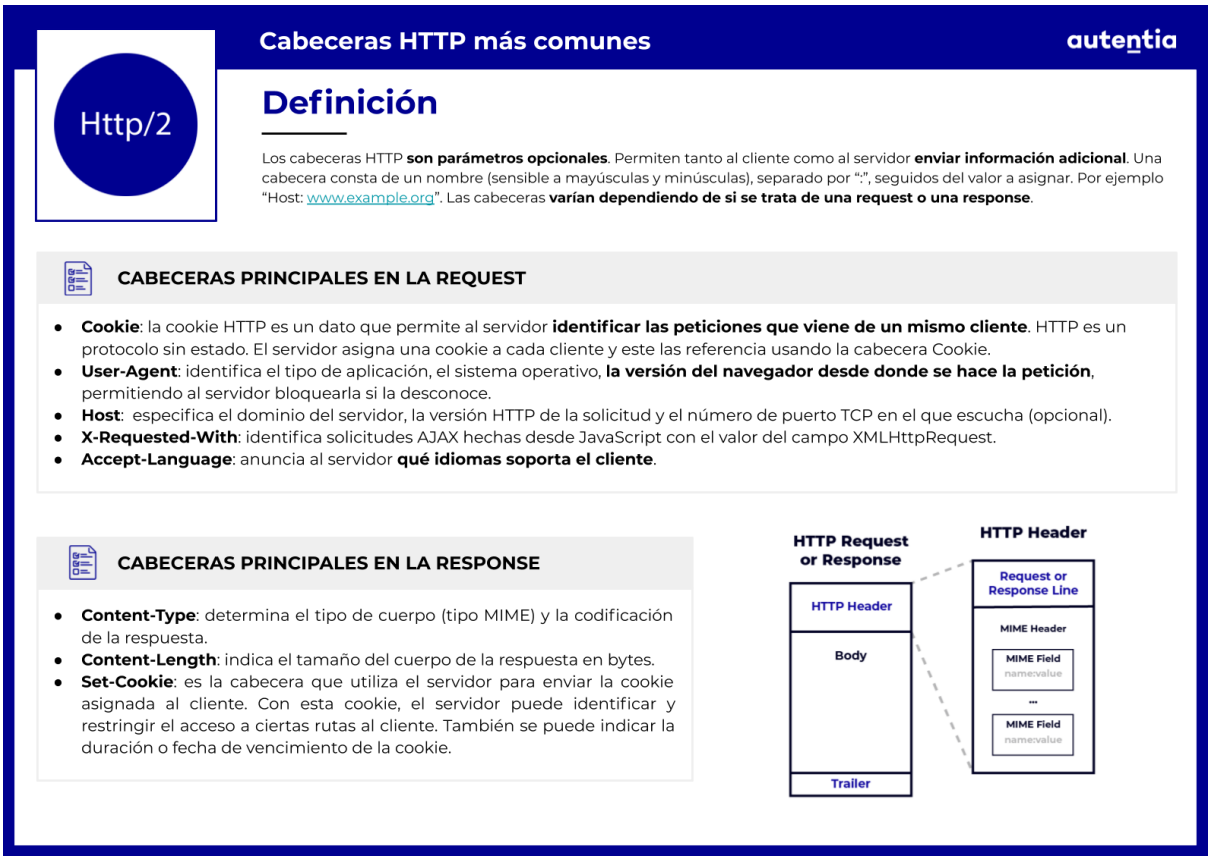
Algunas de las **cabeceras HTTP más usadas** son las que tienen que ver con el caché para mejorar el rendimiento y las que tienen que ver con seguridad para evitar incidentes como secuestros de sesión o ataques XSS (Cross-site scripting), etc.

DE CACHÉ

- **Cache-Control**: esta cabecera se utiliza para especificar parámetros de cacheado en el navegador.
  - ***no-cache***: obliga a validar un recurso caducado contra el servidor.
  - ***no-store***: no permite cachear la respuesta. Muy usada en páginas con datos confidenciales.
  - ***private*** vs. ***public***: si solo se permite la caché del navegador o también a terceros (CDN, Proxies, etc).
  - ***max-age***: vida útil del recurso cacheado (en segundos).
- **Validators:** utiliza la cabecera ETag como hash del recurso que tiene el servidor y verificar si el recurso cacheado caducado ha cambiado. Si no ha cambiado, no lo vuelve a solicitar, actualizando el recurso.
- **Extension Cache-Control:** Estas son algunas cabeceras que extienden la directiva Cache-Control.
  - ***immutable***: no valida con el origen, solo se refresca si caduca.
  - ***stale-while-revalidate***: especifica el tiempo que se mostrará el archivo obsoleto mientra se valida en origen.
  - ***stale-if-error***: especifica tiempo extra de un recurso caducado sin validar.

DE SEGURIDAD

- **HTTP Strict Transport Security (HSTS)**: solo se permite conexiones HTTPS. Esto evita incidentes de seguridad como el secuestro de cookies.
- **Content Security Policy (CSP)**: es una capa de seguridad adicional que ayuda a prevenir ataques de inyección de código.
  - Estableciendo whitelist de contenidos como js, images, font, etc.
  - Permite establecer políticas de navegación (a qué recursos se puede redirigir al usuario).
  - Podemos generar informes o limitar el uso de recursos como plugins.
- **Cross Site Scripting Protection (X-XSS):** habilita un filtro en los navegadores contra ataques XSS.
- **X-Frame-Options:** restringe el renderizado de objetos como <frame>, <iframe>, <embed> o <object> para prevenir los ataques de clickjacking.
- **X-Content-Type-Options:** esta cabecera evita que el usuario pueda determinar el tipo de archivo que espera el servidor e intente camuflar bajo otro tipo de archivos, por ejemplo hacer pasar script de php como imágenes.



El protocolo HTTP cuenta también con lo que se conoce como **parámetros**. Generalmente se usan a modo de filtro. Veamos un ejemplo:

Se hace una llamada **GET** a la URL https://protectoradeanimales.com/api/v2/pets**?**breed=pitbull**&**color=brown

Los parámetros se indican al final de la url, justo después del símbolo de cierre de interrogación y se separan por el símbolo & si estamos enviando más de uno. Son pares de clave - valor. Lo que cabría esperar de esta consulta es que devolviera un listado de animales, de raza pitbull de color marrón.

A veces se utiliza la propia URL para hacer este tipo de filtros. Es muy normal ver peticiones de este tipo:

GET a la URL https://protectoradeanimales.com/api/v2/pets/**346**

Lo que se espera de esa llamada es que nos responda con los datos del animal cuyo identificador es el 346.

No nos tenemos que olvidar que hay veces que sólo con la URL no podemos completar todas las operaciones que se nos pueden ocurrir. Si

queremos dar de alta mascotas y queremos enviar los datos de la mascota al servidor, seguramente estemos haciendo algo parecido a lo siguiente:

POST a la URL https://protectoradeanimales.com/api/v2/pets

Las peticiones de tipo POST pueden ir acompañadas de un cuerpo en la petición. Para el ejemplo vamos a enviar el **body o cuerpo de la petición** en formato Json. Sería algo parecido a lo siguiente:

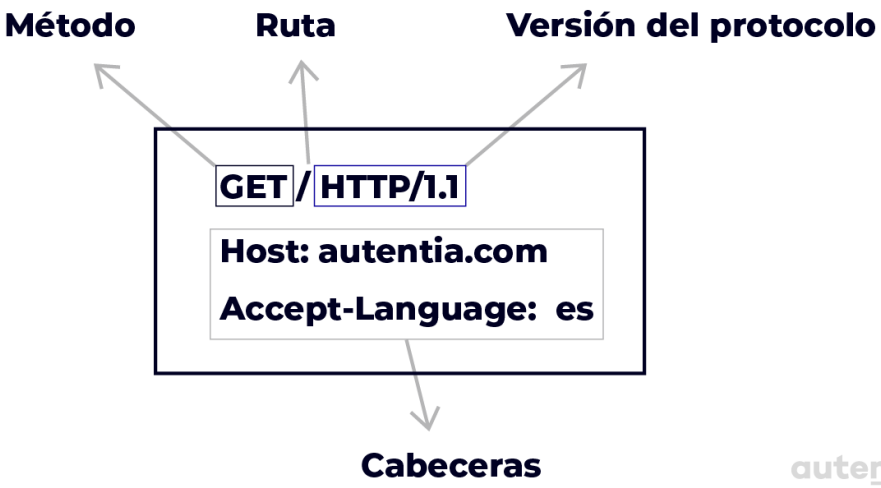
{

“name”:”thor”,

“breed”:”Pitbull”

“birthDate”:”21-07-2019”

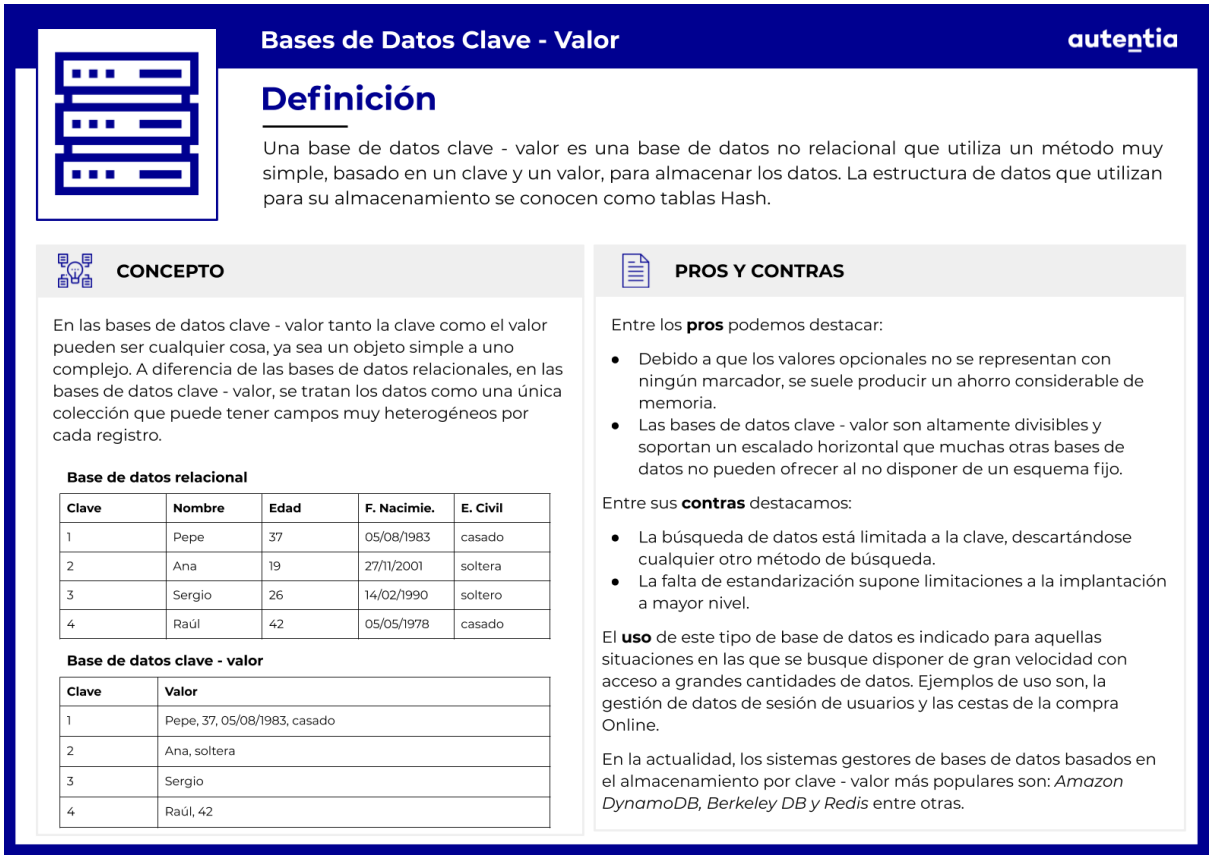
}



DNS

Una vez entendido esto, ¿cómo se conecta un dispositivo a un servidor web? ¿Cómo sabe a qué dirección IP debe realizar las peticiones? Ahí es donde entra en juego el protocolo **DNS** (Domain Name System). El DNS es un sistema de nombramiento jerárquico y descentralizado para ordenadores, servicios, etc., conectados a internet o a una red privada. Básicamente, es lo que **permite traducir un nombre fácilmente memorizable** (por ejemplo, www.google.com) **a una IP** (como puede ser 216.58.209.68). Si se copia y pega esa dirección IP en un navegador, se puede comprobar que corresponde al buscador.

Este sistema (DNS) se mantiene por un sistema de base de datos distribuida.



Bases de Datos Clave - Valor

autentia

Definición

Una base de datos clave - valor es una base de datos no relacional que utiliza un método muy simple, basado en un clave y un valor, para almacenar los datos. La estructura de datos que utilizan para su almacenamiento se conocen como tablas Hash.

CONCEPTO

En las bases de datos clave - valor tanto la clave como el valor pueden ser cualquier cosa, ya sea un objeto simple a uno complejo. A diferencia de las bases de datos relacionales, en las bases de datos clave - valor, se tratan los datos como una única colección que puede tener campos muy heterogéneos por cada registro.

Base de datos relacional

Clave	Nombre	Edad	F. Nacimie.	E. Civil
1	Pepe	37	05/08/1983	casado
2	Ana	19	27/11/2001	soltera
3	Sergio	26	14/02/1990	soltero
4	Raúl	42	05/05/1978	casado

Base de datos clave - valor

Clave	Valor
1	Pepe, 37, 05/08/1983, casado
2	Ana, soltera
3	Sergio
4	Raúl, 42

PROS Y CONTRAS

Entre los **pros** podemos destacar:

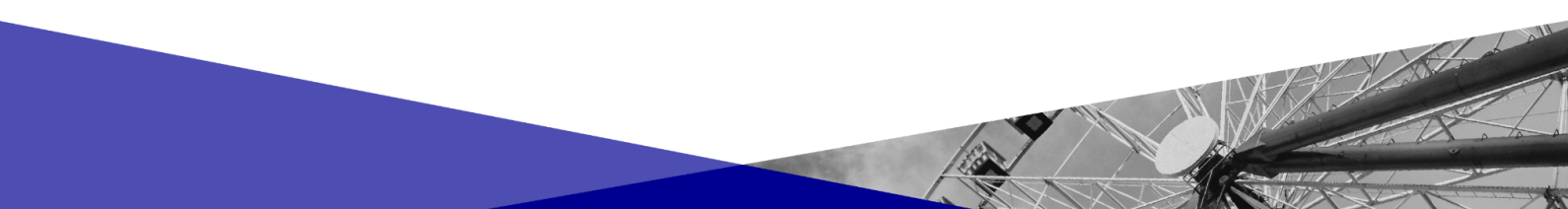
- Debido a que los valores opcionales no se representan con ningún marcador, se suele producir un ahorro considerable de memoria.
- Las bases de datos clave - valor son altamente divisibles y soportan un escalado horizontal que muchas otras bases de datos no pueden ofrecer al no disponer de un esquema fijo.

Entre sus **contras** destacamos:

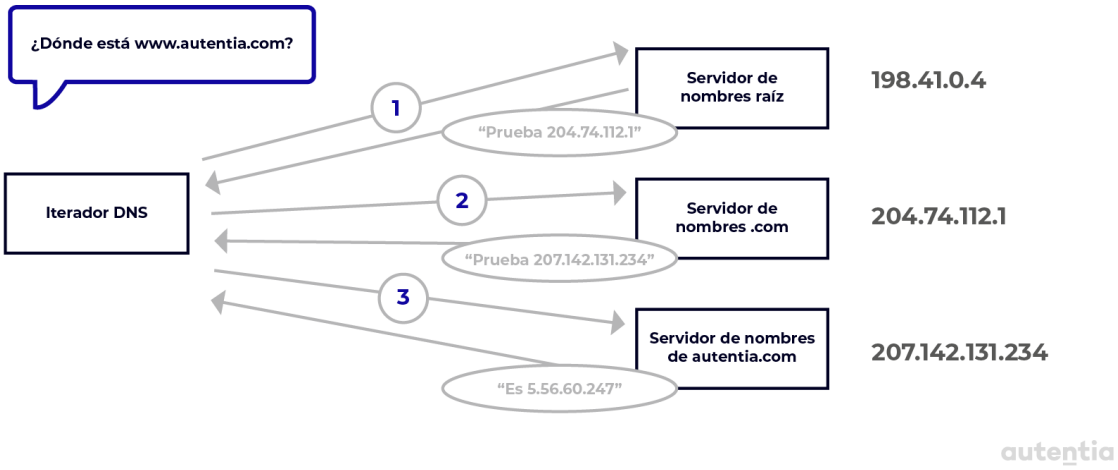
- La búsqueda de datos está limitada a la clave, descartándose cualquier otro método de búsqueda.
- La falta de estandarización supone limitaciones a la implantación a mayor nivel.

El **uso** de este tipo de base de datos es indicado para aquellas situaciones en las que se busque disponer de gran velocidad con acceso a grandes cantidades de datos. Ejemplos de uso son, la gestión de datos de sesión de usuarios y las cestas de la compra Online.

En la actualidad, los sistemas gestores de bases de datos basados en el almacenamiento por clave - valor más populares son: *Amazon DynamoDB, Berkeley DB y Redis* entre otras.



Cada dominio tiene al menos un servidor de nombres (names server) autorizado que publica información sobre el dominio y los servidores de nombres de cualquier dominio subordinado. Estos servidores autorizados son servidores que responden con IPs que no son cacheadas, sino que han sido instaladas en su sistema de configuración. De esta forma, cuando se escribe un dominio, se comienza un proceso de queries para recoger la IP del sitio. Primero, se pregunta al servidor de nombres raíz y este responderá con la IP del servidor de nombres subordinado correspondiente, que será el de la terminación del dominio (.org, .com, .es, etc.). Después, se preguntaría a éste y se recogería la IP del sitio web al que se quiere acceder.



Realmente este mecanismo significa una carga enorme en los servidores raíz, por lo que normalmente se utiliza una caché en los servidores DNS para evitarlo. Al final, estos servidores raíz sólo participan en una fracción muy pequeña de todas las peticiones. Esto es debido a que una vez que conocemos la IP, no volvemos a preguntar. Esta IP se almacena en la caché de nuestro ordenador.

HTTPS

HTTPS (Hypertext Transfer Protocol **Secure**) es una extensión de HTTP utilizada para una comunicación segura, identificando a los interlocutores (servidor y opcionalmente el cliente) y estableciendo una comunicación privada a través de internet. Para ello, **el protocolo de comunicación es encriptado utilizando TLS** (Transport Layer Security). Gracias a esta encriptación bidireccional, se puede determinar con seguridad que nos estamos comunicando con la página web que deseamos sin la interferencia de posibles atacantes. Esta encriptación consiste en una **encriptación simétrica** con unas claves que son generadas de forma única para cada nueva conexión, y están basadas en un secreto compartido que se negocia con el servidor al principio de la sesión, en lo que se denomina **TLS handshake**. El hecho de que sea una encriptación simétrica significa que ambas claves (la que utiliza el servidor y la que utiliza el cliente) son privadas e idénticas.

TLS

autentia

¿Qué es?

Transport Layer Security (TLS) es un protocolo criptográfico que proporciona autenticación y cifrado de la información intercambiada entre las distintas partes que operan sobre una red.

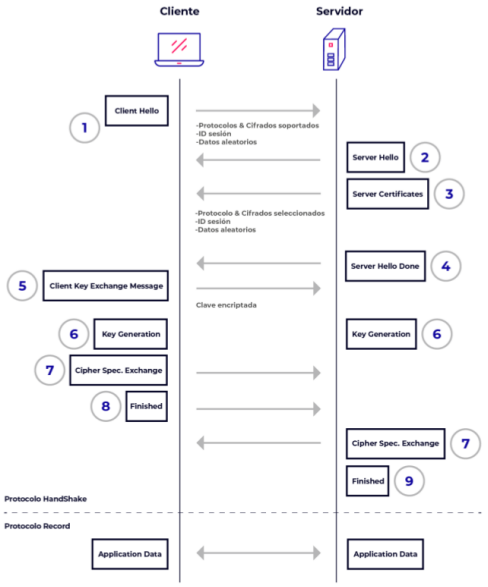
¿EN QUÉ CONSISTE?

TLS es el **sucesor de SSL (Secure Socket Layer)**. Se caracteriza por:

- **Comunicación privada y encriptada** tanto en el lado cliente como en el lado servidor a través de **un cifrado simétrico.**
- **Intercambio de claves** públicas y autenticación a través de **certificados digitales.**
- **Negociación del algoritmo** de intercambio de mensajes entre las partes.

TLS **intercambia registros** entre las partes en un **formato específico**. Cada registro es comprimido, cifrado y empaquetado con un código de MAC. Hay dos protocolos para ello:

- **TLS Handshake:** es un protocolo que sirve para que dos partes se verifiquen entre sí y puedan establecer un tráfico cifrado e intercambiar claves. Las claves generadas para dicho cifrado se consiguen a través de una negociación entre las partes.
- **TLS Record:** se lleva a cabo la autenticación de los datos para que su transmisión sea mediante una conexión fiable y segura (p.e. TCP). Los mensajes que se intercambien estarán cifrados simétricamente mediante las claves negociadas en el Handshake.



Cifrado simétrico

autentia

¿Qué es?

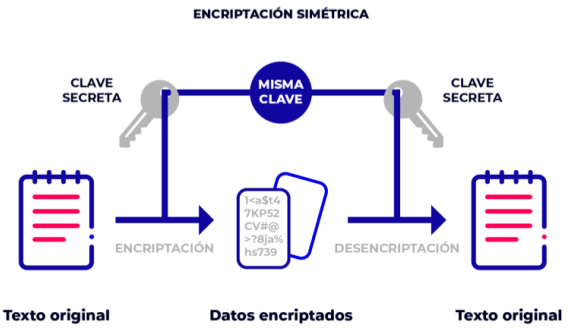
El **cifrado simétrico** es una forma de encriptación en la que sólo se utiliza una clave, tanto para encriptar como para desencriptar.

¿EN QUÉ CONSISTE?

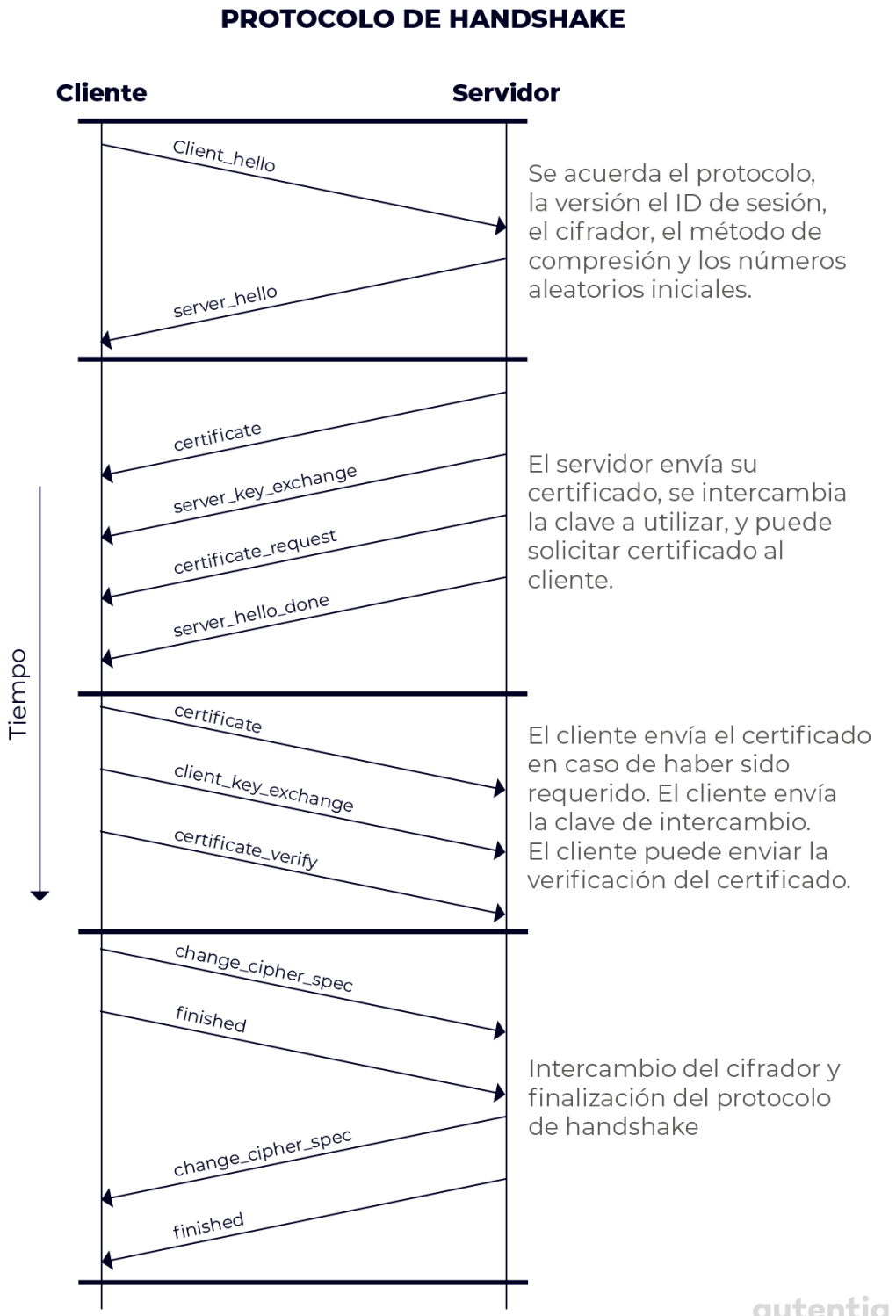
Al haber sólo una clave, esta es **privada**, y las distintas entidades comunicándose **deben compartirla entre ellas** para poder utilizarla en el proceso de desencriptación.

El hecho de que todas las partes tengan acceso a la clave privada es la **principal desventaja** de este tipo de encriptación, ya que hay más probabilidad de que esa clave pueda ser vulnerada. Además, si un atacante consigue esta clave podría desencriptar y leer todos los mensajes y datos de esa conversación. La **ventaja** que tiene sobre la alternativa de usar un cifrado asimétrico es que tiene mejor rendimiento, porque los algoritmos usados son más sencillos.

La clave puede ser una contraseña/código o puede ser una cadena de texto o números aleatoria generada por un software especializado en generar este tipo de claves.



Sin entrar en mucho detalle, el protocolo de handshake donde se negocia esa clave tendría un flujo como el de la imagen.



Aparte de esta encriptación, para confirmar que el servidor es de confianza, el protocolo HTTPS requiere de una autenticación por parte de un tercero de confianza (una **autoridad certificadora**) que firme certificados digitales. Cuando una entidad solicita un certificado, la autoridad certificadora (AC) verifica la identidad del solicitante (ya sea mediante carnés de identidad, huellas dactilares, declaraciones juradas...) y después crea una clave pública (para cifrar los mensajes) y una privada (para descifrarlos). El uso para el cifrado de la clave pública y privada se conoce como **cifrado asimétrico**. Los navegadores suelen incluir por defecto un repositorio de AC de confianza, pero si hay alguna AC en la que en principio no se confía (a lo mejor porque todavía no lleva mucho tiempo en funcionamiento), habrá que instalar manualmente el certificado en el repositorio del navegador. Es importante conocer que **estos certificados caducan**. Una iniciativa interesante desde el punto de vista de devops es alertar sobre la caducidad de los mismos para que así, cuando caduque, no nos quedemos sin peticiones. Esto suele pasar mucho en el entorno profesional. El certificado se renueva, y como no caduca hasta dentro de 1 año o X tiempo, todo el mundo se olvida de ello, hasta que un día, de repente, no funciona y ya tenemos el problema montado en producción. Además **es una buena práctica tener estos certificados en un repositorio** para poder instalarlos en los servidores que se necesite de forma automática.

Certificado

autentia

¿Qué es?

Documento virtual que contiene los datos identificativos de una persona física o de un sitio web y que está autenticado por un organismo oficial encargado de emitir y validar dicha información.

TIPOS

Cuando queremos identificar a una persona física, una entidad o un dominio de manera digital, hablamos de **certificados digitales**. Una **Autoridad Certificadora (AC)** será la encargada de firmar y emitir los certificados, verificando que quien lo solicite es quien dice ser.

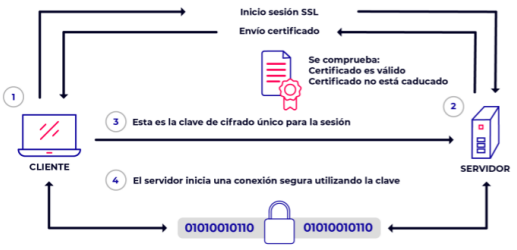
También podemos validar un sitio web a través de certificados. Se configura el servidor para que use dicho certificado y usando el protocolo SSL se cifra la información enviada al servidor. Esto es lo que comúnmente se conoce como **certificado SSL** debido a que este protocolo es el más extendido hoy en día. Los navegadores suelen incluir por defecto un repositorio de AC de confianza, pero si hay alguna AC en la que en principio no se confía (porque lleva poco tiempo en funcionamiento, por ejemplo), habrá que instalar manualmente el certificado en el repositorio del navegador. **Los certificados caducan** y esto se hace para asegurar que toda la información es precisa y demuestra una validez como propietario de confianza del dominio

¿Qué ventajas nos ofrecen los certificados?

- Ahorro de espacio, tiempo y dinero.
- Confidencialidad y seguridad en las comunicaciones.

¿CÓMO FUNCIONA?

Se hace uso de claves públicas y claves privadas. En el caso de un sitio web, la clave privada permanece en el servidor donde se ha configurado el certificado. Cuando un navegador web desea establecer una nueva conexión, el servidor comparte la clave pública con el cliente para establecer un método de cifrado y el cliente confirma que reconoce y confía en la entidad emisora del certificado. Este proceso inicia una sesión segura que protege la privacidad y la integridad del mensaje.



Cifrado asimétrico

autentia

¿Qué es?

El **cifrado asimétrico** es una forma de encriptación en la que las claves vienen en parejas. Lo que una de las llaves encripta, sólo puede ser desencriptado por la otra.

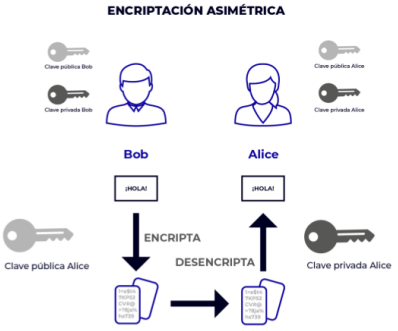
¿EN QUÉ CONSISTE?

El cifrado asimétrico tiene **dos claves** por persona, a diferencia del cifrado simétrico que sólo tiene una. Pongamos que tenemos dos personas: Bob y Alice. Los pasos para encriptar y desencriptar un mensaje que Bob le envía a Alice serían los siguientes:

- Bob dispone de la clave pública de Alice, que sirve para encriptar (las claves públicas las puede ver cualquiera). **Encripta un mensaje con esa clave pública**.
- **Manda el mensaje encriptado a Alice**.
- Los mensajes que se hayan encriptado usando la clave pública de Alice sólo se pueden desencriptar usando la clave privada de Alice, clave que solo ella conoce. **Alice usa su clave privada para desencriptar el mensaje**.

A TENER EN CUENTA

Si un atacante consiguiera la clave privada de Bob podría leer los mensajes que le llegan a Bob, ya que han sido encriptados usando su clave pública, **pero no podrían descifrar mensajes que Bob mande a otros**, ya que estos mensajes se encriptan usando claves públicas de otros.



Ahora que ya hemos aclarado qué es HTTP y HTTPS, es conveniente aclarar que los servidores web típicamente abren 2 puertos. El **puerto 80** de un servidor se utiliza para las comunicaciones HTTP y el **puerto 443** para las comunicaciones HTTPS. Estos puertos no se ponen de forma aleatoria, sino que **están estandarizados** para esta función. Puede verse en el siguiente enlace. En muchas ocasiones y por seguridad, las comunicaciones HTTP están deshabilitadas y lo que se hace es que se redirige el puerto 80 al 443 para permitir únicamente establecer comunicación HTTPS.

¿Qué pasa cuando con un solo servidor no somos capaces de dar abasto porque recibimos muchísimas peticiones?, ¿cómo es posible poner otro? Hasta ahora hemos visto que por DNS si llamamos a https://protectoranimales.com esto va a resolver a una IP del tipo 158.34.56.230. ¿Cómo hacemos si queremos poner 2 servidores web o más? Pues **para eso existen los balanceadores que vamos a ver a continuación**, para apuntar el DNS a un balanceador por delante de los servidores web y no directamente a los servidores web.

DNS autentia

¿Qué es?

El DNS (Domain Name System) es un sistema de nombramiento jerárquico y descentralizado para ordenadores, servicios, etc. conectados a internet o a una red privada. Permite **traducir un nombre fácilmente memorizable para una persona a una IP**.

CONCEPTO

El DNS nació de la necesidad de recordar fácilmente los nombres de los servidores de Internet. **Cada dominio tiene al menos un servidor de nombres** que publica información sobre ese mismo dominio y sobre los servidores de nombres de cualquier dominio subordinado.

Cuando intentas acceder a www.autentia.com comienza un proceso de consultas iterativo para resolver la IP del sitio.

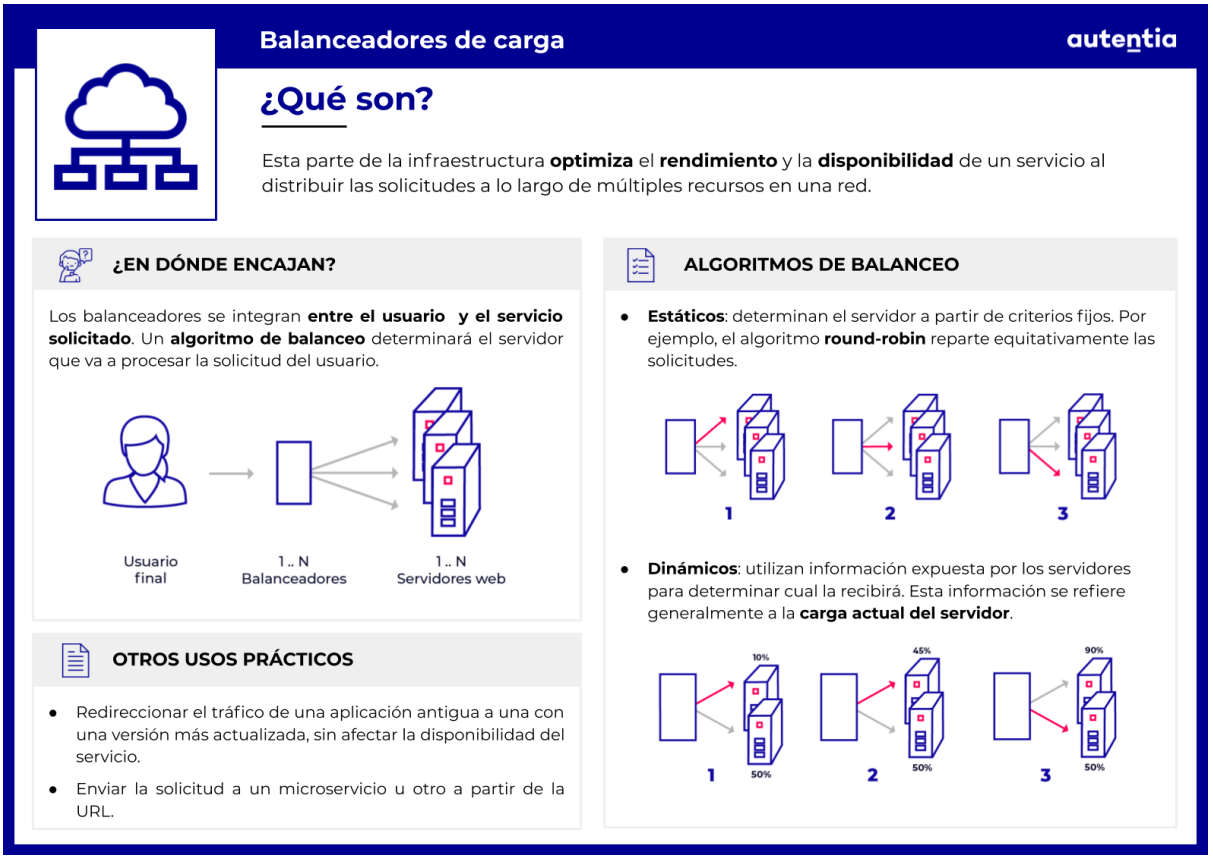
1. Primero se pregunta al servidor de nombres raíz, que responde con la IP de un servidor de nombres subordinado.
2. El servidor de nombres subordinado responde con la dirección del servidor de nombres de autentia.com.
3. Finalmente este servidor conoce la IP del dominio y lo devuelve.

En general, la resolución de los nombres **se hace de forma transparente al usuario a través del cliente** (navegador, aplicación de correo, etc.).

Además, para reducir el número de peticiones, ya que las direcciones se cachean a distintos niveles. Por ejemplo los navegadores cachean los resultados de estas consultas y también los servidores de nombres disponen de cachés para evitar hacer tantas peticiones.

Balanceadores

Los balanceadores son la parte de la infraestructura que se va a encargar de **distribuir la ejecución de procesos y tareas** entre piezas iguales, de tal forma que no haya algunos servidores ociosos mientras otros están muy cargados, dando tiempos peores de finalización de tareas o de respuesta. Se utilizan para **optimizar el uso de recursos hardware**.



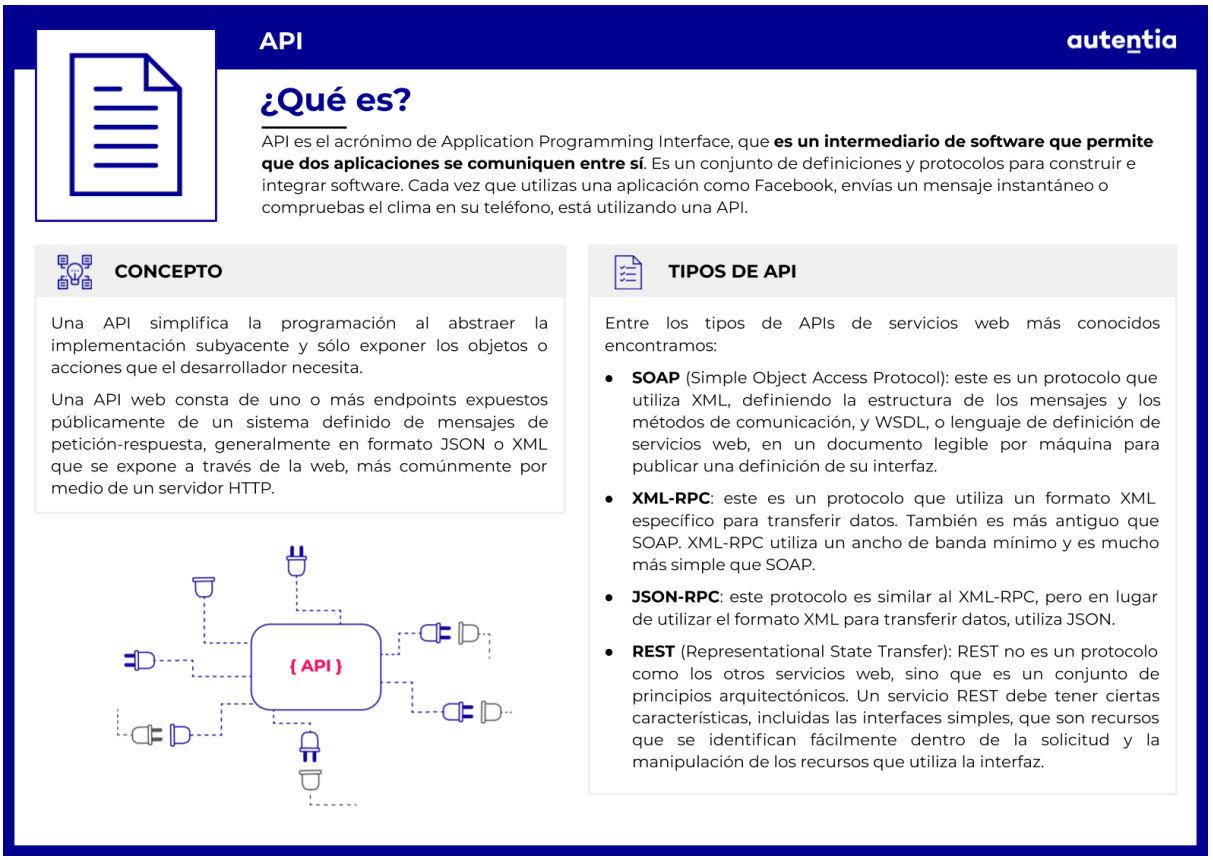
Son entonces una **pieza básica para el rendimiento**.

También son una **pieza importante** en cuanto a la **alta disponibilidad**. Mediante los balanceadores podemos dejar de apuntar a servidores con una aplicación antigua y apuntarlos hacia servidores con una nueva versión de nuestra aplicación. Sirven en este caso para **redireccionar tráfico**.

En algunos casos, hay balanceadores, por ejemplo los de AWS, que tienen **mecanismos de health check**, esto es, comprobar que los servidores a los que están mandando tráfico o tareas, efectivamente se encuentran

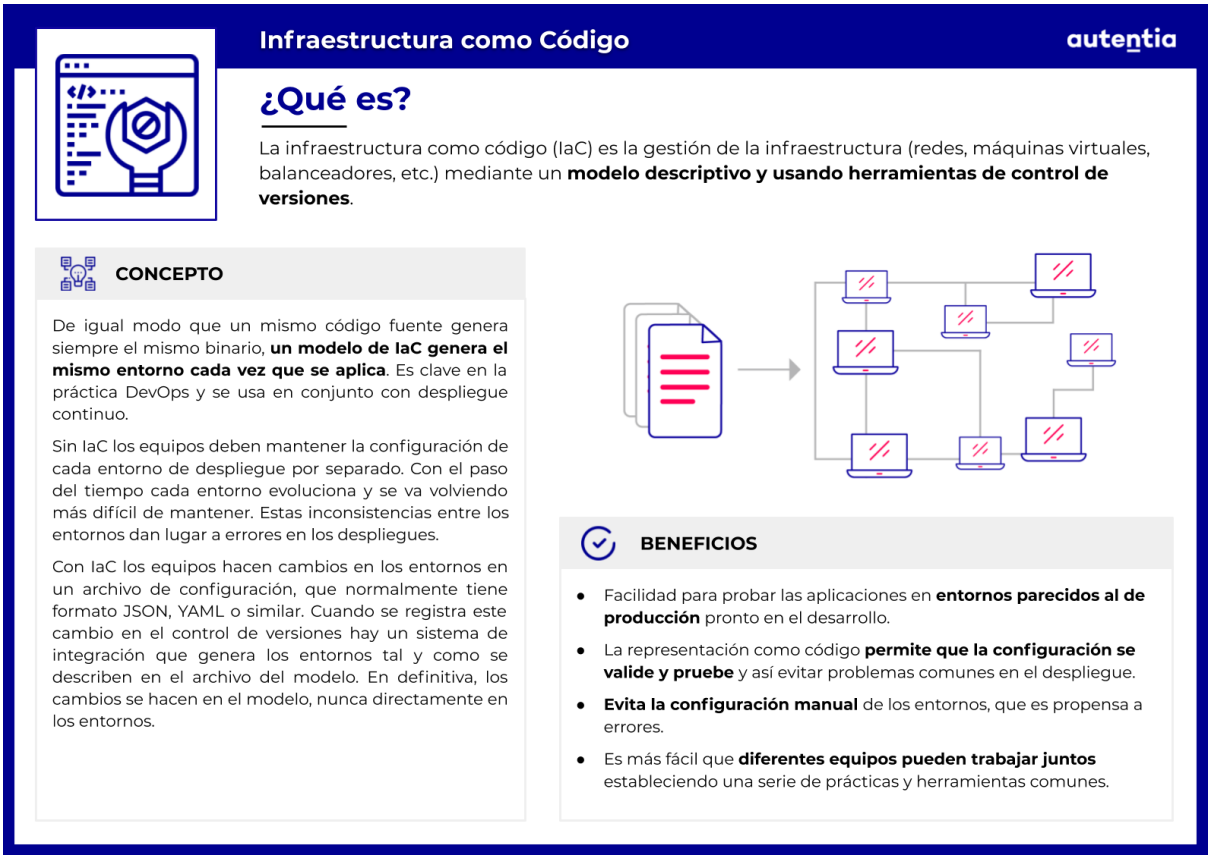
operativos y funcionando. Si no fuera este el caso, a partir de ese momento no enviarían más tareas a esos servidores que no responden.

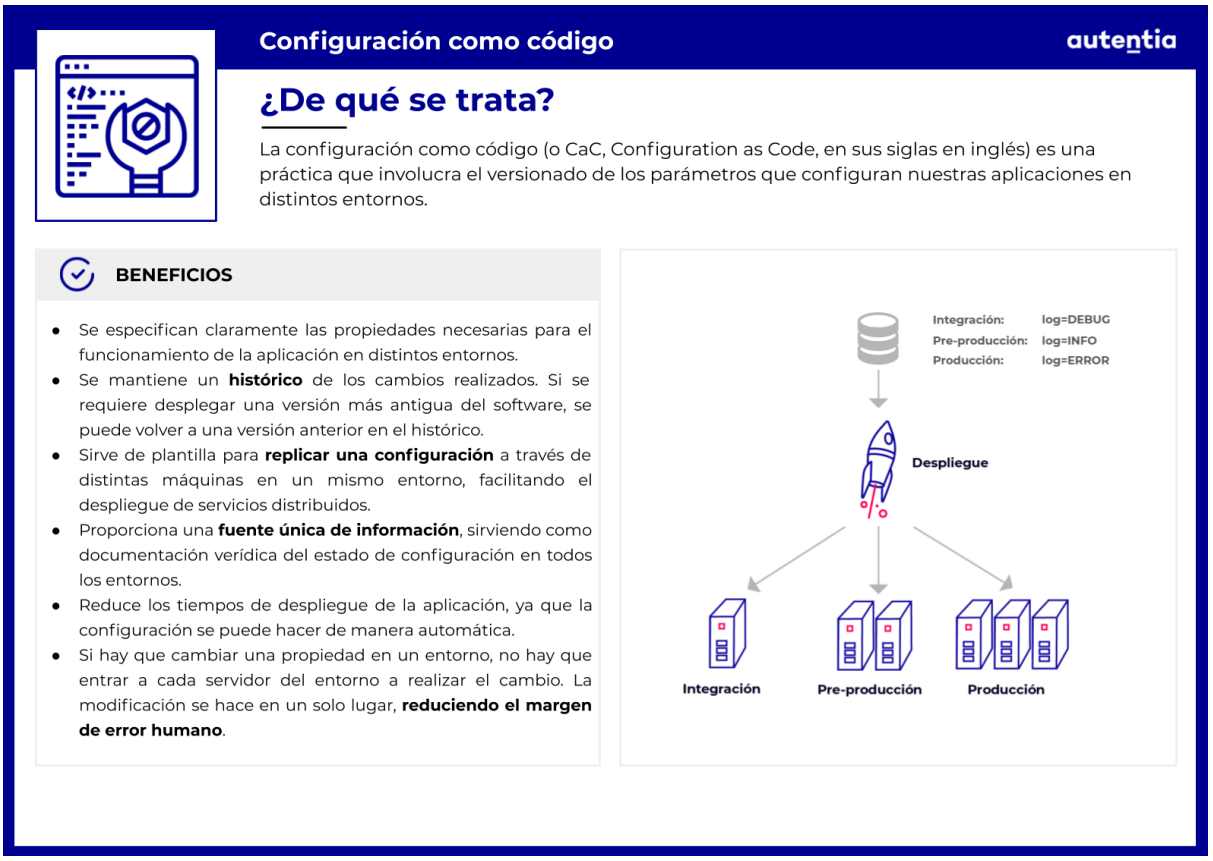
Cuando veamos más adelante los servidores de aplicaciones y cómo éstos son los encargados de exponer APIs pueden surgir dudas. Si los balanceadores son el punto de entrada: ¿cómo vamos a discernir cuándo una petición es para pedir una página estática, por ejemplo a un NGINX, y cuándo para acceder a un API que tenemos en un Tomcat? Estos balanceadores **muchas veces redirigen también entre aplicaciones o microservicios por la url**. En función de una url le mandan la petición a un conjunto de máquinas o a otro.



Hay multitud de software que puede hacer estas tareas de balanceo, pero casi todos comparten una cosa en común, **requieren configuración**. Es muy importante que seamos capaces de cambiar esta configuración en cualquier momento, de volver a una versión anterior... y todo ello de **forma automática**. Es normal tener una infraestructura con muchos balanceadores a lo largo de todos los entornos que hemos ido viendo con anterioridad. Ir modificando esta infraestructura o cambiando esa configuración a mano para los distintos entornos, para tener todos los balanceadores actualizados y configurados, es garantía de que las cosas

van a salir mal y vamos a tener errores. Es por eso que es importante automatizar estas tareas y para ello, existen herramientas como IaC (*Infrastructure as Code*) y CaC (*Configuration as Code* referenciados en la parte uno de este documento).





Es importante también saber que hay distintos algoritmos que se pueden utilizar en los balanceadores. Se dividen en dos grande bloques:

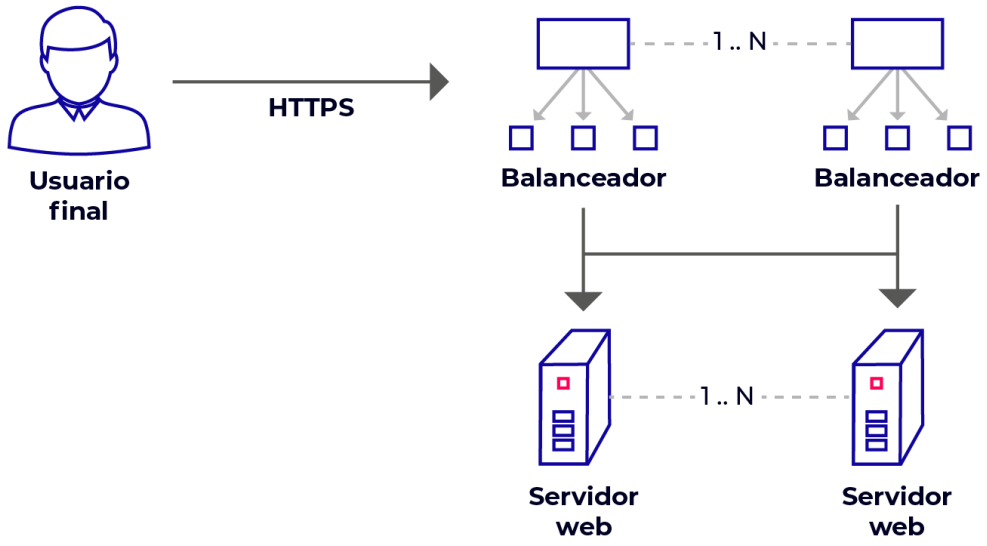
1. **Algoritmos estáticos**: **no tienen en cuenta la carga de las máquinas** a las que envían las peticiones o tareas. Da igual que una máquina esté muy cargada, si le toca recibir tarea, la va a recibir. El algoritmo más común es **round robin**, o sea, si tengo 4 máquinas, le envío por orden primero a la 1ª, acabando por la 4ª y la 5ª tarea se la vuelvo a mandar al primer "nodo". Se conoce como nodo a cada uno de los servidores que forman un conjunto que recibe tareas de un balanceador. Suelen estar agrupados por funcionalidad o similitud de tareas. Este tipo de algoritmos tiene la ventaja de que es **muy sencillo de implementar**, y además, **para ciertos escenarios**, como puede ser el de responder peticiones http, **es muy eficiente**.

2. **Algoritmos dinámicos**: estos algoritmos, al contrario que los anteriores, **sí tienen en cuenta la carga de los nodos** a los que balancean. **Son muy útiles cuando las tareas que se tienen que distribuir varían mucho de unas a otras**, que es lo que puede causar la congestión de un nodo, frente a otro mucho más libre. En estos casos, aplicar estos algoritmos tiene unos resultados excelentes. Un

ejemplo puede ser el procesamiento en batch de lotes de tareas con tamaño dinámico.

Una restricción importante es también la necesidad en algunos sistemas de balancear las peticiones con mismo origen, siempre hacia el mismo servidor destino, vinculando dichas peticiones a las sesiones del usuario. A esto se le denomina “afinidad de sesión” o “sticky session”.

Es frecuente encontrarse servidores web, como puede ser Nginx o Apache, haciendo de balanceadores, pero también hay software específico para realizar esta función.



Servidores de aplicaciones

Aparte de los web servers, existen otro tipo de servidores: los app servers o servidores de aplicaciones. Estos están ligados a la **lógica de negocio** y **son más complejos** que los web servers.

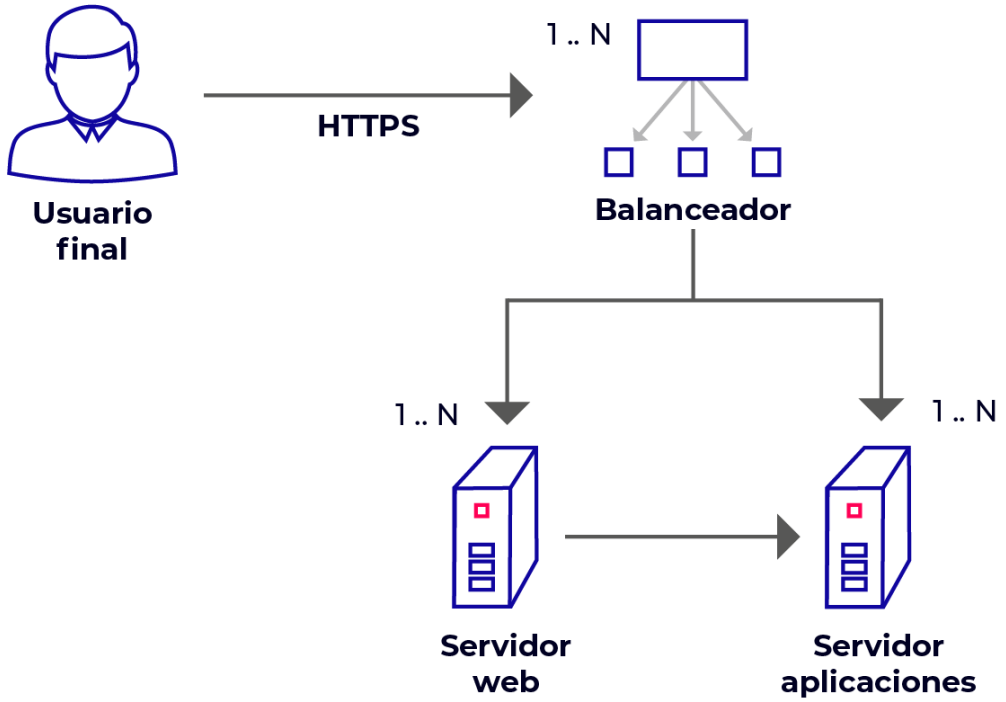
Si bien la tarea principal de los **web servers** es **servir páginas web estáticas**, los **app servers** sirven aplicaciones. Estas aplicaciones se pueden categorizar en varios tipos entre los que se encuentran:

- **Aplicación web dinámica**, la cual es servida a un navegador.
- **API (REST, SOAP, etc.), que expone los recursos disponibles para ser consumidos por los clientes** y realiza operaciones ligadas al negocio. Por ejemplo, si estás utilizando la app de Instagram y la aplicación quiere acceder a la información de tu perfil para pintarla, realizará una petición a un app server.

Mientras que los clientes de un webserver son navegadores, los de un app server pueden ser o bien un navegador, o bien **otros web servers que necesitan** cierta **información antes de servir una página web o responder una petición** o incluso, **otros app servers cuyas aplicaciones necesiten información**.

La conexión a estos servidores se puede realizar tanto por HTTP como por HTTPS, utilizando puertos distintos a los de los servidores web. El protocolo a utilizar dependerá de **si estamos en una red interna de confianza**, donde se puede utilizar **HTTP** o **incluso de la sensibilidad del dato**, que aunque sea una red interna, puede decidirse comunicarse mediante **HTTPS**. **Si la comunicación traspasa nuestra red interna** es recomendable utilizar **HTTPS**.

Algunos de los servidores de aplicaciones más utilizados son Apache Tomcat, Jetty, JBoss Wildfly, IBM Websphere y Liberty, entre otros.


1 .. N
HTTPS
Usuario final
Balanceador
1 .. N
1 .. N
Servidor web
Servidor aplicaciones

Sistemas de bases de datos

Un **sistema de Base de Datos** (**DB** por sus siglas en Inglés) es una herramienta que **captura, organiza y relaciona datos,** permitiendo al usuario, ya sea una persona u otro sistema, **acceder, analizar y/o explotar estos datos**.

Las bases de datos se pueden clasificar de muchas formas según el contexto que se esté manejando, la utilidad de las mismas o las necesidades que satisfagan. Una de las formas de clasificación más utilizada dentro de contexto de TI es la **clasificación según modelo de administración de datos**. Algunas de las categorías más destacadas son:

- **Relacionales**, donde los datos se almacenan y se acceden por medio de relaciones ya establecidas. Estos datos se organizan en un conjunto de **tablas** con filas y columnas. Las tablas se utilizan para contener información sobre los objetos que se representarán en la base de datos. Cada **columna** de una tabla contiene un cierto **tipo de dato** y un campo almacena el valor real de un atributo. Las **filas** en la tabla representan una colección de **valores relacionados de un objeto o entidad**. Cada fila de una tabla se puede marcar con un identificador único llamado clave principal, y las filas entre varias tablas, se pueden relacionar utilizando claves externas. Se puede acceder a estos datos de muchas maneras diferentes sin reorganizar las tablas de la base de datos. El lenguaje más común utilizado en este tipo de bases es **SQL** (Structured Query Language). Algunas de las DB más conocidas de esta categoría son PostgreSQL, MySQL, Oracle y MariaDB.

- **No Relacionales** o NoSQL, son el conjunto de todas aquellas bases de datos que proporcionan un mecanismo para el almacenamiento y la recuperación de datos que se modelan en medios distintos de las relaciones tabulares utilizadas en las bases de datos relacionales. Este tipo de base de datos tiene aplicaciones diversas según su naturaleza, desde cachés distribuidas a almacenamiento primario de datos.

Base de Datos

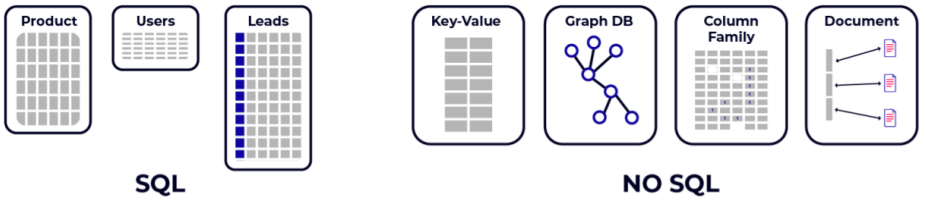
autentia

¿Qué es?

Una base de datos es una especie de "almacén" que nos permite guardar grandes cantidades de información para su posterior **recuperación, análisis y/o transmisión**. Podemos encontrar diversos tipos de bases de datos y se clasifican de acuerdo a las necesidades que busquen solucionar en dos grandes bloques.

TIPOS

No Relacionales	Definición	Ejemplo
Documentales	**Los datos se almacenan y se consultan como documentos tipo JSON**. Los documentos pueden contener muchos pares diferentes clave-valor, o incluso documentos anidados. Son más flexibles al cambio ya que si el modelo de datos necesita cambiar, solo se deben actualizar los documentos afectados y no todo el esquema.	MongoDB
De Gráfos	**Usan nodos para almacenar entidades de datos y aristas para almacenar las relaciones entre nodos**. El valor de estas bases de datos se obtiene de las relaciones entre nodos y no hay un límite para el número de relaciones que un nodo pueda tener. Un borde siempre tiene un nodo de inicio, un nodo final, un tipo y una dirección.	Neo4J, HyperGraphDB



Bases de Datos NoSQL

autentia

Definición

Las bases de datos noSQL manejan grandes cantidades de información no estructurada, almacenando los datos de diferentes formas (documentos, grafos o colecciones de clave-valor, etc.)

VENTAJAS

- Asegura que **la base de datos no se convierta en un cuello de botella** si la cantidad de datos aumenta.
- **Almacena grandes cantidades de datos desestructurados**, pudiendo almacenar cualquier dato sin establecer restricciones por su tipo o estructura.
- Las bases de datos NoSQL **permiten un escalado horizontal en múltiples nodos o servidores** de forma inmediata y sin problemas.
- Debido a su naturaleza no relacional, **no necesita un modelo de datos muy detallado, lo que ahorra tiempo de desarrollo**.

DESVENTAJAS

- La comunidad **NoSQL carece de madurez**, ya que es relativamente nueva frente a los RDBMS de SQL.
- La falta o **escasez de herramientas para generar informes** sobre el análisis y pruebas de rendimiento en noSQL frente a la amplia gama que encontramos en SQL.
- En el lenguaje de consulta en las bases de datos noSQL usa sus propias características, por lo que **no es 100% compatible con el lenguaje SQL** utilizado en las bases de datos relacionales.
- Hay muchas bases de datos NoSQL y **una falta de estandarización** en ellas, pudiendo causar un problemas en las migraciones.

Algunas de las bases de datos noSQL más utilizadas son:

- Redis
- Neo4j
- CouchDB
- Postgres
- Hbase
- MongoDB
- BigTable
- Cassandra

Esta categoría a su vez comprende otras categorías. Las más destacadas son:

- **De tablas, columna ancha o BigTables**, donde se guardan los datos a lo largo de tablas que pueden tener muchísimas columnas. Se suele usar para guardar búsquedas en internet. Ejemplos son Cassandra, HBase...
- **Orientada a Grafos**, donde se representa la información como nodos de un grafo y sus relaciones con las aristas del mismo, de manera que se pueda usar teoría de grafos para recorrer la base de datos ya que ésta puede describir atributos de los nodos (entidades) y las aristas (relaciones). Algunas de las DB más conocidas de esta categoría son Neo4j y Sparksee.
- **Clave-Valor**, donde se almacenan datos como un conjunto de pares clave-valor en los que una clave sirve como un identificador único. Tanto las claves como los valores pueden ser cualquier cosa, desde objetos simples hasta objetos compuestos complejos. Algunas de las DB más conocidas de esta categoría son Redis o Dynamo.
- **Documentales**, que permiten almacenar y consultar datos como documentos de tipo JSON, YAML, XML y otros. En este enlace, podemos aprender cómo funciona una base de datos de este tipo, en este caso **CouchDB**. Las bases de datos de documentos facilitan a los desarrolladores el almacenamiento y la consulta de datos en una base de datos mediante el mismo formato de modelo de documentos que emplean en el código de aplicación. Algunas de las DB más conocidas de esta categoría son MongoDB, CouchDB, SimpleDB o Lucene.

A su vez, las bases de datos pueden ser **centralizadas** (toda la información de la base de datos se encuentra almacenada en un único lugar) o **distribuidas**, que son un conjunto de múltiples bases de datos lógicamente relacionadas y que se encuentran distribuidas en **diferentes espacios lógicos y/o geográficos** e **interconectados** por una red de comunicaciones. Dichas DB tienen la capacidad de realizar **procesamientos autónomos** y permiten realizar operaciones locales o distribuidas.

También se utilizan para Business Intelligence (BI o inteligencia de negocio), que comprende las estrategias y tecnologías utilizadas por las empresas para el análisis de datos de información empresarial. Dos de las formas

más comunes de implementar BI es mediante el uso de sistemas de Data Warehouse (DW o almacén de datos) y Data Lake.

- **Data Warehouse** es un sistema utilizado para reportes y análisis de datos. Son almacenes centralizados de datos integrados de una o más fuentes dispares. Almacenan datos actuales e históricos en un solo lugar que se utilizan para crear informes analíticos para los trabajadores de toda la empresa.

  Los datos almacenados en el almacén se cargan desde los sistemas operativos (como marketing o ventas). Los datos pueden pasar a través de un almacén de datos operativos y pueden requerir la limpieza de datos para operaciones adicionales para garantizar la calidad de los datos antes de ser utilizados en el DW para la presentación de informes.

- **Data Lake** es un sistema o depósito de datos almacenados en su formato natural (o en crudo), generalmente objetos de tipo blob o archivos. Un lago de datos suele ser un único almacén de todos los datos de la empresa, incluidas las copias sin procesar de los datos del sistema de origen y los datos transformados que se utilizan para tareas como informes, visualización, análisis avanzado y aprendizaje automático. Un lago de datos puede incluir datos estructurados de bases de datos relacionales (filas y columnas), datos semiestructurados (CSV, registros, XML y JSON), datos no estructurados (correos electrónicos, documentos y PDF) y datos binarios (imágenes, audio y vídeo). Se puede establecer un lago de datos on-premise (dentro de los centros de datos de una organización) o en la nube (utilizando servicios en la nube de proveedores como Amazon, Google y Microsoft).

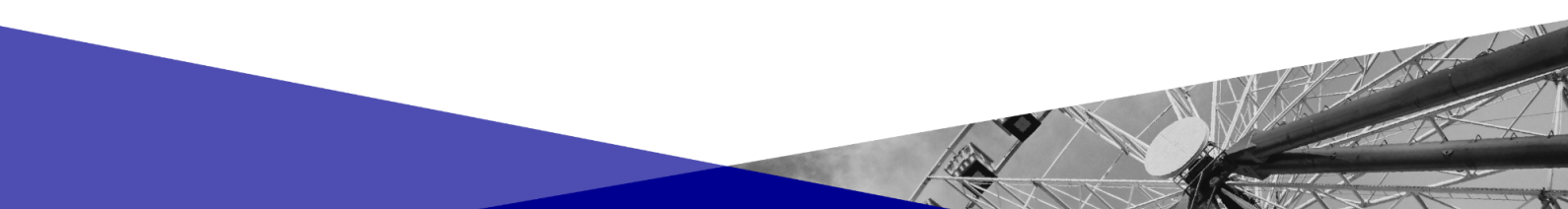
Como podemos ver, las DB almacenan **datos vitales**, no sólo para las aplicaciones, si no para el negocio. Suelen recopilar, por ejemplo, toda la información interesante de los clientes, los potenciales o los ya consolidados, así como aquellos que dejaron de utilizar tus servicios. Es muy útil al poder realizar muchas acciones, como por ejemplo segmentar en base a intereses o analizar comportamientos.

Es por esto que es importante, cuando se trata de algo tan delicado como una base de datos, tener su versionado controlado. Si no, algo como introducir o eliminar una columna en una tabla se puede convertir en un desastre. Por eso existen **herramientas de control de cambios** de bases de

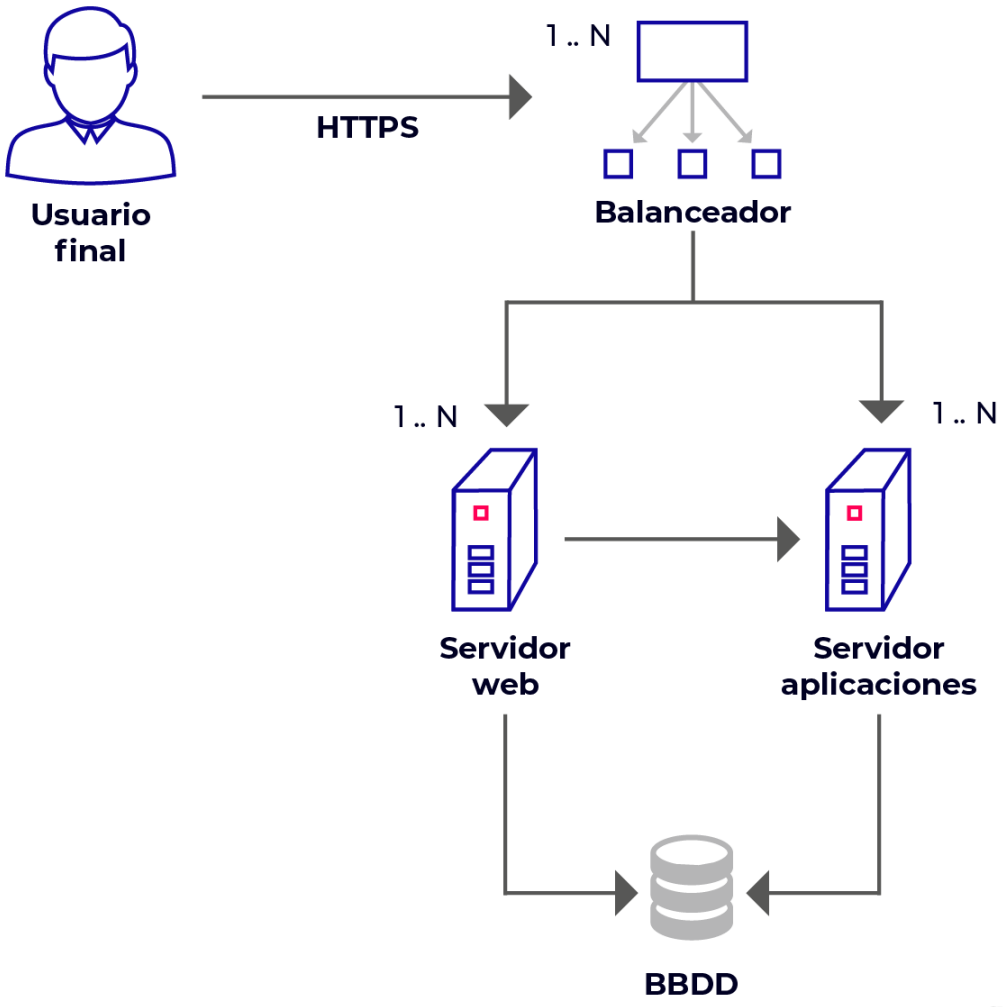
datos, como pueden ser Liquibase o Flyway.

Pongamos, por ejemplo, que tenemos una base de datos de usuarios en la que tenemos los campos “nombre”, “dirección” y “correo electrónico”. Cuando ya llevamos en producción un tiempo, vemos que sería beneficioso guardar también el número de teléfono y decidimos añadir ese campo a la tabla. Si esto lo hiciéramos **a mano**, lo más probable es que **acabe en desastre**. Sin embargo, si usamos estas herramientas que ofrecen la posibilidad de **ir versionando estos cambios**, tendremos más **seguridad de que el cambio funcionará de la forma esperada**. Además, **si hay algún problema** al realizar algún cambio, gracias a estas herramientas podríamos simplemente **volver a la versión anterior** de la base de datos y **deshacer el cambio**.

También, gracias a que **tenemos guardados todos los cambios** que hemos ido realizando, si en algún momento se **crea una base de datos nueva**, estas herramientas lo detectarán y **generarán automáticamente todas las tablas** con todos los cambios que hemos realizado, sin necesidad por nuestra parte de ejecutar **ningún script a mano**.



En lugar de Liquibase o Flyway, se podrían también **idear alternativas** desde DevOps como un **proyecto de bbdd** donde se tengan todos los **scripts almacenados en un repositorio por versión y entorno**.



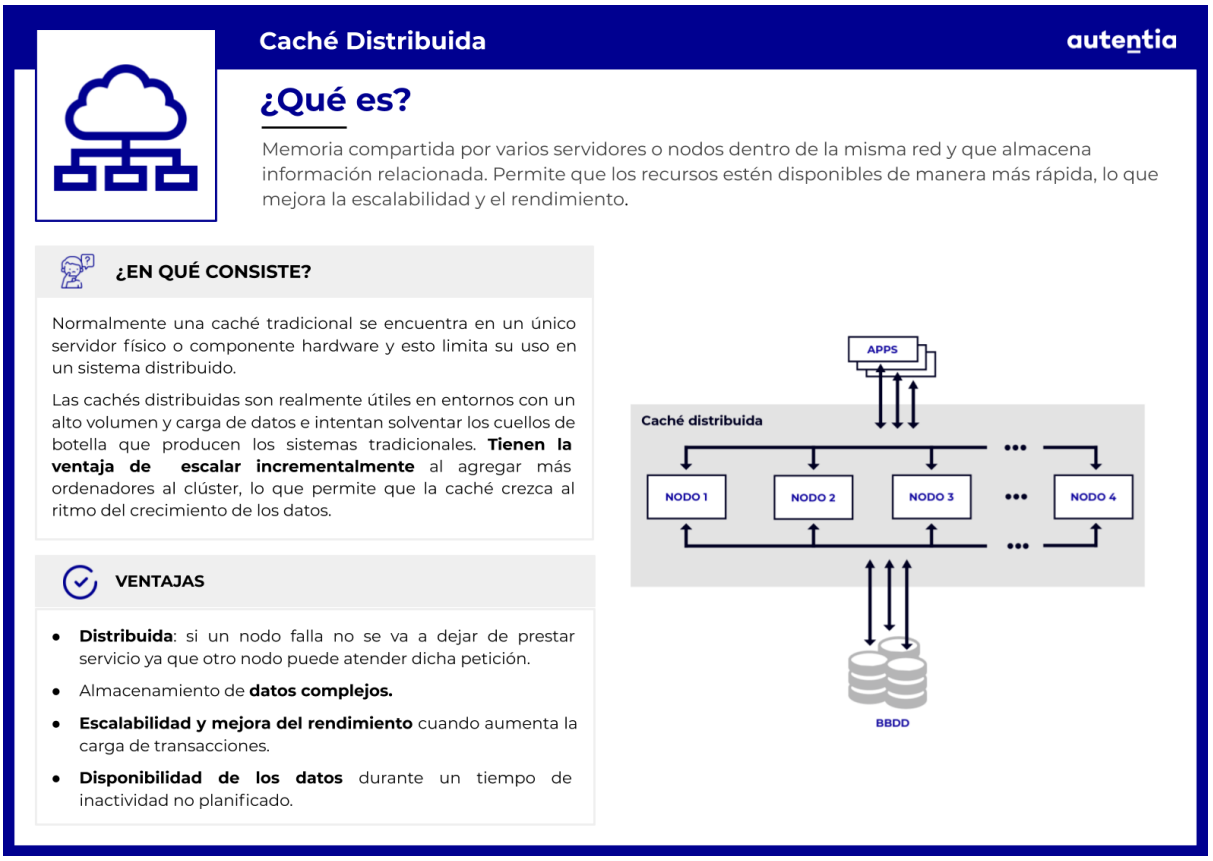
Sistemas de caché

Cuando tenemos un servidor expuesto a un gran número de peticiones, una técnica muy común y eficaz es utilizar un sistema de caché. Una caché es básicamente una **capa de almacenamiento de datos de alta velocidad** que **mantiene datos normalmente temporales** o de naturaleza transitoria, de modo que las **solicitudes de acceso** a esos datos **se atiendan más rápido que si se recogieran de la ubicación principal** del almacenamiento de esos datos. De esta forma, si hay ciertas peticiones que se **repiten mucho** y la **respuesta es siempre la misma**, esa **respuesta** se puede **cachear**, consiguiendo, entre otras, las siguientes ventajas:

- Por una parte, la **respuesta a esa petición será más rápida** que si se tuviera que acceder a la base de datos.
- Se **libera de carga al servidor**, pudiendo **dedicarse a otras peticiones** y por tanto, soportando una mayor carga de peticiones en general.

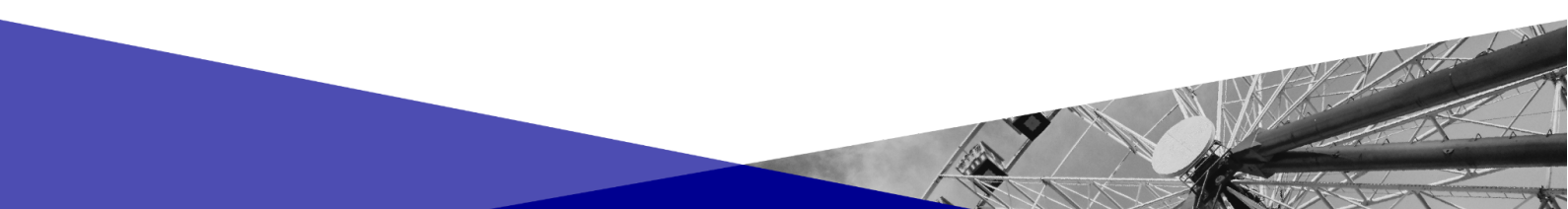
Cuando utilizamos un sistema de caché, es necesario **saber qué es lo que estamos haciendo** y tener en cuenta distintos aspectos como puede ser el **tiempo de vida** de los datos en la caché antes de eliminarlos (**TTL o Time to live**) o saber la **volumetría** de lo que vamos a cachear, es decir, **cuánto espacio** vamos a tener que utilizar según el **peso de los objetos** que vamos a querer guardar (MB, GB). Esto nos ayudará a decidirnos entre los distintos tipos de caché según la situación.

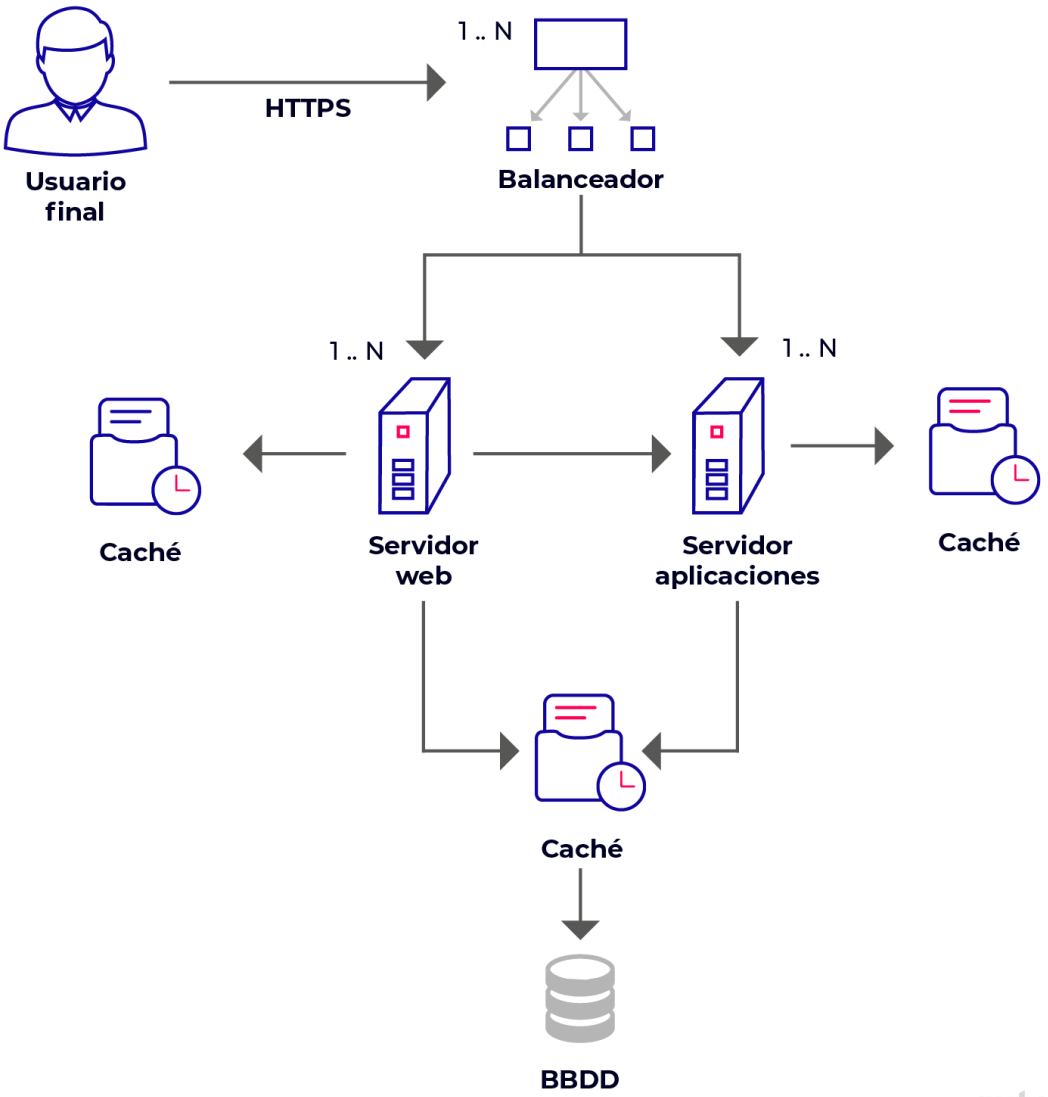
Los tipos de caché se pueden dividir en dos tipos: **no distribuida y distribuida**. La primera se suele utilizar para guardar resultados de operaciones que son **poco pesados**, siendo un ejemplo de esto Ehcache. En el caso de una caché **distribuida**, sin embargo, en vez de utilizar la memoria de la misma máquina, se dedica un **conjunto** de máquinas explícitamente para este fin. Un ejemplo de esto puede ser Redis. Una de las **desventajas** que suelen tener las **cachés distribuidas** es que su acceso es **más lento** que una en memoria, ya que normalmente tendremos que acceder a una **máquina externa**. Por otro lado, este tipo de caché es capaz de guardar **más información**, y por tanto es más útil para almacenar **resultados de operaciones más pesadas**. Por lo tanto, cuando vamos a cachear algo hemos de sopesar, según distintos criterios, en qué caché guardamos esta información.



Aparte de estos dos tipos, tenemos también la **caché en disco**, que se suele utilizar cuando hemos de realizar **procesos muy costosos** para evitar tener que volver a ejecutarlos de nuevo (procesos que a lo mejor **tardan minutos**).

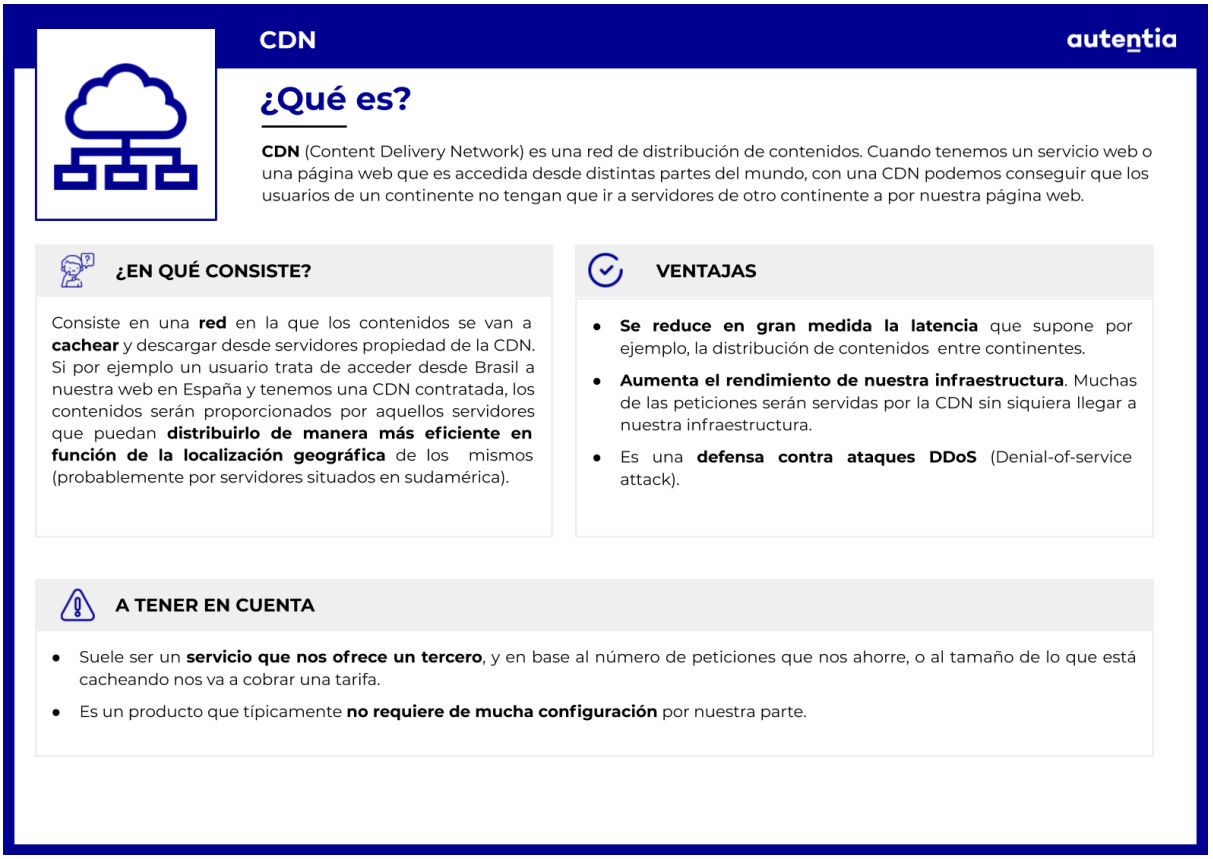
Es posible también que el sistema de caché que utilicemos requiera de **cierta configuración**, que deberemos tener **controlada y almacenada en un repositorio**.

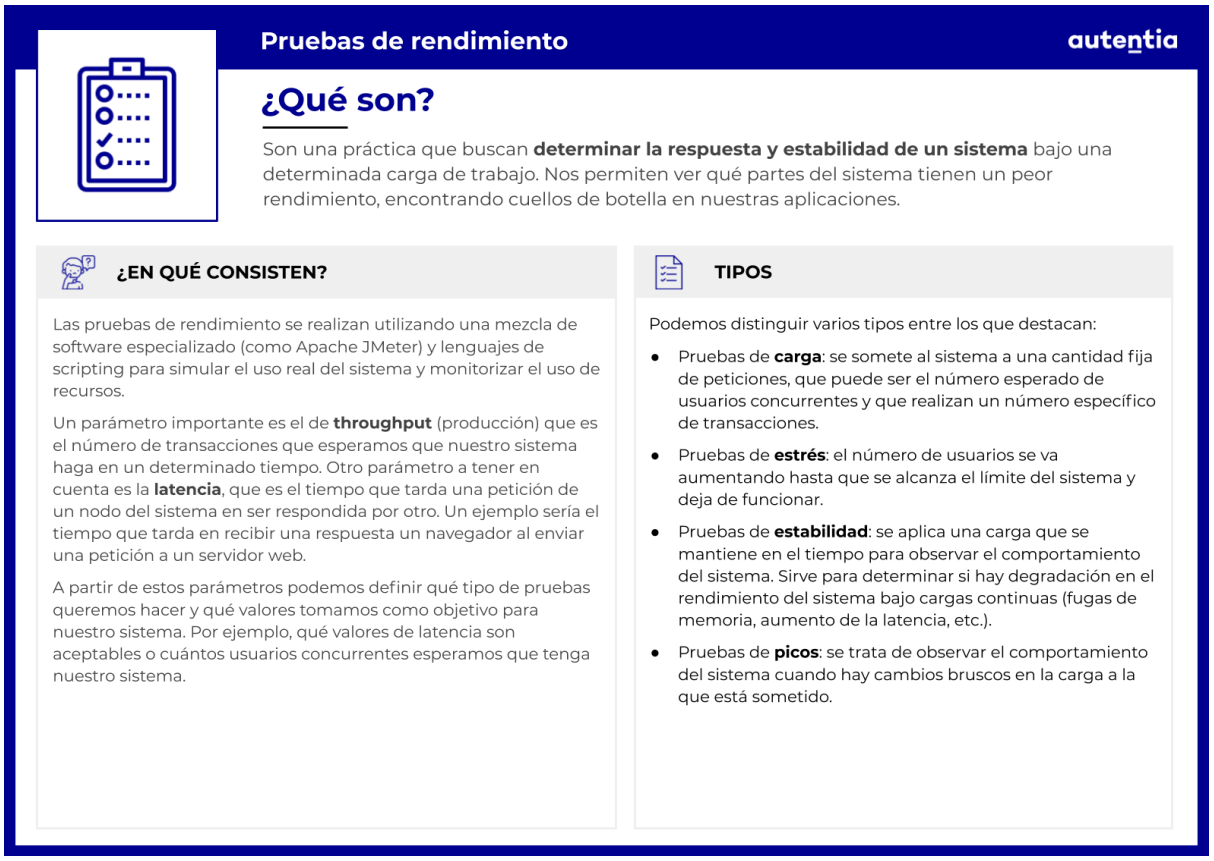



Usuario final
HTTPS
1 .. N
Balanceador
1 .. N
1 .. N
Caché
Servidor web
Servidor aplicaciones
Caché
Caché
BBDD

CDN

Una CDN (Content Delivery Network) es una **red de distribución de contenidos**. Cuando tenemos un servicio web o una página web a la que se accede desde distintas partes del mundo, con una CDN podemos conseguir que los usuarios de otra parte del mundo no tengan que descargarse de servidores en Europa por ejemplo, los contenidos de nuestra página web. Esos contenidos se van a cachear y descargar desde servidores propiedad de la CDN. Si por ejemplo, un usuario trata de acceder desde Brasil, a nuestra web en España y tenemos una CDN contratada, los contenidos serán proporcionados por aquellos servidores que puedan distribuirlo de manera más eficiente en función de la localización geográfica de los mismos (probablemente por servidores situados en sudamérica). De esta forma **se reduce en gran medida la latencia** que supone por ejemplo, la distribución de contenidos entre continentes.



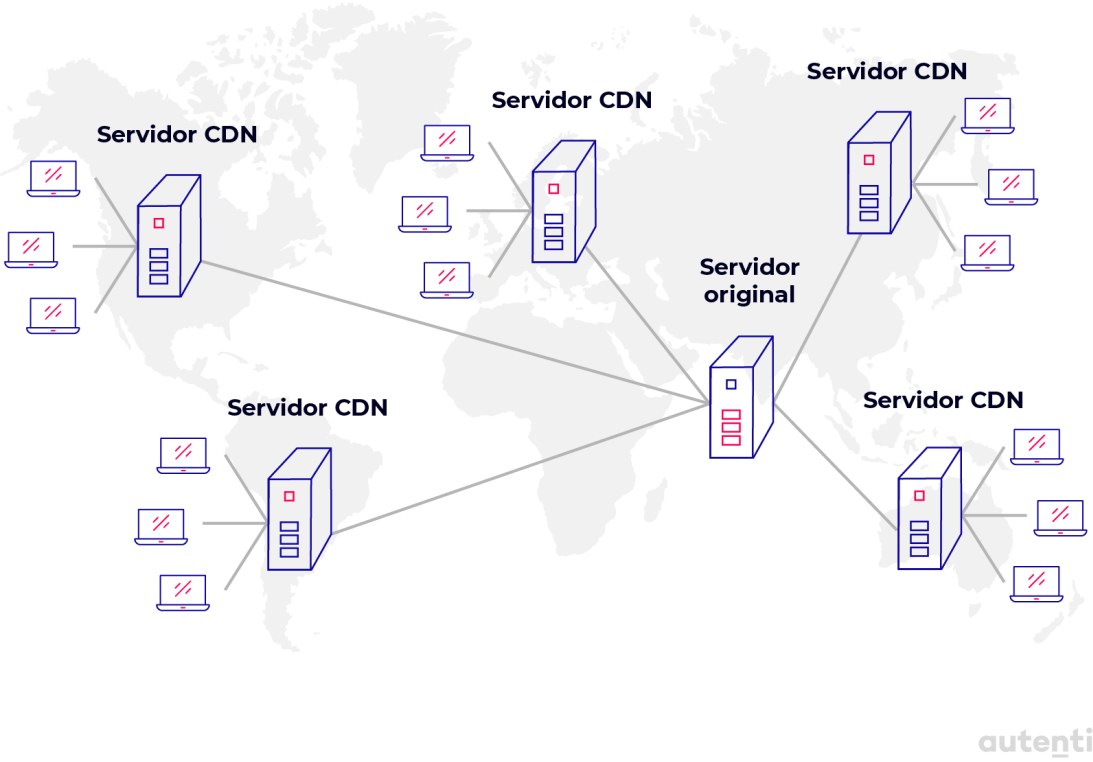


Al ser una caché, aplican los términos vistos anteriormente, como por ejemplo TTL.

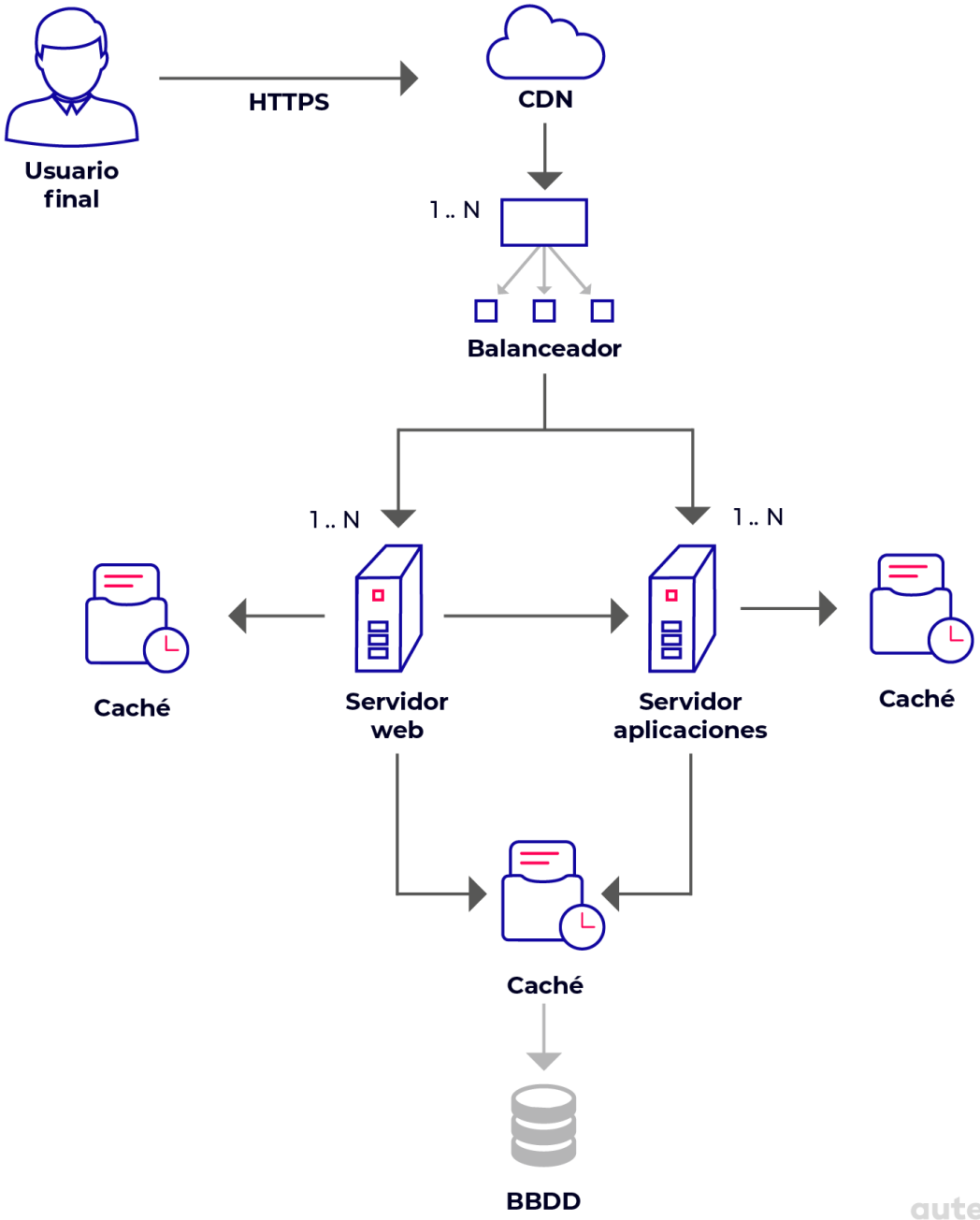
Una CDN es un producto muy útil en ese escenario de presencia internacional, y no sólo porque reduce la latencia al usuario final, sino porque aumenta el rendimiento de nuestra infraestructura. **Muchas de las peticiones van a ser servidas por la CDN sin siquiera llegar a nuestra infraestructura**. Solamente cuando el contenido caduque en la CDN vamos a tener que responder de nuevo para refrescar el contenido.

No sólo **es útil con contenido estático**, sino **también con contenido semi-estático**. Pongamos que somos una empresa de autobuses internacionales. Nuestros orígenes y destinos pueden cambiar, pero no lo van a hacer cada hora y seguramente no lo hagan cada día. No obstante, si tenemos mucho éxito y somos muy grandes, vamos a recibir de forma continuada peticiones sobre nuestros orígenes y destinos. Tener esa información cacheada en la CDN, aunque sea por periodos de 30 minutos, va a liberar a nuestra infraestructura de una cantidad ingente de llamadas. Esto en empresas grandes **puede suponer el ahorro de responder a millones de peticiones diarias con la consiguiente liberación de recursos**, como

puede ser la CPU. No sólo es importante en entornos cloud donde se paga por el uso de CPU, sino también en on-premise, donde dispondremos más recursos para utilizar para otros servicios.



Suele ser un servicio que nos ofrece un tercero, y en base al número de peticiones que nos ahorre o al tamaño de lo que está cacheando, nos va a cobrar una tarifa. Es un producto que típicamente no requiere de mucha configuración por nuestra parte.


Usuario final
HTTPS
CDN
1 .. N
Balanceador
1 .. N
1 .. N
Caché
Servidor web
Servidor aplicaciones
Caché
Caché
BBDD

API Gateway

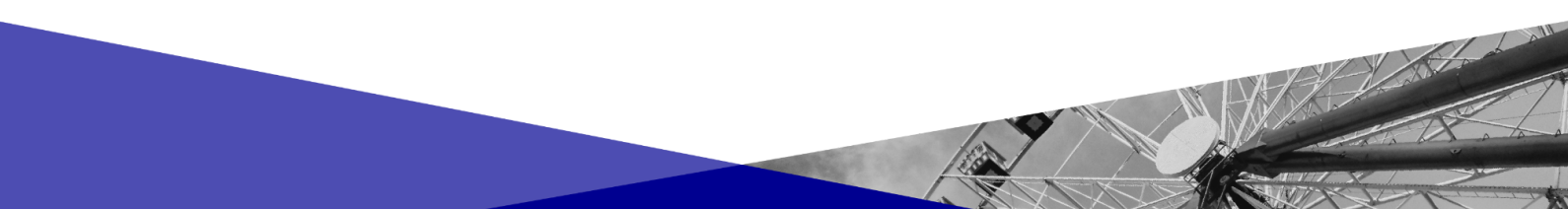
Supongamos que estamos construyendo una tienda virtual utilizando **microservicios** y con **distintos canales** front (web y apps móviles nativas). Imaginemos que actualmente estamos desarrollando la página de detalle del producto. La UI de esta pantalla puede mostrar mucha información sobre un producto. Por ejemplo:

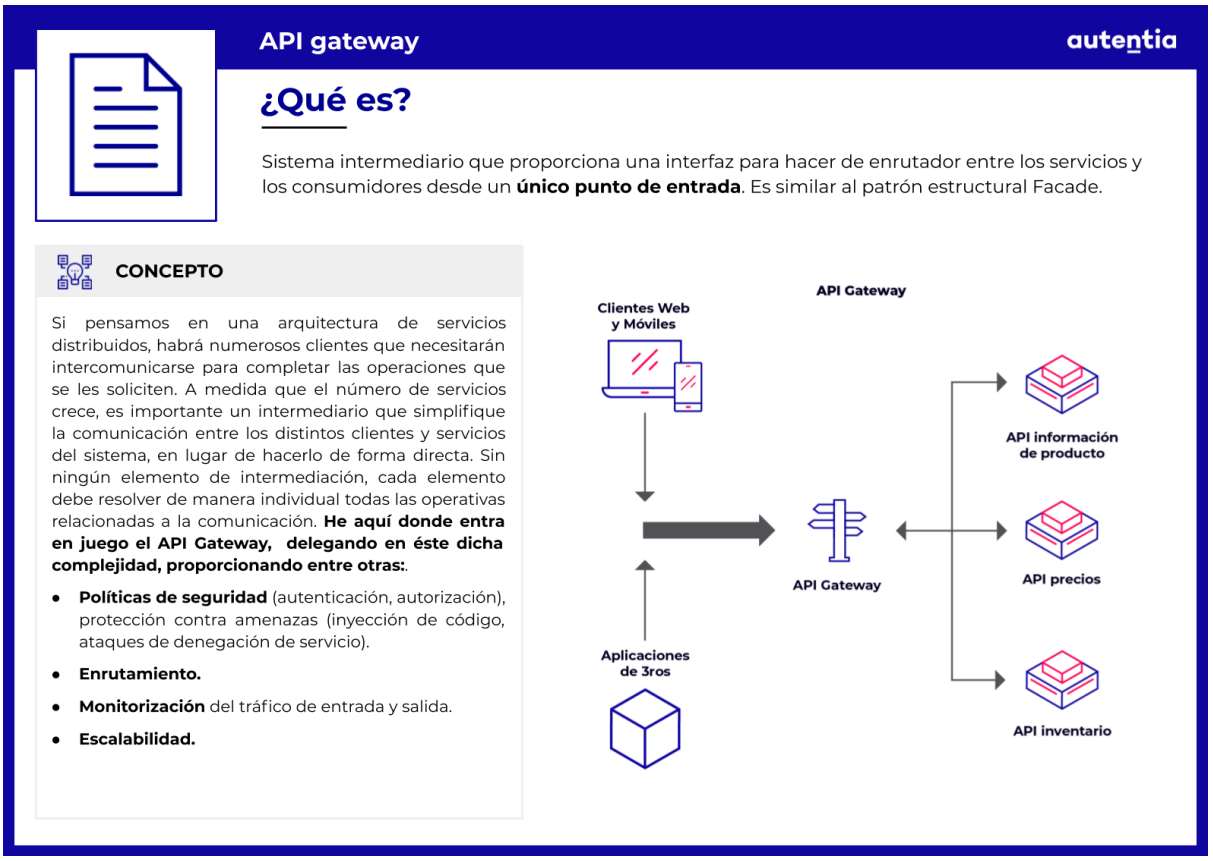
- Información básica sobre el producto.
- Disponibilidad.
- Opciones de compra.
- Valoración de los clientes.

Como estamos implementando microservicios, los datos de detalles del producto se distribuyen en **múltiples servicios**:

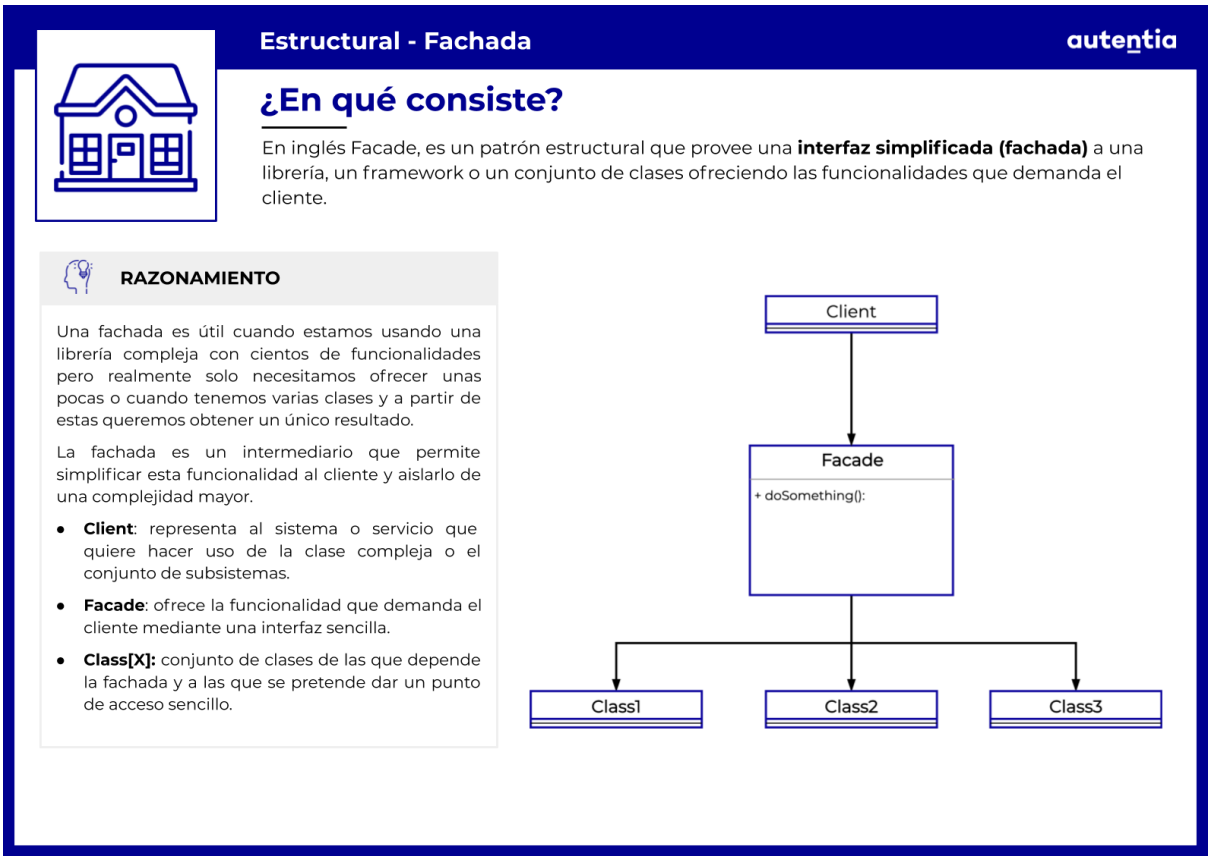
- Servicio de información del producto: información básica sobre el producto, como marca, modelo, características, etc.
- Servicio de precios: precio del producto.
- Servicio de inventario: disponibilidad del producto.

En consecuencia, el código que muestra los detalles del producto necesita obtener información de todos estos servicios. ¿Cómo harán las aplicaciones cliente que necesitemos desarrollar para localizar y acceder a todos estos servicios individuales? Aquí es donde un **API Gateway** nos puede ser de utilidad.





Un API Gateway es un **servidor que actúa como único punto de entrada al sistema**. Es similar al patrón *Fachada (GoF Facade pattern)*. Un API Gateway **encapsula la arquitectura interna del sistema** y **proporciona una API que se adapta a cada cliente**. Es también responsable del enrutamiento de solicitudes, la composición y la traducción del protocolo. Todas las solicitudes de los clientes pasan primero por la API Gateway. Este luego **enruta las solicitudes al microservicio apropiado**. A menudo maneja una solicitud **invocando múltiples microservicios y agregando los resultados**. Puede traducir entre protocolos web como HTTP y WebSocket y protocolos no compatibles con la web que se utilizan internamente.



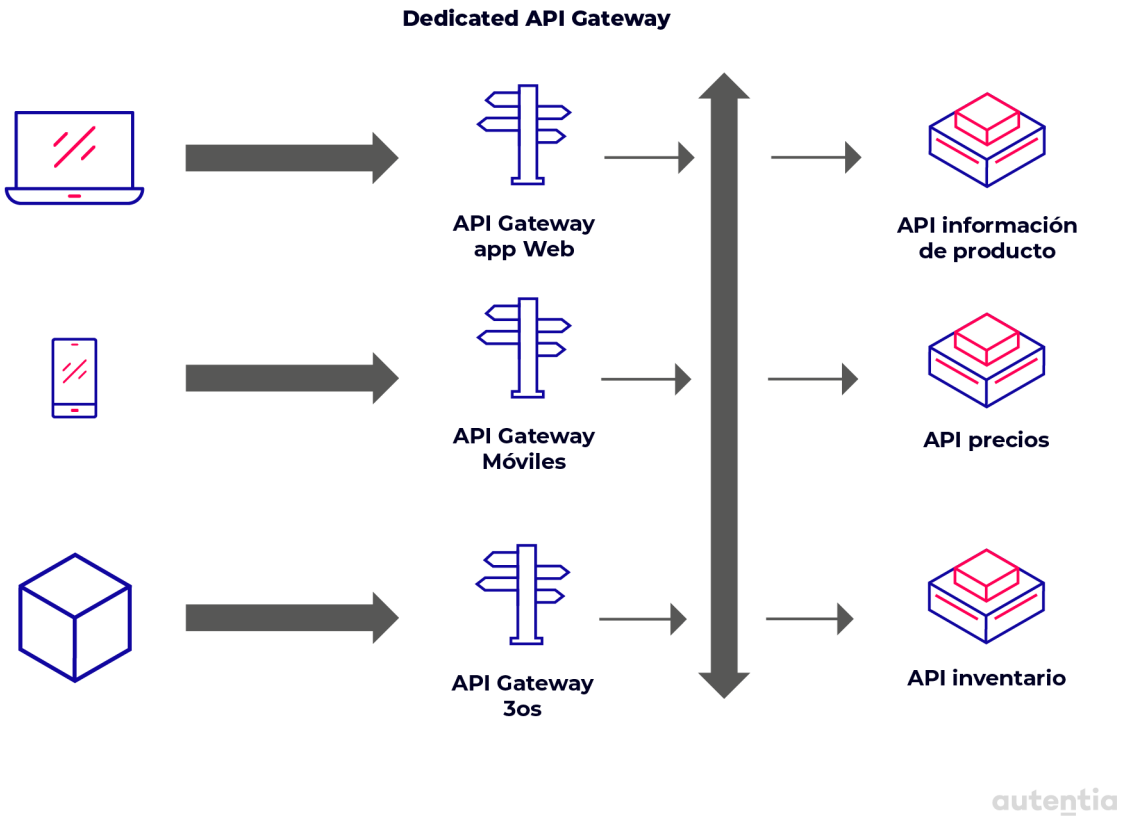
Puede tener **otras responsabilidades**, como autenticación, monitorización, equilibrio de carga, monetización del api, almacenamiento en caché, configuración y administración de solicitudes y manejo de respuesta estática.

Entre las funciones del api gateway podemos encontrar:

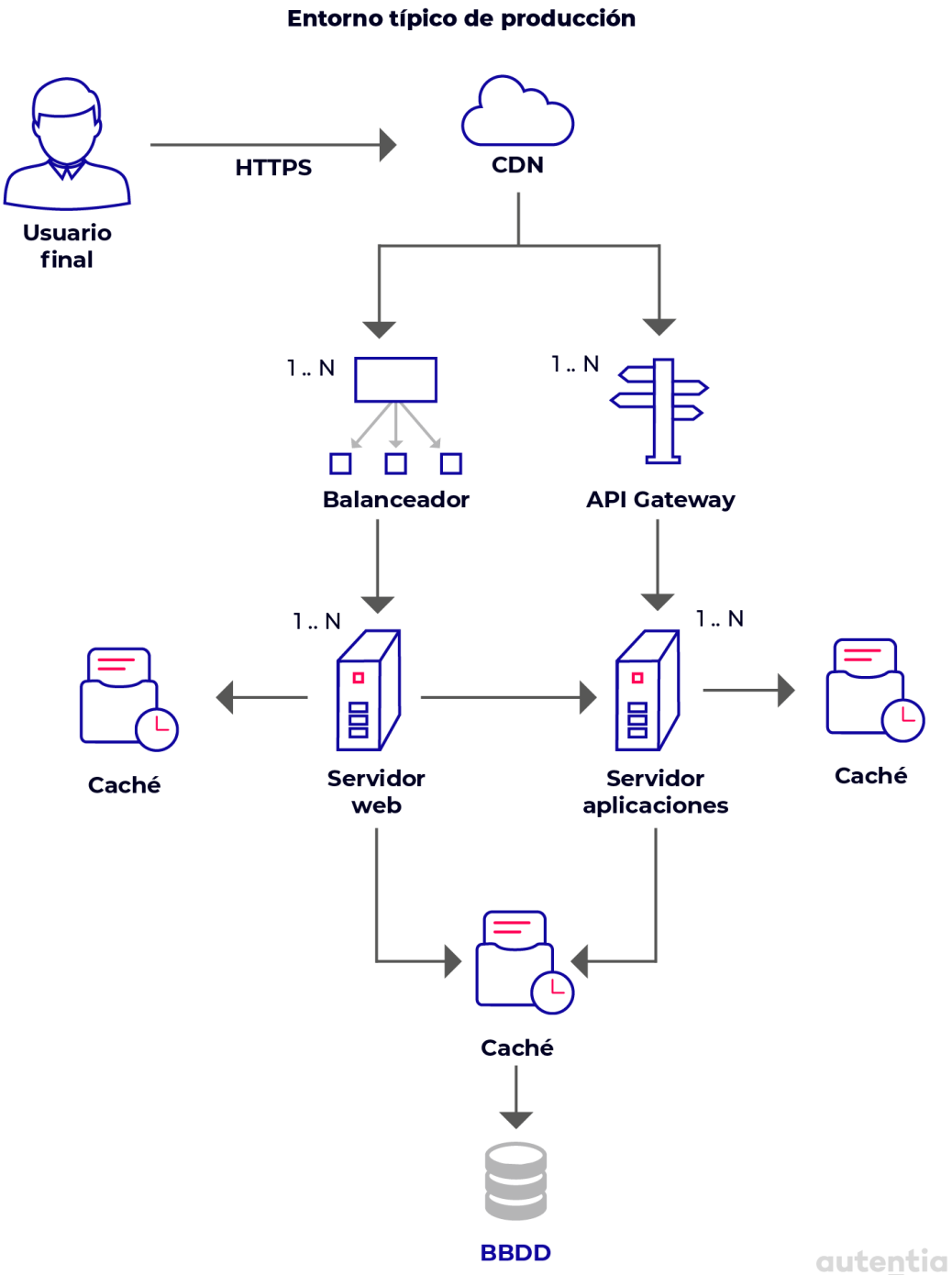
- **Evitar llamadas incorrectas a nuestra infraestructura**, como por ejemplo, aquellas que usan un token de seguridad caducado o es incorrecto. Si ese filtro lo hace el API Gateway, no llega a nuestros microservicios y no se gasta sus recursos.
- **Monetizar el API**, creando distintos paquetes de suscripción, por ejemplo, limitados por cantidad de peticiones.
- **Encapsular la estructura interna de la aplicación**, ya que los clientes se comunican solamente con el gateway en lugar de con recursos específicos.
- Manejar el **versionado del API**, desfasar una versión impidiendo por ejemplo, el acceso a recursos antiguos para ciertos clientes o canales.

- Capacidad para aplicar **throttling o estrangulamiento:** fijar una cuota máxima de llamadas que los distintos canales o terceros pueden hacernos, ya sean totales diarias o máximas por segundo.
- **Permitir o denegar el acceso** a nuestras APIs dependiendo del canal o tercero que nos esté utilizando.
- **Controlar las llamadas** que tiene nuestro sistema y de qué canales o proveedores vienen.

Otra forma bastante común de utilizar estas piezas es configurando varios API Gateways. De esta forma, **podemos tener un gateway exclusivo para cada tipo de cliente** (uno para nuestro cliente web, otro para nuestras apps móviles nativas, otro para consumo de terceros, etc.).



Si incorporamos el concepto de API Gateway a nuestra infraestructura quedaría un entorno resultante como el de la siguiente imagen.


Entorno típico de producción
Usuario final
HTTPS
CDN
1 .. N
Balanceador
1 .. N
API Gateway
1 .. N
Servidor web
1 .. N
Servidor aplicaciones
Caché
Caché
Caché
BBDD
autentia

APM

APM o **Application Performance Management,** es una pieza que va a **monitorizar** una aplicación durante su funcionamiento. Su principal funcionalidad es **detectar problemas de rendimiento** en aplicaciones complejas para de esa forma, mantener un nivel de servicio acorde. Hay dos grupos principales de métricas de rendimiento:

- Aquellas que definen el rendimiento experimentado por el **usuario final**. Aquí se miden cosas como el **tiempo de respuesta** en momentos de picos de carga.
- Aquellas que definen los **recursos computacionales** utilizados por la aplicación. Esto sirve para comprobar si se pueden soportar correctamente los picos de carga, además de poder, por ejemplo, encontrar posibles cuellos de botella.

Dependiendo del APM que se utilice, se deberá realizar una menor o mayor configuración, pero es importante almacenarla en control de versiones en el caso de que la haya.

Las APM más utilizadas son Glowroot, Appdynamics, New Relic o Dynatrace.

Monitorización de infraestructura

La monitorización de infraestructura se basa en la **generación automática de logs** desencadenada por **tráfico en la red** o **actividad** por parte de los **usuarios** a lo largo de **toda la infraestructura**. Estos logs son **detallados**, incluyendo, por ejemplo, la fecha, tipo de evento, usuario que realizó la acción y toda información que se considere útil para la trazabilidad de un evento determinado. Después, estos son **agregados** por un software de monitorización de infraestructura, **agrupándolos en una base de datos** donde se pueden **ordenar**, **realizar búsquedas**, **analizar**, etc., ya sea por personas o por algoritmos.

Gracias a esta monitorización, una empresa puede **detectar problemas operacionales**, **identificar** posibles **brechas de seguridad**, cuellos de botella a nivel de hardware o identificar nuevas áreas de oportunidades de negocio.

Las herramientas más utilizadas son Dynatrace, Metricbeat o Prometheus.

Monitorización funcional

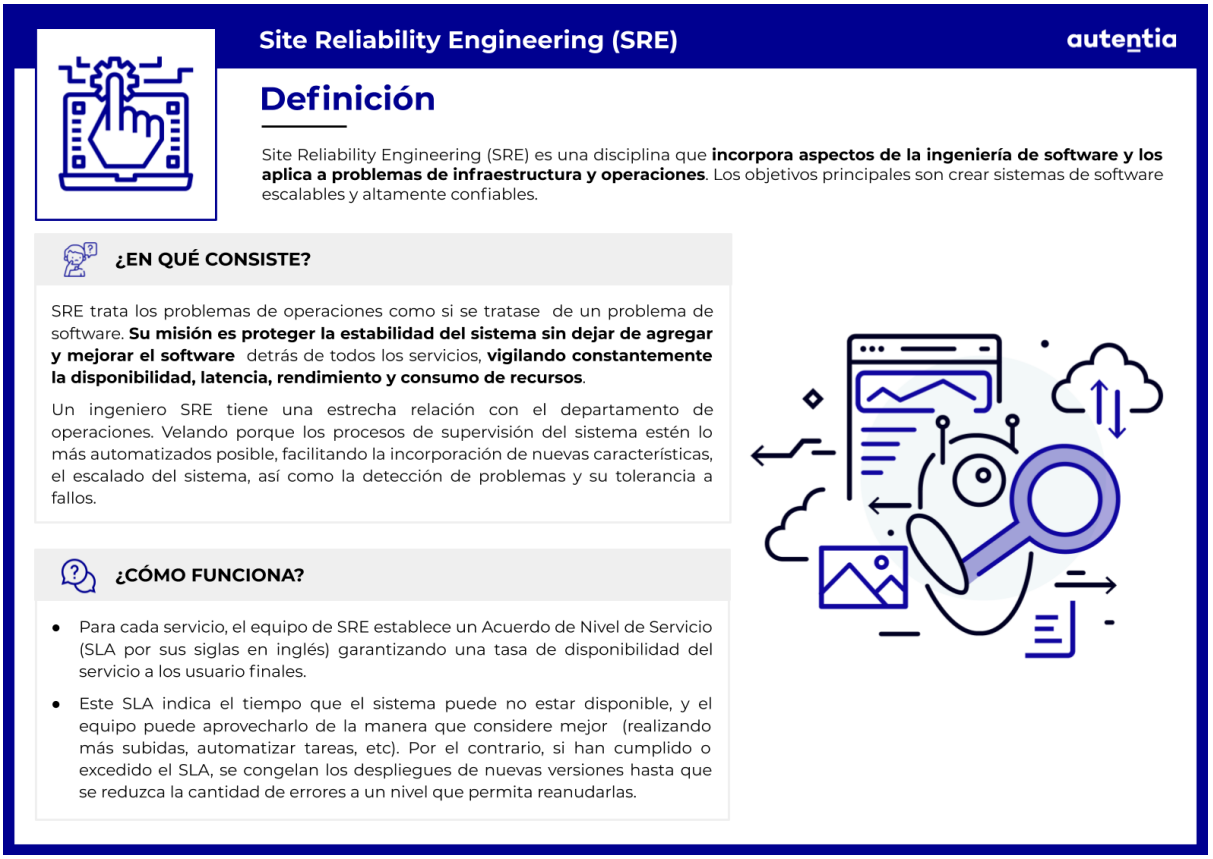
La monitorización funcional permite **observar el rendimiento y funcionalidades** de una aplicación o un sistema distribuido y **determinar si está capacitado** para realizarlas. Este tipo de monitorización se realiza de forma **automática** mediante la ejecución de **scripts**. La monitorización funcional **no soluciona problemas** pero es capaz de **sacarlos a la luz**.

Las herramientas para realizar este tipo de monitorización suelen tener **bastante configuración**, que además es bastante **dinámica** y **dependiente del entorno** (desarrollo, preproducción, producción...), por lo que es muy importante tenerla versionada en repositorio y con **variables**, lo que nos permitirá ahorrarnos numerosos errores.

Un ejemplo en este caso, podría ser Splunk. Es muy útil también para definir alertas funcionales que nos avisen de por ejemplo cuando no estamos vendiendo lo suficiente. Estas alertas pueden poner en evidencia otros problemas.

Perfil SRE

SRE (Site Reliability Engineer) es un término acuñado por Ben Treynor, Vicepresidente de Ingeniería de Google para solventar los problemas entre los equipos de desarrollo y operaciones, preocupándose de establecer un equilibrio entre añadir nueva funcionalidad y la estabilidad del sistema.



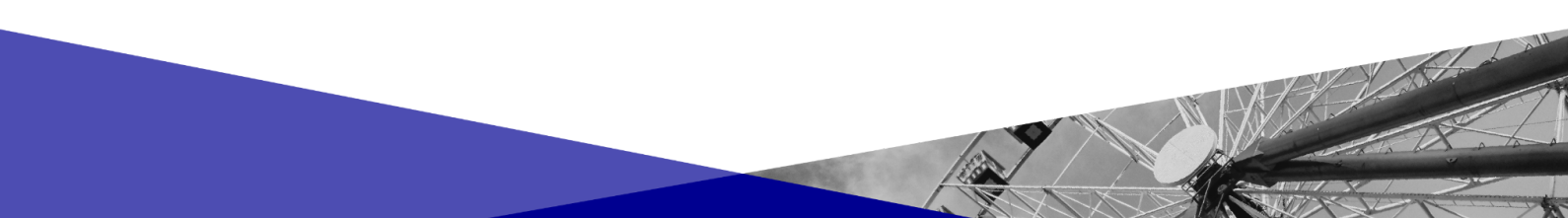
No existe una descripción precisa de sus funciones pero el equipo SRE debe asegurar la disponibilidad, el rendimiento, la monitorización y la respuesta a incidencias de los servicios de la organización. Para realizar este trabajo, “echan mano” de herramientas que les faciliten la tarea como:

- **Disponibilidad**: tanto en Kubernetes como en proveedores de servicios como AWS o Azure nos proveen de herramientas que nos permiten conocer el estado de nuestros servicios.
- **Monitorización**: tal y como hemos comentado en anteriores apartados, tenemos una gran variedad de herramientas que nos permiten monitorizar nuestros servicios en tiempo real y extraer conocimiento operacional sobre los datos generados por el sistema,

permitiéndonos generar informes y tomar decisiones operativas. Algunos ejemplos son BELK Stack, Prometheus, Splunk, etc.

- **Rendimiento**: para medir el rendimiento se utilizan las herramientas de monitorización antes mencionadas, herramientas de análisis proporcionadas por el propio producto como VisualVM de Java, Database Performance Analyzer de Oracle o simplemente comandos o utilidades de la terminal. Otra forma de medir el rendimiento de nuestros servicios es usar herramientas de APM (glowroot, appdynamics, new relic o dynatrace), tal y como hemos comentado en apartados anteriores.
- **Resolución de incidencias:** intentan solucionar las incidencias rápidamente. Realizando un proceso de “postmortem”. Detectando el origen del problema y poniendo las medidas necesarias para que no se vuelva a repetir.

SRE es un intento de concretar las buenas prácticas e implementar el modelo de DevOps en la organización que nos permita derribar los silos y barreras entre los departamentos de desarrollo y operaciones. Siguiendo el enfoque de mejora continua, pasando por la automatización, aportando valor con el uso de metodologías ágiles, pero sin perder de vista la estabilidad de los sistemas.



Parte 3

Administración de sistemas unix

Objetivo

Según la wikipedia el administrador de sistemas es "*la persona que tiene la responsabilidad de implementar, configurar, mantener, monitorizar, documentar y asegurar el correcto funcionamiento de un sistema informático o algún aspecto de este*."

La definición resulta bastante académica, si bien, no focaliza del todo el propósito del mismo que es **reducir los costes**.

La administración del sistema permite que los sistemas estén disponibles, aprovechando todas sus capacidades, rendimiento, reduciendo los tiempos de inactividad y minimizando la probabilidad de fallos en el mismo.

Es por este motivo que los desarrolladores deben tener ciertos conocimientos sobre este área para mantener su principal herramienta, el ordenador, engrasada y lista para la acción. No sólo les será más fácil configurar su entorno de desarrollo para adaptarlo a las necesidades que tengan, sino que, además, podrán reproducir las condiciones de los entornos de producción de forma fiel. Esto también facilitará a su vez las tareas de despliegue de las aplicaciones. En definitiva, se trata de mejorar la eficiencia.

En los siguientes capítulos abordaremos las herramientas básicas necesarias para realizar esta labor de la forma más eficiente.

Contexto

Históricamente, la gestión de la infraestructura de TI era un proceso manual.

El personal cualificado instalaba físicamente los servidores, se instalaba su sistema operativo, se configuraba un entorno y finalmente se instalaba las aplicaciones. Como era de esperar, este proceso manual era lento y a menudo, daba lugar a varios problemas. Algunos de ellos son:

- Coste de infraestructura muy elevado pues cada servidor es estático y cualquier modificación, normalmente, supone su reconfiguración completa.
- Se necesitaba mucho personal para realizar este procedimiento en cada uno de los servidores.
- Complejidad en la gestión, pues las tareas se realizan en un determinado orden sin posibilidad de paralelizar.
- Los entornos son muy variados en función de las aplicaciones que contengan, por lo que la reutilización era mínima.

Una vez se conseguía llegar a este punto, nos enfrentábamos al problema de la escalabilidad y disponibilidad de estas aplicaciones.

Dado que esta configuración manual es lenta, las aplicaciones a menudo tenían problemas con los picos de demanda o el sobredimensionamiento, lo que supone grandes costes tanto por la incapacidad de procesar la demanda, como por el elevado coste del sobredimensionamiento.

Hay que tener en cuenta que no se incluyen los costes asociados al propio centro de datos donde se ubica cada uno de estos servidores, por lo que los costes globales son aún mayores.

Para solventar estas carencias y necesidades surge la virtualización y la infraestructura como código en un entorno habitualmente de tipo cloud.

Veamos con más detalle algunos de estos conceptos preparando un entorno completo, basado en el sistema operativo linux y las herramientas básicas de administración asociadas.

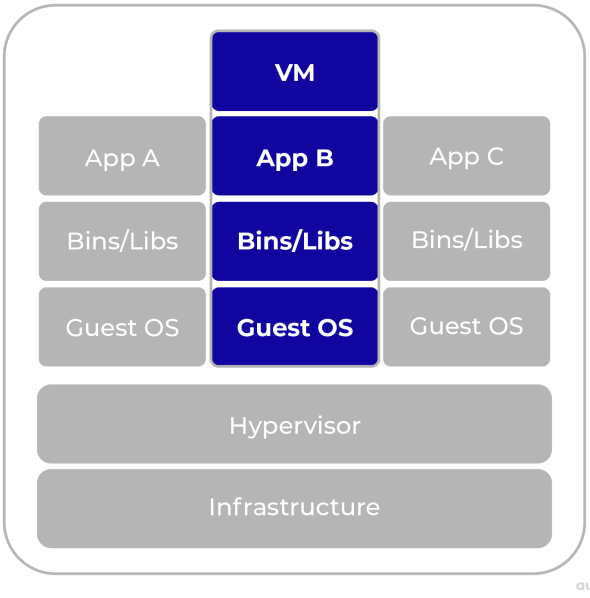
Virtualización de sistemas

El término **virtualización** se refiere a la **simulación software de un recurso tecnológico** (hardware, almacenamiento, dispositivos de red, etc.).

El **hypervisor** es la pieza que provee una capa de abstracción entre la máquina física y la/s virtualizada/s. Esta pieza ayuda a "engañar" al sistema operativo instalado en la máquina virtual (MV) haciéndole pensar que está instalado en una máquina física. El hipervisor también ayuda a gestionar los recursos físicos compartidos entre las distintas máquinas virtuales.

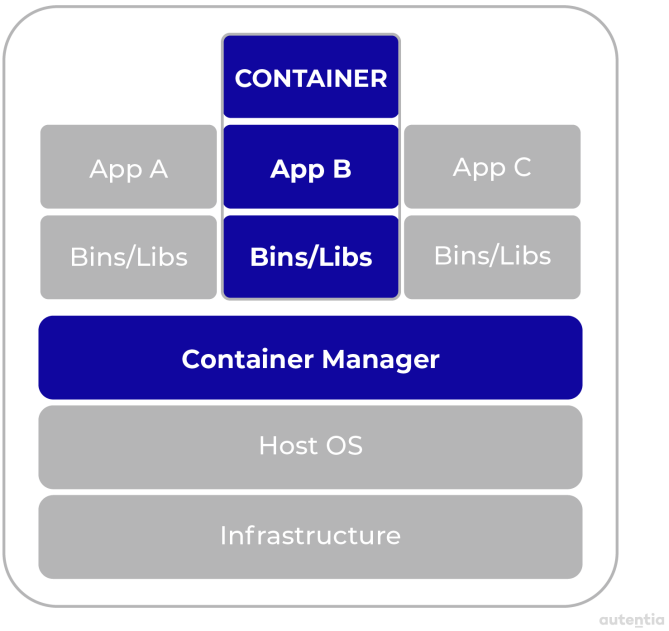
En base a la presencia de hipervisor o no, podemos diferenciar dos tipos de virtualización:

- **Virtualización pesada**: tiene hypervisor. Es la que se realiza en máquinas virtuales. La virtualización es la asignación de recursos que no se comparten: si tienes 8GB de RAM y una MV usa 4GB, sólo se dispondrán de 4 GB para las demás máquinas.



- **Virtualización ligera**: se hace mediante **contenedores**. Los contenedores utilizan el **kernel** de un sistema operativo Linux que se comparte entre todos ellos. La forma de aislamiento de los contenedores se hace mediante namespaces (limitando la visibilidad

de recursos y procesos del contenedor) y cgroups (limitando recursos). La diferencia con las máquinas virtuales es que **se comparte este kernel**, mientras que las máquinas virtuales tienen todas su kernel individual. Otra diferencia es que los contenedores no son sistemas operativos completos. **Sólo llevan aquellos programas imprescindibles para hacer su cometido**. De ahí que se llamen **ligeros**, ya que **ocupan muchísimo menos** que una máquina virtual tradicional.



En los dos tipos de virtualización aparecen los conceptos de host y guest.

Host: mientras que en la virtualización pesada (de MV) identificamos host como el hipervisor, la virtualización ligera (de contenedores) es el servidor donde se encuentra instalado el gestor de contenedores.

Guest: en el caso de la virtualización pesada estaríamos hablando de los SSOO virtualizados en cada MV. Mientras que si hablamos de contenedores, nos referimos al contenedor propiamente dicho.

Virtualización

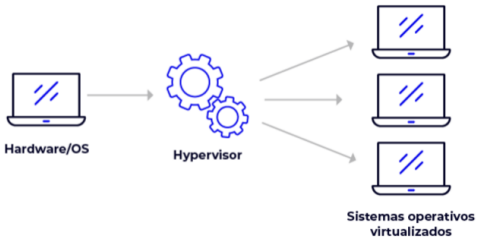
autentia

¿Qué es?

La virtualización comenzó a adoptarse a principios del año 2000 y se define como una simulación software de un recurso tecnológico (hardware, almacenamiento, dispositivos de red, etc) que distribuye sus funcionalidades entre diversos usuarios o entornos permitiendo utilizar toda la capacidad de una máquina física.

HYPERVISOR

El hypervisor es la pieza que provee una capa de abstracción entre la máquina física y la/s virtualizada/s. Esta pieza ayuda a "engañar" al sistema operativo instalado en la máquina virtual haciéndole pensar que está instalado en una máquina física. También ayuda a gestionar los recursos físicos compartidos entre las distintas máquinas virtuales.



TIPOS DE VIRTUALIZACIÓN

En función de si se usa hypervisor o no, podemos encontrar dos tipos de virtualización:

- **Virtualización pesada:** tiene hypervisor. Es la que se realiza en máquinas virtuales. La virtualización es la asignación de recursos que no se comparten: si tienes 8GB de RAM y una MV usa 4GB, sólo se dispondrán de 4 GB para las demás máquinas.
- **Virtualización ligera:** se hace a través de contenedores. Los contenedores utilizan el kernel de un sistema operativo linux que se comparte entre todos ellos.

¿Diferencias? Los contenedores comparten el kernel, mientras que las máquinas virtuales tienen su propio Kernel individual. Además, los contenedores no son sistemas operativos completos, sólo llevan aquellos programas imprescindibles para hacer su cometido. De ahí que se llamen ligeros, ya que ocupan menos espacio que una máquina virtual tradicional.

Diferentes distribuciones

Una distribución o “distro” de Linux no es más que una **versión personalizada** del sistema operativo original, el kernel o núcleo de Linux.

El kernel es el corazón de cualquier sistema operativo y funciona como “mediador” entre las solicitudes de los programas y el hardware. Las distribuciones son un conjunto de programas (clientes de correo, ofimática, etc.) que realizan solicitudes al hardware a través del kernel. Las distribuciones suelen tener un conjunto de programas o comandos para resolver un problema. Un ejemplo de esto es el gestor de paquetes:

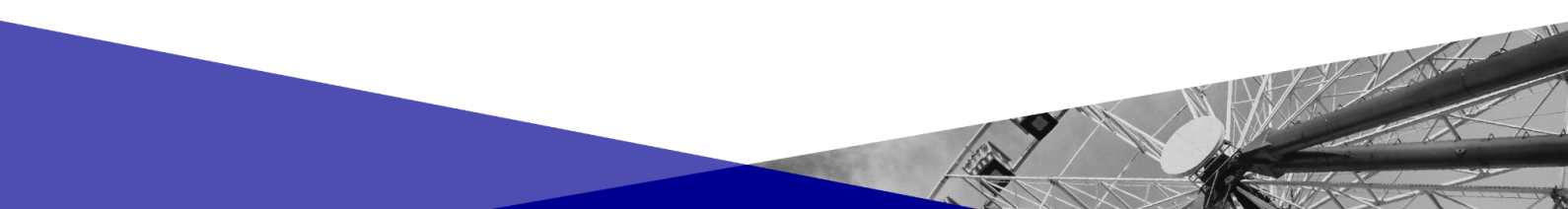
- Para sistemas **Red Hat** es `yum`.
- Para sistemas **Debian** es `apt-get`.

En este documento nos centraremos en una distribución pero puede que a la hora de trabajar en un cliente no sea la misma y las rutas o los comandos no sean iguales (aunque nosotros vamos a explicar comandos que son comunes). Una forma de encontrar una lista de comandos resumida es buscar por “**Cheat Sheet**” + nombre de la distribución en google.

La mayoría de las distribuciones de uso comercial se dividen en:

- Basadas en **Debian**: (Todas son licencias GPL):
  - **Kali Linux**: se utiliza sobre todo para auditorías de seguridad y pruebas de penetración (Pentesting).
  - **Ubuntu**: popularizó Linux en todo el mundo. Es una distribución de Es la beta de su distro principal, Debian.
  - **Debian y Ubuntu Server**.
- Basadas en **Red Hat**:
  - **Fedora**: El Ubuntu de Redhat.
  - **CentOS** (Licencia GPL).
  - **Oracle Linux** (Licencia GPL).
  - **AIX** (Licencia de Pago de IBM adquirida por RedHat).

La licencia **GPL** permite que los usuarios finales (personas, organizaciones, etc.) tengan libertad de usar, estudiar, compartir (copiar) y modificar el software.



Distribución Linux

autentia

¿Qué es?

Coloquialmente conocida como "distro de Linux" es una versión personalizada del sistema operativo original, el kernel o núcleo de Linux y suelen ser versiones compuestas de software libre. La licencia GPL permite que los usuarios finales (personas, organizaciones, etc.) tengan libertad de usar, estudiar, compartir (copiar) y modificar el software.

TIPOS

En general se pueden englobar en tres tipos de distribuciones:

- **de escritorio/domésticas:** son las que puede usar cualquier tipo de usuario pero están enfocadas en aplicaciones de uso común como el correo, navegador web, editor de texto, lectura de ficheros, etc. Ubuntu es una de las más conocidas hoy en dia.
- **Servidores:** se enfocan en equipos de tipo servidor que necesitan dar un servicio alto durante todo el día. Estas distribuciones no suelen tener interfaz gráfica.
- **Empresariales:** se enfocan en servicios más concretos y personalizados. Suelen tener un servicio de soporte.

DEBIAN Y RED HAT

La mayoría de las distribuciones de uso comercial se basan en:

Debian (Todas son licencias GPL):

- Kali Linux: se utiliza sobretodo para auditorías de seguridad y pruebas de penetración (Pentesting).
- Ubuntu: popularizó Linux en todo el mundo. Es la beta de su distro principal, Debian.
- Debian y Ubuntu Server.

Red Hat:

- Fedora: El Ubuntu de Redhat.
- CentOS (Licencia GPL).
- Oracle Linux (Licencia GPL).
- AIX (Licencia de Pago de IBM adquirida por RedHat).

Montando el entorno

En este apartado indicaremos todos los pasos a seguir para montar el entorno local con una distribución linux donde poder ejecutar todos los comandos que aquí se explican.

El software necesario para montar el entorno en macOS es el siguiente:

- Virtualbox: es el SW de virtualización de Oracle para arquitecturas x86/amd64.
- Vagrant: Vagrant es una herramienta que nos permite crear escenarios virtuales de una forma muy simple y replicable. Existen máquinas ya creadas por la comunidad llamadas Boxes. Estas Boxes las podemos encontrar en Vagrant Cloud.
- Homebrew: gestor de paquetes para Mac OS X. Para la instalación de homebrew solo hay que ejecutar el comando que viene en su página de inicio.
- Git: para instalar Git (habiendo instalado previamente Homebrew) solo tenemos que ejecutar este comando: `brew install git`.

Para el entorno de pruebas vamos a descargarnos una distribución de Ubuntu Server. En este caso, utilizaremos una distribución 22.04 LTS que tiene un soporte prolongado en el tiempo (Long Term Support). Para hacer la instalación del entorno hay que seguir lo siguientes pasos:

- Descargamos los dmgs tanto de Virtualbox como de Vagrant. Tienen instaladores al uso sin ninguna configuración adicional.
- También podemos Instalar vagrant con brew: brew install vagrant
- Abrimos un terminal en la ruta donde incluir el fichero de Vagrant (VagrantFile), en nuestro caso /Users/${USER}/Documents/OnBoarding: `vi Vagrantfile`

  Incorporamos el siguiente contenido para generar nuestro servidor linux de pruebas.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# All Vagrant configuration is done below. The "2" in Vagrant.configure
# configures the configuration version (we support older styles for
# backwards compatibility). Please don't change it unless you know what
# you're doing.
Vagrant.configure("2") do |config|
  # The most common configuration options are documented and commented
below.
  # For a complete reference, please see the online documentation at
  # https://docs.vagrantup.com.
  # Every Vagrant development environment requires a box. You can search
for
  # boxes at https://vagrantcloud.com/search.
  config.vm.box = "bento/ubuntu-22.04"

  # Create a forwarded port mapping which allows access to a specific
port
  # within the machine from a port on the host machine and only allow
access
  # via 127.0.0.1 to disable public access
  config.vm.network "forwarded_port", guest: 80, host: 8080, host_ip:
"127.0.0.1"

  # Create a private network, which allows host-only access to the
machine
```

```
  # using a specific IP.
  config.vm.network "private_network", ip: "192.168.56.10"


  # Provider-specific configuration so you can fine-tune various
  # backing providers for Vagrant. These expose provider-specific
options.
  config.vm.provider "virtualbox" do |vb|
  #   # Display the VirtualBox GUI when booting the machine
      vb.name = "Onboarding Environment"
      vb.gui = false
      vb.cpus = 1
  #   # Customize the amount of memory on the VM:
      vb.memory = "1024"

      unless File.exist?('./secondDisk.vdi')
        vb.customize ['createhd', '--filename', './secondDisk.vdi',
'--variant', 'Fixed', '--size', 10*1024]
      end
      vb.customize ['storageattach', :id, '--storagectl', 'SATA
Controller', '--port', 1, '--device', 0, '--type', 'hdd', '--medium',
'./secondDisk.vdi']

      unless File.exist?('./thirdDisk.vdi')
        vb.customize ['createhd', '--filename', './thirdDisk.vdi',
'--variant', 'Fixed', '--size', 10 * 1024]
      end
      vb.customize ['storageattach', :id,  '--storagectl', 'SATA
Controller', '--port', 2, '--device', 0, '--type', 'hdd', '--medium',
'./thirdDisk.vdi']
  #
  end
end
```

- Si quisiéramos crear un Vagrantfile desde cero solo tendríamos que ejecutar el comando `vagrant init bento/ubuntu-22.04`. Para poder lanzar el aprovisionamiento del servidor, ejecutamos `vagrant up`. Este comando construirá y aprovisionará la máquina virtual en Virtualbox que utilizaremos como entorno.

- En las últimas versiones de osx, podemos tener problemas a la hora de que se creen los interfaces de red, tenemos que darle permiso si no lo hemos hecho ya. Para ello nos vamos a Preferencias del sistema

> Seguridad y privacidad > General. Cuando le demos permiso se nos pedirá reiniciar.

- Una vez creada, podemos entrar por ssh con el comando `vagrant ssh`.

- Una vez dentro de la máquina, podríamos instalar un apache. Antes de realizar la instalación, cambiamos de usuario Vagrant a root con el comando `sudo su - root`.

- Ya estamos en disposición para instalar apache2 con el siguiente comando `sudo apt-get update && apt-get install -y apache2`.

- Vagrant nos simplifica el aprovisionamiento del servidor. A través del fichero Vagrantfile, indicamos las características de la máquina sin necesidad de hacerlo de forma manual a través de VirtualBox. Aquí podemos configurar cosas como la cantidad de RAM, número de núcleos, nombre, etc. Lo hemos dejado así:

```
 config.vm.provider "virtualbox" do |vb|
#   # Display the VirtualBox GUI when booting the machine
   vb.name = "Onboarding Environment"
   vb.gui = false
   vb.cpus = 1
# Customize the amount of memory on the VM:
   vb.memory = "1024"
 end
```

- Para cargar las nuevas propiedades no es necesario volver, borrar y volver a crear la máquina virtual. Podemos ejecutar `vagrant reload`. También podríamos parar la máquina ejecutando `vagrant halt` y volverla a arrancar haciendo `vagrant up`. Pero con Vagrant reload conseguimos los mismos efectos utilizando un solo comando.

Si no hubiéramos configurado estas propiedades, en Virtualbox veríamos algo así:

General

Nombre:	03_Administracion_Unix_default_1588748043313_94245
Sistema operativo:	Ubuntu (64-bit)

Sistema

Memoria base:	4096 MB
Procesadores:	2
Orden de arranque:	Disco duro, Óptica
Aceleración:	VT-x/AMD-V, Paginación anidada, PAE/NX, Paravirtualización KVM

Ahora en cambio si abrimos VirtualBox tenemos algo como:

General

Nombre:	Onboarding Environment
Sistema operativo:	Ubuntu (64-bit)

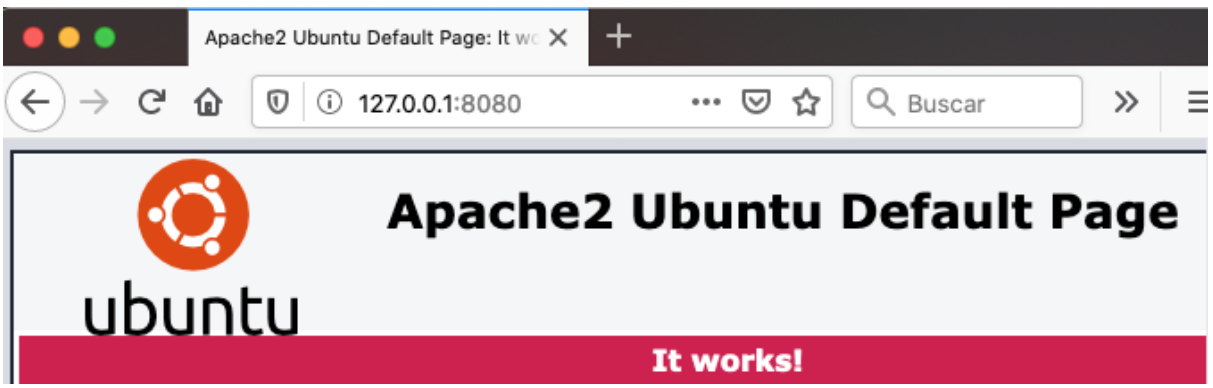
Sistema

Memoria base:	1024 MB
Orden de arranque:	Disco duro, Óptica
Aceleración:	VT-x/AMD-V, Paginación anidada, PAE/NX, Paravirtualización KVM

Ahora que tenemos instalado un apache, si queremos acceder a él desde fuera deberíamos realizar **forwarded port** que veremos más adelante y que sirve para mapear el puerto 80 (guest) al puerto 8080 (host). A través del fichero Vagrantfile podemos indicar la configuración de los puertos y el mapeo de los mismos.

```
 config.vm.network "forwarded_port", guest: 80, host: 8080, host_ip:
"127.0.0.1"
```

Vemos que si en un navegador tratamos de acceder a 127.0.0.1:8080 tenemos acceso a la página de instalación del apache de nuestra máquina virtual.



Gestión de usuarios, permisos y accesos

Al igual que en muchos sistemas operativos como Mac OS X y Windows, los sistemas Unix también permiten la creación de cuentas de usuario. Cada usuario entonces podrá tener sus carpetas a las que solo él puede acceder, una home para él, una configuración personalizada, etc.

Usuarios

Cuando se ha creado el servidor con Vagrant, al generarse el servidor se crean una serie de usuarios del sistema por defecto. Si estamos siguiendo este documento y hemos levantado la máquina virtual con Vagrant, el usuario que se crea por defecto es "vagrant". Para acceder al servidor podemos utilizar `vagrant ssh`.

Estando dentro del servidor podemos ejecutar el comando `cat /etc/passwd` y lo que vamos a ver por la pantalla es información de todos los usuarios. En nuestro caso, algo parecido a lo siguiente:

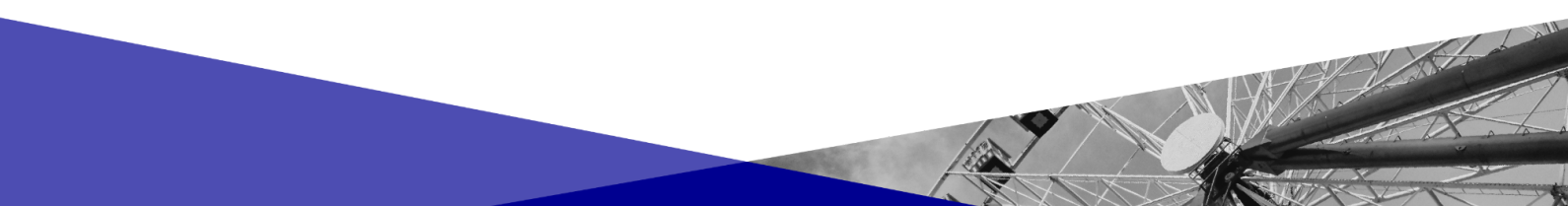
```
syslog:x:102:106::/home/syslog:/usr/sbin/nologin
messagebus:x:103:107::/nonexistent:/usr/sbin/nologin
_apt:x:104:65534::/nonexistent:/usr/sbin/nologin
uuidd:x:105:109::/run/uuidd:/usr/sbin/nologin
sshd:x:106:65534::/run/sshd:/usr/sbin/nologin
vagrant:x:1000:1000:vagrant,,,:/home/vagrant:/bin/bash
```

El formato de la salida es el siguiente:

[username]:[x]:[UID]:[GID]:[Comment]:[Home directory]:[Default shell]

O sea que esto nos quiere decir que:

1. Username indica que es el **usuario** vagrant.
2. La x significa que **nuestra password está oculta**. Podemos verlo en /etc/shadow. Lo vamos a ver en detalle más tarde.
3. El UID es el User Identification o sea, un **identificador del usuario.**
4. El GID es el Group Identification o sea, un **identificador del grupo**.



5. En Comment, en este caso, tenemos **el nombre del usuario**, que en el caso que se instale de forma manual se pregunta durante la instalación.

6. Después tenemos el directorio de **nuestra home**. El home del usuario es el directorio en el que estamos situados cuando accedemos al servidor. Si tras entrar escribimos el comando `pwd` nos va a decir que estamos en /home/vagrant.

7. Por último tenemos el tipo de shell que va a tener el usuario cuando se autentifique en el sistema. Una **shell** no es más que un intérprete de comandos para que escribamos los mismos, se ejecuten y el sistema operativo nos comunique el resultado.

Por seguir uniendo piezas, vamos a ver exactamente qué es lo que hay en /etc/shadow. Para ello ejecutamos el comando `sudo cat /etc/shadow`. Utilizamos sudo porque si ejecutamos el mismo comando sin sudo, nos va a decir que **nuestro usuario no tiene permisos para ver ese fichero**. Simplemente, tenemos que introducir nuestra password.

Ya veremos más adelante qué significa esto. El usuario que hemos creado de primeras, tiene permisos para ejecutar comandos sudo, que por ahora nos interesa saber, que es para ejecutar comandos como si fuéramos el usuario root. El usuario root es el administrador, que tiene privilegios de acceso ilimitados, por lo que **no conviene utilizar ese usuario salvo que sea imprescindible**.

El formato del fichero /etc/shadow sería así:

[username]:[password]:[lastchanged]:[minimum]:[maximum]:[warn]:[inactive]:[expire]:

```
systemd-coredump:!!:18387::::::
abarranco:$6$LM6wS26/x.QzWqB5$lg3iKCijc5WTsoNEmONcd58fbJeYSjlaYy4diqNT2AlbWi.XRCirNohpJ176azAAXtniXH
IPR2GI.W7GPgDEO/:18387:0:99999:7:::
lxd:!:18387::::::
```

Si observamos el usuario abarranco, podemos ver lo siguiente:

1. El identificador abarranco aparece en el primer campo.

2. El segundo campo corresponde a la password, aparece con un alfanumérico, concretamente **es un hash de la password real**. Los 3 primeros caracteres indican cual es el algoritmo usado para ofuscar la password. Tenemos las siguientes opciones:

    a. 1 (MD5)

   b. $2a$ (Blowfish).
   c. $2y$ (Blowfish).
   d. 5 (SHA-256).
   e. **6** (SHA-512). En nuestro caso.

3. El tercer campo es el **último día en que se cambió la password** contando desde el 1 de Enero de 1970. En nuestro caso, hoy es 5 de mayo del 2020 y han pasado 18387 días desde esa fecha, lo que quiere decir, que desde hoy no cambio mi password. Esto es así porque hoy he creado el usuario. Si mañana cambio la password, habría que sumar 1 a este número.
4. El cuarto campo indica el **número mínimo de días que tienen que pasar hasta que pueda volver a cambiar mi password**. El 0 indica que se puede cambiar en cualquier momento.
5. El quinto campo es **el número máximo de días en los que la password es válida** antes de que obliguen a ese usuario a cambiarla. En este caso, 99999, indica que no tiene obligación de cambiarla.
6. El sexto campo muestra el **número de días de preaviso para cambiar su password**, en este caso una semana.
7. El séptimo campo es **el número de días que una cuenta está deshabilitada** cuando le caduca la password. En nuestro caso no viene nada, así que no aplica.
8. El octavo campo son **los días que la cuenta lleva deshabilitada**, de nuevo contados desde el 1 de Enero de 1970. Una fecha absoluta indica que el login no puede volver a ser usado.

A continuación veremos algunas operaciones que podemos hacer para gestionar usuarios como, su creación, cómo cambiar de usuario, borrado y bloqueo de usuarios

Para crear un usuario tenemos que seguir los siguientes pasos:

- Creamos un usuario con el comando `sudo useradd -m removeMe`. El parámetro -m es para que cree la home.
- Establecemos su contraseña con `sudo passwd removeMe`.

Con fines ilustrativos, repetiremos estos dos pasos creando un usuario denominado ***abarranco*** con password ***verysecret*** que utilizaremos posteriormente. Es decir:

```
sudo useradd -m abarranco
sudo passwd abarranco
usermod -aG sudo abarranco
```

En la última sentencia se le ha incluido en el grupo de administradores "*sudoers*" puesto que realizaremos acciones privilegiadas en las siguientes secciones.

Para cambiar del usuario actual al nuevo usuario creado, utilizaremos el comando `su - removeMe`. Vemos que el terminal ahora es distinto. Solo nos sale el símbolo "$" para escribir los comandos. Si ejecutamos el comando `cat /etc/passwd`, vamos a ver que el shell que tiene por defecto es **/bin/sh**. Por eso es un poco distinto este terminal comparado con el de root o el usuario vagrant.

Si no nos gusta la shell bin/sh, podemos cambiarla. Para poder realizar el cambio de shell de un usuario, necesitamos que nuestro usuario tenga permisos de sudo o pertenecer al grupo de sudo. Para añadir a nuestro usuario al grupo de sudo tenemos que ejecutar los siguientes comandos:

1. Estando logueados con el usuario removeMe debemos **hacer un logout** ejecutando el comando `exit` y **volver a la sesión del usuario vagrant** para poder concederle permisos de sudo al usuario removeMe.
2. Para agregar al usuario removeMe al grupo sudoer ejecutamos el siguiente comando: `sudo usermod -a -G sudo removeMe`
3. Ahora nos podemos logar con removeMe ejecutando el comando `su - removeMe`

Para modificar el shell de un usuario necesitamos permisos de root. Podemos logearnos como root con `sudo su - root` o ejecutarlos con el usuario removeMe ya que tiene permisos de sudo con los siguiente comandos:

```
sudo usermod --shell /bin/bash removeMe
sudo grep removeMe /etc/passwd
```

Ahora podemos comprobar que si nos cambiamos a ese usuario (`su - removeMe`), el terminal es como el que teníamos antes. En vez de salirnos solo “$”, nos aparece “removeMe@localhost:~$”.

Para bloquear la cuenta de un usuario tenemos diferentes formas:

- Con el comando `sudo usermod -L removeMe` o `passwd -l removeMe`. Este comando añade la marca ! en la segunda columna del usuario (donde estaba la x que hace referencia al password `cat /etc/shadow`). Esta forma no bloquea el login por clave pública que veremos más adelante.
- Con el comando `sudo usermod -s /sbin/nologin removeMe` o `usermod -s /bin/false removeMe` cambiamos la shell en el /etc/passwd por /bin/false o /sbin/nologin impide que el usuario pueda autenticarse.
- Con el comando `sudo chage -E0 testuser` hacemos que la cuenta expire. Además cambia la shell a nologin.

Para borrar un usuario utilizaremos el comando `sudo userdel -r removeMe`. La opción -r es para borrar todos los datos del mismo (carpeta home, mails, etc.).

Home del usuario

La home es el directorio por defecto que se le asigna a los usuarios. Existen un par de trucos que conviene saber:

1. El comando `cd` **nos va a llevar directamente a la home**.
2. El directorio home suele estar en **/home/{NOMBRE_DE_USUARIO}**. El nombre de usuario lo podemos ver a la izquierda, en nuestra shell, ya que normalmente, esta parte izquierda está compuesta de {NOMBRE_DE_USUARIO}@{NOMBRE_DE_SERVIDOR}:

Si utilizamos cualquiera de estos trucos y nos vamos a nuestra home podemos ejecutar el comando `ls -la` para ver qué tenemos aquí.

```
abarranco@server1:~$ ls -la
total 24
drwxr-xr-x 3 abarranco abarranco 4096 may  5 16:23 .
drwxr-xr-x 3 root      root      4096 may  5 16:10 ..
-rw-r--r-- 1 abarranco abarranco  220 feb 25 12:03 .bash_logout
-rw-r--r-- 1 abarranco abarranco 3771 feb 25 12:03 .bashrc
drwx------ 2 abarranco abarranco 4096 may  5 16:13 .cache
-rw-r--r-- 1 abarranco abarranco  807 feb 25 12:03 .profile
-rw-r--r-- 1 abarranco abarranco    0 may  5 16:23 .sudo_as_admin_successful
abarranco@server1:~$ _
```

Vemos que este comando nos dice el total del espacio en disco que ocupa este directorio. Puede ser el total de bloques pero en este caso, es el total en Kilobytes que ocupa el directorio. De hecho, si cambiamos un poco el comando y al programa `ls` le pasamos el parámetro h ejecutando `ls -lha`, nos va a presentar el resultado en formato legible por los humanos, es decir, con unidades de medida, pudiendo ver fácilmente cuánto ocupa cada fichero.

Vemos que la salida después tiene información en columnas, donde de izquierda a derecha tenemos:

- Una serie de letras y guiones que son los **permisos del fichero**, que veremos más adelante.
- El **número de links**. Algunas plataformas tienen nociones distintas de lo que es un link. Podríamos decir que un link es un equivalente a un acceso directo en Windows.
- El **propietario del fichero**.
- El **grupo del fichero**. Si un usuario está en un grupo va a tener los

permisos sobre el fichero que correspondan al grupo. Lo veremos más adelante.

- El **tamaño** del fichero.
- La **fecha** de última modificación del fichero.
- El **nombre** del fichero.

.bash_profile

Este fichero se encuentra en la home de los usuarios. Se carga cuando un usuario entra en un servidor. Este fichero nos permite configurar nuestro entorno cada vez que se inicie sesión (o se cargue el fichero manualmente con `source ~/.bash_profile`), estableciendo valores en variables de entorno, declarando alias como atajos para comandos largos, realizar tareas al iniciar sesion como el montaje de discos, etc.

Vamos a crearnos nuestro propio archivo de **.bash_profile** para ver un ejemplo. Ejecutamos el comando `vi .bash_profile` para crear el fichero, que por ahora no existe.

Una vez estamos dentro del programa vi, pulsamos la tecla *i* para poder insertar texto.

Escribimos lo siguiente:

```
alias prueba="echo Esto es una prueba"
```

Para escribir esto en el fichero de forma permanente y salir de él tenemos que dar a la *tecla escape* y luego escribir `:wq`.

Con esto hemos creado un alias, de tal forma que cada vez que en la terminal escribamos `prueba`, nos va a salir por pantalla el mensaje “Esto es una prueba”. Compruébalo.

Te habrás dado cuenta que el terminal te dice algo como “Command not found”. Esto es porque ya estamos logados en nuestra cuenta en este servidor y el archivo .bash_profile no se recarga cuando cambia su contenido, sino cuando nos logamos de nuevo. También podemos recargar el .bash_profile ejecutando `source ~/.bash_profile` (la virgulilla ~ hace referencia a la ruta de tu home, en nuestro caso /home/abarranco y se obtiene pulsando ALT + Ñ).

Para logarnos de nuevo en el servidor podemos ejecutar el comando `exit` y escribir nuestro usuario de nuevo y nuestra password. Si ahora escribimos `prueba` en el terminal nos saldrá el mensaje que escribimos antes por pantalla.

Conexión por SSH

Se denomina SSH tanto al protocolo como al programa para acceder de forma remota a un servidor. En este caso se accede por el puerto estándar designado, que es el 22.

Vamos a probar a acceder con nuestro usuario a nuestra máquina virtual. Recordad que previamente se ha provisionado el servidor a través de Vagrant y que disponemos de un usuario (*abarranco*) que se creó previamente.

Necesitaremos conocer la dirección IP de nuestro servidor. Podemos saberlo abriendo una consola con Vagrant y consultando al servidor. Es decir, se ejecutan los comandos:

```
vagrant ssh
ip addr show
```

```
vagrant@vagrant:~$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:64:75:a1 brd ff:ff:ff:ff:ff:ff
    altname enp0s3
    inet 10.0.2.15/24 metric 100 brd 10.0.2.255 scope global dynamic eth0
       valid_lft 86275sec preferred_lft 86275sec
    inet6 fe80::a00:27ff:fe64:75a1/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:e6:cd:9f brd ff:ff:ff:ff:ff:ff
    altname enp0s8
    inet 192.168.56.10/24 brd 192.168.56.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fee6:cd9f/64 scope link
       valid_lft forever preferred_lft forever
```

Una vez conocemos la ip, podemos abrir un nuevo terminal y utilizar el cliente ssh para conectarnos directamente sin utilizar Vagrant utilizando el comando `ssh {NOMBRE_USUARIO}@{IP_SERVIDOR}`, lo que en nuestro caso corresponde a `ssh abarranco@192.168.56.10`.

Cuando lo hagamos, si es la primera vez que tratamos de conectarnos a ese servidor, nos va a preguntar si realmente decidimos confiar en él. Si contestamos que sí, se genera una entrada en [HOME_USUARIO}/.ssh/**known_host**.

Observamos que la máquina remota nos pide la password del usuario con el que nos estamos intentando autenticar para comprobar que realmente somos ese usuario. Hay otra forma más cómoda de hacer esto y es con un par de claves, pública y privada.

Para generarlas, ejecutamos el comando `ssh-keygen -t rsa` dentro de nuestra máquina y vamos contestando a las preguntas. Si todo acaba de forma correcta, ahora tendremos dos archivos más en el directorio $HOME/.ssh/. Un archivo del nombre que hayamos puesto y otro igual pero con extensión .pub. Esta última es nuestra clave pública que podemos subir a cualquier sitio con el que queramos autenticarnos sin meter nuestra password para demostrar que somos nosotros.

En los servidores existe un fichero {HOME_USUARIO}/.ssh/**authorized_keys** donde se registran todas las claves de los usuarios autorizados. Podemos crear el fichero con vi y copiar el contenido con `cat id_rsa.pub >> authorized_keys` estando dentro de la carpeta .ssh del usuario que queremos logar sin meter su password.

Si ahora volvemos a probar a logarnos con ssh, no nos debería pedir la password.

Grupos

Un usuario puede pertenecer a uno o a varios grupos. Esto es bueno ya que de cara al mantenimiento, es más fácil dar permisos sobre un fichero a un grupo de usuarios que a muchos usuarios de forma individual. Los sistemas pueden tener decenas o incluso más de un centenar de usuarios. La agrupación de los mismos, facilita mucho la administración.

Podemos ver todos los grupos y los usuarios que pertenecen a los mismos, con el comando `vi /etc/group`. El formato de la salida es el siguiente:

[Group name]:[Group password]:[GID]:[Group members]

Aquí vemos, por ejemplo, el grupo sudo, donde la X en el password quiere decir que las password del grupo no se utilizan, el tercer campo (GID) es el identificador del grupo y por último, tenemos una lista de usuarios separados por coma de los usuarios que forman ese grupo.

Para ver los grupos de un usuario podemos utilizar el comando `groups {NOMBRE_DEL_USUARIO}` o directamente el comando `groups` si estamos logados con nuestro usuario y queremos ver nuestros grupos.

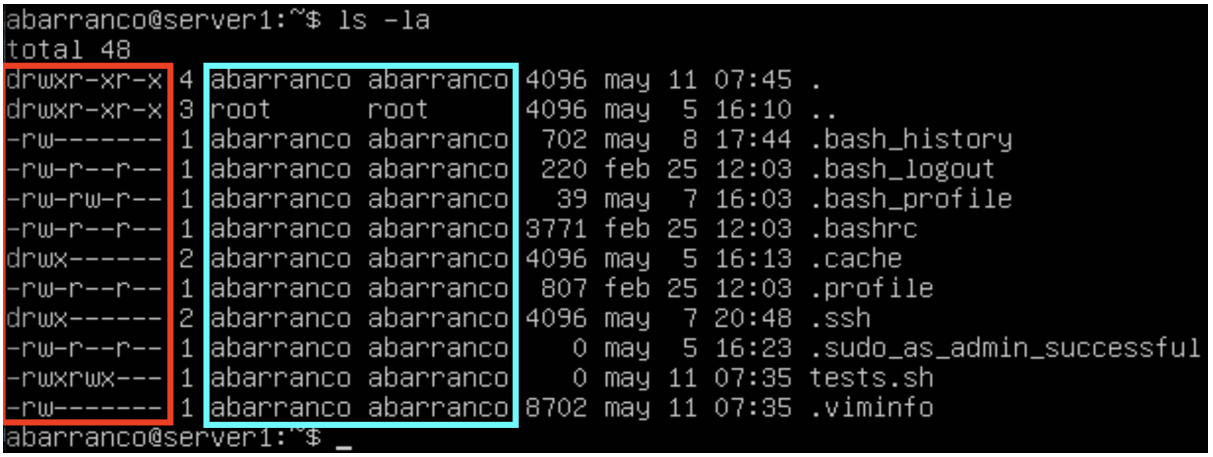
Para **añadir un usuario a otros grupos** podemos utilizar el comando `usermod --append --groups man,mail {NOMBRE_USUARIO}`. **Es necesario volver a logarse con el usuario para ver los nuevos grupos añadidos.**

Permisos

Los permisos son la forma que tiene el sistema operativo de indicar cómo y quién puede acceder a los ficheros o directorios que contiene el mismo.

Para ilustrar los permisos, podemos crear un fichero vacío llamado tests.sh y utilizarlo para realizar las diferentes pruebas. Para crearlo, en el directorio home podemos lanzar el comando `touch $HOME/tests.sh`.

Observemos, por ejemplo, los ficheros que tenemos en nuestra home y cuáles serían sus permisos.



Los permisos se encuentran a la izquierda del todo (encuadrados en rojo) y están estructurados de la siguiente forma:

- En primer lugar tenemos un símbolo que nos indica de qué tipo es esa entrada. Si tenemos **"-"** estaremos ante un **fichero** normal y corriente. En el caso de tener una **"d"** estamos ante un **directorio**. Y si tenemos una **"c"** estamos ante un **fichero con caracteres especiales**.

- Seguido del tipo, hay tres grupos de tres símbolos cada uno. **El primer grupo hace referencia a los permisos del usuario, el segundo a los permisos del grupo, y el tercero a los permisos del resto de**

usuarios. Los tres grupos funcionan de la misma manera:

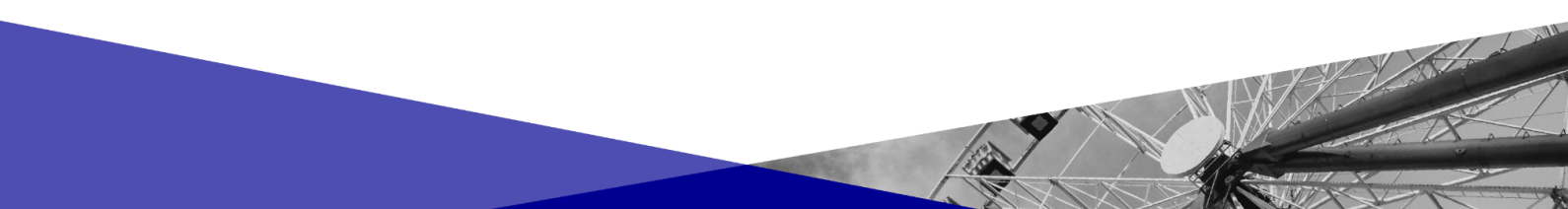
- El **primer carácter** indica los permisos de lectura.
  - Si es una **“r” tenemos permisos de lectura.**
  - Si es **“-” no tenemos permisos de lectura.**
- El **segundo carácter** indica los permisos de escritura.
  - Si es una **“w” tenemos permisos de escritura.**
  - Si es **“-” no tenemos permisos de escritura.**
- El **tercer carácter** indica los permisos de ejecución.
  - Si es una **“x” tenemos permisos de ejecución.**
  - Si es **“-” no tenemos permisos de ejecución.**

Encuadrado en azul arriba, tenemos a la izquierda el usuario propietario del fichero y a la derecha el grupo al que está asociado dicho fichero.

Según la imagen de arriba, si analizamos el fichero *.profile*, aparece “*-rw-r--r-- 1 abarranco abarranco 807*” indicando:

1. El fichero pertenece al usuario abarranco y al grupo abarranco.
2. Se trata de un fichero ya que en rojo podemos ver que el primer caracter es un “-”.
3. El usuario abarranco tiene permisos de lectura y escritura. El grupo abarranco, tienen permisos únicamente de lectura.
4. Los usuarios que no son abarranco ni están en el grupo abarranco, no tienen permisos de escritura ni de lectura.
5. El número 1 indica el número de enlaces a este fichero (hard links).
6. El número 807 indica la longitud del mismo en bytes.

En los sistemas unix, los permisos se almacenan en tres bloques de tres bits pero a la hora de visualizarlos se pueden presentar con letras o en formato decimal. No es más que una traducción del binario de los permisos de una entrada del disco, donde el primer bloque indica los permisos de usuario, el segundo bloque indica permisos del grupo y el tercero es para el resto de usuarios. El valor máximo de un bloque de permiso es 7 en decimal, 111 en binario y rwx utilizando letras, siendo el mínimo permiso 0 en decimal, 000 en binario y --- utilizando letras.



En la siguiente tabla se ilustran algunos ejemplos:

Permisos con caracteres	Permisos con números	Significado
----------	0000	No hay permisos.
-rwx------	0700	Lectura, escritura y ejecución solo para el usuario.
-rwxrwx---	0770	Lectura, escritura y ejecución solo para el usuario y el grupo.
---x--x--x	0111	Permisos de ejecución para todos.
--w--w--w-	0222	Permisos de escritura para todos.
--wx-wx-wx	0333	Escritura y ejecución para todos.
-r--r--r--	0444	Permisos de lectura para todos.
-r-xr-xr-x	0555	Permisos de lectura y ejecución para todos.
-rw-rw-rw-	0666	Permisos de escritura y lectura para todos.
-rwxr-----	0740	El usuario puede hacer todo, y el grupo solo lectura. El resto nada.

Normalmente, estos comandos se suelen utilizar solo con 3 dígitos (los 3 de la derecha) ya que cuando se omite el primero se asume un 0. El propósito de este primer dígito es para setear el Id de usuario, de grupo y de los atributos “sticky”, lo que queda fuera del ámbito de esta guía.

Vamos a ejecutar, por ejemplo, el comando `chmod 400 tests.sh` para ver qué permisos nos da. En este caso, modificamos los permisos utilizando el formato decimal para indicar los permisos de cada bloque (usuario, grupo y otros usuario). El primer bloque (valor 4) afecta a los permisos a nivel de usuario. El número 4 en binario corresponde a los bits 100, siendo “r--” si se

utilizan letras. Es decir, el usuario solo va a tener permisos de lectura. Los otros dos bloques al estar a cero, indican que no tienen permisos ni el grupo, ni el resto de usuarios.

Además, tenemos el comando chown, para cambiar el usuario y grupo propietario del fichero. El formato es **chown {OPTIONS} {USER}:{GROUP} {FILE(s)}**. Por ejemplo, de nuevo en la home podemos lanzar `chown abarranco:adm tests.sh`. Esto va a cambiar el grupo del fichero a adm. Esto se puede aplicar de forma recursiva a directorios enteros, podemos hacer la siguiente prueba `chown -R abarranco:adm /home/abarranco`. A menudo el -R se puede añadir a diversos comandos para hacerlo recursivo y que aplique a todos los directorios y ficheros dentro del path indicado. Funciona por ejemplo, también con `chmod -R 760 /home/abarranco`. **Puede que en algún momento y haciendo pruebas, veamos que no podemos hacer cd a nuestra home. Es el escenario perfecto para practicar estos comandos que estamos viendo hasta volver a poder acceder**.

Particiones en Linux

Las particiones son la forma que tienen los sistemas operativos de dividir una unidad de almacenamiento en unidades utilizables más pequeñas que se pueden tratar como si fueran una unidad de almacenamiento independiente y destinarla a propósitos diferentes. Particionar las unidades de almacenamiento nos aporta mayor flexibilidad en el futuro.

La **tabla de particiones** se utiliza para especificar al sistema operativo, el número y el tipo de particiones que tiene nuestro sistema. Además de indicar al sistema dónde se encuentra la partición de arranque.

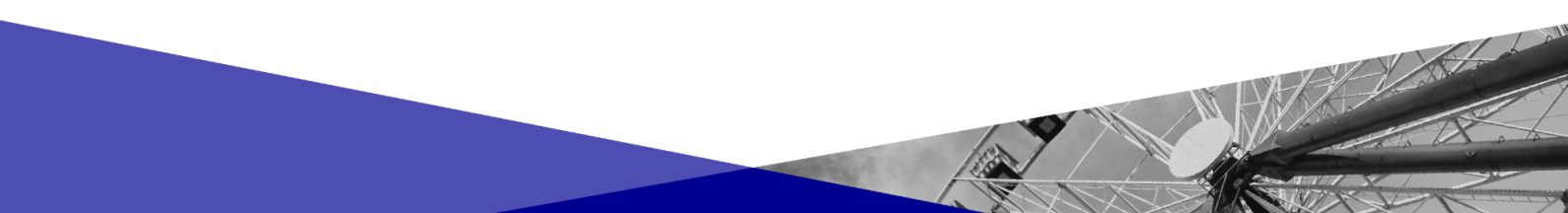
Existen distintos esquemas de particiones para la tabla de particiones de disco. Los más conocidos y difundidos son **MBR** (Master Boot Record) y **GPT** (GUID Partition Table).

- **MBR**: es el sistema de partición tradicional pero tiene algunas limitaciones:
  - No se puede usar para discos de más de 2 TB.
  - Solo puede tener un máximo de cuatro particiones primarias.
- **GPT**: es un esquema de partición más moderno que intenta resolver algunos de los problemas inherentes a MBR.
  - No tiene limitación del número de particiones y del tamaño del disco.
  - Tiene disponible la tabla de particiones en varias ubicaciones para evitar que se corrompa.

En la mayoría de los casos, **GPT es la mejor opción** a menos que tu sistema operativo o herramientas te impidan usarlo.

Para ver qué tipo de tabla de particiones tenemos en el entorno, podemos hacerlo de dos formas:

- Usando **fdisk**: con el comando `sudo fdisk -l`, la salida contendrá, entre otros, el disco `/dev/sda`



```
vagrant@vagrant:~$ sudo fdisk -l /dev/sda
Disk /dev/sda: 64 GiB, 68719476736 bytes, 134217728 sectors
Disk model: VBOX HARDDISK
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: EA49BFFA-1811-42C4-B9D3-08275C18710B

Device       Start       End   Sectors Size Type
/dev/sda1     2048      4095      2048   1M BIOS boot
/dev/sda2     4096   4198399   4194304   2G Linux filesystem
/dev/sda3  4198400 134215679 130017280  62G Linux filesystem
```

- Usando **GParted**: GParted es una herramienta de código abierto para gestionar particiones en entornos GNU/Linux.
    - Lo primero siempre es actualizar los repositorios de Ubuntu. Ejecutamos `sudo apt-get update`.
    - Los instalamos ejecutando `sudo apt install gparted -y`.
    - Ejecutamos *`sudo parted -l`*. En la salida podremos reconocer **msdos (es equivalente a MBR)** o **GPT**

```
vagrant@vagrant:~$ sudo parted -l
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sda: 68.7GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:

Number  Start   End     Size    File system  Name  Flags
 1      1049kB  2097kB  1049kB                     bios_grub
```

Para cambiar el tipo de tabla de particiones de MBR a GPT se puede utilizar el CD-Live de GParted y ejecutar GParted tal y como se indica en este hilo. También se podría incluir un script que lo cambiase dentro del Vagrantfile como se ve en este repositorio.

Tipos de particiones

Un disco duro puede ser dividido en particiones de los siguientes tipos:

- **Partición primaria**: son las particiones que **se pueden arrancar**. Solo puede haber una partición primaria activa. Cuando hay varios sistemas operativos instalados, la partición activa tiene un pequeño programa llamado gestor de arranque que presenta un pequeño menú que permite elegir qué sistema operativo se arranca.
- **Partición extendida o partición secundaria**: es la manera que tiene

Linux de ampliar el número de particiones en un solo disco. Solo puede haber una partición de este tipo por disco y **sirve para contener varias particiones lógicas**.

- **Partición lógica**: son el tipo de particiones no arrancables y **ocupan parte o la totalidad de una partición extendida**. Una partición lógica muy usada para facilitar el swapping (pasar páginas de memoria a disco) en el proceso de paginación (proceso de grabar en disco información de memoria no usada recientemente) es la extensión **SWAP** para ordenadores con poca memoria RAM. Normalmente el tamaño que se le establecía era el doble que la RAM disponible en el equipo. Esta partición no tiene sistema de ficheros.

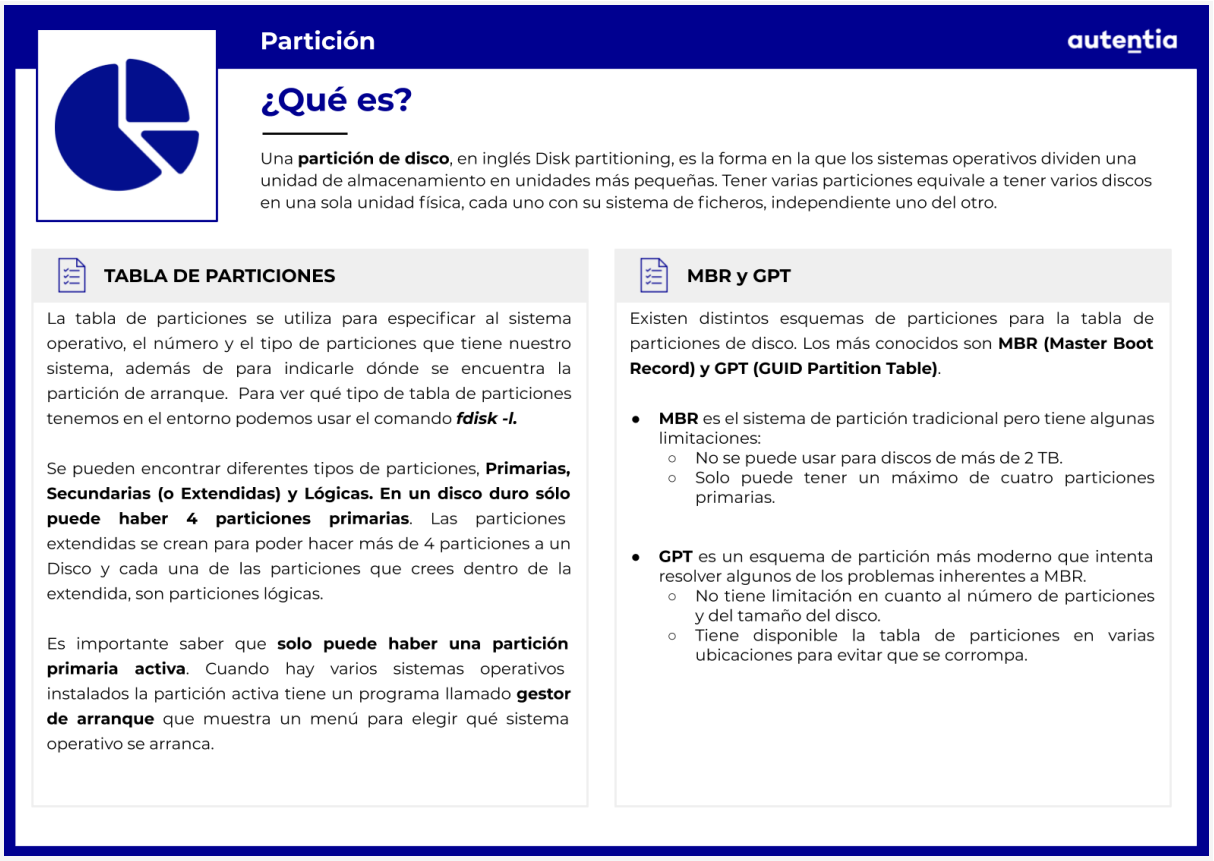
Para comprobar el tipo de particiones que tenemos, podemos recurrir a GParted con el siguiente comando:

```
sudo parted /dev/sda print
```

```
vagrant@vagrant:~$ sudo parted /dev/sda print
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sda: 68.7GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:

Number  Start   End     Size    File system  Name  Flags
 1      1049kB  2097kB  1049kB                     bios_grub
 2      2097kB  2150MB  2147MB  ext4
 3      2150MB  68.7GB  66.6GB
```

En nuestro caso, al usar GPT, no tenemos restricciones sobre el número de particiones primarias a utilizar, por tanto, todas son primarias. Por otro lado, en particiones con tabla MBR sí existe esta limitación, y el resultado del comando quedaría así:

```
vagrant@localhost:/usr/sbin$ sudo parted /dev/sda print
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sda: 53.7GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type     File system  Flags
 1      1049kB  53.7GB  53.7GB  primary  ext4         boot
```



Sistemas de Ficheros

Un **sistema de archivos** consta de los métodos y estructuras de datos que un sistema operativo utiliza para organizar los archivos de un disco o partición. Los tipos más comunes son:

- **Ext2** (Sistema de archivos Extendido, versión 2): es el primer sistema de archivos utilizado por GNU/Linux. Como todos los sistemas de archivos de Linux es asíncrono, es decir, no escribe inmediatamente los metadatos en el dispositivo de almacenamiento, sino que lo hace de manera periódica aprovechando los tiempos muertos de la CPU mejorando el rendimiento. También permite la recuperación de la información en caso de fallo (detectando particiones desmontadas erróneamente).

- **Ext3** (Sistema de archivos Extendido, versión 3): es compatible con Ext2. Añade la implementación de **Journaling**, un mecanismo que almacena las transacciones (operaciones de lectura y escritura de archivos) que se realizan en el sistema y que permitirá la recuperación de los datos en caso de fallo.

- **Ext4** (Sistema de archivos Extendido, versión 4): mantiene

retrocompatibilidad, posee “journaling”, reduce la fragmentación de archivos (mejorando con ello el rendimiento) y permite dispositivos de almacenamiento de más capacidad.

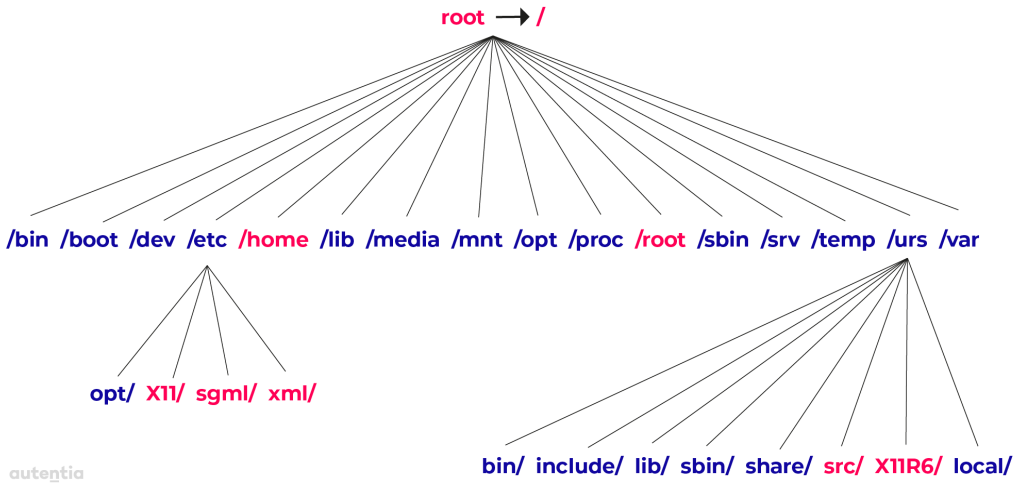
- **ReiserFS**: la versión más reciente de este sistema de archivos se denomina “Reiser4” y además de las características antes indicadas, posee mecanismos que le permiten trabajar con gran cantidad de archivos y una estructura de archivos optimizada.
- **XFS**: destaca por su alta escalabilidad y fiabilidad (admite redireccionamiento de 64 bits, implementación paralelizada y sobre todo, porque es capaz de trabajar con archivos muy grandes.
- **JFS**: posee un eficiente “journaling” que le permite trabajar cómodamente con archivos de gran tamaño como con otros más pequeños. Las particiones JFS pueden ser dinámicamente redimensionadas (como RaiserFS), pero no comprimidas (como RaiserFS y XFS).

Como hemos visto en anteriores pantallazos, el sistema de ficheros que utiliza nuestra partición es ext4.

Estructura de Directorios en Linux

Pero no solo el sistema de archivos de nuestro sistema operativo Linux es importante para conocer aspectos que afectan al rendimiento de nuestro equipo, también es importante que conozcamos la forma en la que está estructurado el sistema de directorios.

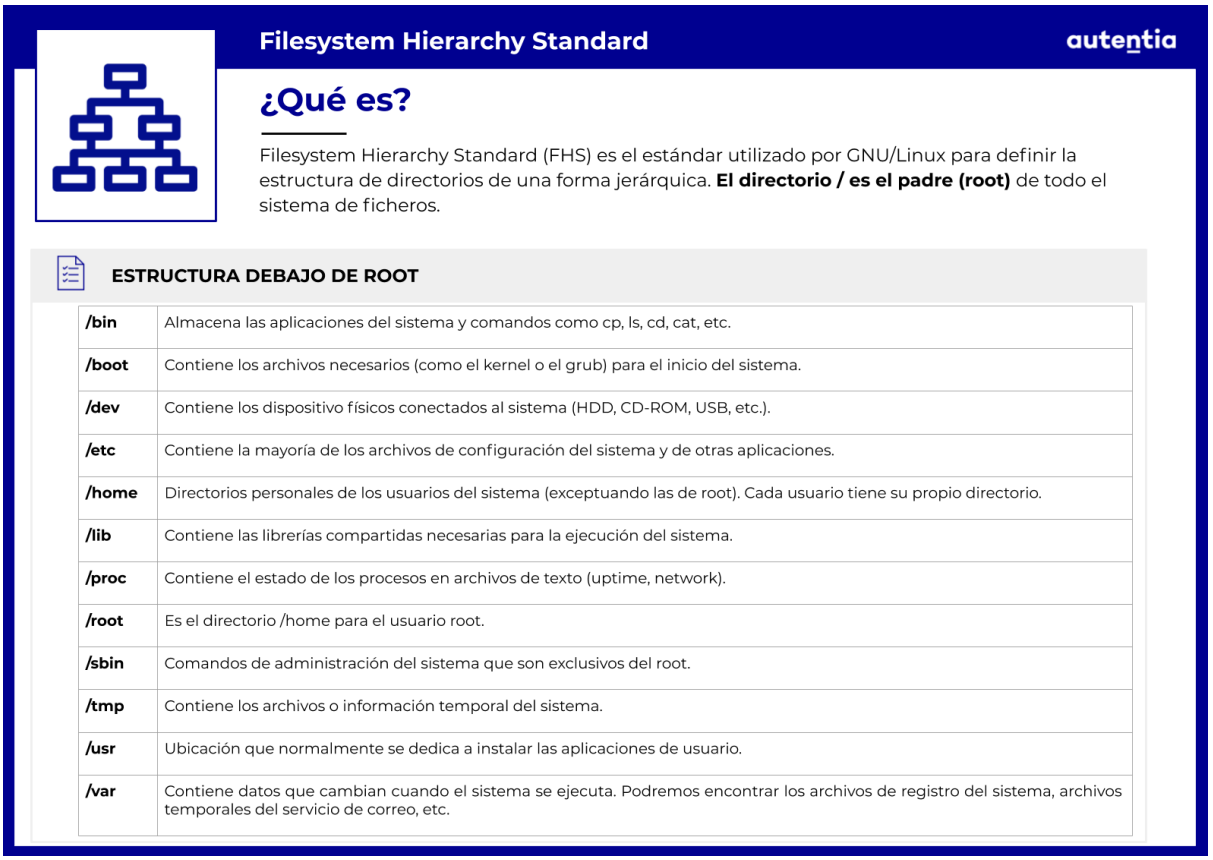
El estándar utilizado por GNU/Linux para organizar la información se denomina **FHS** (Filesystem Hierarchy Standard). Este sistema se encarga de organizar la información de forma jerárquica.



Partiendo de un “root”(/) encontraremos los siguientes directorios:

- **bin** Almacena las aplicaciones del sistema.
- **boot** Aquí se encontrarán los archivos necesarios (como el kernel o el grub) para el inicio del sistema.
- **dev** Cada uno de los archivos representa a un dispositivo del sistema (HDD, CD-ROM, USB, etc.).
- **etc** Es el directorio donde se encontrarán la mayoría de los archivos de configuración del sistema y de otras aplicaciones.
- **home** Donde se encontrarán los directorios personales de los usuarios del sistema.
- **lib** Librerías compartidas necesarias para la ejecución del sistema.
- **proc** Es donde se almacenan los ficheros con el estado (de procesos, de dispositivos) del sistema.

- **root** Es el directorio personal del administrador del sistema.
- **sbin** Comandos de administración del sistema.
- **tmp** Carpeta donde el sistema almacena información temporal.
- **usr** Ubicación que normalmente se dedica para instalar las aplicaciones de usuario.
- **var** Contiene datos que cambian cuando el sistema se ejecuta normalmente. En él podremos encontrar los archivos de registro del sistema, archivos temporales del servicio de correo o el directorio de trabajo del servidor de páginas web.



Filesystem Hierarchy Standard autentia

¿Qué es?

Filesystem Hierarchy Standard (FHS) es el estándar utilizado por GNU/Linux para definir la estructura de directorios de una forma jerárquica. **El directorio / es el padre (root)** de todo el sistema de ficheros.

ESTRUCTURA DEBAJO DE ROOT

/bin	Almacena las aplicaciones del sistema y comandos como cp, ls, cd, cat, etc.
/boot	Contiene los archivos necesarios (como el kernel o el grub) para el inicio del sistema.
/dev	Contiene los dispositivo físicos conectados al sistema (HDD, CD-ROM, USB, etc.).
/etc	Contiene la mayoría de los archivos de configuración del sistema y de otras aplicaciones.
/home	Directorios personales de los usuarios del sistema (exceptuando las de root). Cada usuario tiene su propio directorio.
/lib	Contiene las librerías compartidas necesarias para la ejecución del sistema.
/proc	Contiene el estado de los procesos en archivos de texto (uptime, network).
/root	Es el directorio /home para el usuario root.
/sbin	Comandos de administración del sistema que son exclusivos del root.
/tmp	Contiene los archivos o información temporal del sistema.
/usr	Ubicación que normalmente se dedica a instalar las aplicaciones de usuario.
/var	Contiene datos que cambian cuando el sistema se ejecuta. Podremos encontrar los archivos de registro del sistema, archivos temporales del servicio de correo, etc.

Recomendaciones y buenas prácticas

El estándar FHS da las siguientes recomendaciones relacionadas con estos directorios y las particiones:

- Se recomienda que directorios como '/tmp', '/var' y '/home' tengan su partición propia para que un crecimiento desmesurado de sus datos no afecten al resto del sistema.
- También se aconseja que el directorio '/home' se encuentre en una partición aparte para que no le afecte una posible actualización del sistema.
- Creando una partición propia para el directorio /boot conseguimos reducir la complejidad del sistema, facilitando el arranque, pudiendo dar una capa de seguridad evitando el automontaje de esta partición y permitir la instalación en equipos montados en RAID 1 o LVM que muchas BIOS no son compatibles.
- En el directorio / deben estar los directorios /etc, /lib o /bin que nunca deberían separarse en particiones.
- Se aconseja hacer copias de seguridad del directorio /etc y /home, ya que almacenan datos de usuarios y ficheros de configuración.

Añadir un nuevo disco

Para cimentar todos los conocimientos anteriormente explicados vamos añadir un nuevo disco, particionarlo y mover el directorio /home a la nueva partición.

Para añadir un nuevo disco hay que agregar las siguientes líneas en el Vagrantfile:

```
     unless File.exist?('./secondDisk.vdi')
        vb.customize ['createhd', '--filename',
'./secondDisk.vdi', '--variant', 'Fixed', '--size', 10 * 1024]
     end
     vb.customize ['storageattach', :id,  '--storagectl', 'SATA
Controller', '--port', 1, '--device', 0, '--type', 'hdd',
'--medium', './secondDisk.vdi']
```

Podemos consultar el significado de cada propiedad en esta página.

Una vez añadidas, ejecutamos `vagrant reload` para aplicar los cambios. En nuestro caso ya las añadimos en el documento inicial.

Una manera de conocer los valores que tenemos que añadir es ver el fichero /Users/${USER}/.vagrant.d/boxes/${BOX}/${VERSION}/virtualbox/box.ovf de nuestra **máquina host**.

- User: en el ejemplo ddelcastillo.
- Box: tal y como aparece la box en Vagrant Cloud.
- Version: la versión de la box bajada.

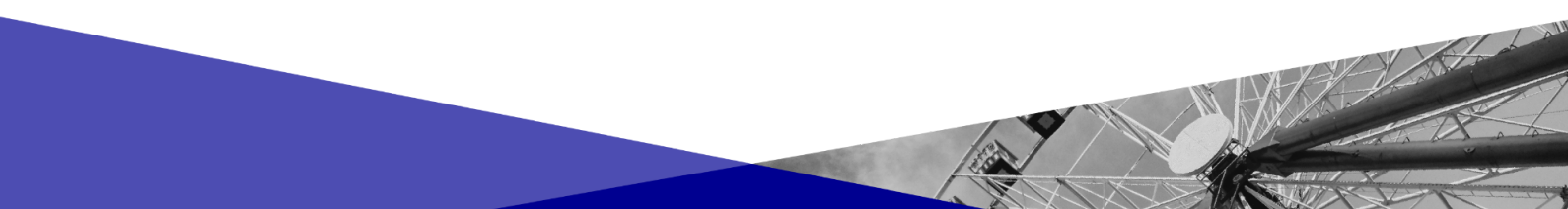
De esta manera, podemos conocer los valores de la propiedad "**--storagectl**".

Ejecutamos los comandos:

```
cd
/Users/${USER}/.vagrant.d/boxes/peru-VAGRANTSLASH-ubuntu-18.04-ser
ver-amd64/20200501.01/virtualbox
cat box.ovf | grep -i storagecontroller
```

Si nos vamos a /dev en la máquina guest y listamos el contenido de dev con el patrón `ls sd?` o ejecutamos el comando `sudo fdisk -l /dev/sd?` para listar las particiones y disco, podemos ver el nuevo disco.

```
vagrant@vagrant:/dev$ ls sd?
sda  sdb  sdc
vagrant@vagrant:/dev$ sudo fdisk -l /dev/sd?
Disk /dev/sda: 64 GiB, 68719476736 bytes, 134217728 sectors
Disk model: VBOX HARDDISK
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: EA49BFFA-1811-42C4-B9D3-08275C18710B

Device       Start       End   Sectors Size Type
/dev/sda1     2048      4095      2048   1M BIOS boot
/dev/sda2     4096   4198399   4194304   2G Linux filesystem
/dev/sda3  4198400 134215679 130017280  62G Linux filesystem


Disk /dev/sdb: 10 GiB, 10737418240 bytes, 20971520 sectors
Disk model: VBOX HARDDISK
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/sdc: 10 GiB, 10737418240 bytes, 20971520 sectors
```



Crear una partición

Vamos a utilizar Parted para crear la partición en el nuevo disco. Para crearla hay que realizar los siguientes pasos:

- Listamos los discos y particiones `sudo parted -l`. Podemos ver el nuevo disco montado en el directorio /dev/sdb y sin particionar.

```
vagrant@vagrant:~$ sudo parted -l
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sda: 68.7GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:

Number  Start   End     Size    File system  Name  Flags
 1      1049kB  2097kB  1049kB                     bios_grub
 2      2097kB  2150MB  2147MB  ext4
 3      2150MB  68.7GB  66.6GB


Error: /dev/sdb: unrecognised disk label
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sdb: 10.7GB
Sector size (logical/physical): 512B/512B
Partition Table: unknown
Disk Flags:
```

- Seleccionamos el disco montado en **/dev/sdb** con `sudo parted /dev/sdb`.

- Creamos la tabla de particiones en el nuevo disco con el comando `mklabel msdos`.. Los tipos de tabla de particiones soportados se pueden consultar aquí. Listamos la tabla de particiones de /dev/sdb y vemos que el anterior comando ha tenido efecto.

```
(parted) print list
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sdb: 10.7GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:
```

- Creamos una partición primaria con el FS de tipo ext4 volviendo a entrar en sdb con Parted con el comando `sudo parted /dev/sdb` y después ejecutamos `mkpart primary ext4 0 10.7GB` ignorando el warning.

```
(parted) select /dev/sdb
Using /dev/sdb
(parted) mkpart primary ext4 0 10.7GB
Warning: The resulting partition is not properly aligned fo
Ignore/Cancel? I
(parted) print
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sdb: 10.7GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start  End     Size    Type     File system  Flags
 1      512B   10.7GB  10.7GB  primary  ext4         lba
```

- Si listamos las particiones con el comando `print list` dentro de Parted podemos ver que nuestra nueva partición no está formateada por ext4.

```
(parted) print list
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sdb: 10.7GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start  End     Size    Type     File system  Flags
 1      512B   10.7GB  10.7GB  primary


Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sda: 68.7GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:

Number  Start   End     Size    File system  Name  Flags
 1      1049kB  2097kB  1049kB                     bios_grub
 2      2097kB  2150MB  2147MB  ext4
 3      2150MB  68.7GB  66.6GB
```

- Antes de crear el FS en nuestra partición, tenemos que conocer la ruta donde crearlo para ello, salimos de Parted con `quit` y ejecutamos `sudo fdisk -l /dev/sdb`

```
vagrant@vagrant:~$ sudo fdisk -l /dev/sdb
Disk /dev/sdb: 10 GiB, 10737418240 bytes, 20971520 sectors
Disk model: VBOX HARDDISK
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x03a31fe2

Device     Boot Start      End  Sectors Size Id Type
/dev/sdb1           1 20971519 20971519  10G 83 Linux
```

- Una vez que conocemos el dispositivo, podemos formatear la partición con el comando `sudo mkfs.ext4 /dev/sdb1`

```
vagrant@vagrant:~$ sudo mkfs.ext4 /dev/sdb1
mke2fs 1.46.5 (30-Dec-2021)
Creating filesystem with 2621439 4k blocks and 655360 inodes
Filesystem UUID: b9bc17c6-967a-4519-8572-f148c2413bcd
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done
```

- Por último, para poder acceder a la nueva partición recién creada, tenemos que montarla en un directorio.
    - Creamos el directorio donde montar la partición: `sudo mkdir -p /mnt/sdb1`.
    - Montamos la partición en el directorio creado: `sudo mount /dev/sdb1 /mnt/sdb1`.

```
vagrant@vagrant:~$ sudo mkdir -p /mnt/sdb1
vagrant@vagrant:~$ sudo mount /dev/sdb1 /mnt/sdb1
vagrant@vagrant:~$ df -hT
Filesystem                         Type    Size  Used Avail Use% Mounted on
tmpfs                              tmpfs    98M 1004K   97M   2% /run
/dev/mapper/ubuntu--vg-ubuntu--lv ext4     31G  3.9G   25G  14% /
tmpfs                              tmpfs   486M     0  486M   0% /dev/shm
tmpfs                              tmpfs   5.0M     0  5.0M   0% /run/lock
/dev/sda2                          ext4    2.0G  126M  1.7G   7% /boot
vagrant                            vboxsf  466G  127G  340G  28% /vagrant
tmpfs                              tmpfs    98M  4.0K   98M   1% /run/user/1001
/dev/sdb1                          ext4    9.8G   24K  9.3G   1% /mnt/sdb1
```

    - Vemos que se ha montado correctamente gracias al comando `df -hT`. Este comando nos muestra más información como el tipo de FS, su tamaño (diferenciando entre usado y disponible), así como su punto de montaje.

Mover a la nueva partición

Si nos hemos quedado sin espacio o queremos anticiparnos a que esto ocurra, podemos mover directorios que son susceptibles de crecer como /home a una nueva partición montada en un nuevo disco con más capacidad.

En este apartado, mostraremos los pasos a seguir para mover el directorio /home a la partición creada en apartados anteriores:

- **Copiamos todos los archivos de manera recursiva** (todos los ficheros de un directorio y repetimos el proceso para los subdirectorios) del directorio de anclaje de la nueva partición con el comando `sudo cp -aR /home/* /mnt/sdb1`. Una forma de comprobar que este paso se ha realizado correctamente es utilizar la herramienta **diff** y ver que no hay diferencia entre los archivos de una ruta y otra con el siguiente comando `sudo diff -r /home /mnt/sdb1`.

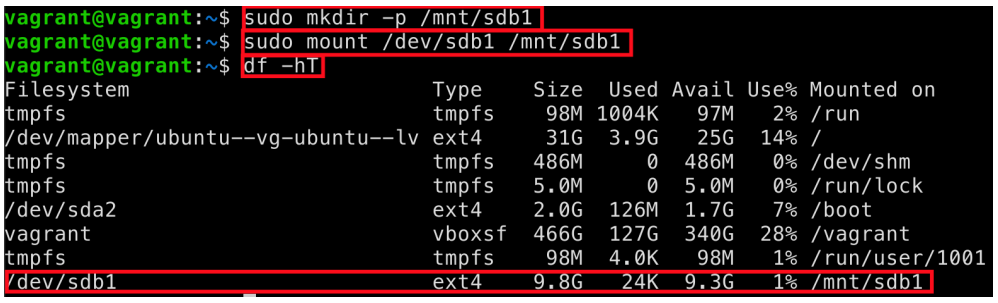
- Como ya tenemos los datos de /home en la nueva partición. Los **borramos** de la antigua con el siguiente comando `sudo rm -rf /home/*`.

- Desmontamos la nueva partición de **/mnt/sdb1** ejecutando `sudo umount /mnt/sdb1`. Antes de ejecutar este comando, asegúrate de no encontrarte dentro del directorio a desmontar.

- Montamos la nueva partición de **/home** ejecutando `sudo mount /dev/sdb1 /home`.

- Si hacemos una comparativa de tamaño con el apartado anterior, apreciamos que el tamaño usado de la nueva partición se ha incrementado. No podemos ver la disminución de tamaño usado en la antigua porque es mínima y no se aprecia al medirse en GB.

    - ANTES



- DESPUÉS

```
vagrant@vagrant:~$ df -hT
Filesystem                         Type     Size  Used Avail Use% Mounted on
tmpfs                              tmpfs     98M 1004K   97M   2% /run
/dev/mapper/ubuntu--vg-ubuntu--lv ext4      31G  3.9G   25G  14% /
tmpfs                              tmpfs    486M     0  486M   0% /dev/shm
tmpfs                              tmpfs    5.0M     0  5.0M   0% /run/lock
/dev/sda2                          ext4     2.0G  126M  1.7G   7% /boot
vagrant                            vboxsf   466G  126G  341G  27% /vagrant
tmpfs                              tmpfs     98M  4.0K   98M   1% /run/user/1001
/dev/sdb1                          ext4     9.8G   92K  9.3G   1% /home
```

Si queremos que el anclaje de la nueva partición se haga permanente y no se pierda al reiniciar la máquina, tenemos que tocar **fstab**. El fstab es el fichero que se consulta al inicio para saber qué directorios montar. Tenemos que añadir una línea en el **/etc/fstab**. La línea está compuesta por lo siguiente:

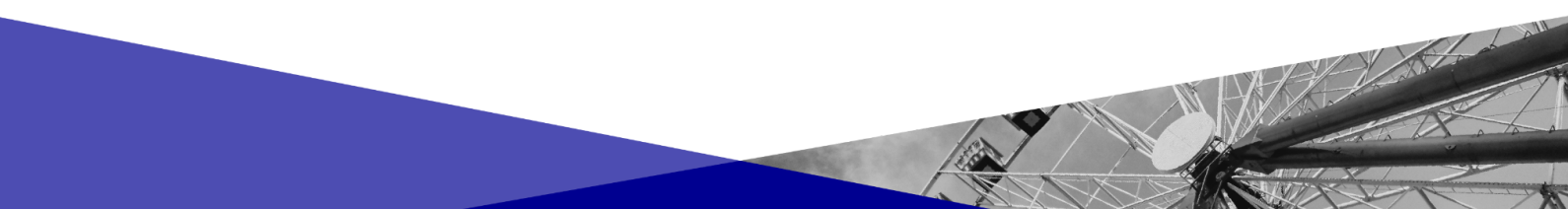
[Device] [Mount Point] [File System Type] [Options] [Dump] [Pass]

- **Device**: la localización (/dev/sdb1) o el UUID (ejecutando `sudo blkid /dev/sdb1`).
- **Mount Point**: el directorio por el cual será accesible la partición.
- **File system type**: el tipo de File System.
- **Options**: opciones de acceso a la partición.
- **Dump**: habilitar (o no) la partición para hacer una copia de seguridad al ejecutar el comando dump.
- **Pass**: orden en el que fsck comprueba el dispositivo/partición en busca de errores en el momento del arranque. El dispositivo raíz debería ser el 1. Las otras particiones deberían ser 2 o 0 para desactivar la comprobación.

  Por lo que nuestra línea quedará de la siguiente manera:

  /dev/sdb1 /home ext4 defaults 0 0

  Después de reiniciar (`sudo shutdown -r now`), podemos ver que que la línea está añadida en /etc/fstab:



```
vagrant@vagrant:~$ sudo cat /etc/fstab
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/ubuntu-vg/ubuntu-lv during curtin installation
/dev/disk/by-id/dm-uuid-LVM-qcAv8viYIFO62I78z51D50ArWPhAUmuVMWjEphp6Q07A5FQo7R4jNr
# /boot was on /dev/sda2 during curtin installation
/dev/disk/by-uuid/27b9d749-1470-4a8c-b837-2daab86aad8d /boot ext4 defaults 0 1
/swap.img       none    swap    sw      0       0
#VAGRANT-BEGIN
# The contents below are automatically generated by Vagrant. Do not modify.
vagrant /vagrant vboxsf uid=1000,gid=1000,_netdev 0 0
#VAGRANT-END
/dev/sdb1 /home  ext4   defaults        0       0

vagrant@vagrant:~$ df -hT
Filesystem                         Type    Size  Used Avail Use% Mounted on
tmpfs                              tmpfs    98M 1004K   97M   2% /run
/dev/mapper/ubuntu--vg-ubuntu--lv ext4     31G  3.9G   25G  14% /
tmpfs                              tmpfs   486M     0  486M   0% /dev/shm
tmpfs                              tmpfs   5.0M     0  5.0M   0% /run/lock
/dev/sda2                          ext4    2.0G  126M  1.7G   7% /boot
vagrant                            vboxsf  466G  126G  341G  27% /vagrant
tmpfs                              tmpfs    98M  4.0K   98M   1% /run/user/1001
/dev/sdb1                          ext4    9.8G   96K  9.3G   1% /home
```

Borrar una partición

Para explicar este apartado vamos a añadir otro disco de 10 GB con dos particiones de 5 GB cada una. En el archivo Vagrantfile ya se incluye este disco, nos quedaría particionarlo como hemos visto antes. Una vez lo tenemos hecho, tendríamos algo como lo siguiente:

```
vagrant@vagrant:~$ df -hT
Filesystem                         Type    Size  Used Avail Use% Mounted on
tmpfs                              tmpfs    98M 1012K   97M   2% /run
/dev/mapper/ubuntu--vg-ubuntu--lv ext4     31G  3.9G   25G  14% /
tmpfs                              tmpfs   486M     0  486M   0% /dev/shm
tmpfs                              tmpfs   5.0M     0  5.0M   0% /run/lock
/dev/sda2                          ext4    2.0G  126M  1.7G   7% /boot
vagrant                            vboxsf  466G  126G  341G  27% /vagrant
tmpfs                              tmpfs    98M  4.0K   98M   1% /run/user/1001
/dev/sdb1                          ext4    9.8G   96K  9.3G   1% /home
/dev/sdc1                          ext4    4.9G   24K  4.6G   1% /mnt/sdc1
/dev/sdc2                          ext4    4.9G   24K  4.6G   1% /mnt/sdc2
```

Se ha movido a otro disco el contenido de la partición **/dev/sdc2** y nos piden borrarla para liberar espacio. Para ello debemos seguir los siguientes pasos:

- Desmontamos el punto de anclaje con `sudo umount /mnt/sdc2`.
- Abrimos **Parted** con el tercer disco añadido con `sudo parted`

`/dev/sdc`.

- Imprimimos las particiones que tiene el disco con `print` montrando:

```
(parted) print
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sdc: 10.7GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type     File system
 1      512B    5350MB  5350MB  primary  ext4
 2      5350MB  10.7GB  5387MB  primary  ext4
```

- Ejecutamos el comando `rm 2` para borrar la segunda partición.
- Para ver que la partición se ha eliminado correctamente podemos ejecutar `print free`.

```
(parted) print free
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sdc: 10.7GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type     File system
        0.00B   511B    512B             Free Space
 1      512B    5350MB  5350MB  primary  ext4
        5350MB  10.7GB  5387MB           Free Space
```

Redimensionar una partición

Nos han pedido redimensionar la partición /dev/sdc1 para aprovechar el espacio desasignado de la partición borrada en el apartado anterior. Para hacer este procedimiento de manera segura y sin pérdida de datos hay que realizar los siguientes pasos:

- Desmontamos el punto de anclaje con `sudo umount /mnt/sdc1`.
- Ejecutamos `sudo parted /dev/sdc`.
- Ejecutamos `print` para ver el número de la partición a redimensionar.
- Ejecutamos el comando `resizepart 1 100%` para redimensionar la partición 1.

- Si imprimimos las particiones vemos que el redimensionamiento se ha realizado correctamente.

```
(parted) print
Model: ATA VBOX HARDDISK (scsi)
Disk /dev/sdc: 10.7GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start  End     Size    Type     File system
 1      512B   10.7GB  10.7GB  primary  ext4
```

- Pero si montamos la partición y listamos el espacio disponible con `df -hT` vemos que no tenemos el espacio reasignado disponible.

```
vagrant@vagrant:~$ sudo mount /dev/sdc1 /mnt/sdc1
vagrant@vagrant:~$ df -hT
Filesystem                         Type    Size  Used Avail Use% Mounted on
tmpfs                              tmpfs    98M 1008K   97M   2% /run
/dev/mapper/ubuntu--vg-ubuntu--lv ext4     31G  3.9G   25G  14% /
tmpfs                              tmpfs   486M     0  486M   0% /dev/shm
tmpfs                              tmpfs   5.0M     0  5.0M   0% /run/lock
/dev/sda2                          ext4    2.0G  126M  1.7G   7% /boot
vagrant                            vboxsf  466G  127G  340G  28% /vagrant
tmpfs                              tmpfs    98M  4.0K   98M   1% /run/user/1001
/dev/sdb1                          ext4    9.8G   96K  9.3G   1% /home
/dev/sdc1                          ext4    4.9G   24K  4.6G   1% /mnt/sdc1
```

- Para tener disponible el espacio reasignado hay que ejecutar el comando `sudo resize2fs /dev/sdc1` y veremos que ahora sí aparece el tamaño correcto.

```
vagrant@vagrant:~$ sudo resize2fs /dev/sdc1
resize2fs 1.46.5 (30-Dec-2021)
Filesystem at /dev/sdc1 is mounted on /mnt/sdc1; on-line resizing required
old_desc_blocks = 1, new_desc_blocks = 2
The filesystem on /dev/sdc1 is now 2621439 (4k) blocks long.

vagrant@vagrant:~$ df -hT
Filesystem                         Type    Size  Used Avail Use% Mounted on
tmpfs                              tmpfs    98M 1008K   97M   2% /run
/dev/mapper/ubuntu--vg-ubuntu--lv ext4     31G  3.9G   25G  14% /
tmpfs                              tmpfs   486M     0  486M   0% /dev/shm
tmpfs                              tmpfs   5.0M     0  5.0M   0% /run/lock
/dev/sda2                          ext4    2.0G  126M  1.7G   7% /boot
vagrant                            vboxsf  466G  127G  340G  28% /vagrant
tmpfs                              tmpfs    98M  4.0K   98M   1% /run/user/1001
/dev/sdb1                          ext4    9.8G   96K  9.3G   1% /home
/dev/sdc1                          ext4    9.8G   23M  9.3G   1% /mnt/sdc1
```

Editor Vi

El editor Vi es el más usado dentro de la familia Linux. Algunas de las razones que lo convierten en un editor ampliamente utilizado son:

1. Está disponible en casi todas las distribuciones de Linux.
2. Funciona igual en diferentes plataformas y distribuciones.
3. Es fácil de usar aunque es difícil conocer todas las funciones que ofrece para editar un archivo.

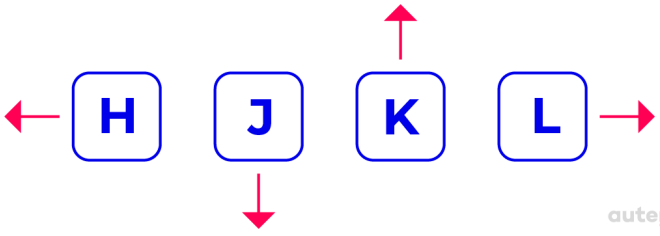
Modos

Para trabajar en el editor Vi, es necesario entender sus modos de funcionamiento. Existen tres modos en vi:

- **Modo comando**: este es el modo en el que se encuentra el editor al iniciarse. Puede ejecutar comandos como mover el cursor así como copiar, cortar o pegar texto. Podemos salir del modo inserción al modo comando pulsando ESC.
- **Modo inserción**: este es el modo que se usa para insertar texto. Una forma de entrar en este modo es pulsar la tecla **i** desde el modo de comando aunque existen otras formas que veremos en siguientes apartados.
- **Modo ex**: en este modo se pueden realizar búsquedas y sustituciones. Todos los comandos que empiezan con ":" son comandos en modo ex.

Moverse con el cursor

Vi permite desplazarse por el texto utilizando las siguientes teclas:

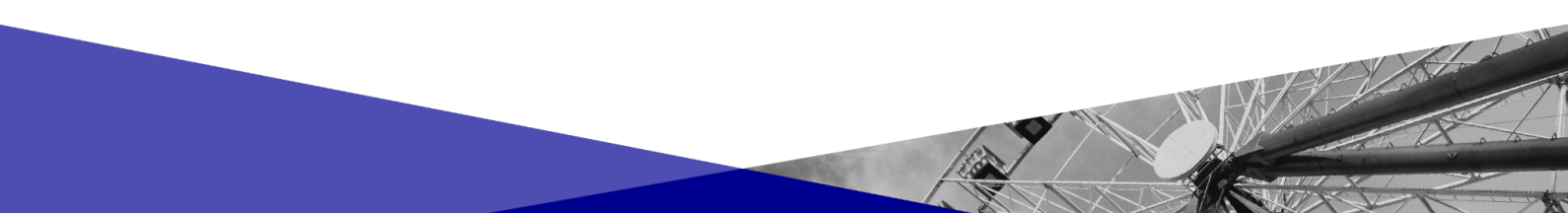


Comando (teclas)	Acción
Flechas	Mover en la dirección de la flecha.
h	Mover hacia la izquierda
l	Mover hacia la derecha.
k	Mover hacia arriba.
j	Mover hacia abajo.
1G	Lleva el curso hasta el comienzo del archivo.
G	Lleva el curso hasta el final del archivo.

Cambiar de Modo

En este apartado mostramos los teclas que ofrece vi para cambiar de modo comando a modo inserción.

Comando (teclas)	Acción
i	Insertar texto a la izquierda del cursor.
a	Insertar texto a la derecha del cursor.
A	Inserta texto al final de la línea donde se encuentre el cursor.
I	Inserta texto al comienzo de la línea donde se encuentre el cursor.
o	Insertar una línea por debajo de la actual.
O	Insertar una línea por encima de la actual.



Borrar texto

En este apartado mostramos las teclas que ofrece vi para borrar texto en el modo comando (sin entrar en el modo inserción). Aclarar que todo lo que se borra queda almacenado en un buffer y puede recuperarse inmediatamente (sin ejecutar otros comandos) pulsando la tecla **p**.

Comando (teclas)	Acción
x	Borra el caracter bajo el cursor.
dd	Borra la línea donde se encuentra el cursor.
ndd	Borra las próximas n líneas.
D	Borra desde donde se encuentra el cursor hasta el final de la línea.
dw	Borra desde donde se encuentra el cursor hasta el final de una palabra.

Copiar, cortar y pegar

Vi ofrece muchas opciones para copiar y pegar, incluyendo la capacidad de copiar y pegar desde múltiples buffers. Los siguientes comandos permiten copiar y pegar:

Comando (teclas)	Acción
yy	Copia la línea donde se encuentra el cursor.
nyy	Copia las próximas n líneas.
p	Pega una línea por debajo de la actual.
P	Pega una línea por encima de la actual.
u	Deshacer el cambio anterior, equivalente a :redu.

Como hemos dicho en el apartado anterior, al borrar una línea con el comando **d**, este es almacenado en el buffer y es lo que usaremos como equivalente del cortar.

Otra forma de copiar, cortar y pegar es usando marcas. Para ello podemos usar el comando **m** (y el identificador de la marca), por ejemplo mc. Este comando se tiene que introducir al principio del bloque que queramos copiar o cortar. Al final del bloque que queremos copiar o cortar introduciremos (siguiendo con el ejemplo):

- Para **copiar** el bloque de la marca c se introduciría **y'c**.
- Para **cortar** el bloque de la marca c se introduciría **d'c**.

Una vez copiado o cortado, solo habría que usar el comando **p o P** para pegar el contenido almacenado en el buffer. También podemos entrar en modo visual pulsando **v.**

Buscar y reemplazar

Algunos de los comando para realizar búsquedas podrían ser:

Comando(teclas)	**Acción**
/string	Búsqueda hacia adelante de la cadena.
?string	Búsqueda hacia atrás de la cadena.
n	Siguiente ocurrencia en la dirección de la búsqueda.
N	Siguiente ocurrencia en dirección opuesta.
fchar	Buscar un caracter en la misma línea.
;	Siguiente ocurrencia hacia adelante en la misma línea.
'	Siguiente ocurrencia hacia atrás en la misma línea.
:%s/old/new/g	Buscar y reemplazar en todo el fichero.
:%s/old/new/gc	Buscar y reemplazar en todo el fichero pidiendo confirmación.

Manipulación de ficheros

Algunos de los comandos son los siguientes:

Comando (teclas)	Acción
:e newfile	Abrir un fichero en modo edición sin salir del vi.
:r newfile	Abrir un fichero en modo lectura sin salir del vi.
vi file* o file1 file2	Se pueden abrir varios ficheros usando wildcard o listas de ficheros.
:n o :bn	Abrir el siguiente fichero de la lista de ficheros abiertos.
:bp	Abrir el anterior fichero de la lista de ficheros abiertos.
:w filename	Guardar los cambios en otro fichero.
:wq o ZZ	Guardar y salir.
:r !command	Ejecutar un comando y almacenar la salida en el vi
:!command	Ejecutar el comando y volver al vi.
:set	Forma de habilitar/deshabilitar configuraciones en vi (set number para que aparezca en número de la línea).

Una herramienta útil puede ser esta chuleta de comandos de vi.

Procesos, servicios y tareas programadas

Un proceso es un programa en ejecución. Cuando ejecutamos una aplicación, se ejecutan uno o varios procesos que necesitan hacer uso de ciertos recursos como son memoria, dispositivos E/S, tiempo de CPU, archivos, etc. La asignación de los recursos se hace cuando se crea y durante su ejecución. Un proceso tiene asignado un identificador único llamado PID, que se usará para referenciarlo en las diferentes herramientas de gestión de procesos.

Herramientas para el manejo de procesos

En Linux hay herramientas que nos permiten ver los procesos en ejecución y los recursos consumidos. En los siguientes apartados veremos algunas de estas herramientas.

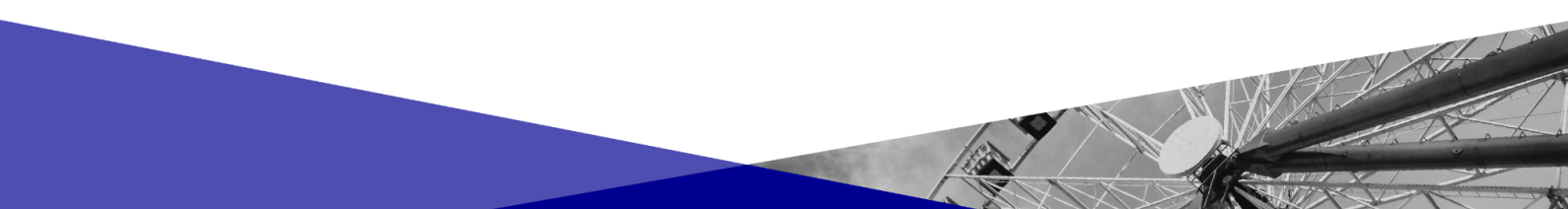
Ver los procesos en ejecución

Existen varias herramientas para ver los procesos en ejecución, la más importante es el comando `ps`.

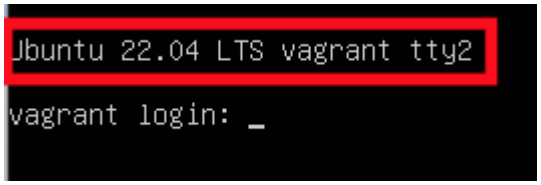
```
vagrant@localhost:~$ ps
  PID TTY          TIME CMD
 1957 pts/0    00:00:00 bash
 1968 pts/0    00:00:00 ps
```

Las cuatro columnas que aparecen se identifican con:

- **PID**: el identificador del proceso.
- **TTY**: el nombre de la consola a la que el usuario se ha logado. Normalmente para cambiar de terminal se hace con CTRL/CMD + ALT + Tecla de Función (F1 para tty1, F2 para tty2, etc.). Para probarlo en nuestro entorno debéis hacer login (vagrant/vagrant) en la máquina virtual en vez de conectarnos por ssh. Os aparecerá algo similar a esto:
    - TTY1 (CTRL/CMD + ALT + F1).



- TTY2 (CTRL/CMD + ALT + F2)



- **TIME**: el tiempo de CPU usado en el proceso.
- **CMD**: el comando que ha provocado el proceso.

PS nos ofrece muchas opciones tanto de filtrado como de formato para mostrar los procesos del sistema.

Una de las más utilizadas es `ps -aux`, que muestra todos los procesos en ejecución. La salida de este comando se puede ir procesado mediante la concatenación de otros comandos usando pipes (|). Un ejemplo sería procesar la salida con **less** (similar a **more**) para paginar la salida de ps quedando de la siguiente manera:

```
ps -aux | less
```

Pero se pueden hacer cosas más complicadas como listar los 10 procesos con más consumo de CPU.

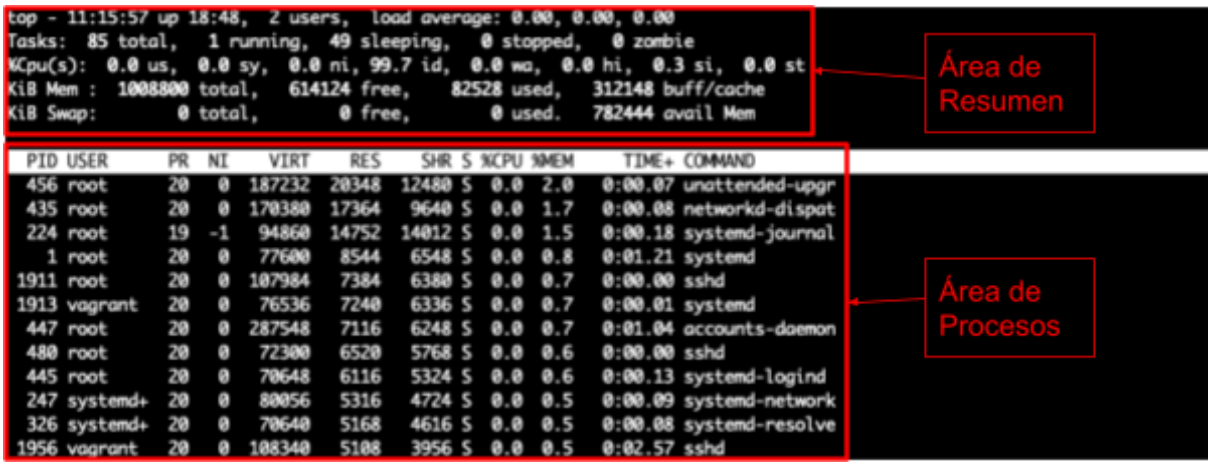
```
ps -eo pcpu,pid,user,args | sort -k1 -r -n | head -10
```

Aunque hay muchas formas de hacer lo mismo con diferentes comandos y herramientas tal y como indica este artículo.

El comando **ps** da una versión estática de los procesos. **Top** nos da una lista actualizada dinámicamente. Por defecto, se refresca cada 3 segundos. Para cambiar el tiempo de refresco a 1 segundo sería:

```
top -d 1.0
```

Mostrándonos lo siguiente:



Podemos ver dos zonas diferenciadas:

- **Área de resumen:** en este área podemos ver información sobre el número de sesiones, el uso de la memoria, tiempo de actividad, unas estadísticas de los estados de los procesos, uso de CPU, etc.
- **Área de procesos:** aparece un listado de procesos activos, divididos por diferentes columnas como (PID, USER, %CPU, etc.).

También ofrece la posibilidad de **filtrar** la lista pulsando la tecla 'o' e indicando el criterio del filtro, por ejemplo USER=vagrant. Esto nos mostrará todos los procesos del usuario vagrant. Para resetear los filtros hay que pulsar '='.

Podemos **añadir nuevas columnas** como la columna SWAP, que no aparece por defecto y que nos ayuda a detectar qué procesos realizan swapping (utilizar espacio de disco como memoria). Para añadirla, pulsamos 'f', seleccionamos las columnas que queramos añadir con 'espacio'. También podemos especificar la columna por la que queremos ordenar, seleccionando y pulsando 's'. Para salir pulsamos 'q' y, si quisiéramos un orden inverso del campo SWAP, pulsaremos 'R'.

Las siguientes teclas ordenan según estos criterios:

- **M**: ordena por memoria.
- **N**: ordena por PID.
- **P**: ordena por %CPU.
- **T**: ordena por tiempo.

Normalmente este comando se utiliza para:

- Matar procesos: si pulsamos 'k' podemos indicar el PID del proceso a matar y el SIGTERM 9 (**SIGKILL**).

- Filtrar por usuario, ordenando por diferentes columnas, mostrar los procesos en modo árbol (padre e hijos), etc.

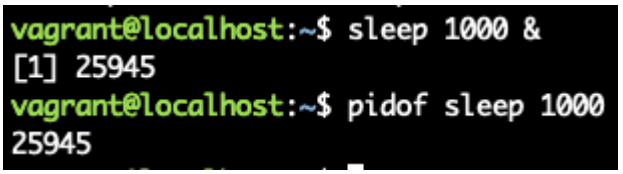
Una lectura muy recomendable para entender todo lo que ofrece el comando top es este post.

Matar procesos

Para matar procesos utilizaremos el comando **kill -[SIGTERM] [PID,..]**. El SIGTERM indica la señal que se le va a mandar al PID (id del proceso) o PIDs indicados. Normalmente se usan estos **SIGTERM**:

- **9**: para matar el proceso.
- **15**: SIGTERM por defecto si no se especifica. Parar el proceso.

Para conocer el PID podemos usar todas las herramientas explicadas en apartados anteriores o **pidof** :



Listar los ficheros abiertos por procesos

Si desea ver el nombre de los archivos que han sido abiertos por un proceso en particular, el comando **lsof** nos permite hacerlo. Un archivo abierto podría ser un archivo normal, directorio, biblioteca, un programa en ejecución o incluso una secuencia o un archivo de red. Algunas de las utilidades que podemos darle a lsof son:

- Listar todos los ficheros abiertos ejecutando simplemente `lsof`.
- Listar los ficheros abiertos por los procesos de un usuario específico con `lsof -u vagrant`.
- Listar los archivos abiertos en función de la dirección a internet (IPV4 o IPV6) con `lsof -i 4`(IPV4). Esta opción también se suele utilizar para listar los puertos abierto con `lsof -i -P -n | grep LISTEN`.
- Listar los ficheros abiertos por un proceso con `lsof -p 26107`. Podemos ejecutar un proceso en segundo plano, donde abrimos un fichero de prueba y esperamos 1000 segundos.

```
vagrant@localhost:~$ vi prueba && sleep 1000 &
[1] 26107
vagrant@localhost:~$ lsof -p 26107
COMMAND   PID    USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
bash    26107 vagrant  cwd    DIR    8,1     4096 1703938 /home/vagrant
bash    26107 vagrant  rtd    DIR    8,1     4096       2 /
bash    26107 vagrant  txt    REG    8,1  1113504 2359418 /bin/bash
bash    26107 vagrant  mem    REG    8,1    47568 2228258 /lib/x86_64-linux-gnu/libnss_files-2.27.so
bash    26107 vagrant  mem    REG    8,1    97176 2228255 /lib/x86_64-linux-gnu/libnsl-2.27.so
bash    26107 vagrant  mem    REG    8,1    47576 2228260 /lib/x86_64-linux-gnu/libnss_nis-2.27.so
bash    26107 vagrant  mem    REG    8,1    39744 2228256 /lib/x86_64-linux-gnu/libnss_compat-2.27.so
bash    26107 vagrant  mem    REG    8,1  3004464 3014678 /usr/lib/locale/locale-archive
bash    26107 vagrant  mem    REG    8,1  2030544 2228248 /lib/x86_64-linux-gnu/libc-2.27.so
bash    26107 vagrant  mem    REG    8,1    14560 2228251 /lib/x86_64-linux-gnu/libdl-2.27.so
bash    26107 vagrant  mem    REG    8,1   170784 2228296 /lib/x86_64-linux-gnu/libtinfo.so.5.9
bash    26107 vagrant  mem    REG    8,1   170960 2228239 /lib/x86_64-linux-gnu/ld-2.27.so
bash    26107 vagrant  mem    REG    8,1    26376 3017178 /usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache
bash    26107 vagrant    0u   CHR  136,0      0t0       3 /dev/pts/0
bash    26107 vagrant    1u   CHR  136,0      0t0       3 /dev/pts/0
bash    26107 vagrant    2u   CHR  136,0      0t0       3 /dev/pts/0
bash    26107 vagrant  255u   CHR  136,0      0t0       3 /dev/pts/0

[1]+  Stopped                 vi prueba && sleep 1000
vagrant@localhost:~$
```

- Listar los ficheros abiertos indicando un directorio concreto con `lsof +D /tmp/`. Siguiendo con el ejemplo anterior, podemos ver la salida de este comando.

```
vagrant@localhost:~$ vi /tmp/prueba && sleep 1000 &
[5] 26140
vagrant@localhost:~$ sudo lsof +D /tmp/

[5]+  Stopped                 vi /tmp/prueba && sleep 1000
vagrant@localhost:~$
```

Arrancando una aplicación como servicio

Un servicio no es más que una aplicación que se inicia al arrancar el servidor, interactúa con el usuario de alguna manera (proveyendo páginas web en el caso del servidor web, suministrando o persistiendo datos en el caso de base de datos, etc.).

En este apartado instalamos un NGINX como servicio. Para ello, debemos seguir los siguiente pasos:

1. Actualizamos el listado de paquetes y sus versiones ejecutando `sudo apt-get update`.
2. Se crea un fichero de texto `sudo nano /etc/systemd/user/nginx.service` del tipo “unit file”. Este fichero de texto “unit file” en Linux es cualquier recurso que se puede administrar, como nuestro nuevo servicio. El fichero contiene las

directivas de configuración que define el comportamiento de nuestro servicio. Añadimos las siguientes líneas dando una breve descripción de la directiva como comentario en las más relevantes:

```
[Unit]
Description=A high performance web server and a reverse proxy server
# Indica que la red debe estar activa para ejecutarse el servicio
After=network.target
[Service]
# Simple (Por defecto, como en primer plano), forking(Como en segundo
# plano). Consultad más tipos aquí.
Type=forking
# Fichero donde se almacena el PID para monitorizarse en caso de forking
PIDFile=/var/run/nginx.pid
# Comando que se deben ejecutar antes de arrancar, al arrancar, parar,
# o reiniciar
ExecStartPre=/usr/sbin/nginx -t -q -g 'daemon on; master_process on;'
ExecStart=/usr/sbin/nginx -g 'daemon on; master_process on;'
ExecReload=/usr/sbin/nginx -g 'daemon on; master_process on;' -s reload
ExecStop=-/sbin/start-stop-daemon --quiet --stop --retry QUIT/5
--pidfile /var/run/nginx.pid


# Timeout antes de parar o matar el servicio
TimeoutStopSec=5
# Especifica cómo se matarán los procesos de esta unidad.
# Con mixed se envía la SIGTERM de suspender solo al proceso padre
KillMode=mixed
[Install]
# Espera a que se inicien los servicios de nivel 2 (los servicios sin
red)
WantedBy=multi-user.target
```

Hemos cogido el unit file de la página oficial de NGINX. Si queréis entender mejor la anatomía de los 'Unit File', la lectura de este artículo es muy recomendable.

Una vez creado el fichero, podemos interactuar con el servicio con los siguientes comandos:

- `sudo systemctl start nginx`: para iniciar el servicio.
- `sudo systemctl stop nginx`: para parar el servicio.
- `sudo systemctl restart nginx`: para reiniciar el servicio.

- `sudo systemctl reload nginx`: para cargar ficheros de configuración sin necesidad de reiniciar.
- `sudo systemctl enable nginx`: habilitar que el servicio NGINX se arranque al iniciar.
- `sudo systemctl disable nginx`: deshabilitar que el servicio NGINX se arranque al iniciar.
- `sudo systemctl status nginx`: conocer el estado del servicio, si está arrancado o parado.
- `sudo journalctl -f -n 10 -u nginx.service`: conocer a tiempo real las 10 últimas líneas del log del servicio NGINX.

Tareas programadas

Las tareas programadas en Linux nos ayudan a la hora de realizar tareas como:

- Realizar copias de seguridad.
- Comprobar el espacio y mandar un mail si corresponde.
- Borrar archivos temporales.

Se recomienda el uso de tareas programadas cuando se quiera realizar una tarea, generalmente pesada en un momento determinado, normalmente cuando el servidor tiene menos carga.

Crontab es el fichero por usuario donde se especifican las tareas, programas o jobs a ejecutar. Un ejemplo de una línea podría ser el siguiente:

```
# m h  dom mon dow   command
*/1    *    *    *    *    /sbin/ping -c 1 127.0.0.1; ls -la >> /var/log/cronrun
```

En las primeras 5 columnas se especifica el tiempo en el que queremos establecer la tarea: minutos (0 al 59), horas (0 al 24), día del mes (1 al 31), mes (1 al 12) y día de la semana (0-6, siendo 0 el domingo). Y por último, el comando o comandos (separados por ; a ejecutar).

La expresión que se ve en la imagen indica que se ejecute el cron cada minuto, que es equivalente a poner los 5 asteriscos (* * * * *). Pero queríamos enseñar esta forma de especificarlo porque podemos extrapolar fácilmente:

EXPRESSION CRON (m, h, d, mes, sem)	**EXPLICACIÓN**
***/1 * * * ***	Cada minuto.
*** */1 * * ***	El primer minuto de cada hora (cada hora). Otra forma de expresarlo sería con @hourly.
*** */8 * * ***	El primer minuto cada 8 horas.
0 8 * * *	Todos los días a las 8 am. No confundir con la anterior expresión.
*** * */1 * ***	El primer día del mes (cada mes). Otra forma de expresarlo sería con @monthly.

Os dejamos esta página para que podáis evaluar vuestras expresiones.

Para manejar el crontab podemos utilizar los siguientes comandos:

`crontab -l`: lista todas las tareas del crontab sin editarlo.

`crontab -e`: permite editar el crontab.

`crontab -u abarranco -e`: edita el crontab del usuario abarranco.

ADVERTENCIA: Cuidado con los permisos de los script y quién los ejecuta ya que un script en el que tenemos permisos de escritura, cuyo propietario es root puede hacer que alguien con nuestras credenciales escale a root.

Imaginemos que tenemos una tarea que hace backups, su propietario es root y por descuido o desconocimiento le hemos concedido todos los permisos para que cualquiera pueda modificarla sin tener que darle las credenciales de root. Veamos los pasos que puede seguir un atacante para escalar a root por este error en nuestra tarea programada:

- Vamos a concederle a backup.sh todos los permisos mediante el comando `chmod 777 backup.sh`.

```
vagrant@localhost:~$ cat backup.sh
#!/bin/bash
echo 'Haciendo backup';
vagrant@localhost:~$ sudo chmod 777 backup.sh
vagrant@localhost:~$ sudo chown root backup.sh
vagrant@localhost:~$ ls -la backup.sh
-rwxrwxrwx 1 root vagrant 36 May 13 08:00 backup.sh
vagrant@localhost:~$
```

- Si intentamos modificar el fichero /etc/passwd directamente, el sistema lo impedirá ya que no tenemos permisos para ello (no somos root). Si cambiamos a root, modificamos el script y añadimos la línea `chmod 777 /etc/passwd`, nos permite conceder todos los permisos (incluido el de escritura) y podremos modificarlo.

```
#!/bin/bash
echo 'Haciendo backup';
chmod 777 /etc/passwd;
```

- Ahora nos centraremos en generar una password y establecerla para el usuario root en **/etc/passwd/**. Para conocer el método de encriptación podemos hacer un *`cat /etc/login.defs | grep ENCRYPT_METHOD`*.

- Si generamos una password usamos `openssl passwd`, que genera la password con el método de encriptación de sistema por defecto si no le especificamos nada. Introducimos nuestra password y copiamos la generada.

```
vagrant@localhost:~$ openssl passwd
Password:
Verifying - Password:
sJ0Cy6It/DMvY
```

- Editamos el /etc/passwd y cambiamos la **x** de root por la password que hemos generado en el paso anterior. Al reemplazar la x, hacemos que no intente validar la password almacenada en **/etc/shadow.**

```
GNU nano 2.9.3                                    /etc/passwd

root:sJ0Cy6It/DMvY:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

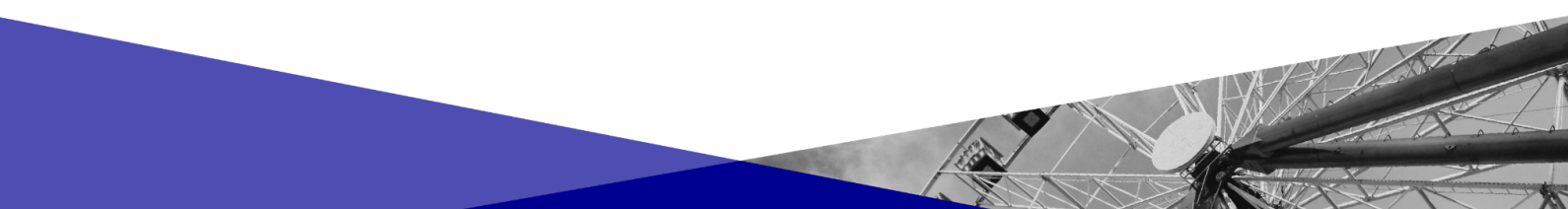
- Hacemos `su -root` e introducimos nuestra password. Vemos que ya somos root y todo por conceder más permisos de los necesarios en una tarea programada.

```
vagrant@localhost:~$ cat /etc/passwd | grep root
root:sJ0Cy6It/DMvY:0:0:root:/root:/bin/bash
vagrant@localhost:~$ su - root
Password:
root@localhost:~#
```

Debemos tener mucho cuidado con los permisos que concedemos y siempre intentar conceder los mínimos imprescindibles para evitar problemas.

De Crontab a Timers de SystemD

Otra forma de establecer tareas programadas es convirtiendo los comandos a ejecutar en scripts, estos scripts se convierten en un servicio y utilizan timers. Cron no es un estándar, mientras que SystemD sí lo es.



Vamos con un ejemplo. Imaginemos que tenemos una tarea programada en Crontab que cada minuto loga ‘Este mensaje aparece cada minuto” en /tmp/myMessages. Para ello tenemos que seguir los siguientes pasos:

- Vemos la tarea crontab a extraer. Podemos ver que la tarea imprime ‘Este mensaje aparece cada minuto desde Crontab” en /tmp/myMessages.

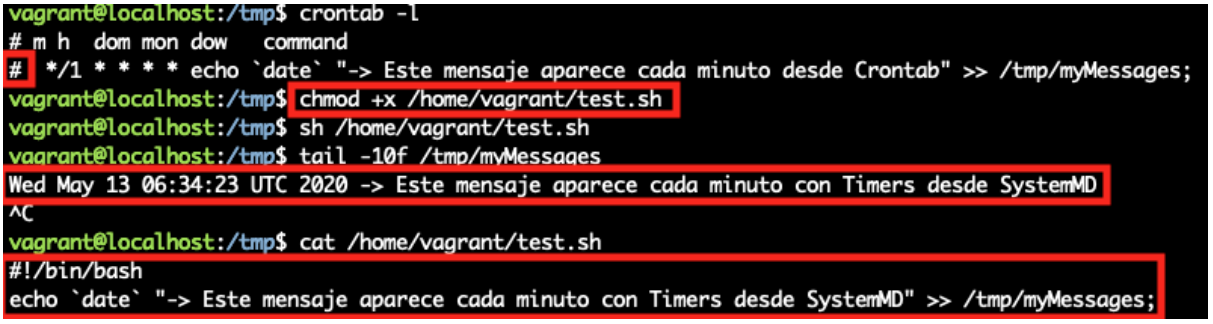
```
vagrant@localhost:/tmp$ crontab -l
# m h  dom mon dow   command
  */1 * * * * echo `date` "-> Este mensaje aparece cada minuto desde Crontab" >> /tmp/myMessages;
vagrant@localhost:/tmp$ tail -10f myMessages
Wed May 13 06:23:01 UTC 2020 -> Este mensaje aparece cada minuto desde Crontab
Wed May 13 06:24:01 UTC 2020 -> Este mensaje aparece cada minuto desde Crontab
```

- Extraemos la tarea del crontab realizando las siguientes acciones:
  - Comentar la tarea con # ejecutando `crontab -u vagrant -e`.
  - Creamos el scrip test.sh con las siguientes lineas:

```
#!/bin/bash
echo `date` "-> Este mensaje aparece cada minuto con Timers desde SystemD" >> /tmp/myMessages;
```

  - Le damos permisos de ejecución con `chmod +x test.sh`
  - Lo ejecutamos y vemos que loga el nuevo mensaje.

Todas estas acciones las podéis ver en la siguiente imagen:



- Creamos el archivo de **test.service** para que ejecute nuestro script en **/etc/systemd/user** con las siguiente líneas:

```
[Unit]
Description=De Crontab a Service de SystemMD con Timers
[Service]
Type=oneshot
ExecStart=/bin/bash /home/vagrant/test.sh
```

- Creamos el archivo de **test.timer** para fijar nuestro timer en **/etc/systemd/user** con las siguiente líneas:

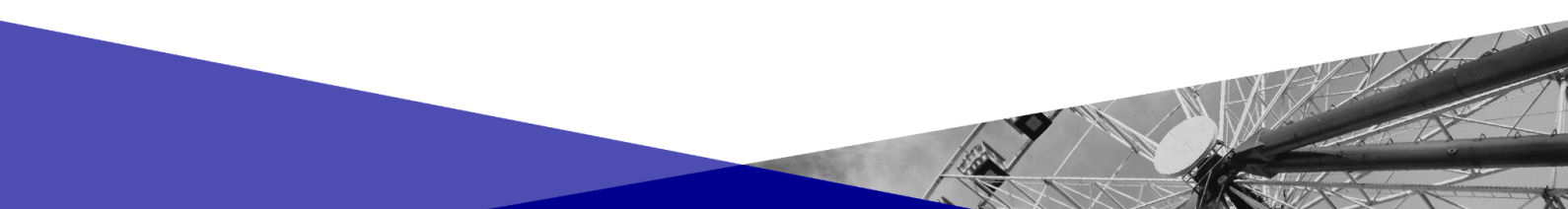
```
[Unit]
Description=Ejecutar cada 1 minutes test.sh
Requires=test.service
[Timer]
OnCalendar=*:0/1
[Install]
WantedBy=timers.target
```

Todos estos pasos los podemos ver en la siguiente imagen:

```
vagrant@localhost:~$ cd /etc/systemd/user/
vagrant@localhost:/etc/systemd/user$ cat test.service
[Unit]
Description=De Crontab a Service de SystemMD con Timers

[Service]
Type=oneshot
ExecStart=/bin/bash /home/vagrant/test.sh
vagrant@localhost:/etc/systemd/user$ cat test.timer

[Unit]
Description=Ejecutar cada 1 minutes test.sh
Requires=test.service

[Timer]
OnCalendar=*:0/1

[Install]
WantedBy=timers.target
```

- Por último, habilitamos el timer para que se arranque como servicio ejecutando `systemctl --user enable test.timer`. Tanto nuestro servicio como nuestro timer funciona con las mismas acciones de start, stop, enable, status, etc., visto en el apartado donde hemos hecho que NGINX sea un servicio.
- Reiniciamos la máquina y vemos que nuestro timer funciona correctamente.

```
vagrant@localhost:/etc/systemd/user$ cat /tmp/myMessages
Wed May 13 07:21:47 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:21:56 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:22:33 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:23:33 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:24:33 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:25:33 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:26:33 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:27:33 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:28:33 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:29:33 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:30:33 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:31:33 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
Wed May 13 07:32:33 UTC 2020 -> Este mensaje aparece cada minuto con Timers desde SystemMD
```



Bash Scripting

Bash es el intérprete de comandos más extendido en distribuciones GNU, el cual ejecuta comandos y devuelve los resultados. También puede recibir un fichero como entrada llamado script. Un script no es más que una secuencia de comandos con cierta lógica, permitiendo la ejecución automática en lugar de ejecutarlo de forma interactiva en el terminal uno a uno.

Para sustituir esta interactividad, Bash Scripting nos provee de sentencias de control como if, while, case, for, etc., que nos permiten tomar decisiones sobre los resultados obtenidos por los comandos, pudiendo ejecutarlo de forma automática y sin necesidad de interacción por el usuario.

En la primera línea de un script siempre aparece algo similar a **#!/bin/bash**. En esta línea se establece el intérprete que se va a utilizar. En este apartado siempre utilizaremos `/bin/bash` pero pueden ser otro:

- **#!/bin/sh**: utiliza la shell del sistema.
- **#!/usr/bin/env python**: utiliza el intérprete de Python.
- **#!/bin/false:** hace que el script no haga nada.

Ojo que aunque pongamos en el script el uso de un intérprete, por ejemplo **#!/bin/bash**, si luego ejecutamos el script con otro (**/bin/sh**), por ejemplo `sh test.sh`, prevalece el intérprete con el que estamos intentando ejecutarlo (en este caso **/bin/sh**) pudiendo provocar errores e incompatibilidades.

Variables

Una variable es la forma que tenemos de almacenar temporalmente información. En nuestro caso, suelen ser los resultados devueltos al ejecutar comandos. El contenido de una variable puede establecerse o leerse.

```
#!/bin/bash
# Escribimos en la variable archivos el número de ficheros en la ruta
# del usuario usando el comando wc
archivos=$(find $HOME -maxdepth 1 -type f | wc -l)
# Leemos e imprimimos el contenido de las variables $HOME y $archivos
echo "En $HOME hay $archivos archivos"
```

Como podéis ver en este ejemplo, establecemos el número de ficheros que tenemos en la carpeta **`/home/vagrant`** usando el comando **`wc`** para después leerlo e imprimirlo por pantalla.

Vemos que para leer el contenido de una variable usamos el **`$`** junto al nombre de la variable **`$varName`**. No confundir con **`$(commands)`**, el cual ejecuta los comandos dentro de los paréntesis y mostrando el contenido. De esta manera, hemos establecido el contenido de la variable **`archivos`**.

A continuación, se muestra un ejemplo que devuelve el mismo resultado:

```
echo "En $HOME hay $archivos archivos"
echo "En $(echo $HOME) hay $(find $HOME -maxdepth 1 -type f | wc -l)
archivos"
```

Según lo dicho anteriormente, la variable **`HOME`** la estás leyendo pero, ¿dónde la has declarado?

En bash existen unas variables especiales que nos proveen de información útil como (número de argumentos de script, el valor de los argumentos, el id de proceso, etc.). Algunas de estas variables son:

VARIABLE ESPECIAL	CONTENIDO
$#	Número de argumentos.
$?	Salida del último comando ejecutado (0 es OK).
$$	El PID de la shell que se está ejecutando.
$@	Listado de los argumentos separados por espacios.
$0 - $9	Valor de los argumentos del 1 al 9. El argumento $0 es el nombre del script.
$*	Listado de los argumentos separados por lo establecido en la variable IFS (si no se establece, se comporta como $@).

También existen lo que llamamos **shell variables**, otro conjunto de variables a las que podemos acceder desde el script. Algunos ejemplos son:

SHELL VARIABLE	CONTENIDO
$RANDOM	Variable que contiene un número aleatorio cada vez que se consulte.
$PWD	Variable que contiene la ruta actual.
$BASHPID	El PID de la shell que se está ejecutando. Equivalentes al $$.

Para obtener un listado completo de estas variables ejecutamos el comando `declare -p`.

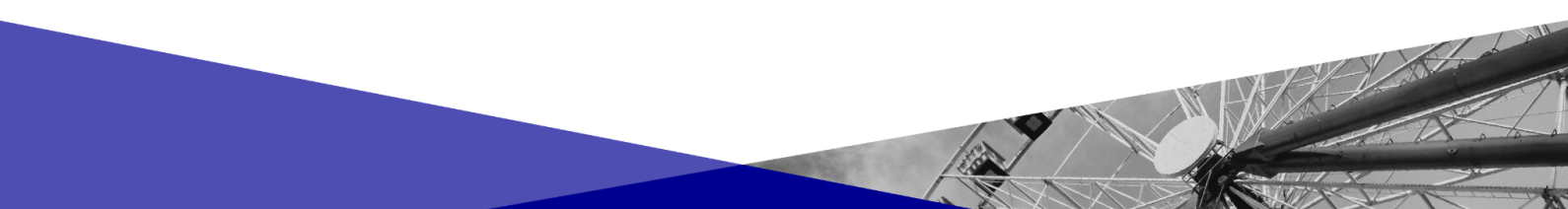
La variable $HOME se puede ver como una shell variable o una variable de entorno. Las variables de entorno son valores dinámicos que el sistema operativo y otros programas utilizan para determinar una información específica. Algunos ejemplos serían:

ENVIRONMENT VARIABLE	CONTENIDO
$HOME	Carpeta Home del usuario logado.
$LOGNAME	Usuario logado.
$SHELL	Shell establecida en el sistema.

Para obtener un listado de las variables de entorno establecidas, podemos ejecutar los comandos `env` o `printenv`. Las variables de entorno se pueden utilizar en los scripts e implementar diferentes flujos en función de su valor. En cualquier momento se pueden modificar.

Para añadir o sobrescribir las variables de entorno se usa el comando *`export`*. En este ejemplo vemos cómo se añade la ruta /misBinarios a la variable $PATH.

```
export $PATH = $PATH:/misBinarios;
```



Sentencias condicionales

If

La primera sentencia condicional que se aprende en todos los lenguajes de programación es el if y en Bash Scripting no va a ser una excepción. En el siguiente ejemplo, podemos ver un código para determinar si el número introducido es par o impar donde se hace uso de la sentencia if-else.

```
#!/bin/bash
echo -n "Introduce un número: "
read num
if [[ $num%2 -eq 0 ]]
then
  echo "El número $num es par"
else
  echo "El número $num es impar"
fi
```

En bash es importante el uso correcto de las tabulaciones para separar las sentencias contenidas en la sección del `if` o la sección del `else`. También podemos hacer if concatenados usando `elif`.

Case

Una alternativa a los `if` anidados es el uso de `case`. Se suelen usar para la construcción de **menús interactivos** en bash. Un ejemplo podría ser el siguiente:

```
  #!/bin/bash
  echo -n "Sabes usar case en bash?(s o SÍ/n o NO): "
  read opcion
  case $opcion in
      s|SI)
         echo "pulso la opción SÍ"
      ;;
      n|NO)
         echo "pulso la opción NO"
      ;;
      *)
         echo "desconozco esa opción"
      ;;
  esac
```

Como ves, la opción por defecto se especifica con `*)` y `;;` para terminar la ejecución de cada caso (equivalente a break en otros lenguajes).

Operadores

En el anterior apartado hemos visto el uso del operador -eq para validar si el número introducido tenía resto 0. -eq es uno de los muchos operadores ofrecidos por bash. Alguno de ellos son:

OPERADOR	CONTENIDO
-n $n	Devuelve true si la longitud del contenido de la variable es mayor a 0.
-z $n	Devuelve true la variable está vacía.
-eq, -nq, -gt. -ge, -lt, -le	Son la equivalencia a (==, !=, >, >=, <, <=) respectivamente y son intercambiables. Ej. $num1 -eq $num2
-h FILE	Devuelve true si el fichero tiene un enlace simbólico.
-e FILE	Devuelve true si el fichero existe (fichero, directorio, etc.).
-r, -w, -x	Devuelve true si el fichero se puede leer, escribir o ejecutar respectivamente. Ej. -r FILE

Bucles

En este apartado mostramos ejemplos de cómo codificar los bucles **while**, **until** y **for** en bash. En los siguientes ejemplos veremos un script en bash donde se le pasa por parámetro los días de la semana y va iterando e imprimiéndolos. Mostramos las diferentes implementaciones con cada bucle.

While

El bucle while itera mientras la condición se cumpla.

```
#!/bin/bash
# Ejecutar bash test.sh Lunes Martes Miercoles Jueves Viernes    #
Sabado Domingo
```

```
week=( "$@" );
echo "Los días de la semana son ${week[@]}"
i=0;
while [[ $i -lt $# ]]
do
 echo "Hoy es ${week[$i]}" && i=$[i+1];
done
```

Until

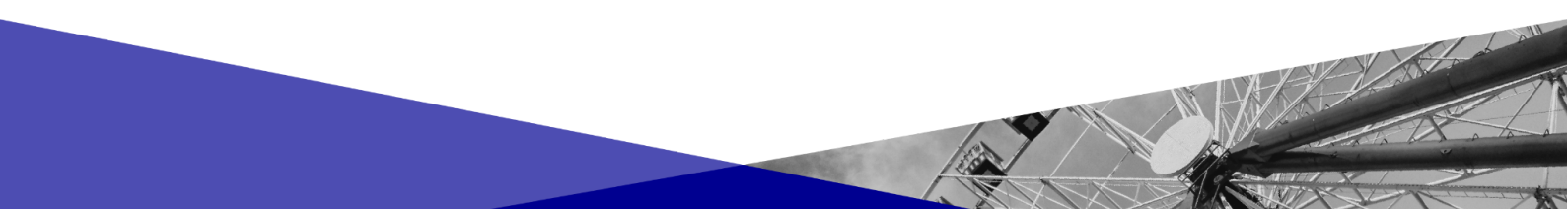
El bucle until itera hasta que la condición no se cumpla. Si te fijas es el mismo código que en el ejemplo anterior pero con la lógica negada.

```
#!/bin/bash
# Ejecutar bash test.sh Lunes Martes Miercoles Jueves Viernes
# Sabado Domingo
week=( "$@" );
echo "Los días de la semana son ${week[@]}"
i=0;
until [[ $i -ge $# ]]
do
 echo "Hoy es ${week[$i]}" && i=$[i+1];
done
```

For

La implementación del bucle for es mucho más sencilla ya que nos permite iterar sobre la lista de argumentos sin necesidad de gestionar el índice.

```
#!/bin/bash
# Ejecutar bash test.sh Lunes Martes Miercoles Jueves Viernes    #
Sabado Domingo
echo "Los días de la semana son $@"
for day in "$@"
do
    echo "Hoy es $day"
done
```



Manejo de ficheros

En este apartado nos centraremos en diferentes operaciones que podemos realizar con ficheros en bash como pueden ser leer, escribir o añadir líneas a un fichero. Para simplificar los ejemplos, no haremos validaciones sobre la existencia de los ficheros con los operadores vistos en anteriores apartados como (-d, -e, -f) y supondremos que el fichero a leer, escribir o modificar no provoca errores.

Leer un fichero

Para explicar cómo leer un fichero en bash haremos comentarios sobre una implementación que convierte un fichero de varias líneas en una sola:

```
#!/bin/bash
# Ejecutar bash test.sh file.txt
declare -a ARRAY
let count=0
# Utilizamos la variable IFS para dividir el fichero por líneas
# Se utiliza el carácter de nueva línea como separador
while IFS = read -r line
do
 ARRAY[$count]=$line
 ((count++))
done <<< $(cat $1)
echo 'El contenido en una línea del fichero es >>' ${ARRAY[@]}
```

Si te fijas, leemos el fichero teniendo como entrada, la salida del comando `cat $1` (primer parámetro que representa el fichero a leer).

En la siguiente imagen podemos ver el contenido del fichero file.txt y el resultado al pasarlo al script como entrada.

```
vagrant@localhost:~$ cat file.txt
Ya
se
leer
ficheros
en
bash
!!!
vagrant@localhost:~$ ./test.sh ./file.txt
El contenido en una lina del fichero es >> Ya se leer ficheros en bash !!!
vagrant@localhost:~$
```

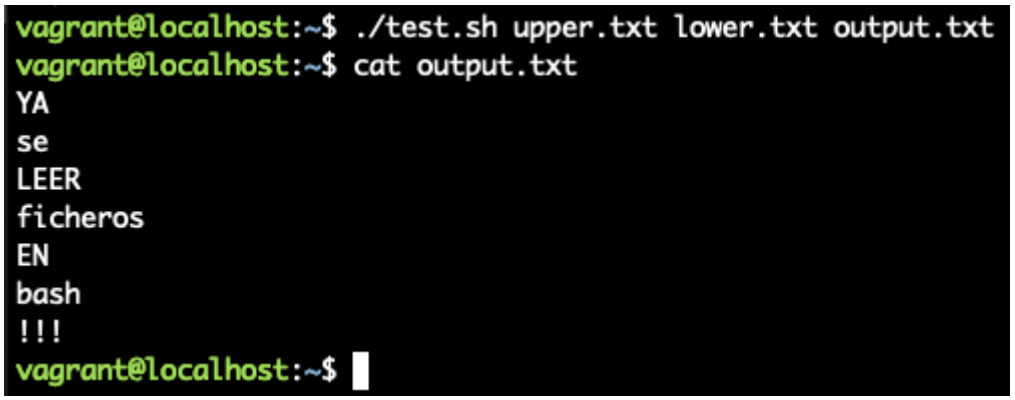
Escribir un fichero

Para ver cómo se escribe en un fichero en bash, vamos a modificar la implementación anterior para que acepte un segundo parámetro que represente el nombre de fichero de salida.

El objetivo de esta implementación es, en base a un fichero de entrada, generar un fichero de salida que cambie todas las vocales por i. Teniendo esto en cuenta, la implementación es la siguiente:

```
#!/bin/bash
# Ejecutar bash test.sh file.txt output.txt
declare -a ARRAY
let count=0
while IFS= read -r line
do
 ARRAY[$count]=$(echo $line | tr 'aeou' 'i')
 ((count++))
done <<< $(cat $1)
output_file=$2
echo "*** Este fichero ha sido generado por NINI-GENERATOR ***" >>
output_file
echo "El fichero ${output_file} transformado en una línea NINI es >>"
${ARRAY[@]} >> output_file
```

También podríamos querer leer varios ficheros a la vez para generar un solo fichero. Si mezclásemos los ficheros upper.txt y lower.txt intercalando sus líneas (1 - primera upper, 2 - segunda lower, 3 - tercera upper, etc.), la salida después de ejecutarlo debería ser algo así:



Una posible implementación podría ser la siguiente:

```
#!/bin/bash
# Ejecutar bash test.sh file.txt output.txt
let count=1
while IFS= read -u3 -r a && IFS= read -u4 -r b;
do
 if [[ $count%2 -ne 0 ]]
 then
  echo "${a}"
 else
  echo "${b}"
 fi
 ((count++))
done 3<$1 4<$2
```

En cada función read leemos el identificador de fichero y en el done asignamos cada parámetro a su identificador.

Modificar un fichero

Para modificar ficheros, normalmente, utilizamos un editor. Pero hay tareas, como aplicar una transformación a un fichero, por ejemplo, hacer que las líneas pares se escriban en mayúsculas, para las cuales puede ser costoso utilizarlos. Para este tipo de tareas, awk es nuestro aliado. La sentencia para hacerlo sería la siguiente:

```
echo "$(awk '{if (NR % 2 == 0){print toupper($0)}else{print $0}}'
file.txt)" > file.txt
```

El resultado del comando sería:

Un ejemplo de reemplazo podría ser cambiar la palabra 'ficheros' por 'archivos' utilizando awk:

```
echo "$(sed '4s/ficheros/archivos/g' file.txt)" > file.txt
```

Con esta sentencia reemplazamos la palabra ficheros de la línea 4 por la palabra archivos. Aunque se podría reemplazar toda la línea en un script de bash. Si quieres conocer más sobre awk puedes consultar esta página. También es interesante esta web, donde aparecen 30 usos de awk.

En bash se suele hacer uso de ficheros temporales y luego sobreescribir el fichero original:

```
echo "$(sed '4s/ficheros/archivos/g' file.txt)" > tmp.txt && mv tmp.txt
file.txt
```

Si quieres conocer más sobre el comando sed, consulta linuxconfig.org.

Funciones

Para mostrar un ejemplo de uso de funciones en bash vamos a implementar un script que tras introducir un número de DNI el usuario, le calcule la letra.

```
#!/bin/bash
  function get_DNI_letter(){
   local my_array=(T R W A G M Y F P D X B N J Z S Q V H L C K E)
   local index=$(($1%23))
   echo ${my_array[$index]}
  }
```

```
echo -n "Introduce el número de tu DNI (solo números): "
read num
printf "Tu DNI completo es %s - %s\n" "$num" "$(get_DNI_letter num)"
```

Las funciones en bash tienen varias peculiaridades:

- Las funciones se suelen declarar con la cláusula **function**, aunque se puede omitir y solo indicar el nombre de la función. Nosotros la utilizaremos para mayor legibilidad.
- En las funciones no se especifica el tipo de datos devuelto, ni el número de parámetros en la firma. Los parámetros de la función se establecen con `$n` como hemos visto en apartados anteriores ($1, $2, etc.).
- En las funciones podemos hacer uso de variables locales con `local my_var` (el ámbito de las variables se restringe a la función).
- Tenemos diferentes formas de devolver valores en una función en bash:
    - Si se usa la cláusula **return,** tenemos que devolver un entero entre 0 y 255. Bash entiende que cualquier entero entre 1 y 255 que devuelva la función es un código de error, siendo 0 el código para indicar un correcto funcionamiento de la función
    - Estableciendo el valor de una variable global. A las variables globales se tiene acceso en cualquier momento dentro del script. En todos los ejemplos hasta este apartado se ha hecho uso de variables globales. Una forma de devolver un valor en una función es establecerlo en una variable global.
    - Hacer un `echo "$my_var"` de una variable local de la función.
    - Utilizar el operador `$?` para recuperar el valor devuelto del último comando ejecutado.

```
my_function(){
    return 5
}
my_function
echo $?
```

En el siguiente apartado veremos un ejemplo que pone en práctica cada una de las peculiaridades aquí expuestas.

Control de Errores

Imaginemos que nos piden mejorar el programa anterior usando funciones y validando los datos introducidos, sacando un mensaje de error, si son erróneos. Asentado todo esto, nos piden que el programa tenga los siguiente requisitos con respecto al control de errores:

- La función que obtenga la letra del DNI tiene que devolver 0 si su salida es correcta (devolviendo la letra calculada) y otro número en cualquier otro caso.
- El programa principal tiene que devolver un mensaje descriptivo que indique el código de error obtenido.
- Se debe solicitar un nuevo DNI cada vez, a no ser que interrumpa la ejecución pulsando CTRL+C, en cuyo caso, se debe mostrar un mensaje indicativo de que esta combinación ha sido pulsada.

Una posible implementación cumpliendo los requisitos anteriores podría ser la siguiente:

```
#!/bin/bash
# VARIABLES GLOBALES
let resultado
# FUNCIONES
  function ctrl_c() {
    printf "\n\n*** Programa interrumpido por el usuario pulsando CTRL-C
***\n"
    exit 0
  }
  function get_DNI_letter(){
    local my_array=(T R W A G M Y F P D X B N J Z S Q V H L C K E)
    if [[ $1 =~ ^[0-9]{8}+$ ]]
    then
     local index=$(($1%23))
     resultado=$(echo ${my_array[$index]})
     return 0
    else
     return 1
    fi
  }
  function show_DNI(){
    get_DNI_letter $1
    if [[ $? -eq 0 ]]
    then
```

```
     printf "Tu DNI completo es %s - %s\n" "$1" "$resultado"
    else
     printf "El DNI introducido es inválido, se esperaba un número de
longitud 8"
    fi
  }
```

```
#PROGRAMA PRINCIPAL
trap ctrl_c INT
while  true;
do
  echo -n "Introduce en número de tu DNI (solo números): "
  read dni
  show_DNI $dni
  printf "\n\n*** Presiona cualquier tecla para introducir otro DNI ***"
  read dni
  clear
done
```

Queremos que se preste atención en los siguientes puntos de la implementación:

- Se han creado varias funciones:
    - **ctrl_c**: función que controla cuando el usuario pulsa CTRL+C, sacando un mensaje descriptivo y saliéndose de forma controlada de la ejecución del programa. Normalmente, en estas funciones se borran ficheros generados por el programa, se cierran sockets, etc. para salir de manera controlada. En nuestro caso siempre devolvemos 0 porque no propagamos el código de estado de la función **get_DNI_letter** ya que es controlado por la función **show_DNI**.
    - **get_DNI_letter**: función que dado el número de DNI (8 dígitos) calcula la letra que le corresponde. Esta función devuelve el código de estado (0, almacenando la letra en la variable global resultado y 1 si no).
    - **show_DNI**: Función que muestra un mensaje representativo en base al código de error devuelto por **get_DNI_letter**. Para recuperar el código de error, se hace uso del operador $?.
- Utilizamos variables globales porque necesitamos que la función **get_DNI_letter** tenga 2 salidas, la letra calculada y el código de

estado.

- Hacemos uso de la sentencia trap y capturamos la señal de interrupción (INT) y ejecutamos la función de **ctrl_c** que nos permite parar la ejecución de manera limpia (borrando o cerrando todo lo necesario) y controlada.
- Para realizar la validación de los datos introducidos por el usuario utilizamos expresiones regulares.

Control de excepciones

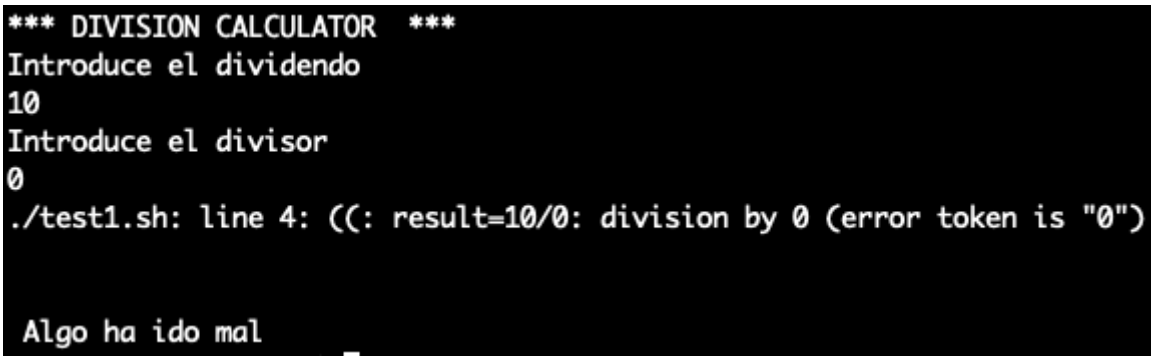
Bash nos ofrece diferentes formas de controlar los errores. Una de ellas ya la hemos visto en apartados anteriores, que es consultar el operador $? para obtener el código de estado de la última sentencia ejecutada. Un ejemplo donde usamos este operador es en la siguiente función que hace una división después de pedirle el dividendo y el divisor al usuario. Una posible implementación podría ser la siguiente:

```
#!/bin/bash
  let result
  function calculate_division(){
     ((result=$1/$2)) 2>&-
     show_division $1 $2
  }
  function show_division(){
    if [[ $? -eq 0 ]]
    then
     printf "\n\n El resultado de dividir %s entre %s es %s \n" "$1"
"$2"  "$result"
    else
     printf "\n\n Algo ha ido mal\n"
    fi
  }
printf "\n\n*** DIVISION CALCULATOR  ***\n"
printf "Introduce el dividendo\n"
read  dividend
printf "Introduce el divisor\n"
read  divider
calculate_division $dividend $divider
```

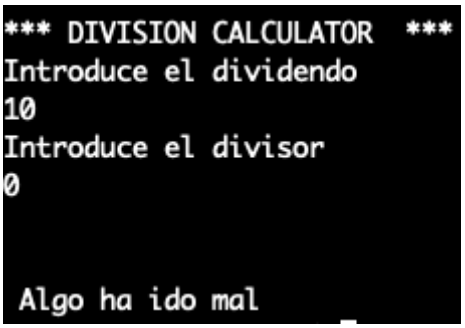
Si te fijas, en la función **calculate_division** es donde se realiza la operación de división en sí. Lo hace con las siguientes sentencias:

```
    ((result=$1/$2)) 2>&-
    show_division $1 $2
```

Asignamos el resultado de la división a la variable global **result**. Para evitar que posibles errores (como por ejemplo, una división por 0), aparezcan en la pantalla al realizar la división, redirigimos la salida de errores (strderr) hacia **ningún** identificador de fichero **&-** (es equivalente a redirigir a /dev/null). De esta manera, descartamos o desechamos todos los mensajes de error que se produzcan. Si no incluyéramos esta redirección, en una ejecución donde se produjera una división por 0 aparecería lo siguiente:



Y si redirigimos la salida de error al realizar la operación con `((result=$1/$2)) 2>&-` o `((result=$1/$2)) 2>/dev/null` aparecera asi:



Otra manera de controlar errores es utilizar los operadores || y &&. Estos operadores nos permiten concatenar comandos y ejecutarlos dependiendo del código de estado del comando que le precede.

- CMD1 && CMD2: si el CMD1 falla, el CMD2 no se intentará ejecutar (equivalente al operador AND en otros lenguajes de programación pero con comandos).
- CMD1 || CMD2: si el CMD1 tiene éxito, el CMD2 no se intentará ejecutar (equivalente al operador OR en otros lenguajes de programación pero con comandos).

En el siguiente script de bash podéis ver de una forma más explícita las diferentes combinaciones que se pueden dar con estos operadores:

```
#!/bin/bash
echo "CMD: true && echo 'Siempre se mostrará'  EXEC: " && (true && echo
"Siempre se mostrará")
echo "CMD: false && echo 'Nunca se mostrará'  EXEC: " && (false && echo
"Nunca se mostrará")
echo "CMD: true || echo 'Nunca se mostrará'  EXEC: " && (true || echo
"Nunca se mostrará")
echo "CMD: false || echo 'Siempre se mostrará'  EXEC: " && (false ||
echo "Siempre se mostrará")
```

Estos operadores se pueden usar para concatenar operaciones a realizar y la función que controle el error. Una implementación que usan estos operadores para hacer un script que ejecute un LS de forma segura podría ser:

```
#!/bin/bash
function error_catch(){
 echo "Algo fue mal"
}
if [ $# == 1 ]; then
  (ls $1 2>/dev/null) || error_catch
fi
```

De esta manera, si nos pasan por parámetro una ruta que no existe, el script lanzará la función **error_catch**. En la siguiente imagen podemos ver una ejecución del script con una ruta que existe y otra que no:

El último método para controlar los errores en bash es usar la sentencia trap para capturar los errores en el script. Ya hemos utilizado esta sentencia para capturar la señal de interrupción al pulsar CTRL+C en anteriores apartados. Un ejemplo de implementación que use la sentencia trap para capturar los errores podría ser el siguiente:

```
#!/bin/bash
set -e
function error_catch(){
```

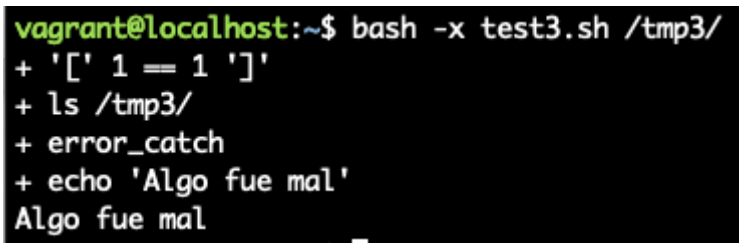
```
 echo "Algo fue mal"
}
trap "error_catch" ERR
if [ $# == 1 ]; then
  ls $1 2>&-
fi
```

Una peculiaridad que tiene esta implementación que no tienen las anteriores, es el uso de la sentencia set (nos permite habilitar/deshabilitar opciones y parámetros en shell). Alguno de los parámetros más utilizados de set son:

- **set -e**: interrumpir la ejecución si se produce algún error, evitando ejecutar las demás sentencias.
- **set +u**: no provocará error si la variable no está definida.
- **set -x**: habilitar el modo debug en el script. Nunca usar en producción.

Si queréis conocer más opciones que se pueden establecer, podéis consultar linuxcommand.org.

También podemos establecer estos valores incluyéndolos como parámetros en la forma de ejecutar el script. Si quisiéramos ejecutar la última implementación en modo debug para ver por dónde pasa cuando le pasamos una ruta que no existe, ejecutariamos `bash -x test3.sh /tmp3/`, lo que nos genera la siguiente salida:



Utilidades esenciales en UNIX

En este último apartado os daremos un listado de los comandos de UNIX que consideramos que es importante que conozca cualquier persona inmersa en la cultura DevOps o un SysAdmin.

Comandos de red

Ping

Ping es el comando que utilizamos para saber si tenemos conectividad con un servidor, host o puerta de enlace. Algunos usos que le podemos dar a ping son:

- `ping -i 5 www.google.es`: enviar a Google un paquete cada 5 segundos.
- `ping -c 5 www.google.es`: enviar a Google solo 5 paquetes.
- `ping -R 10.0.2.255`: imprime la ruta por la cual se envía y recibe el paquete. Hemos puesto la puerta de enlace (obtenida con ip addr) para que veamos al menos dos ip por las que pasa (la puerta de enlace y tu ip).

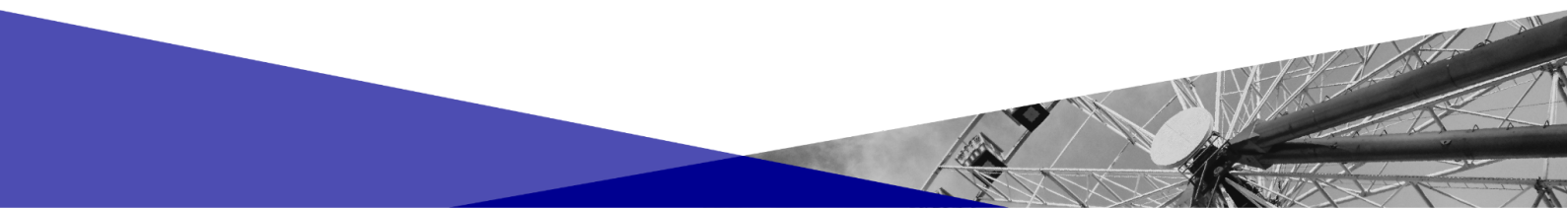
Si quieres conocer más usos que podemos darle al comando ping, puedes visitar este tutorial.

Traceroute

Traceroute nos muestra las rutas ip y RTT (Round Trip Times, tiempo de ida y vuelta del paquete enviado). Por defecto, traceroute envía tres paquetes para hacer la medición del tiempo de ida y vuelta; el tiempo obtenido en la medición de cada uno de estos paquetes se corresponde con una columna de RTT (RTT1, RTT2, RTT3). El esquema de la salida de traceroute seria:

Hop RTT1 RTT2 RTT3 Domain Name [IP Address]

Un ejemplo de salida podría ser el siguiente:



```
root@localhost:~# traceroute www.google.es
traceroute to www.google.es (216.58.211.227), 30 hops max, 60 byte packets
 1  _gateway (10.0.2.2)  0.098 ms  0.067 ms  0.126 ms
 2  192.168.0.1 (192.168.0.1)  0.693 ms  0.757 ms  0.682 ms
 3  10.195.72.1 (10.195.72.1)  15.865 ms  15.811 ms *
```

Los * en traceroute significan que no se ha recibido respuesta del paquete enviado o que éste ha tardado más de 4 segundos.

Algunas de las opciones que ofrece traceroute son:

- `traceroute -n www.google.es`: limitar el uso de DNS para que no afecte en la medición.
- `traceroute -q 1 www.google.es`: aumentar/disminuir el número de mediciones o columnas de RTT. Por defecto 3.
- `traceroute -m 10 www.google.es`: establecer el número máximo de saltos para llegar a la ip destino.
- `traceroute www.google.es 1000`: fijar el tamaño de byte del paquete enviado.

Si queréis saber más sobre traceroute, os recomiendo mirar la ayuda del comando con `traceroute --helptraceroute --help` o visitar esta página.

Algunos de los problemas que nos ayuda a diagnosticar son:

- Problemas de latencia.
- Problemas de configuración en firewall.
- Detectar bucles en las rutas.
- Detectar problemas de conectividad entre redes.

SCP

SCP (Secure Copy) es un comando que se utiliza para copiar ficheros a servidores remotos de manera segura.

Si queremos probar SCP se deben cumplir los siguientes requisitos:

- Conocer las credenciales de acceso por ssh a la máquina virtual levantada por Vagrant (usuario: vagrant pass: vagrant).
- Establecer una red privada entre el Host (nuestro Mac) y el Guest (la MV). Para ello, tenemos que incluir la siguiente línea en el Vagrantfile **config.vm.network "private_network", ip: "192.168.33.10"**

- Crear los directorios y ficheros que se vayan a transferir antes de ejecutar los comandos. En nuestro caso, transferimos **/tmp/vagrantFolder** a **/tmp/hostFolder** junto con todos los ficheros que creemos para las pruebas.

Algunos de las opciones que ofrece scp son:

- Copiar /tmp/vagrantFolder/file de vagrant a /tmp/hostFolder/ del host (nuestro mac). Ejecutaremos el siguiente comando desde una terminal del host:

```
scp vagrant@192.168.33.10:/tmp/vagrantFolder/file /tmp/hostFolder
```

- Si quisiéramos hacerlo en sentido contrario (del host a la MV) ejecutamos el siguiente comando desde una terminal del Host:

```
scp /tmp/hostFolder/file vagrant@192.168.33.10:/tmp/vagrantFolder
```

- Si quisiéramos copiar todo un directorio (por ejemplo hostFolder) del host a vagrant, lo haríamos con el siguiente comando:

```
scp -r /tmp/hostFolder vagrant@192.168.33.10:/tmp
```

Como hemos visto en los comandos anteriores, si se quiere invertir la dirección de transferencia solo es necesario intercambiar las rutas.

- Ahora necesitamos transferir varios ficheros de la máquina host a la máquina virtual. Lo hacemos con el siguiente comando:

```
scp /tmp/hostFolder/file1 file2 file3
vagrant@192.168.33.10:/tmp/vagrantFolder
```

- Su homólogo para transferir un listado de directorios sería:

```
scp -r/tmp/hostFolder/dir1 dir2 dir3
vagrant@192.168.33.10:/tmp/vagrantFolder
```

Una forma de mejorar el rendimiento a la hora de transferir un fichero de un host remoto es fijar otro algoritmos de cifrado al de por defecto (lo hemos obtenido con la opción **-v** al ejecutar un scp sin la opción **-c**).

Hemos hecho un script en bash para realizar una comparativa de tiempo al realizar una transferencia de un fichero de 1 GB a la máquina virtual. El código del script es el siguiente:

```
#!/bin/bash
  function ctrl_c() {
    printf "\n\n*** Programa interrumpido por el usuario pulsando CTRL-C
***\n"
```

```
    exit 0
  }
trap ctrl_c INT
for i in "chacha20-poly1305@openssh.com" "aes128-ctr" "aes192-ct
do
   dd if=/dev/zero of=/tmp/file  bs=1m  count=1000 &> /dev/null
   scp -vCc $i /tmp/file vagrant@192.168.33.10:/tmp/ &> /tmp/scp.log
   cmd=$(eval "grep -i Transferred /tmp/scp.log | egrep -o 'in
[0-9]+.[0-9]+ seconds' | egrep -o '[0-9]+.[0-9]+'")
   echo "La tasa de transferencia con $i ha tardado $cmd segundos"
   rm /tmp/scp.log
done
```

Para que este script funcione de manera no interactiva y no solicite la password por cada prueba, es necesario realizar la autenticación por clave pública sin establecer passphrase como se ha explicado en apartados anteriores. La salida que hemos obtenido del script es la siguiente:

```
La tasa de transferencia con chacha20-poly1305@openssh.com ha tardado 26.0 segundos
La tasa de transferencia con aes128-ctr ha tardado 19.7 segundos
La tasa de transferencia con aes192-ctr ha tardado 24.0 segundos
La tasa de transferencia con aes256-ctr ha tardado 21.9 segundos
La tasa de transferencia con aes128-gcm@openssh.com ha tardado 16.9 segundos
La tasa de transferencia con aes256-gcm@openssh.com ha tardado 21.0 segundos
```

A la vista de los resultados, vemos que podemos bajar hasta 10 segundos si seleccionamos aes128-gc@openssh.com frente al de por defecto (chacha20-poly305@openssh.com).

Esta parte surge porque en algunos blogs indican la utilización del cifrado blowfish-cbc, que mejora hasta 10 veces la velocidad de transferencia con respecto al cifrado por defecto que había antes (3des-cbc).

La lista de cifrados que comparten entre el host y la máquina virtual la hemos obtenido intentando hacer la transferencia de ficheros con un cifrado no soportado como es 3des-cbc.

```
➜ ./test.sh
Generando fichero para transferirlo por el cifrado 3des-cbc
2400+0 records in
2400+0 records out
2516582400 bytes transferred in 2.419960 secs (1039927262 bytes/sec)
Unable to negotiate with 192.168.33.10 port 22: no matching cipher found. Their offer: chacha20-poly1305@openssh.com,aes128-ctr,aes192-ctr,aes256-ctr,aes128-gcm@openss
h.com,aes256-gcm@openssh.com
lost connection
```

Para obtener el listado de cifrados soportados ejecutamos `ssh -Q cipher` o editamos el fichero **/etc/ssh/ssh_config**

Port Forwarding

El port forwarding es una técnica que permite la reasignación de puertos. Al principio de este documento hicimos un port forwarding redirigiendo el puerto 8080 de la máquina virtual al puerto 80 del host.

Para este ejemplo vamos a cambiar el puerto para conectarnos por ssh (puerto 22) sin modificar la propiedad **port** en el fichero de configuración **/etc/ssh/sshd_config**. Para hacerlo, añadiremos nuevas reglas al filtro de NAT de iptables.

Iptables es un módulo del núcleo de Linux que se encarga de filtrar los paquetes de red, determinando cuáles llegan al servidor y cuáles no.

La tabla NAT permite definir reglas de comunicación entre la red externa (Internet) y nuestra red interna.

Existen diferentes tipos de reglas para NAT en iptables:

- **PREROUTING**: reglas que modifican la dirección o el puerto de los paquetes nada más llegar al equipo.
- **OUTPUT**: reglas que aplican a los paquetes que van a ser enrutados como salida.
- **POSTROUTING**: reglas que aplican a los paquetes que ya van a salir.

Para hacer nuestro Port Forwarding del puerto 22 al 3333 incluiremos reglas de redireccionamiento en las 2 primeras. Las reglas que tenemos que incluir en el NAT de iptables son las siguientes:

- Para incluir una regla en PREROUTING para redirigir todos los paquetes TCP que entran por el puerto 3333 al puerto 22 debemos ejecutar este comando:

```
sudo iptables -t nat -I PREROUTING -p tcp --dport 3333 -j REDIRECT
--to-ports 22
```

- Para incluir una regla en OUTPUT para redirigir todos los paquetes TCP de la interfaz (localhost) que salgan por el puerto 3333 al puerto 22 debemos ejecutar este comando:

```
sudo iptables -t nat -I OUTPUT -p tcp -o lo --dport 3333 -j REDIRECT
--to-ports 22
```

Si ejecutamos un nmap para ver los puertos abiertos dentro de la máquina Vagrant antes de incluir la reglas en el iptables podemos ver lo siguiente:

```
root@localhost:~# nmap localhost

Starting Nmap 7.60 ( https://nmap.org ) at 2020-05-22 08:09 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000050s latency).
Other addresses for localhost (not scanned): ::1 127.0.1.1
Not shown: 999 closed ports
PORT   STATE SERVICE
22/tcp open  ssh

Nmap done: 1 IP address (1 host up) scanned in 1.60 seconds
```

Si lo hacemos después de incluir las reglas, podemos ver que ya reconoce el puerto 3333 como un puerto abierto.

```
root@localhost:~# nmap localhost

Starting Nmap 7.60 ( https://nmap.org ) at 2020-05-22 08:11 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000060s latency).
Other addresses for localhost (not scanned): ::1 127.0.1.1
Not shown: 998 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
3333/tcp open  dec-notes
```

Nos conectamos por ssh usando el puerto 3333 para comprobar que nuestro port forwarding funciona correctamente:

```
▶   ssh vagrant@192.168.33.10 -p 3333
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-99-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

Last login: Fri May 22 06:43:02 2020 from 10.0.2.2
```

nmap

Nmap es una herramienta que nos permite realizar escaneos de puertos abiertos. Para instalarla tenemos que ejecutar `sudo apt-get install nmap -y`. Alguna de las opciones que nos ofrece son las siguientes:

- `nmap localhost -sU`: escaneo de puertos UDP. Por defecto son de TCP SYN.
- `nmap localhost -sV`: nos permite conocer las versiones de los puertos abiertos.
- `nmap localhost -p 22,3333`: el parámetro -p nos permite establecer listados o rangos de los puertos a escanear.
- `nmap localhost -A / nmap localhost -O`: nos permite detectar el sistema operativo.

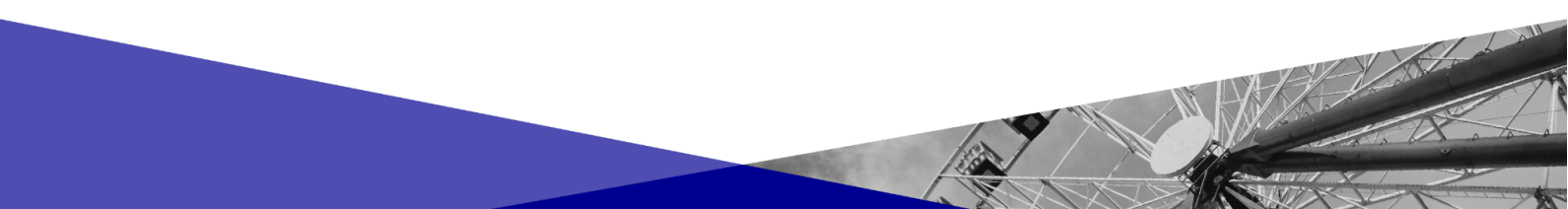
El escaneo de puertos puede tardar bastante. Aquí os damos algunas de las opciones que podéis incluir para evitarlo:

- `nmap localhost --top-ports 100`: limitar el rango de puertos a escanear con --top-ports o con la opción -p vista anteriormente.
- `nmap localhost -T5`: hace un escaneo más rápido porque inunda la red de paquetes. Solo es recomendable en entornos controlados y redes rápidas, ya que puede penalizar en su rendimiento.
- `nmap localhost -n`: deshabilitar la resolución DNS.
- `nmap localhost --min-rate 500`: establecer una tasa mínima de envío de paquetes por segundo.

Nmap también nos sirve para detectar posibles vulnerabilidades en los puertos que tenemos expuestos al exterior con el uso de script NSE (Nmap Scripting Engine). Estos script se encuentran en **/usr/share/nmap/scripts**. Un script que podemos utilizar para ver si el puerto 80 (donde tenemos desplegado nuestro apache) es vulnerable, es utilizando el script http-enum con el siguiente comando:

```
nmap --script http-enum -p80 localhost
```

El script hace fuzzing (prueba a acceder a las urls normalmente activas en los servicios a escanear) en el puerto 80 y nos detecta que tenemos habilitada la url /status-server/:



```
root@localhost:/usr/share/nmap/scripts# nmap --script http-enum -p80 localhost

Starting Nmap 7.60 ( https://nmap.org ) at 2020-05-22 09:45 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000040s latency).
Other addresses for localhost (not scanned): ::1 127.0.1.1

PORT   STATE SERVICE
80/tcp open  http
| http-enum:
|_  /server-status/: Potentially interesting folder
```

Para mitigar el problema, lo que hay que hacer es incluir las siguientes líneas que deshabilitan la directiva para evaluar el estado del servidor en el fichero **/etc/apache2/apache2.conf**:

```
<Location /server-status>
 SetHandler server-status
 Order deny,allow
 Deny from all
</Location>
```

Curl

Curl es una herramienta de transferencia de archivos que soporta múltiples formatos: FTP, HTTP, TFTP, SCP, SFTP, Telnet, DICT, FILE y LDAP, entre otros. Tienes que instalarlo con *`apt-get install curl`*. Algunos de los usos que le podemos dar son:

- Descargar un fichero o página: `sudo curl www.google.com --output google.html`
- Hacer la redirección: `sudo curl -IL www.autentia.es`. Con este comando podemos ver que aunque accedemos a www.autentia.es, nos redirige a www.autentia.com.
- Parar una descarga y continuar después:
    - Empezamos a descargar hasta que pulsamos **CTRL+C**

```
sudo curl -L
https://cdimage.debian.org/debian-cd/current/amd64/iso-cd/debian-1
0.4.0-amd64-xfce-CD-1.iso --output cd.iso
```

    - Continuamos descargando:

```
sudo curl -L -O -C -
```

```
https://cdimage.debian.org/debian-cd/current/amd64/iso-cd/debian-10.4.0-amd64-xfce-CD-1.iso --output cd.iso
```

- Logarse: `curl -u user:password ftp://192.168.0.19/`. Cuando uses este comando en pruebas de conectividad, acuérdate de borrarlo del historial (por ejemplo para borrar la línea 12 del historial `history -d 12`), ya que quedarían en claro las credenciales.
- Conectarse mediante proxy: `curl -x 192.168.1.1:8080 http://example.com`
- Utilizat verbos HTTP: `curl -d "param1=value1&param2=value2" -X POST http://localhost:3000/data`

nc

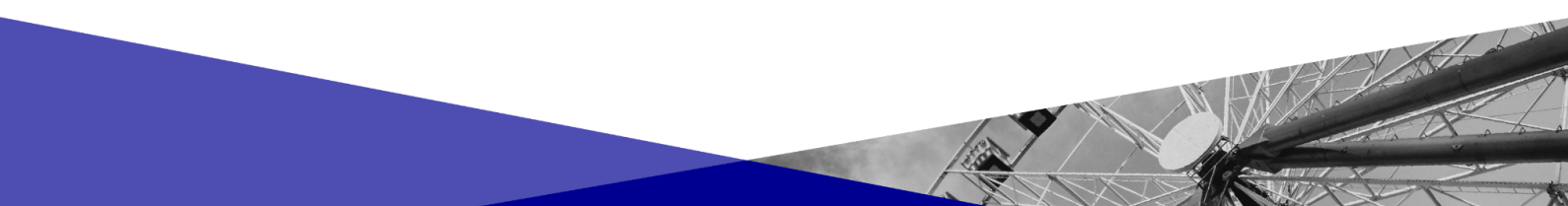
Netcat es una herramienta de red con una sintaxis sencilla que permite abrir puertos, asociar una shell a un puerto en concreto, etc.

Antes de evaluar las diferentes operaciones que podemos realizar con netcat tenemos que realizar los siguientes pasos:

- Desinstalar netcat-openbsd: `sudo apt-get remove --purge netcat-openBSD -y`
- Instalar netcat: `sudo apt-get install netcat -y`

Ahora sí, estas son algunas de la operaciones que nos ofrece:

- Verificar si un puerto está abierto: *`nc -v localhost 80`*
- Escaneo de puertos: `nc -zv localhost 1-6000 2>&1 | grep succeeded`
- Establecer una conexión cliente-servidor:
  - Para el server(Vagrant): `nc -l 4444`
  - Para el cliente(Mac): `nc 192.168.33.10 4444`
- Transferir archivos:
  - Para el server(Vagrant): `nc -l 4444 > myFile`
  - Para el cliente(Mac): `nc 192.168.33.10 4444 < myFile`
- Enlazar un proceso a un puerto:
  - Para el server (Vagrant Terminal 1): `nc -lvp 4444`. Aquí es donde tendremos nuestra reverse shell donde poder ejecutar comandos una vez se asocie el puerto.
  - Para el server (Vagrant Terminal 2): `nc 192.168.33.10 4444 -e /bin/sh`. Asociamos la shell al puerto 4444.



Comandos entrada y salida

wc

Wc (word count) es un comando que nos permite contar palabras en Linux. Es muy utilizado para obtener datos cuantitativos después de aplicar filtros a ficheros o comandos.

`wc -l`**:** contamos el número de líneas del fichero.

`wc -w`**:** contamos el número de palabras del fichero.

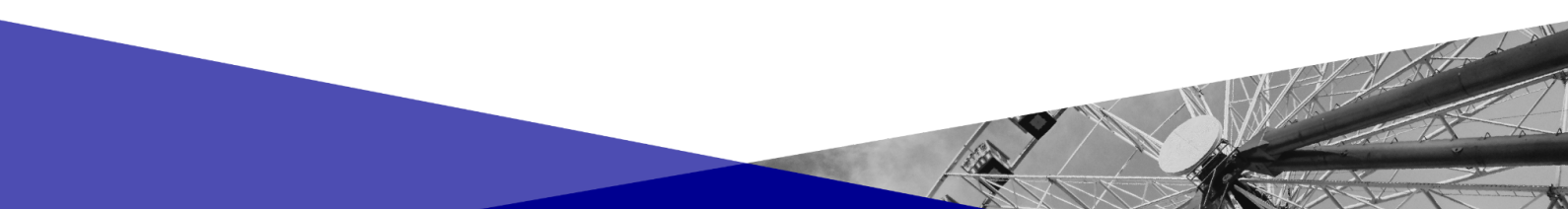
Utilizando uno de los comandos anteriores, si quisiéramos saber el número de puertos abiertos en la máquina, podríamos obtenerlos ejecutando los siguientes comandos encadenados:

```
nc -zv localhost 1-6000 2>&1 | grep open | wc -l
```

Cat, more, less

Cat es la abreviatura de “concatenate”. Uno de los muchos usos de cat es concatenar ficheros o redirigir el contenido del fichero a la salida estándar u otro comando. Con more y less podemos limitar la salida de un fichero cuando no cabe todo su contenido en una página, de esta forma, navegamos a través de él de una manera más fácil y cómoda. Algunos de los usos más utilizados son:

- Mostrar el contenido de un fichero: *`cat /etc/passwd`*
- Concatenar varios ficheros: *`cat file1 file2 > file3`*
- También se podría utilizar wildcards para concatenar varios ficheros: *`cat file* > file3`*
- Crear un fichero: *`cat > file`* (Se escribe el contenido y para guardarlo pulsamos CTRL+D)
- Usar less o more para limitar la salida de cat en ficheros grandes:
    - *`cat file | more`*
    - *`cat file | less`*
- Cat también nos ofrece opciones para mostrar el contenido:
    - *`cat -n file`*: muestra de forma enumerada las líneas del fichero.
    - *`cat -e file`*: muestra **$** en cada final de línea, final de fichero o espacio entre párrafos.



- *`cat -T file`*: muestra **^I** por cada tabulador introducido en el fichero.

Redirecciones

stdin (standard input): 0; **stdout (standard output):** 1; **stderr (standard error):** 2;

A lo largo de este documento ya hemos visto redirecciones cuando redirigimos la salida de stderr a /dev/null. Recordemos que el operador **>** redirecciona la salida al fichero que especifiquemos y sobreescribe su contenido. El operador **>>** hace lo mismo que **>** pero sin sobrescribir los datos. **2>&1** redirecciona stderr a stdout. ¿Por qué se usa &? Porque si solo escribimos 2>1, se entendería como "redirígeme stderr al fichero con nombre 1". Hay un pequeño hilo en Stackoverflow que lo explica.

Algunos usos que le podemos dar a las redirecciones son:

- Redirigir el resultado de comandos hacia un fichero: `cat file1 | sort > file1_sorted`
- Redirigir el resultado de comandos hacia un dispositivo: `cat music.mp3 > /dev/audio`
- Redirigir el resultado de comandos para añadirlo a un fichero: `echo "Esto se añadirá al final del fichero" >> file1`
- Redirigir la entrada estándar a un comando: `cat < file1`. (Equivalente a cat file). Esto lo hemos usado en los bucles de bash.
- Desechar la salida de errores: `(echo "$((4/0))") 2>/dev/null`

echo

Echo es un comando que se usa para mostrar variables, cadenas o comandos por la salida estándar. Algunos de los usos más comunes de este comando son:

- Con la opción **-e** interpretamos todos los caracteres precedidos por `\.` Algunos de estos caracteres son:
  - **\b**: borra los espacios: `echo -e "Esto \bse \bva \bmostrar \bjunto"`
  - **\n**: inserta una nueva línea: `echo -e "\nUna \nlinea \npor \npalabra"`
  - **\t**: inserta un tabulador: `echo -e "\nLista de tareas: \n\ttask1 \n\ttask1 \n\ttask1"`

    - **\a**: produce un sonido: `echo -e “Esto va a pitar\a”`. A partir de Mac OS este comando no funciona, un equivalente podría ser `printf "Esto va a pitar\n\a"`

- Redirigir hacia un fichero: `echo “Mi primer fichero” > file1`
- Listar ficheros (similar a ls): `echo *.jpg`

Comandos de filtrado

Grep

Grep es un comando que se usa para buscar textos. Normalmente, se utiliza para buscar coincidencias en líneas de ficheros o en la salida estándar tras la ejecución de comandos. Algunos de los usos más comunes son:

- Obtener las líneas de un fichero donde se encuentre cierta cadena: `grep vagrant /etc/passwd`.
- Buscar una cadena en varios ficheros: `grep -r "secreto" file*`. Si quisiéramos mostrar el nombre de los ficheros donde aparece lo haríamos con la opción **-l**.
- La opción -i devuelve las líneas donde existen ocurrencias en la búsqueda. `echo -e "He is a VIP \nin the ISSIS" | grep -i "is"`
- Buscar palabras, no ocurrencias. En el ejemplo anterior, si quisiéramos obtener las líneas donde apareciese la palabra “is” utilizaremos la opción **-w**: `echo -e "He is a VIP \nin the ISSIS" | grep -wi "is"`
- Usar expresiones regulares: `echo "Extrae el número 1234" | grep -Eo '[0-9]{1,4}'`
- Mostrar las líneas que hay antes o después de la ocurrencia. En este ejemplo se muestra la primera línea después de la ocurrencia: `echo -e "Antes\n**secret**\nDespues" | grep -A 1 "secret"`
- Invertir la búsqueda. Buscar las líneas que no tienen estas ocurrencias. En este ejemplo no se mostrará la línea con la palabra “secret”: `echo -e "No muestres el\n**secret**\nDespues" | grep -v "secret"`
- Contar el número de veces que aparece la ocurrencia con la opción -c (similar a utilizar wc). `echo -e "He is a VIP \nin the ISSIS" | grep -ci "is"`

- Mostrar la posición (linea:carácter donde empieza) o el número de línea con las opciones **-ob** y **-n** respectivamente.

Sed

Sed (stream editor) no es un editor de texto aunque pueda modificar su contenido. Sed recibe las cadenas de texto como stream y las modifica. Algunos de los usos más comunes de sed son:

- Visualizar un rango de líneas. En este ejemplo veremos las 3 primeras líneas del fichero /etc/passwd: `sed -n '1,3p' /etc/passwd`. Se pueden incluir varios rangos con la opción **-e**.
- También podemos excluir rangos. El comando para excluir las 3 primeras lineas seria `sed '1,3d' /etc/passwd`
- Reemplazar una cadena por otra. En este ejemplo, reemplazamos archivos por ficheros `echo -e “ls muestra los archivos del directorio.\n Los archivos son:” | sed 's/archivos/ficheros/'`. Pero no reemplaza en la segunda línea. Sed ofrece diferentes opciones para reemplazar (con la opción s, substitute) un string. Algunas de ellas son:
    - Reemplazar todas las ocurrencias de ‘is’ por ‘os’ con la opción global **g**: `echo -e "He is a VIP \nin the ISIS" | sed 's/is/os/g'`. No reemplaza mayúsculas.
    - Con la opción **gi** si reemplazaría mayúsculas: `echo -e "He is a VIP \nin the ISIS" | sed 's/is/os/gi'`.
    - Los rangos vistos en el primer punto de este apartado también se pueden usar para delimitar el área donde realizar las sustituciones de cadenas.

Tail

Tail es un comando que te muestra las últimas líneas de un fichero. Se suele utilizar mucho para ver el contenido de logs de servicios en funcionamiento, ya que su salida se va actualizando. Un ejemplo del uso de tail para ver las 10 últimas líneas de syslog seria:

```
tail -n 10 -f /var/log/syslog
```

Comandos para buscar

Find

Find es un comando que se utiliza para buscar ficheros o directorios con ciertas características. Algunos usos del comando son:

- Buscar en el directorio actual por el nombre del fichero: `find . -name file1`
- Ejecutar comando de forma que su salida sea la entrada del find: `find . -name file1 -exec ls -l {} \;`
- Hacer tareas de mantenimiento:
    - Buscar ficheros vacíos: `find /home/vagrant -empty`
    - Buscar los 5 ficheros más pesados: `find /home/vagrant -type f -exec ls -s {} \; | sort -n  | head -5`

Con `type -f` buscamos solo ficheros y con `type -d` buscamos directorios.

Locate

Locate es un comando que nos permite conocer de una forma rápida la ruta de un fichero. Para ello se utiliza la base de datos creada por **updatedb**.

```
locate file1
```

Whereis

Otro comando que al igual que located, permite comprobar si existe un fichero y saber dónde está. En el ejemplo buscamos dónde se encuentra el comando ls.

```
whereis ls
```

Miscelanea

History

History es un comando que nos muestra un listado de comandos ejecutados en la terminal. Alguno de los usos más frecuentes de este

comando son:

- Pulsar **CTRL+R** para buscar en el historial.
- Saber cuándo se ejecutó el comando con `export HISTTIMEFORMAT='%F %T '`
- Ejecutar un comando específico del historial con *!n* (siendo n la posición en el historial).
- Borrar un comando del historial (útil cuando se usan credenciales en claro) con `history -d n` (siendo n la posición en el historial) o todo el historial con *history -c*.

Nohup

Nohup es un comando que se utiliza para mantener los procesos en segundo plano aunque cerremos la sesión.

```
#!/bin/bash
SECONDS=0
echo "Esto va a tardar ..."
sleep 180
duration=$SECONDS
echo "He tardado $(($duration % 60)) segundos"
```

```
nohup ./slow_script &
```

Si le damos permisos de ejecución y ejecutamos el script en segundo plano con nohup, nos devuelve el PID. Si salimos de la sesión y volvemos a entrar, podemos comprobar que el PID se sigue ejecutando al hacer un **`ps -aux | grep PID`**.

Nohup es muy útil cuando se necesita ejecutar un script de shell o un comando que tarda mucho en finalizar y no se quiere esperar a que se complete.

Free

Free es un comando que nos permite ver el estado de la memoria mostrándonos la cantidad de memoria disponible, en uso y swap:

free [OPTIONS]

Algunas de las opciones que nos ofrece son:

- **-g,-k,-m**: muestra la memoria en gigas, kilobytes o megas respectivamente.
- **-t**: muestra la suma total de la memoria sumando la swap a la disponible en ram.

Du, df y tree

Estos comandos nos permiten ver el tamaño de carpetas y ficheros, así como saber el espacio en disco.

El comando **du** lo utilizamos para saber el tamaño de carpetas y archivos. Si quisiéramos saber cuánto ocupa la carpeta home, lo haríamos de la siguiente manera:

```
du -bsh /home/vagrant
```

También funciona con ficheros:

```
du -bsh /etc/passwd
```

Si quisiéramos una forma más visual de ver los tamaños de los directorios y los ficheros podemos utilizar el comando **tree** (hay que instalarlo con apt install). Tree nos permite ejecutar el comando anterior y desglosar los diferentes tamaños en forma de árbol. El ejemplo anterior con tree seria:

```
tree /home/vagrant --du -h
```

Por último, el comando **df** nos permite conocer el uso de espacio en disco, mostrando cuánta capacidad de disco hay, cuánta se usa, cuánta hay libre y dónde está montada. Un ejemplo de ejecución del comando df podría ser:

```
df -h
```

Diff

Diff es un comando que nos permite encontrar fácilmente diferencias entre 2 ficheros. Vamos a generar dos ficheros para después compararlos. Utilizamos los saltos de línea \n para que cada palabra se cree en una nueva línea del fichero:

File1: `echo -e "archivo\nfile1\npara\ncomparar" > file1`

File2: `cat file1 | sed 's/1/2/g' | sed 's/archivo/fichero/g'> file2`

Si comparamos los dos ficheros con `diff file1 file2`, podemos ver lo siguiente:

```
vagrant@localhost:~$ diff file1 file2
1,2c1,2
< archivo
< file1
---
> fichero
> file2
```

- **<** nos indica que aparece en file1 y no está en file2
- **>** nos indica que aparece en file2 y no está en file1
- **1,2c1,2** significa que las líneas 1 y 2 de file1 junto con las líneas 1 y 2 de file2 se deben cambiar para que coincidan.

Una de las opciones que se puede usar en diff es **-w** para que no detecte como cambio los espacios en blanco.

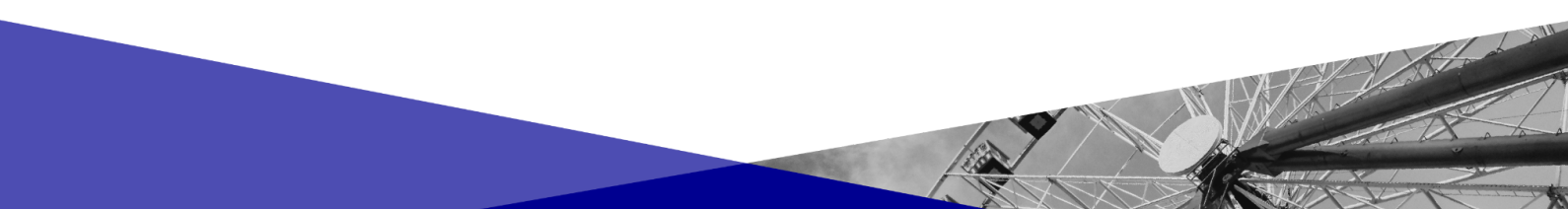
Tar

Utilizamos tar para comprimir y descomprimir los ficheros con extensión tar.gz o .tar.bz2. Algunas de las tareas que podemos hacer con tar son:

- Comprimir una carpeta: `tar cvf home_vagrant.tar /home/vagrant`
- Descomprimir un fichero tar: `tar xvf home_vagrant.tar /tmp`
- Listar el contenido de un fichero tar: `tar tvf home_vagrant.tar`
- Añadir un fichero a un archivo comprimido ya creado: `tar rvf home_vagrant.tar newFile`
- Estimar el tamaño que ocupará una carpeta comprimida (en KB): `tar -cf - /tmp | wc -c`

El significado de las opciones que hemos visto en los ejemplos anteriores es:

- **c:** crea el archivo.
- **v:** muestra la información de una forma amigable para el usuario.
- **f:** crea el archivo con el nombre especificado.
- **x:** extrae el archivo.
- **t:** muestra el contenido o la lista de ficheros del archivo.



Date

Date es un comando que nos ayuda a mostrar la fecha del sistema en diferentes formatos y nos permite hacer operaciones con ella. Algunas de las opciones que nos da son las siguientes:

- Convertir una fecha (como cadena) en el formato del sistema: `date --date="25 May 2020"`. También puede hacerlo con un fichero como entrada: `date --file=datefile`
- Calcular fechas relativas (pasadas o futuras): `date --date="next fri"` o `date --date="1 day ago"`
- Fijar la fecha del sistema: `date -s "Mon May 25 16:00:00 PDT 2020"`
- Mostrar la fecha en formato universal (UTC): `date -u`
- Mostrar la fecha en diferentes formatos: `date +%N`.

Si quieres conocer más formatos puedes hacer `man date` o consultar esta página.

Parte 4

Docker

Introducción a Docker

¿Qué es Docker?

Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos.

Es una plataforma de virtualización a nivel de sistema operativo que permite crear una aplicación y empaquetarla en un contenedor junto con sus dependencias. Esto garantiza que la aplicación funcione sin problemas en cualquier otra máquina o entorno, independientemente de sus configuraciones. Esta característica permite promocionar de manera más rápida entre entornos.

Docker consta de cuatro componentes, que incluyen:

- Docker Client y Daemon.
- Imágenes.
- Registros de Docker.
- Contenedores.

Definiremos cada componente en los siguientes apartados.

Docker emplea características del kernel de Linux (namespaces y cgroups) para construir contenedores sobre un sistema operativo y automatizar la implementación de la aplicación en los contenedores. El software que alojan estos contenedores se conoce como Docker Engine.

Básicamente, Docker funciona de manera muy similar a una máquina virtual (VM). Sin embargo, hay una gran diferencia entre ambos. Mientras una máquina virtual crea un sistema operativo virtual completo, Docker permite que las aplicaciones usen el mismo kernel de Linux del sistema en el que se ejecutan.

Solo necesita enviar la aplicación y sus dependencias (las que no comparta con el host) dentro de un contenedor para poderlo ejecutar. El proceso de contenedorización no solo reduce el tamaño de la aplicación, sino que

también aumenta la capacidad de aprovechamiento de los recursos y facilita la escalabilidad.

¿Por qué se usa?

Docker ofrece muchas ventajas a las empresas. Es capaz de reducir los costes operativos en infraestructura y mantenimiento. Al mismo tiempo, reduce el time to market permitiendo ofrecer nuevas funcionalidades de manera rápida y escalable. Otras de las ventajas que podemos destacar de Docker son:

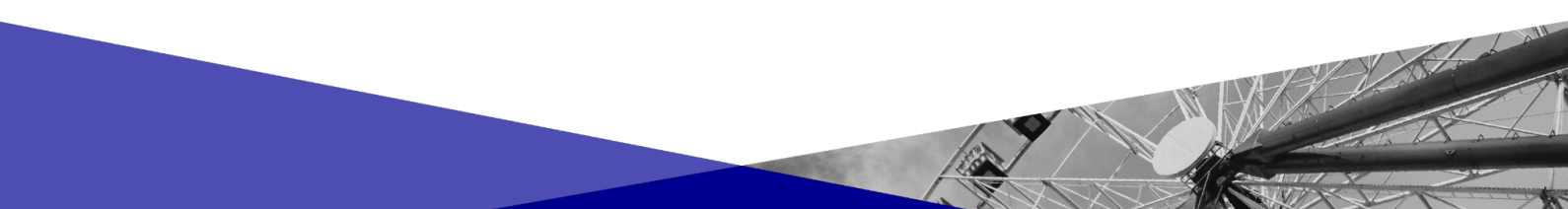
- **Uso eficiente de los recursos del sistema.**
  Las aplicaciones en contenedores requieren menos memoria (gracias al tamaño reducido de la aplicación al compartir librerías con el kernel de sistema operativo). Esto permite que las aplicaciones puedan crearse, usarse y destruirse para liberar recursos rápidamente.

  La capacidad que nos ofrecen los contenedores de empaquetar aplicaciones y su bajo consumo, nos permite desplegar multitud de contenedores dentro de un host, reduciendo el coste en infraestructura y aumentando la capacidad de carga de trabajo de la aplicación, optimizando el uso del procesamiento.

  Además, se produce un ahorro considerable en licencias de SO, ya que se necesitarán menos instancias para ejecutar la misma carga que con las máquinas virtuales.

- **Ciclos de entrega de software más rápido.**
  En un mercado tan competitivo y cambiante, las empresas tienen que responder rápidamente a las necesidades del mercado. Esto significa, no sólo ser capaces de adaptarse a las necesidades cambiantes de sus clientes, si no a la actualización constante de los productos, servicios, aplicaciones, librerías que su cliente utiliza. Estas dos tareas se pueden realizar usando contenedores de Docker.

  Con contenedores podemos desplegar, en poco tiempo, una nueva funcionalidad, así como una versión anterior al que se le agrega una nueva funcionalidad. También nos permite dar “marcha atrás” a una versión anterior. Lo que nos permite ciclos de entrega más rápidos.



- **Permite la portabilidad de la aplicación.**
  Un contenedor puede instanciarse y ejecutarse en cualquier máquina que tenga Docker instalado. Esto permite que las aplicaciones embebidas en contenedores puedan promocionarse sin problemas de configuración o rendimiento entre los diferentes entornos.

- **Mejora la productividad de los desarrolladores.**
  Gracias a los contenedores de Docker, los desarrolladores no tienen que crear el entorno cada vez que la aplicaciones se tienen que ejecutar en un entorno o máquina diferente, dejando atrás la frase “en mi máquina funciona” y permitiéndoles ser más productivos. La estandarización de los entornos gracias a los contenedores también le facilita la tarea al equipo de DevOps, pudiendo acelerar los tiempos y la frecuencia de las implantaciones.

- **Aumenta la eficiencia operativa.**
  Los contenedores de Docker mejoran la agilidad operativa de las empresas. Simplificando la cadena de suministros y automatizando la gestión de múltiples aplicaciones en un único modelo operativo centralizado. Tener un modelo operativo único implica una reducción en el tiempo promedio empleado en la resolución de incidencias, lo que tiene un impacto directo en la satisfacción del cliente.

¿Qué es un contenedor?

Un contenedor es un conjunto de uno o más procesos que están aislados del resto del sistema. Un contenedor no es más que una instancia en ejecución de una imagen. Un contenedor de Docker lo podemos descomponer en:

- Una imagen de Docker.
- Un entorno de ejecución.
- Un conjunto de instrucciones a ejecutar.

¿Cómo funciona Docker?

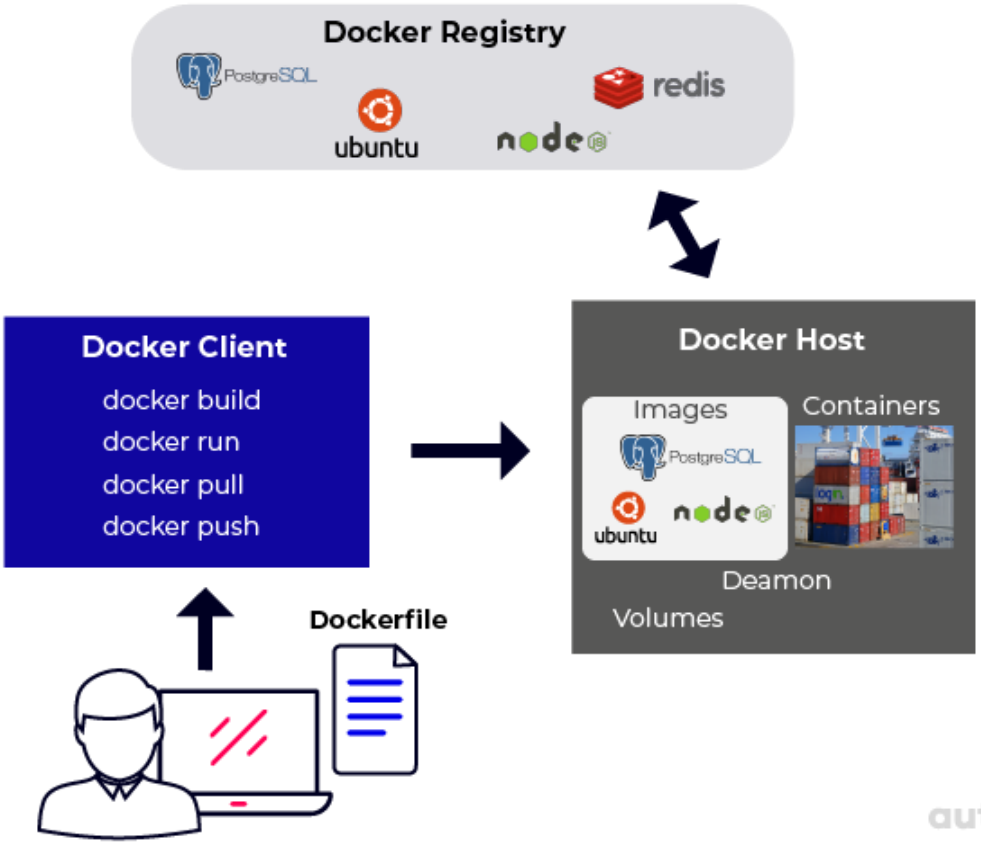
Para entender cómo funciona Docker debemos conocer las partes fundamentales de las que se compone:

- **Docker Engine**

  El motor de Docker es la capa sobre la que funciona Docker. Es una herramienta ligera que administra contenedores, imágenes, builds, etc. Se ejecuta de forma nativa en sistemas Linux y está compuesto por:

  1. Un Docker Daemon que se ejecuta en el host.
  2. Un Docker Client que se comunica con el Docker Daemon para ejecutar comandos (docker run, docker ps, docker rm, etc.).
  3. Una API REST para interactuar con el Docker Daemon de forma remota.

- **Docker Client**



Docker Client es la interfaz que le ofrece Docker al usuario final para comunicarse con el Docker Daemon. El usuario nunca se comunica directamente con el Docker Daemon. Se puede ejecutar desde la máquina host, pero no es obligatorio. Puede ejecutarse desde otra maquina. En posteriores apartados veremos las operaciones que puede ejecutar el usuario con Docker Client.

- **Docker Daemon**
  El Docker Daemon es realmente quien ejecuta los comandos enviados por Docker Client, cómo construir, arrancar o parar contenedores. El Docker Daemon se ejecuta desde el host.

- **Dockerfile**
  Un Dockerfile es el fichero donde se escriben las instrucciones para construir una imagen de Docker. En posteriores apartados veremos en mayor detalle qué es y cómo se construye un Dockerfile.

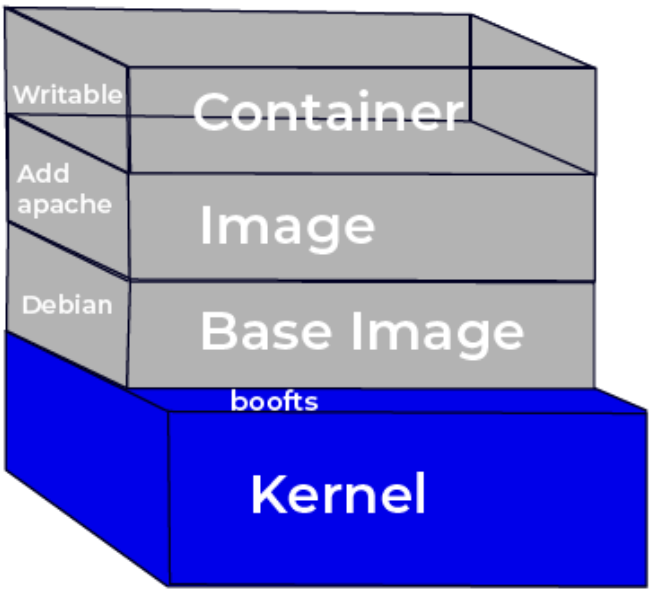
- **Docker Images**
  Las Docker images son plantillas de solo lectura con un conjunto de instrucciones escritas en su Dockerfile para construir el contenedor. En la Docker Images se define tanto la aplicación a empaquetar como las dependencias que necesita para arrancar.

  La imagen Docker se construye utilizando un Dockerfile. Cada instrucción en el Dockerfile añade una nueva "capa" a la imagen. Las capas representan una porción del sistema de archivos de la imagen que añade o reemplaza los que tiene la imagen base de la que parte. Las capas son clave para hacer que los contenedores sean ligeros. Docker utiliza un sistema de archivos llamado “Union File Systems” para lograrlo.

- **Union File System**
  Docker usa Union File System para construir una imagen. Podríamos verlo como un sistema de archivos apilable, donde cada archivo o directorio del sistema de archivos separado en ramas, puede superponerse formando un único sistema de archivos. Esta operación es transparente para el contenedor.

  Se controla mediante punteros las diferentes versiones de un mismo recurso en cada capa. De tal manera, que si se necesita modificar una capa (por algún cambio en el Dockerfile), se hace una copia de la original y esa capa es la que se modifica. Así es como los sistemas de archivos simulan ser “escribibles” sin realmente permitir la escritura.

(En otras palabras, un sistema de "copia sobre escritura").



Los sistemas de capas ofrecen dos beneficios principales:

1. **Evitan la duplicación**: evita la duplicación de archivos al crear o ejecutar contenedores, haciéndola más rápida y eficiente.
2. **Segregación de capas**: hacer un cambio es mucho más rápido, ya que cuando cambias una imagen (modificando sentencias del Dockerfile) solo se actualiza la capa a la que afecta el cambio.

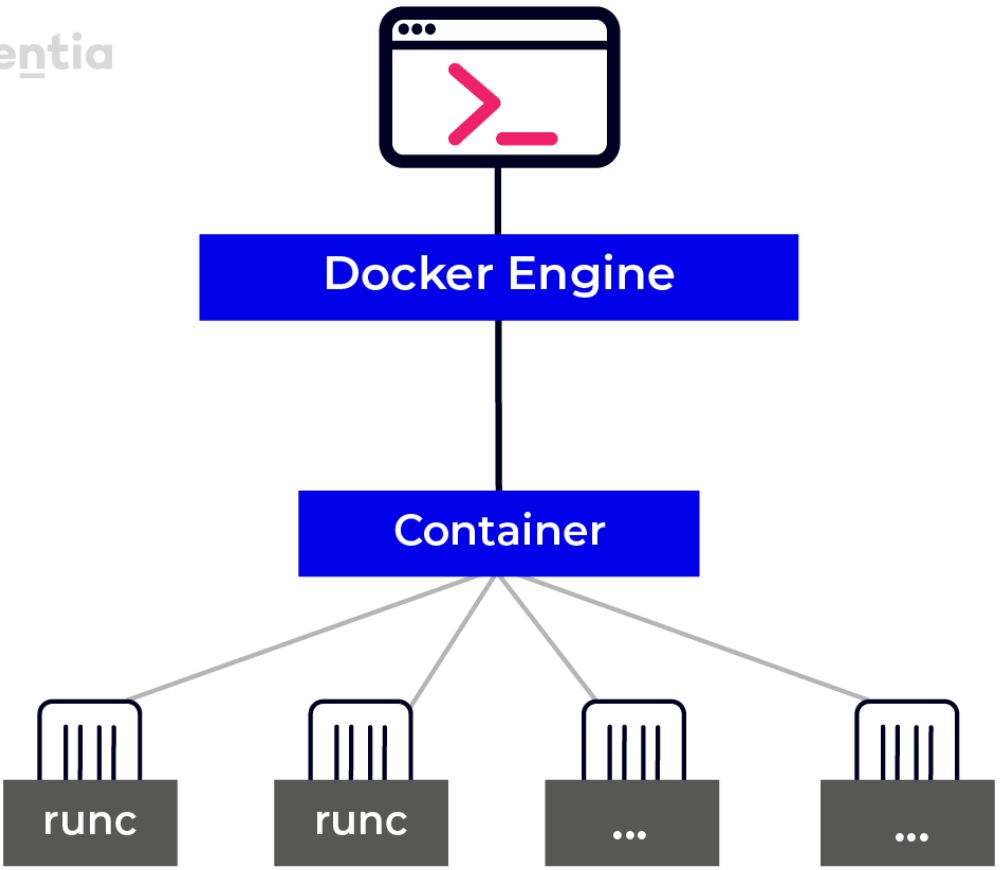
- **Volúmenes**
  Los volúmenes son la parte de "datos" de un contenedor y se inician en su fase de creación. Los volúmenes son la forma que tiene el contenedor de persistir y compartir los datos. Los volúmenes de datos están separados del Union File System predeterminado y existen como directorios y archivos normales del sistema de archivos del host. Por lo que, si un contenedor se borra o actualiza, sus volúmenes permanecerán intactos. Un mismo volumen puede ser usado por varios contenedores.

- **Contenedores Docker**
  Un contenedor Docker, como se ha dicho en apartados anteriores, no es más que una instancia en ejecución de una imagen.

Anatomía de contenedores

Arquitectura

Docker Engine proporciona una interfaz REST que puede usar cualquier otra herramienta como por ejemplo Docker CLI, Docker para MacOs o para Windows y también Kubernetes. Por debajo de Docker Engine se encuentra Containerd y runC para gestionar el ciclo de vida de los contenedores. Para hacer esta gestión, Containerd y runC, aprovechan funcionalidades del kernel de Linux como cgroups, namespaces y UnionFS.



Namespaces

Docker utiliza una función del kernel de Linux llamada *namespaces* para proporcionar encapsulamiento y restringir la visibilidad al exterior de los contenedores.

Los namespaces son el mecanismo que utiliza docker para que los procesos que tiene dentro piensen que son una instancia independiente del resto del sistema.

Podemos restringir diferentes elementos del sistema dependiendo del tipo de *namespace* que utilicemos para establecer las restricciones, pudiendo restringir la visibilidad de PIDs, de redes, del sistema de archivos, etc. Se puede obtener más información sobre los namespaces y Docker en la documentación oficial.

Control groups (cgroups)

La función *groups* es la encargada de gestionar el uso de recursos en los contenedores. Esto incluye limitar y aislar recursos de dichos procesos. El uso de cgroup nos permite:

- Limitar recursos (memoria, CPU, I/O, uso de red, etc.).
- Priorizar unos grupos con respecto a otros.
- Seguimiento del uso de recursos por motivos de facturación.
- Control sobre la ejecución de los procesos.

Union Filesystem (UnionFS)

Como se ha explicado en el apartado anterior, la función de UnionFS es la superposición de sistemas de archivos para formar un sistema de archivos único.

De este modo, los contenidos de directorios en las distintas capas que tienen la misma ruta en distintos sistemas aparecerán juntos en un único directorio. Construyendo un nuevo sistema de archivos virtual de la composición de muchos.

Fontanería de contenedores: runC y Containerd

Las dos tecnologías sobre las que funciona Docker son runC y Containerd.

runC es una herramienta que funciona como cliente que permite construir y ejecutar contenedores siguiendo la especificación de **Open Container Initiative (OCI)**, aportando funcionalidades de alto nivel como la distribución de las imágenes, el almacenamiento, la gestión de las redes, etc.

Por otro lado, **Containerd** es un demonio que actúa como una API Fachada para los contenedores y el SO, la fachada les provee de una capa superior de abstracción que les permite trabajar con entidades de alto nivel como son snapshots y contenedores.

Instalación de Docker

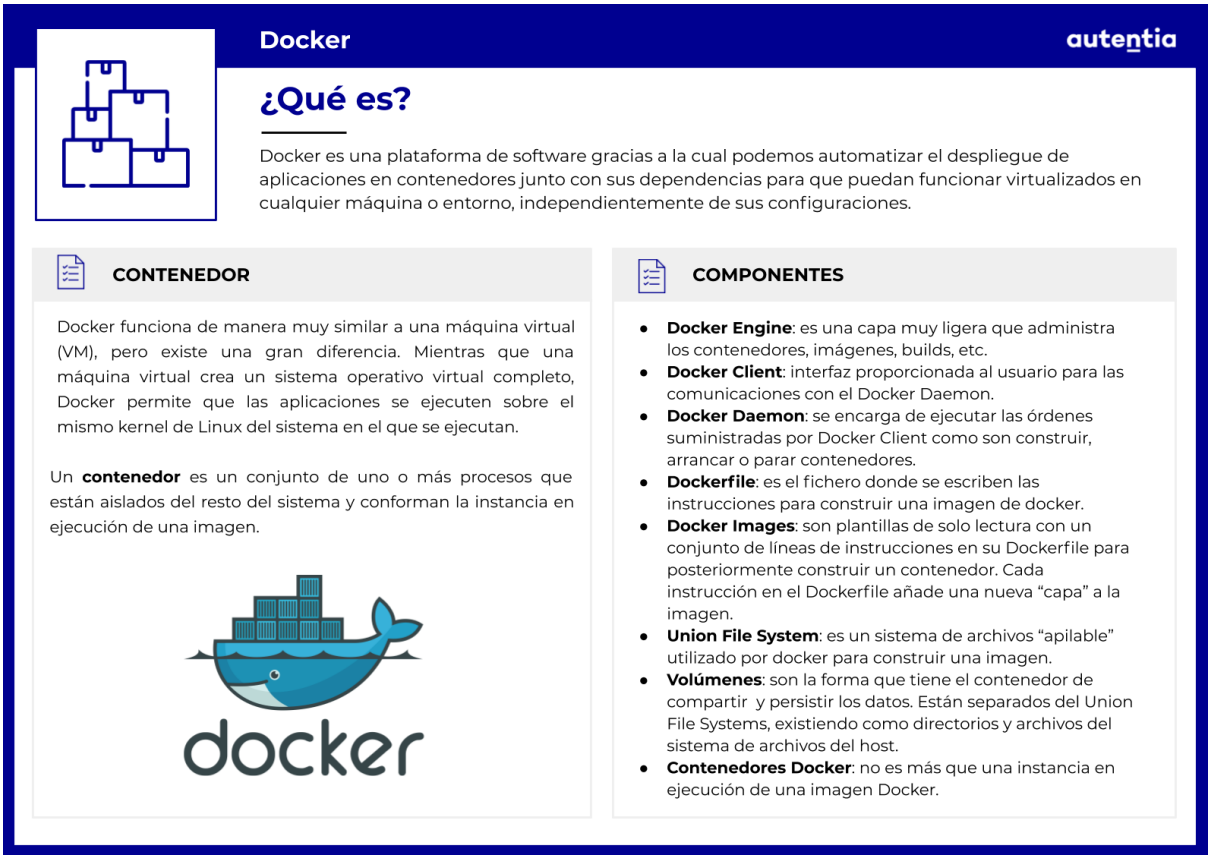
Docker en Mac se puede instalar de diferentes maneras. Tres de las más habituales son:

- **Descargar la .dmg de Docker.com**
  Como primera opción, podemos ir a la web oficial de Docker. Para instalar solo necesitas descargar y ejecutar el fichero .dmg. Probablemente esta sea la manera más fácil. Docker Desktop no tiene necesidad de máquinas virtuales.

- **Con Docker Toolbox**
  Docker Toolbox está soportada para Mac OS y Windows. La principal característica de esta herramienta es el uso de Virtualbox para utilizar el Kernel de Linux. Por lo que, tenemos docker en una máquina diferente al entorno de docker. Docker Toolbox proporciona las herramientas docker-compose y Kinematic, las cuales permiten gestionar los contenedores usando tanto GUI como la terminal.

- **Con Homebrew**
  Por último, también podemos utilizar el gestor de paquetes Homebrew utilizado en anteriores documentos. El resultado de esta instalación es equivalente a descargarlo de la web (instalación nativa). Para instalar docker con Homebrew simplemente tenemos que ejecutar el siguiente comando:

```
brew cask install docker
```



Trasteando con Docker

Como primeros pasos con Docker conviene comprobar que la instalación ha sido correcta. Esto se puede hacer mediante la ejecución del comando docker version, para comprobar la versión que se ha instalado.

En los siguientes apartados se van a mostrar los casos de uso más frecuentes a la hora de usar Docker CLI.

Arrancar mi primer contenedor

Arrancar un contenedor se hace con el comando `docker container run` y a continuación el nombre de una imagen que se va a usar para instanciarlo. Como ejemplo se va a arrancar un contenedor con la imagen de Nodejs:

```
$ docker container run node:12
```

La primera vez que se trate de arrancar una imagen, va a tener que ser descargada de un repositorio online llamado Docker Hub.

El argumento `--name` nos permite especificar un nombre al contenedor para poder hacer referencia a él más fácilmente.

```
$ docker container run --name mycontainer node:12
```

Otro argumento muy común es -p, relativo a los puertos. Con esto se van a relacionar dos puertos, el primero es el de la máquina local y el segundo es el del contenedor (Port Forwarding).

De este modo, si instalamos una aplicación en nuestro contenedor, podemos acceder a ella desde la máquina local por el puerto 8181. Para ello, arrancamos nuestro contenedor de la siguiente manera:

```
$ docker container run --name mycontainer -p 8181:8080 node:12
```

Para identificar qué puerto se le asigna al host y cuál al contenedor, recordad que el de la izquierda se refiere al del host (8181) y el de la derecha al del contenedor (8080).

Una vez está arrancado nuestro primer contenedor podemos obtener información acerca de él: sobre su estado, configuración, variables de entorno, redes, etc., ejecutando el comando `docker container inspect mycontainer`.

También se puede mostrar una lista con todos los contenedores que actualmente están arrancados en nuestra máquina con `docker ps`. Otra forma alternativa es usando `docker container ls`.

```
macbook-javiersepulveda-2:~ javier$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
b34d64dc251d        node:12             "docker-entrypoint.s…"   About a minute ago
macbook-javiersepulveda-2:~ javier$
```

Arrancar, parar, borrar contenedores

Cuando se tiene un contenedor arrancado existen dos posibilidades para pararlo. La primera es parar (stop) el contenedor, lo que implica terminar con los procesos asociados a él. El comando stop envía una señal SIGTERM para terminar con el proceso. La segunda opción es utilizar el comando de kill que envía la señal SIGKILL al proceso. Los comandos para parar y matar un contendor son:

```
docker container stop mycontainer
docker container kill mycontainer
```

Un contendor que se haya parado puede eliminarse con `docker container rm mycontainer`. Si está ejecutándose se puede borrar añadiendo `docker rm --force mycontainer`. En el caso de tener muchos contenedores y querer borrar todos se usa `docker rm $(docker ps -aq)`.

Dado un contenedor parado se puede ejecutar `docker container start mycontainer` para volverlo a arrancar.

Hay ocasiones en las que la vida de un contenedor es muy corta. Por ejemplo, en el caso de que utilicemos contenedores para compilar aplicaciones y una vez generado el binario en un volumen para compartirlo con la máquina host. Una vez realizada la compilación, ese contenedor no tiene razón de ser y está consumiendo recursos.

En ese caso, utilizaremos la opción --rm para que de manera automática lo borre una vez haya terminado. Otro uso que le podemos dar es como

laboratorio de pruebas (por ejemplo de Nodejs), arrancándolo en modo interactivo ejecutando `docker container run -it --rm node:12`.

Monitorizar los logs de un contenedor

Por defecto, Docker guarda los logs generados en un archivo con formato JSON. Este archivo clasifica los logs según su origen (stdout/stderr) y la fecha en la que se ha emitido. Para cada contenedor hay un archivo JSON con sus logs.

Para recuperar los logs de un contenedor se usa el comando `docker logs mycontainer`. Pudiendo recuperar los logs a partir de una fecha concreta o relativa como por ejemplo los de la última hora. Algunos ejemplos pueden ser:

```
docker logs -t 2019-09-16T06:17:46.000000000Z mycontainer
docker logs --since 3h mycontainer
docker logs --tail 5 mycontainer
```

Docker organiza los logs en distintos niveles y estos se encuentran separados. En primer lugar están los logs de la aplicación que se ejecuta dentro de Docker, luego los logs de la máquina *host* y finalmente los del Docker Daemon. Existen distintas estrategias para visualizar los logs:

- A través de la aplicación.
- Persistiendo en volúmenes.
- Configurando *logging drivers* en Docker.
- Usando un contenedor dedicado.

Ejecutar una shell dentro de Docker

El comando docker exec permite ejecutar un comando en un contenedor arrancado. El comando se ejecutará en el directorio de trabajo actual.

Para ejecutar una shell dentro de docker usamos el comando `docker exec -it` seguido del id del contenedor o el nombre que tenga asignado y terminando con bash (que identifica la shell a ejecutar, también podría ser /bin/sh). Los argumentos -it son la versión abreviada de las opciones `--interactive` y `--tty` (*teletypewriter*). Estas opciones permiten al usuario operar con el contenedor de docker de una forma interactiva (y no declarativa, como en el caso del Dockerfile).

```
$ docker exec -it mycontainer bash
```

Una vez ejecutado el comando en nuestra terminal vamos a tener una *shell* dentro del contenedor. Si ejecutamos el comando id podemos ver que hemos entrado al contenedor como el usuario root.

```
macbook-javiersepulveda-2:~ javier$ docker exec -it mycontainer bash
root@b34d64dc251d:/# id
uid=0(root) gid=0(root) groups=0(root)
root@b34d64dc251d:/#
```

Configurando contenedores

Docker CLI ofrece una serie de comandos para listar los distintos objetos que maneja Docker, como contenedores, volúmenes, redes, imágenes guardadas, etc.

Como se ha visto anteriormente, docker ps muestra en formato tabla los distintos contenedores que hay arrancados. Añadiendo el argumento de -a se muestran todos los contenedores instalados en la máquina (arrancados, parados, creados, pausados, etc.).

Podemos mostrar todas las redes de Docker con el comando `docker network ls`.

Con `docker network inspect` y el id o el nombre de la red podemos ver información específica de una red así como los contenedores que

pertenecen a ella.

De igual forma, podemos mostrar todos los volúmenes con `docker volume ls` y obtener información detallada para un volúmen en concreto con `docker volume inspect <volume id>.`

Es muy común el uso de variables de entorno en la construcción de una imagen para establecer parámetros de configuración con el nombre de usuario, la password o el directorio de trabajo. Sin embargo, una aplicación funcionando dentro de un contenedor no es capaz de ver las variables de entorno de la máquina local sobre la que se está ejecutando el contenedor. Docker usa sus propias variables de entorno a la hora de ejecutar el contenedor.

Tenemos dos posibilidades para añadir variables de entorno al arrancar un contenedor:

- -e, --env para definir variables una a una.
- --env-file cuando las variables se encuentran en un fichero.

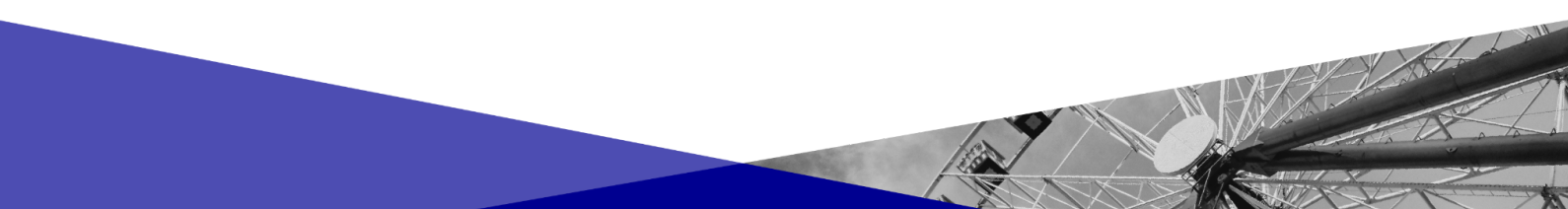
Otra forma (aunque estática) de definir valores por defecto de variables de entorno es su definición en el Dockerfile. En los siguientes apartados, veremos algunos ejemplos de los métodos mencionados para añadir variables de entorno.

Variables de entorno

Un ejemplo de cómo establecer la contraseña del usuario postgres para poder conectarnos a la base de datos sería:

```
$ docker run -e POSTGRES_USER=myuser -e
POSTGRES_PASSWORD=mysecretpassword postgres.
```

Establecer credenciales utilizando comandos es una mala práctica, ya que quedan registrados en el histórico de comandos (en claro). Para evitarlo, recomendamos el uso de ficheros de configuración.



Ficheros de configuración

Para el ejemplo anterior de la conexión a la base de datos se podrían pasar las variables de entorno con los datos del usuario en un fichero. Se usa el siguiente comando, en el que se especifica la ruta al fichero con los datos, `docker run --env-file ./postgres.config postgres`.

El fichero de configuración tiene el siguiente formato:

```
POSTGRES_USER=myuser
POSTGRES_PASSWORD=mysecretpassword
```

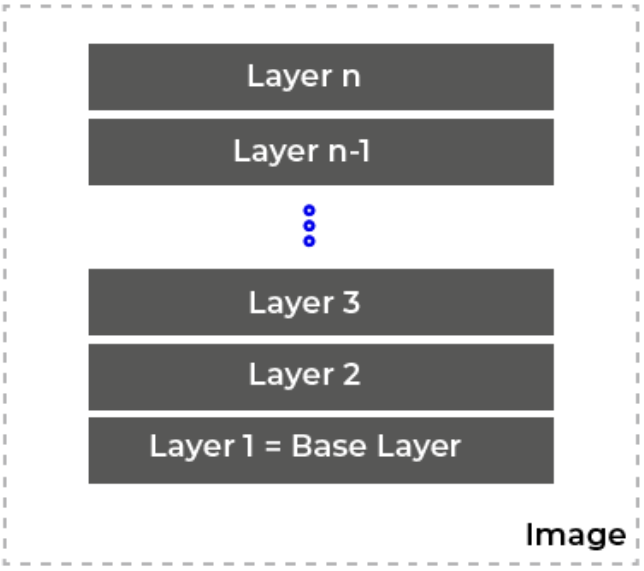
Construyendo imágenes a medida

¿Qué es una imagen?

Como sabemos, en Linux todo es un archivo, por lo que podríamos definir el sistema operativo como un sistema de archivos y carpetas almacenados en disco.

Teniendo esto en cuenta, podríamos decir que la imagen de un contenedor no es más que un gran archivo comprimido que contiene un sistema de archivos en capas.

Como hemos dicho en apartados anteriores, las imágenes de contenedor son plantillas a partir de las cuales se crean contenedores. Estas imágenes se componen de muchas capas. La primera capa en la imagen también se llama capa base:

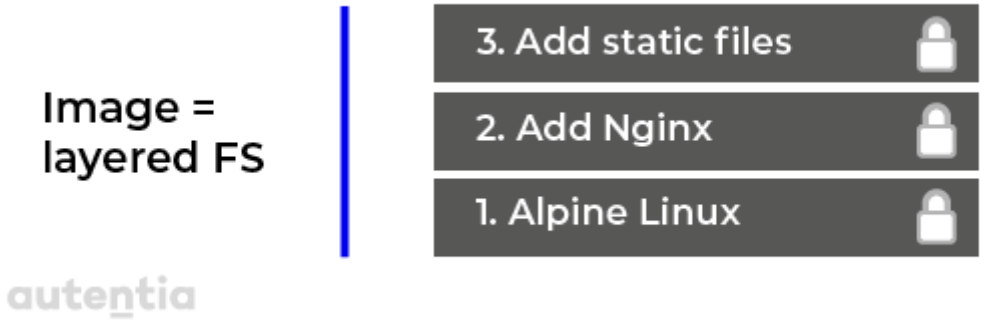


Una imagen de Docker se puede ver como una pila de capas en la que cada capa contiene archivos y carpetas que han cambiado (la delta) con respecto a las anteriores.

Como hemos visto en apartados anteriores, Docker utiliza Union File

System para crear un sistema de archivos virtual con todas las capas. Las capas son inmutables y la única acción que les puede afectar es su eliminación.

La inmutabilidad de las capas es importante porque nos permite cachear capas sin miedo a que se queden obsoletas. Esto nos permite reducir considerablemente los tiempos de construcción y carga de los contenedores así como el espacio que ocupan las imágenes.



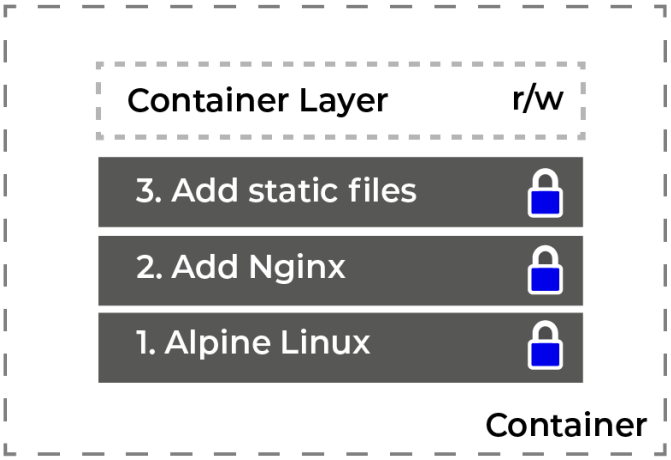
En la imagen anterior podemos ver el stack de capas donde se está montando un Nginx junto con otra capa con los ficheros estáticos (CSS, Javascript, HTML) sobre la imagen base Alpine Linux. Todas las imágenes parten de una imagen base.

Por lo general, esta imagen base se encuentra en los repositorios oficiales, en Docker Hub. Docker Hub es un repositorio público de imágenes de contenedores de Docker que nos permite centralizar las imágenes y poder compartirlas con la comunidad.

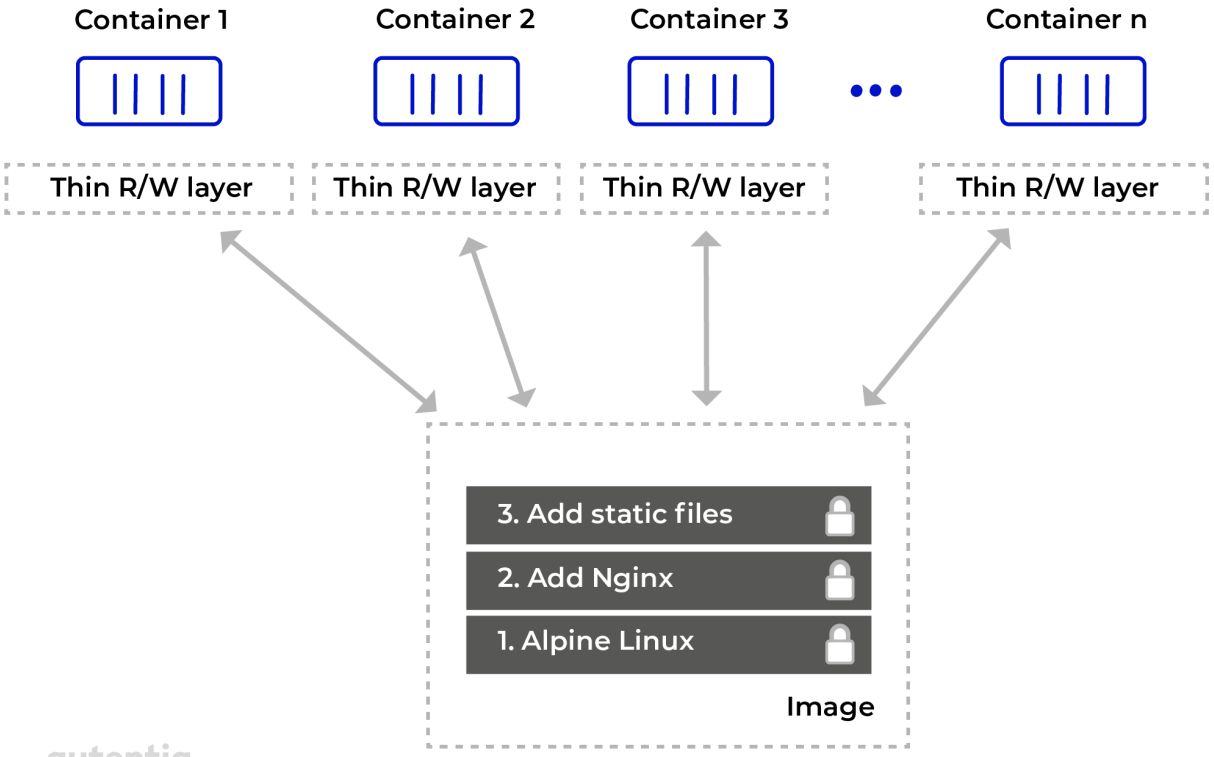
El contenido de cada capa se almacena en la ruta /var/lib/docker/ como una subcarpeta dentro del sistema host.

Pero si las capas son inmutables, ¿cómo hace Docker para que pueda escribir en los contenedores?

Cuando el motor Docker crea un contenedor a partir de dicha imagen, agrega una capa con permisos de lectura/escritura (r/w) en la parte superior dejando las demás capas inmutables. La pila de nuestro contenedor quedaría de la siguiente manera:



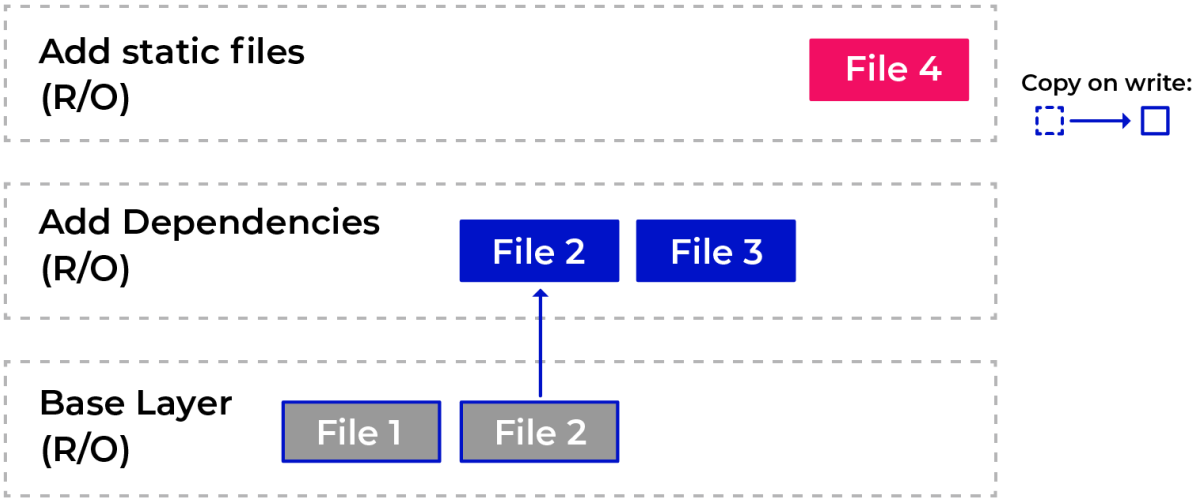
Otra ventaja de la inmutabilidad de las capas es que una misma imagen puede ser compartida por varios contenedores.



Esto supone una reducción en los recursos que se consumen y reduce el tipo de carga de un contenedor, ya que las capas solo se cargan una vez en memoria (con el primer contenedor).

Para poder modificar archivos que se están compartiendo entre las capas

de una imagen, Docker utiliza la técnica de “Copy-on-write”. Se usa cuando una capa quiere modificar un fichero de una capa más baja, primero se hace una copia del fichero a modificar en su capa y después lo modifica. Podemos ver el uso de esta técnica en la siguiente imagen:



autentia

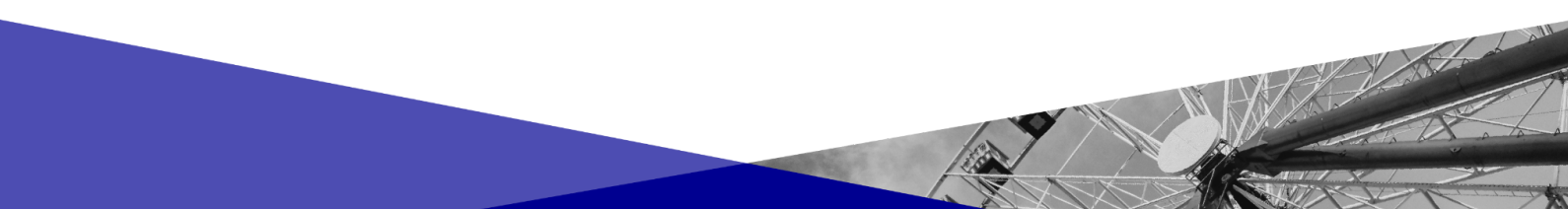
La segunda capa quiere modificar el File 2, que está presente en la capa base. Por lo tanto, la copió y luego la modificó. A la vista de la capa superior, ésta solo verá su propio fichero File 4 y los ficheros de la capa 2.

Creando imágenes

Hay tres formas de crear una nueva imagen de contenedor:

- La primera es a través de la **construcción interactiva** mediante la modificación de un contenedor que contenga todos los cambios que se desean realizar en la nueva imagen.
- La segunda y más utilizada es **usar un Dockerfile** para describir la secuencia de pasos a ejecutar para construir la nueva imagen.
- Finalmente, la tercera forma de crear una imagen es **exportandola e importándola** a otro equipo.

En los siguientes apartados, veremos estas 3 formas más en detalle.



Creación Interactiva

La primera forma para crear una imagen es a partir de un contenedor de Docker. Para ver cómo sería, vamos a construir de manera interactiva una imagen de alpine con las iputils instaladas. Para ello deberíamos seguir los siguientes pasos:

- Ejecutamos un contenedor que tenga como imagen base alpine de manera interactiva (pudiendo acceder dentro del contenedor mediante shell). Para ello tenemos que ejecutar el siguiente comando:

```
$ docker container run -it --name sample alpine /bin/sh
```

- Una vez dentro de contenedor, actualizamos el sistema e instalamos las iputils ejecutando el siguiente comando:

```
$ apk update && apk add iputils
```

- Como último paso, antes de salir del contenedor, hacemos un ping a localhost para verificar que las iputils están instaladas y funcionan correctamente.

```
/ # ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.041 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.046 ms
```

- Para salir del contenedor escribimos exit. Si hacemos un docker ps -a podemos ver que nuestro contenedor existe y se encuentra en estado “Exited”.

```
▶ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
cdc1a7192089        alpine              "/bin/sh"           8 minutes ago       Exited (0) 6 minutes ago                        sample
```

- Si queremos comprobar que los cambios realizados en el contenedor se persisten, podemos ejecutar el comando docker container diff sample que nos permite saber las diferencias de ficheros entre un contendor y su imagen base (A para añadido, C para cambiado y D para borrado).

- Por último, para persistir las modificaciones realizadas en el contenedor en una nueva imagen ejecutamos el siguiente comando:

```
$ docker container commit sample my-alpine
```

- Podemos comprobar con el comando docker image ls | grep my-alpine, que se ha creado nueva imagen con el nombre my-alpine:

Este método es muy útil para explorar y crear prototipos. Pero tiene la desventaja de que es manual, por lo que no se puede repetir ni escalar. También es propensa a errores, como cualquier proceso manual.

Usando Dockerfile

Una alternativa mejor que la anterior es el uso de Dockerfile. Dockerfile es un archivo de texto que contiene instrucciones para construir nuestra imagen a medida..

Un ejemplo de Dockerfile podría ser el siguiente:

```
FROM python:3.9
RUN mkdir -p /app
WORKDIR /app
COPY ./requirements.txt /app/
RUN pip install -r requirements.txt
```

```
CMD ["python", "main.py"]
```

Este ejemplo parte de la imagen base python:2.7. Vemos que el fichero contiene 6 líneas y que cada línea comienza con una palabra clave (los comandos) en mayúsculas (no es obligatorio, se hace por convención) como FROM, RUN, CMD, etc..

Cada línea equivale a una capa en la imagen final.

Comandos

En este apartado veremos algunos de los comandos más utilizados a la hora de construir Dockerfile.

FROM

Con esta palabra clave empiezan todos los Dockerfile, ya que indica la imagen base de la que se va a partir para construir la futura imagen. Aunque si solo está esa línea en el Dockerfile, no generaría imagen. Un ejemplo de línea usando FROM seria:

```
FROM centos:7
```

Se recomienda fijar siempre una versión, ya que si no lo hacemos nos bajaría la última y esto puede provocar errores e incompatibilidades. También recomendamos el uso de distribuciones ligeras como base para construir nuestras imágenes ampliando la imagen según vayamos necesitando. Podéis encontrar algunas de estas distribuciones ligeras y optimizadas para Docker en este enlace.

RUN

RUN es uno de los comandos más utilizados en los Dockerfile. Con él podemos ejecutar cualquier comando de Linux pasándoselo como argumento. Un ejemplo podría ser:

```
RUN apt-get update \
  && apt-get install -y --no-install-recommends \
    ca-certificates \
    libexpat1 \
```

```
    libffi6 \
    libgdbm3 \
    libreadline7 \
    libsqlite3-0 \
    libssl1.1 \
  && rm -rf /var/lib/apt/lists/*
```

Es muy común utilizar operadores como && o || para condicionar la ejecución del siguiente comando. En el ejemplo estamos indicando que solo queremos instalar las librerías si la actualización del sistema se ha completado con éxito.

Si el comando es muy largo y necesitas usar varias líneas, puedes partir la línea con backslash (\) como se puede ver en el ejemplo.

COPY y ADD

COPY y ADD son comandos que nos permiten añadir ficheros y directorios del host a la imagen base. Los dos comandos son muy parecidos, con la excepción de que ADD nos permite descomprimir ficheros e incluir URL como fuente de datos como se puede ver en el ejemplo:

```
# Copia todos los directorios y ficheros del directorio actual
# en la carpeta /app dentro del contenedor
COPY . /app

# Copia toda la carpeta /web (host) dentro de /app/web
(contenedor)
COPY ./web /app/web

# Copia el fichero sample.txt (host) en /data/my-sample.txt
(contenedor)
COPY sample.txt /data/my-sample.txt

# Descomprimir sample.tar (host) en  /app/bin/ (contenedor)
ADD sample.tar /app/bin/

# Copiar sample.txt (URL) en  /data/ (contenedor)
ADD http://example.com/sample.txt /data/
```

Se permite utilizar comodines en la ruta de origen:

```
# Copia todos los ficheros que empiecen por /sample
COPY ./sample* /mydir/
```

Como vimos en el módulo de Administración de Linux, se puede conseguir escalar privilegios de root por un descuido en los permisos que se le asignan a un directorio. En los contenedores de Docker tenemos que tener el mismo cuidado. Por defecto, las carpetas y ficheros añadidos dentro del contenedor tienen el UID y GID a 0, es decir, root. Una forma de cambiar los permisos con ADD es la siguiente:

```
# Copia toda la carpeta /data (host) dentro de /app/data
(contenedor)
# Le asigna el UID(ddelcastillo) y GID(22) a los ficheros
ADD --chown=ddelcastillo:22 ./data /app/data/
```

También se pueden usar nombres de usuarios y grupos definidos en el /etc/passwd y en el /etc/groups. Si indicamos usuarios o grupos que no existen, la imagen fallará al intentar construirse.

WORKDIR

La palabra WORKDIR establece el directorio de trabajo donde se ejecutará el contenedor de nuestra nueva imagen, por lo que si queremos establecer una carpeta dentro de la imagen lo haremos con el siguiente comando:

```
WORKDIR /app/bin
```

Todo lo que pase a partir de esta línea en el Dockerfile utilizará /app/bin como directorio de trabajo.

El trozo de código que se muestra a continuación:

```
RUN cd /app/bin
RUN touch sample.txt
```

No es igual con el código siguiente:

```
WORKDIR /app/bin
```

```
RUN touch sample.txt
```

El primero crea el fichero sample.txt en (/) aunque parezca que no por el comando CD ya que el comando CD no persiste entre capas.

Con el segundo se crea el fichero sample.txt en /app/bin/, al establecer el directorio de trabajo. WORKDIR es el único comando capaz de establecer contexto entre las capas.

CMD y ENTRYPOINT

Los comandos CMD y ENTRYPOINT tienen una peculiaridad especial que no tienen el resto de comandos. Sin embargo, el resto de comandos se ejecutarán en la fase de construcción de la imagen, CMD y ENTRYPOINT lo harán al arrancar el contenedor definido con nuestra imagen.

Como hemos dicho en otros apartados, los contenedores son la forma que tenemos de ejecutar un proceso o aplicación de forma aislada. CMD y ENTRYPOINT nos permiten especificar qué proceso o aplicación se iniciará al arrancar nuestro contenedor y cómo lo hará.

En el fichero Dockerfile, ENTRYPOINT se utiliza para definir el comando a ejecutar y con CMD definimos los parámetros del parámetro a ejecutar.

Si al arrancar un contenedor definido con nuestra imagen quisiéramos hacer un ping a localhost que envíe 3 paquetes ICMP, el comando en la terminal tradicional sería `ping 127.0.0.1 -c 3`. La manera de expresar este comando en el Dockerfile sería la siguiente:

```
FROM alpine:latest
ENTRYPOINT ["ping"]
CMD ["127.0.0.1", "-c", "3"]
```

Otra forma de ejecutar el comando usando únicamente el ENTRYPOINT sería:

```
FROM alpine:latest
ENTRYPOINT ["ping","127.0.0.1", "-c", "3"]
```

O su equivalente usando solo CMD:

```
FROM alpine:latest
CMD ["ping","127.0.0.1", "-c", "3"]
```

La ejecución de los comandos tanto para CMD como ENTRYPOINT también se puede tratar como una lista de tokens separados por espacios.

```
FROM alpine:latest
CMD ping localhost -c 3
```

Si construimos la imagen usando este Dockerfile:

```
docker image build -t pinger .
```

y arrancamos el contenedor en modo interactivo,

```
docker container run --rm -it pinger
```

podemos ver que se ha ejecutado el ping.

```
ddelcastillo@MacBook-Pro test % docker container run --rm -it pinger
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: seq=0 ttl=64 time=0.063 ms
64 bytes from 127.0.0.1: seq=1 ttl=64 time=0.091 ms
64 bytes from 127.0.0.1: seq=2 ttl=64 time=0.076 ms
```

No se recomienda usar únicamente el comando CMD para ejecutar comandos, ya que al no especificar ENTRYPOINT, por defecto se utiliza /bin/sh -c , lo que nos permite saltarnos la ejecución del comando definido en CMD utilizando otra shell.

```
docker container run --rm -it pinger /bin/sh
```

Si ejecutamos el comando anterior, podemos ver como no se ejecuta el ping:

```
ddelcastillo@MacBook-Pro test % docker run -it pinger /bin/sh
/ # echo "No se ha ejecutado el ping"
No se ha ejecutado el ping
/ #
```

También podemos sobreescribir el ENTRYPOINT del contenedor con la propiedad --entrypoint:

```
docker container run --rm -it --entrypoint /bin/sh pinger
```

Obtenemos la siguiente salida después de ejecutar el comando anterior sobre la imagen del Dockerfile que usaba solamente ENTRYPOINT:

```
ddelcastillo@MacBook-Pro test % docker container run --rm -it --entrypoint /bin/sh pinger
/ # echo "Sobreescribiendo el entrypoint"
Sobreescribiendo el entrypoint
/ #
```

Por todo lo dicho, recomendamos el uso de CMD para los argumentos y ENTRYPOINT para especificar el comando a ejecutar para evitar este problema.

Construyendo la imagen

Como hemos visto en el apartado anterior, podemos construir una imagen de Docker a partir del Dockerfile ejecutando el siguiente comando:

```
docker image build -t pinger -f ./Dockerfile .
```

Con la opción -t (tag), le damos nombre a nuestra imagen.

Con la opción -f, podemos especificar la ruta donde se encuentra el

Dockerfile. Si no la pones, buscará el Dockerfile en el directorio actual.

Con el punto del final estamos especificando la ruta donde se encuentran los ficheros para establecer el contexto del contenedor (jar, js, css, fichero de configuración, etc.). Todos los ficheros que construyen el contexto son comprimidos y enviados a Docker Daemon. Ya que puede ser que el Docker Client se esté ejecutando remotamente, en esta ruta podemos definir un fichero **.dockerignore** (con la misma estructura que .gitignore) para evitar enviar ciertos ficheros.

La salida que nos ofrece el comando es la siguiente:

```
ddelcastillo@MacBook-Pro test % docker image build -t pinger -f ./Dockerfile .
Sending build context to Docker daemon  9.216kB
Step 1/2 : FROM alpine:latest
 ---> f70734b6a266
Step 2/2 : ENTRYPOINT ["ping","127.0.0.1", "-c", "3"]
 ---> Using cache
 ---> 08906e939f21
Successfully built 08906e939f21
Successfully tagged pinger:latest
```

Podemos ver la siguiente información:

- El peso del contexto enviado al Docker Daemon.
- El número de sentencias o capas que tiene nuestra imagen.
- El identificador de caché (si ya está en el host y ha sido cacheada) y de la imagen.
- Y la etiqueta con la que requeriste a esta imagen al hacer un docker run.

Una de las ventajas que nos ofrece que Dockerfile sea multicapa es que podemos tener una imagen para las fases de desarrollo (con herramientas, el SDK y librerías). Elementos que pueden consumir muchos recursos. Y utilizar esa misma imagen quitando las capas necesarias para el desarrollo, sustituyéndolas por una distribución ligera (vista en apartado anteriores), nos permite aprovechar mejor los recursos, generando contenedores pequeños con lo mínimo imprescindible y fáciles de escalar.

Un ejemplo que usa la ventaja de las multicapas en los Dockerfile podría ser este:

- Imaginemos que tenemos que construir un programa complejísimo en C como este hola mundo.

```c
#include <stdio.h>
int main (void)
{
    printf ("¡Hola, mundo! \ n");
    return 0;
}
```

- Nos han dicho que utilicemos Docker como entorno de compilación facilitándonos el siguiente Dockerfile:

```
FROM alpine: 3.7
RUN apk update && apk add --update alpine-sdk
RUN mkdir / app
WORKDIR / app
COPY. / app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin / hello
CMD / app / bin / hello
```

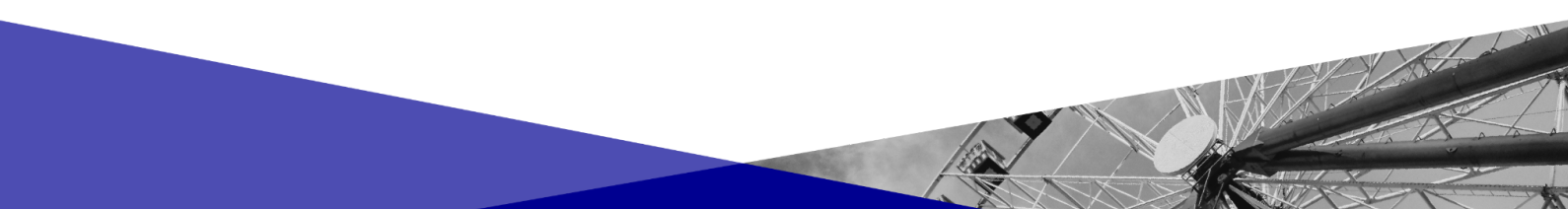
- Como somos un poco curiosos, queremos saber cuánto pesa la imagen después de que la construyamos con `docker image build -t hello-world .` Si ejecutamos el comando `docker image ls | grep hello-world` podemos ver el peso de la imagen:

```
ddelcastillo@MacBook-Pro test % docker image ls | grep hello-world
hello-world         latest              835e7f9aa621        57 seconds ago      178MB
```

- 178 MB!!! ¿Pero no se supone que los contenedores son una virtualización ligera? Se lo comentamos a nuestro jefe y nos dice que pesa tanto por incluir el SDK para desarrollar. Nos pide que miremos cuánto pesaría la imagen que se utilizará en producción utilizando el SDK solo para construir el binario. El Dockerfile que se utilizará en producción es el siguiente:

```
FROM alpine:3.7 AS build
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN mkdir bin
RUN gcc hello.c -o bin/hello

FROM alpine:3.7
COPY --from=build /app/bin/hello /app/hello
CMD /app/hello
```

- Para construir la imagen que se utilizará en producción ejecutamos `docker image build -t hello-world-small .` Si ejecutamos el comando `docker image ls | grep hello-world` y comparamos el



peso de las dos imágenes, podemos ver lo siguiente:

Hemos pasado de 178 a 4.2 MB gracias a que Docker utiliza multicapa a la hora de construir las imágenes. Con este ejemplo se entiende porqué los contenedores de Docker son ligeros, consumen pocos recursos y son muy escalables.

Buenas prácticas con Dockerfile

Aunque a lo largo del documento hemos ido dando algunas recomendaciones, otras buenas prácticas a la hora de definir un Dockerfile son:

- Considerar que los contenedores son efímeros, su tiempo de uso es limitado en el tiempo. Es importante tener esto en cuenta intentando realizar los pasos de instalación y configuración mínimos para no aumentar demasiado el tiempo de construcción de un contenedor.
- El orden de los comandos en el Dockerfile es importante, ya que nos permite aprovechar la caché. Algunas de las reglas que podemos seguir para ordenarlos correctamente son:
    - Pon los comandos estáticos primero (EXPOSE, VOLUMEN, WORKDIR, ENTRYPOINT, etc.).
    - Pon las instrucciones dinámicas (ENV, ADD, COPY, etc.) o que usan contenido que puede cambiar. lo más abajo posible.
    - Dentro de las instrucciones dinámicas, pon ADD y COPY siempre después del RUN.
- Intenta combinar los comandos RUN en uno solo (sobre todo si comparten contexto), ya que se construirían en la misma capa y su ejecución sería más rápida. Un ejemplo de esto podría ser el siguiente:

```
RUN mkdir myproject && cd myproject
RUN echo "hello" > hello
RUN echo -e "FROM busybox\nCOPY /hello/\nRUN cat /hello" >
Dockerfile
```

```
RUN docker build --no-cache -t helloapp:v2 -f
dockerfiles/Dockerfile context
```

En él podemos ver cómo se crea el fichero hello en la segunda sentencia que luego va a ser consultado por el Dockerfile de la tercera. Si unimos la segunda y tercera sentencia con el operador &&, los dos comandos se ejecutarán en la misma capa teniendo el mismo contexto y evitando recurrir a la caché.

Como hemos visto en el ejemplo del apartado anterior (“Construyendo la imagen”) es muy importante construir las imágenes lo más pequeñas posibles. Esto nos aporta beneficios como:

- Reducción del tiempo de carga.
- Necesitamos menos ancho de banda a la hora de descargar las imágenes.
- Necesitamos menos espacio para almacenar la imagen en el host y menos memoria para cargarla.
- Las imágenes pequeñas reducen la superficie de ataque en el caso de tener algún incidente o problema de seguridad.

Algunas de las buenas prácticas para hacer nuestra imagen más pequeña son:

- Utilizar el .dockerignore para excluir todos los ficheros que no son necesarios en nuestra imagen.
- Instalar solo los paquetes imprescindibles evitando instalar paquetes innecesarios o no utilizados que ocupen espacio.
- Por último, intentar hacer compilaciones en varias etapas.

Guardar y cargar una imagen

En este punto mostramos la tercera manera que tenemos de crear una imagen. Los pasos a seguir son los siguientes:

- Guardamos la imagen construida en apartados anteriores como my-alpine.tar ejecutando el siguiente comando:

```
$ docker image save -o ./my-alpine.tar my-alpine
```

- Borramos la imagen con `docker rmi $(docker images | grep 'my-alpine')` para simular que la estamos ejecutando en un equipo que no la tiene. Podemos cargarla en el host de nuevo utilizando el siguiente comando:

```
$ docker image load -i ./my-alpine.tar
```

Compartiendo imagenes: subida y etiquetado

Una vez que ya sabemos cómo crear y usar nuestras imágenes, podemos querer subirlas a un sitio centralizado y compartirlas para que todo el mundo pueda descargarlas. Docker tiene un repositorio online de imágenes llamado DockerHub y es donde vamos a aprender a subir nuestra imagen. Para subir una imagen a DockerHub tenemos que seguir los siguientes pasos:

- Una vez tenemos construida la imagen que queremos compartir, tenemos que hacer un commit. Para realizar el commit de la imagen my-alpine ejecutaremos el siguiente comando (usado en apartados anteriores):

```
docker commit my-alpine
```

- Asignarle una etiqueta. Es lo que indicamos después de los : cuando nos descargamos una imagen al hacer un `docker image pull alpine:3.5`. Normalmente, este campo está destinado al número de versión. Si no le asignamos una etiqueta, Docker le pone latest por defecto. Para etiquetar nuestra imagen ejecutemos el siguiente comando:

```
docker tag my-alpine:latest ddelcastillo/my-alpine:1.0
```

  En el comando podemos ver como la imagen local etiquetada como my-alpine:lastest va a ser etiquetada con ddelcastillo/my-alpine:1.0 que corresponde a el nombre del repositorio en Docker Hub, el nombre de la imagen y su versión, respectivamente.

- Ya estamos listos para subirla, antes de ejecutar el comando push tenemos que iniciar sesión en Docker Hub, lo que implica tener

cuenta. Existe una versión gratuita en la que todos nuestras imágenes son públicas. Por lo que, no utilices este servicio si no pueden ser públicas, aunque para nuestro ejemplo nos vale. Para iniciar sesión utilizamos el comando `docker login`. No especifiques las credenciales con -u y -p haciendo el login interactivo y que nos las solicite.

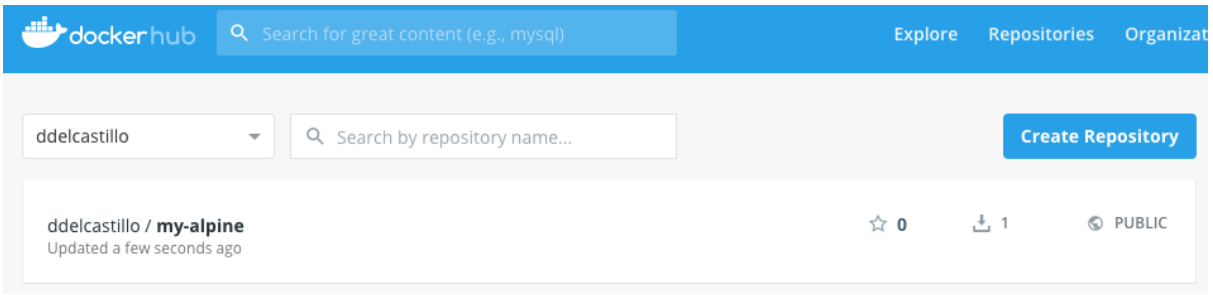
- Por último, ejecutamos el siguiente comando para subir la imagen a Docker Hub:

```
image push ddelcastillo/my-alpine:1.0
```

En la salida podemos ver como Docker Hub ha creado un repositorio público **my-alpine** en la cuenta ddelcastillo para subir nuestra imagen.

```
ddelcastillo@MacBook-Pro test % docker image push ddelcastillo/my-alpine:1.0
The push refers to repository [docker.io/ddelcastillo/my-alpine]
d1316209b7a1: Pushed
3e207b409db3: Mounted from library/alpine
1.0: digest: sha256:05881496b0f37e3dff5311dd5ff6a638dd3138d2ee5e149d585f7b98d4bd921a size: 739
```

También podemos verlo, en nuestra cuenta en Docker Hub.



Para comprobar si podemos hacer pull de nuestra imagen recién subida a Docker Hub, vamos a borrarla para simular que se está descargando de un ordenador distinto al nuestro. Para ello ejecutamos los siguientes comando:

- Hacemos un `docker images ls` para obtener el id de nuestra imagen.

```
ddelcastillo@MacBook-Pro test % docker image ls
REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
ddelcastillo/alpine   1.0                 f70734b6a266        5 weeks ago         5.61MB
alpine                latest              f70734b6a266        5 weeks ago         5.61MB
alpine                3.7                 6d1ef012b567        14 months ago       4.21MB
```

- Con el id de nuestra imagen ejecutamos el siguiente comando para obtener el id de la imagen padre (la imagen de la que ha partido la nuestra):

```
docker inspect --format='{{.Id}} {{.Parent}}' $(docker images
--filter since=f70734b6a266 -q)
```

- Con el id de la imagen padre ejecutamos el siguiente comando:

docker rmi ba854b47fe

Si no lo hacemos así, al haber hecho todo el proceso de creación de la imagen en el mismo ordenador nos saltará el siguiente error:

```
Error response from daemon: conflict: unable to delete
471b6a2bb7cd (cannot be forced) - image has dependent child images
```

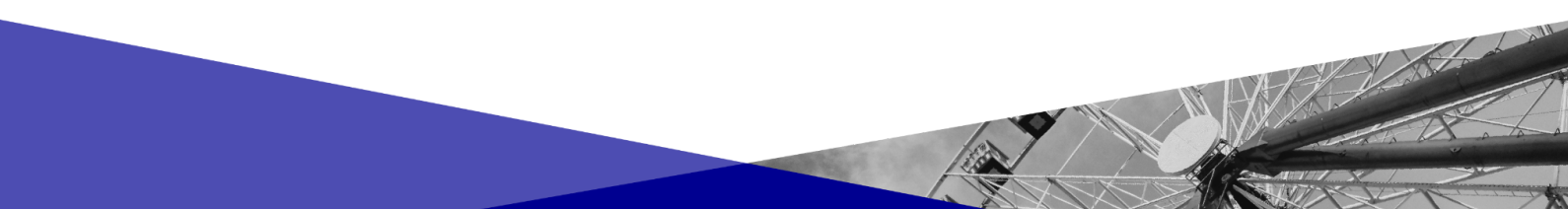
También existe la opción de borrar todos los contenedores con `docker rmi $(docker images -q)`. Esta es una opción más selectiva. Después de realizar los pasos tendremos la imagen **ddelcastillo/my-alpine:1.0** en nuestro local.

Por último, ejecutamos run partiendo de nuestra imagen con el siguiente comando:

```
docker container run --name my_container
ddelcastillo/my-alpine:1.0
```

En la salida podemos ver cómo se ha descargado al hacer el pull de nuestro repositorio.

```
ddelcastillo@MacBook-Pro test % docker container run --name my_container  ddelcastillo/my-alpine:1.0
Unable to find image 'ddelcastillo/my-alpine:1.0' locally
1.0: Pulling from ddelcastillo/my-alpine
cbdbe7a5bc2a: Already exists
db72cf65bcb4: Pull complete
Digest: sha256:05881496b0f37e3dff5311dd5ff6a638dd3138d2ee5e149d585f7b98d4bd921a
Status: Downloaded newer image for ddelcastillo/my-alpine:1.0
ddelcastillo@MacBook-Pro test %
```



Docker Hub

autentia

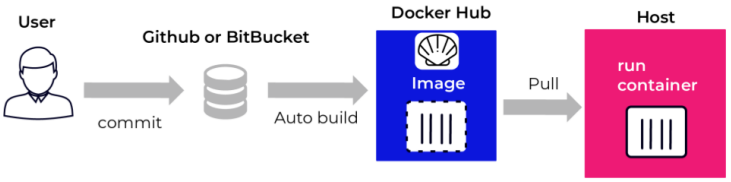
¿Qué es?

Repositorio público en la nube que permite crear, almacenar y distribuir imágenes de Docker gratuitas para ser utilizadas por la comunidad. También permite a los usuarios o empresas crear sus propios repositorios privados para almacenar imágenes propias.

CARACTERÍSTICAS

Algunas de las características más destacables son:

- Permite **descargar y subir imágenes** de Docker ya sea de forma pública o en un repositorio privado. Actualmente, y con el plan gratuito, solo podemos tener un repositorio privado. En caso de que queramos tener más de uno, debemos cambiarnos al plan de pago.
- Permite la **integración con repositorios** como Github o Bitbucket.
- Si se realiza cualquier actualización en el código fuente, se **actualiza automáticamente** y construye la imagen desde el repositorio de Github o Bitbucket.
- Existe el concepto de **Webhook**. Los webhooks son callbacks programadas que se ejecutan a través de un evento de usuario. Los propietarios del repositorio pueden usarlos para integrar Docker Hub con otros servicios.
- Muchas de las imágenes más conocidas y subidas por la comunidad tienen soporte por parte de Docker siempre y cuando hayan pasado una revisión previa.



SUBIR UNA IMAGEN

Una vez hayamos instalado Docker, para poder subir una imagen a nuestro repositorio debemos seguir los siguientes pasos:

- Iniciamos sesión desde la consola con ***docker login***.
- Hacemos un commit de la imagen ***docker commit nombre_imagen.***
- Le podemos añadir una etiqueta con ***docker tag*** y posteriormente subimos la imagen con ***docker push usuario/nombre_imagen.***

Un comando útil para buscar imágenes públicas desde la consola es ***docker search nombre_imagen.*** En caso de querer descargarla, debemos ejecutar ***docker pull nombre_imagen.***

Volúmenes de datos

Los volúmenes se utilizan para mantener el estado en los contenedores pudiendo consumir o generar ficheros persistiendo el estado del contenedor. Esto es debido a que los ciclos de vida de un contenedor y un volumen son diferentes. Mientras que el estado de un contenedor es efímero y todas las modificaciones realizadas en su sistema de archivos desaparecen con él, un volumen mantiene su información incluso cuando el contenedor ha sido eliminado.

Esto resulta útil, por ejemplo, para montar un entorno de pruebas de base de datos para los test de integración sin preocuparse de dejar la base de datos inconsistente. Se arranca la imagen de una base de datos donde poder ejecutar los tests de integración y al terminar se destruye el contenedor.

En los casos en los que queremos persistir archivos dentro de un contenedor, tenemos que trabajar con volúmenes. En los siguientes puntos vamos a explicar cómo se trabaja con ellos.

Gestión de los volúmenes

Según la forma en la que se crean los volúmenes, podemos clasificarlos en:

1. **Con nombre**: Docker gestiona el directorio donde se almacenan los archivos y les damos un nombre para poder hacer referencia a ellos cuando arranquemos otros contenedores.
2. **Anónimos**: se denominan anónimos porque el identificador que se les asigna como nombre es una cadena en SHA-256 que genera Docker. Este tipo de volúmenes no se suelen referenciar.
3. **Del host**: se monta un directorio del host en un directorio del contenedor.

En los siguientes apartados explicaremos cada una de las formas de crear volúmenes en Docker con mayor detalle.

Volúmenes con nombre

Estos volúmenes los creamos fácilmente mediante el comando `docker volume create myvolume`. Esto va a crear un volumen con nombre llamado **myvolume** que luego se puede montar en los contenedores. El volumen va a almacenar los archivos en un directorio gestionado por Docker.

Una vez creado el volumen puede verse la información (fecha de creación, nombre, ruta de los ficheros, el driver utilizado, etc.) relativa a él a través del comando `docker volume inspect myvolume`.

La forma de montar este volumen recién creado es con la opción `-v`, seguida del nombre del volumen y finalmente la ruta en el contenedor:

```
$ docker container run -v myvolume:/data node
```

La ruta /data se refiere a un directorio dentro del contenedor donde van a estar todos los archivos almacenados dentro de la carpeta que destine Docker para el volumen de nombre myvolume. Una vez el volumen deja de estar en uso (los contenedores que lo usaban ya no existen), se puede borrar con el comando `docker volume rm myvolume`.

Hay que tener cuidado y evitar condiciones de carrera cuando **varios contenedores tengan acceso a un mismo volumen**. Es conveniente que a las

aplicaciones se le asignen roles diferenciados dentro del volumen, en las que haya contenedores productores (aplicaciones que crean datos) y consumidores (el resto de aplicaciones que los leen).

Para evitar las condiciones de carrera, podemos restringir los permisos que tiene un contenedor en un determinado volumen añadiendo **:ro** ("read only"), como en el siguiente ejemplo:

```
$ docker container run -v myvolume:/data:ro node
```

El contendor de la imagen "node" sólo va a poder leer los archivos de **myvolume**, pero no podrá escribir en ellos.

Volúmenes anónimos

Cuando definimos un volumen en un Dockerfile y se arranca el contenedor de esa imagen, es el propio Docker el que le asigna un nombre. Este nombre es un código alfanumérico muy largo y no sabemos su valor hasta que arrancamos el contenedor. Esto hace que sea difícil hacer referencia a ellos y por eso los llamamos anónimos.

Para definir un volumen dentro de un Dockerfile se usa la instrucción VOLUME. Existen distintas maneras de definir el directorio del volumen.

```
# Definir un volumen
VOLUME /app/data
# Múltiples volúmenes anónimos separadas por espacios
VOLUME /app/data /usr/src /var/logs
# Sentencia equivalente como una lista de tokens
VOLUME ["/app/data", "/usr/src", "/var/logs"]
```

Cuando un contenedor se arranca, se crea un nuevo volumen con cualquier archivo que exista en esa dirección específica de la imagen. Veamos el siguiente Dockerfile de ejemplo:

```
FROM ubuntu
RUN mkdir /data && echo "hello world" > /data/example
VOLUME /data
```

Este Dockerfile va a crear el directorio /data y luego va a crear el archivo

example. La instrucción VOLUME crea un volumen en /data. Si hay otra instrucción que hace cambios dentro de /data después de la línea en la que se ha declarado VOLUME, estos cambios no tendrán efecto.

Para montar estos volúmenes anónimos tenemos distintas alternativas.

La primera de ellas es obtener el ID que Docker ha asignado a ese volumen inspeccionando su contenedor con `docker inspect mycontainer`. Luego se montaría con `-v <ID>:/container-path`.

Otra forma, es haciendo referencia al contenedor que lo montó desde otro contenedor opción *`--volumes-from`*. En este ejemplo se arranca un contenedor llamado app1 y se montan los volúmenes del contenedor app2:

```
$ docker run  --volumes-from app2 --name app1
```

Volúmenes del host

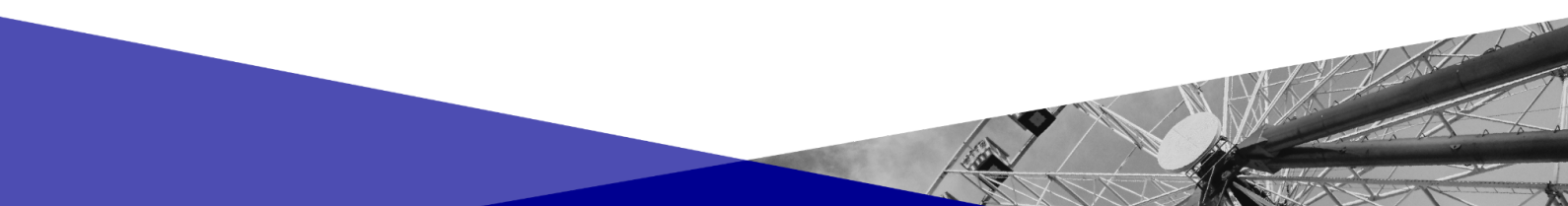
En la mayoría de los casos se utilizan volúmenes con nombre, en los que no conocemos la ruta donde se guardarán los archivos, dejamos que Docker se encargue de ello. Sin embargo, Docker permite que el usuario monte volúmenes en directorios concretos del host. Hay que tener en cuenta que esto crea una dependencia entre el contenedor y el entorno donde se ejecuta, ya que ese directorio no tiene porqué existir en todos los hosts donde pueda ejecutarse el contenedor. Por ese motivo, los volúmenes que se utilizan a la hora de definir los Dockerfile son anónimos, permitiendo que las imágenes que se construyan sean portables y reutilizables en distintos hosts.

Aún así, hay escenarios en los que es útil consumir datos de un directorio específico. Por ejemplo, cuando se está desarrollando una aplicación y se quiere tener el código con las últimas modificaciones, sin necesidad de construir la imagen una y otra vez.

El comando para montar directorios del host es igual al de los volúmenes con nombre, pero en lugar del nombre, hay que indicarle el directorio del host donde se quiere montar. En el siguiente ejemplo, se monta el directorio /var/log del host en el directorio /data del contenedor:

```
$ docker container run --rm -it -v /var/log:/data ubuntu
```

Cuando paremos el contenedor vamos a tener en /var/log todos los datos



generados durante su ejecución.

También podemos restringir el volumen con :ro para permitir solo lectura de ficheros:

```
$ docker container run --rm -it -v /var/log:/data:ro ubuntu
```

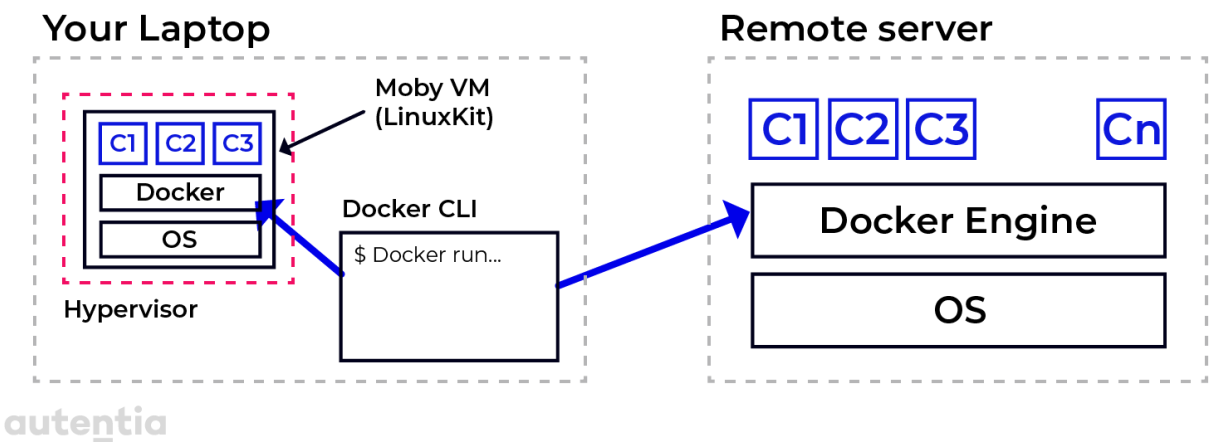
Información del Sistema en Docker

En este apartado mostraremos algunos comandos esenciales para la resolución de problemas con Docker.

Unos de los comandos esenciales es `docker version`, nos proporciona información sobre el cliente y el servidor de Docker tal y como se puede ver en la imagen:

En nuestra configuración actual, tanto el cliente como el servidor están instalados en el mismo host. Queremos aclarar que es común una configuración en la que el cliente no se encuentre en el mismo host que el servidor y se conecte a él de forma remota tal y como indica el siguiente

diagrama:



De esta forma, configurando correctamente el Docker Client podemos gestionar diferentes servidores.

Otro comando importante es `docker system info`, que nos proporciona información sobre el motor de Docker como el resumen sobre el estado de los contenedores, número de imágenes, versiones (del kernel, Containerd, runC, etc.). Nuestra salida después de ejecutar el comando es la siguiente:

```
ddelcastillo@MacBook-Pro ~ % docker system info
Client:
 Debug Mode: false

Server:
 Containers: 2
  Running: 1
  Paused: 0
  Stopped: 1
 Images: 3
 Server Version: 19.03.8
 Storage Driver: overlay2
  Backing Filesystem: <unknown>
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: inactive
 Runtimes: runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: 7ad184331fa3e55e52b890ea95e65ba581ae3429
 runc version: dc9208a3303feef5b3839f4323d9beb36df0a9dd
 init version: fec3683
 Security Options:
  seccomp
   Profile: default
 Kernel Version: 4.19.76-linuxkit
 Operating System: Docker Desktop
 OSType: linux
 Architecture: x86_64
 CPUs: 4
 Total Memory: 1.944GiB
 Name: docker-desktop
 ID: PCG3:PLB7:KU6D:GSVU:6VUA:DMZN:MHMC:Q6BF:2XX6:6J56:FZIO:4EOD
 Docker Root Dir: /var/lib/docker
 Debug Mode: true
  File Descriptors: 51
  Goroutines: 64
  System Time: 2020-06-01T08:48:39.511858142Z
  EventsListeners: 3
 HTTP Proxy: gateway.docker.internal:3128
 HTTPS Proxy: gateway.docker.internal:3129
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: true
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false
 Product License: Community Engine
```

Consumo de recursos

Tras trabajar con un host Docker, este puede generar bastantes recursos: imágenes, contenedores y volúmenes en memoria o disco. Debemos mantener nuestro entorno limpio y libre de recursos, liberando aquellos que no se utilizan para evitar quedarnos sin memoria.

El Client Docker nos proporciona un comando **system** que nos permite listar los recursos que se están utilizando actualmente y cuánto espacio se está usando. El comando es:

```
$ docker system df
```

La salida de este comando es la siguiente:

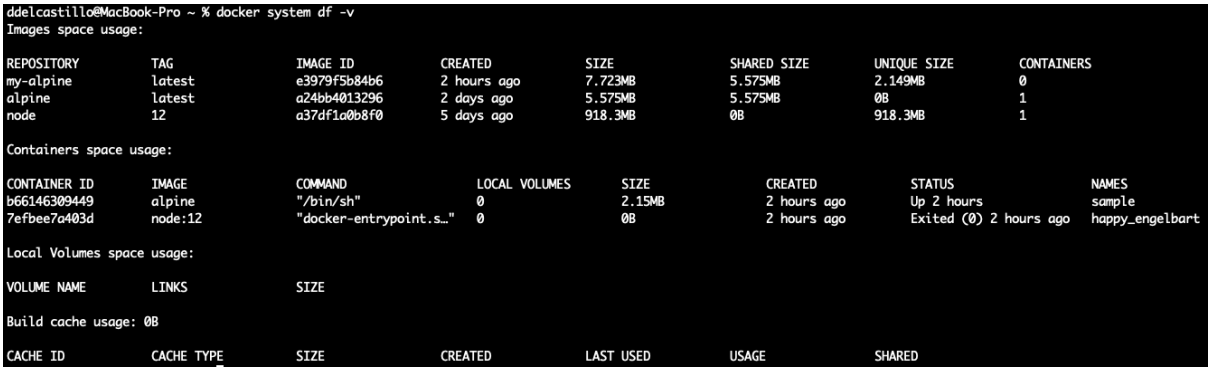
```
ddelcastillo@MacBook-Pro ~ % docker system df
TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
Images          3         2         926.1MB   7.723MB (0%)
Containers      2         1         2.149MB   0B (0%)
Local Volumes   0         0         0B        0B
Build Cache     0         0         0B        0B
```

Si analizamos la información de la salida del comando podemos ver lo siguiente:

- Las imágenes cacheadas y las que están en activo (con algún contenedor ejecutándose).
- La relación entre los contenedores instalados en el host y los que están activos.
- También lista el espacio que ocupan los volúmenes activos (en MB).

Una columna interesante es RECLAIMABLE, que nos indica la cantidad de recursos que se podrían liberar sin afectar al funcionamiento actual del sistema.

Si queremos obtener información más detallada sobre los recursos consumidos (tanto imágenes, como contenedores, volúmenes, etc.), se puede añadir la opción **-v** (verbose) al comando anterior. La salida obtenida después de ejecutar el comando `docker system df -v` es:



Liberar recursos no utilizados

Una vez que hemos detectado que necesitamos liberar recursos, Docker nos proporciona un comando que nos permite hacerlo. Los recursos como las imágenes, contenedores, volúmenes o redes tienen su propio comando **prune**.

Contenedores

En este apartado veremos cómo recuperar los recursos no utilizados por contenedores:

```
$ docker container prune
```

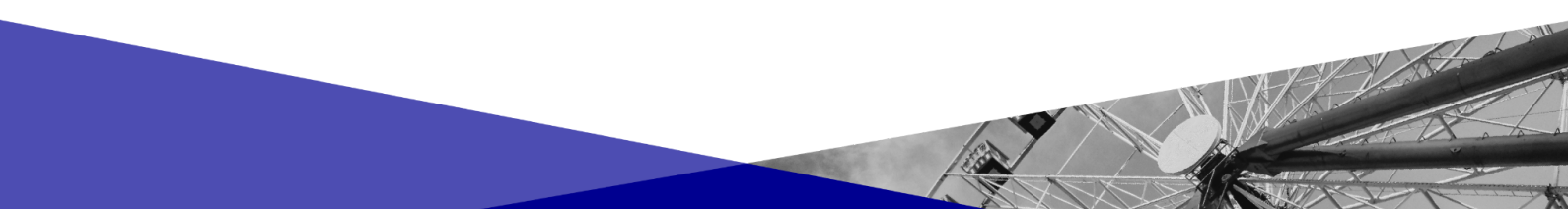
El comando anterior, borraría los contenedores que no se estén ejecutando, pidiendo confirmación antes de hacerlo. Si se quiere ejecutar de forma desatendida hay que añadirle la opción **-f**,

```
$ docker container prune -f
```

En alguna circunstancia, puede ser que necesitemos liberar incluso los contenedores que se están ejecutando. Para esto no podemos utilizar **system prune**. Una alternativa podría ser la ejecución del siguiente comando:

```
$ docker container rm -f $ (docker container ls -aq)
```

ADVERTENCIA: Usa con precaución el comando anterior, ya que borra todos



los contenedores sin pedir confirmación. Se recomienda solo su uso si se entiende perfectamente lo que hace y el por qué lo ejecutamos.

Imágenes

Para liberar el espacio ocupado por las imágenes no usadas, utilizaremos el siguiente comando:

```
$ docker image prune
```

Como hemos explicado en apartados anteriores, las imágenes se componen por capas y éstas son inmutables. Cada vez que hacemos un cambio en una capa, se crea una copia de la anterior con los cambios, dejando huérfana a la capa anterior. Con el comando anterior, liberamos el espacio de estas capas huérfanas, pidiéndonos confirmación antes de hacerlo. Si queremos la ejecución de este comando de forma desatendida, le añadimos la opción **-f**, de la siguiente forma:

```
$ docker image prune -f
```

El comando que nos permite el borrado de todas las imágenes que no se estén utilizando en nuestro local, no solo las capas huérfanas, es el siguiente:

```
$ docker image prune --force --all
```

Volúmenes

Como hemos visto en apartados anteriores, los volúmenes son la manera de compartir y persistir datos por parte de los contenedores. Estos datos pueden ser importantes, así que pedimos que se haga un uso de los comandos de este apartado con precaución.

Si queremos eliminar los volúmenes que no se estén utilizando, ejecutaremos el siguiente comando:

```
$ docker volume prune
```

Tened en cuenta que este comando es irreversible y no se puede deshacer. Se recomienda hacer copias de seguridad de los volúmenes antes de la

ejecución de este comando para evitar problemas.

Para evitar que el sistema se corrompa o tenga un mal funcionamiento, Docker solo eliminará los volúmenes que no estén siendo utilizados por ningún contenedor. En el caso de que un volumen se use, aunque sea por un contenedor parado y éste se desee liberar, primero habrá que eliminar el contenedor.

Una forma fácil de reducir el campo de acción a la hora de liberar volúmenes es el uso de filtros con la opción **--filter** de la siguiente forma:

```
$ docker volume prune --filter 'label=demo' --filter 'label=test'
```

De esta manera, solo se liberarán los volúmenes que cumplan con los criterios especificados. En el ejemplo anterior, se están intentando liberar los volúmenes con la etiqueta demo o test. La opción --filter también puede usarse a la hora de liberar contenedores o imágenes.

Redes

Aunque veremos con mayor detalle las redes en Docker en el siguiente apartado, la manera de eliminar las redes no utilizadas por ningún contenedor o servicio es ejecutando el siguiente comando:

```
$ docker network prune
```

Todo

Si quisiéramos liberar todo a la vez sin especificar el recurso podríamos ejecutar el siguiente comando:

```
$ docker system prune
```

Nos pedirá confirmación para borrar todos los contenedores, imágenes, volúmenes y redes no utilizadas. Recuerda que para hacerlo de forma desatendida, como en anteriores comandos, añadiremos la opción **-f**.

Eventos

Por último, comentar que cada vez que se libera un recurso, esto provoca

un evento que puede ser consumido por una aplicación externa. El comando que nos permite ver los eventos producidos es el siguiente:

```
$ docker system events
```

La ejecución de este comando es bloqueante, y deja inutilizada la sesión donde se ejecute. Se recomienda abrir una segunda terminal para ejecutarlo. La salida después de ejecutar el comando `docker system events` es la siguiente (recordad que los eventos registran toda actividad, no solo el último comando, esto es solo un ejemplo):

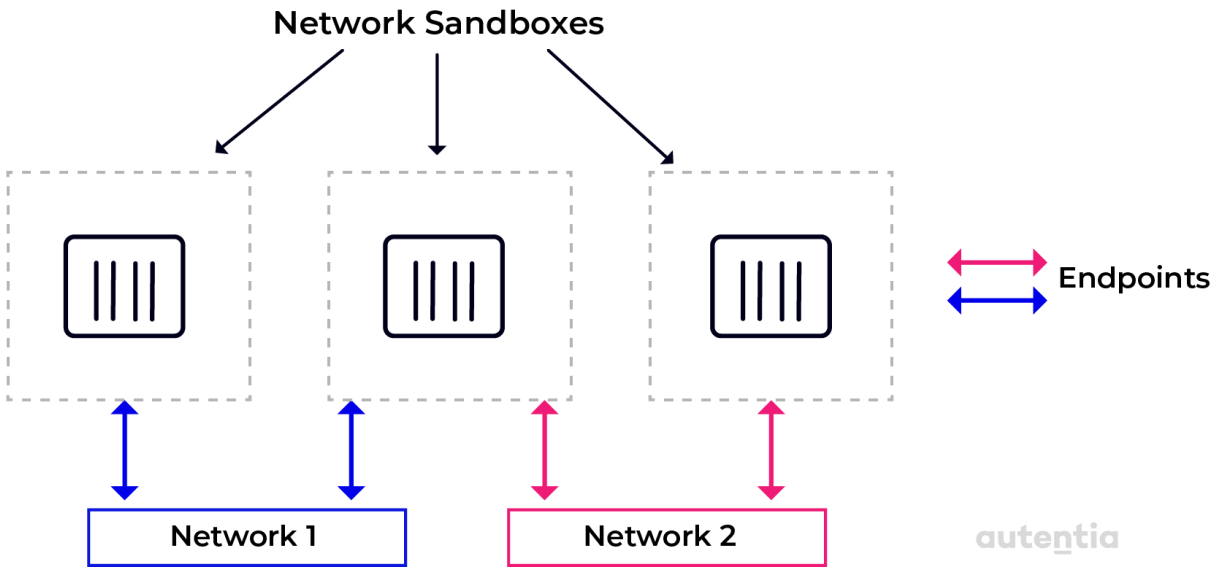
Una forma de hacer la salida más legible es modificar la opción **--format** de la misma. Esto es útil para extraer solo la información que nos interesa de los eventos. Para extraer únicamente el tipo de recurso, su imagen y la acción realizada podemos indicarle el siguiente formato:

```
$ docker system events --format 'Type={{.Type}}
Image={{.Actor.Attributes.image}} Action={{.Action}}'
```

Redes

Modelo de red en un contenedor

Hasta ahora, hemos estado trabajando con contenedores individuales. Pero, en realidad, una aplicación consta de varios contenedores que colaboran para lograr que todo funcione correctamente. Para lograr esa colaboración, necesitamos una forma de que los contenedores se comuniquen entre sí. Esto se logra estableciendo redes de Docker. Estas redes permiten el envío de paquetes entre contenedores basadas en el modelo CNM (container network model). En la siguiente imagen podemos ver una representación gráfica de una CNM con 3 contenedores y 2 redes:



El modelo CNM consta de tres elementos principales:

- **Sandbox**: es la pieza que aísla al contenedor del mundo exterior. No permite conexiones entrantes al contenedor, manteniéndolo aislado. Solo es posible comunicarse con él a través de **endpoints**.
- **Endpoint**: es una puerta de enlace controlada entre el mundo exterior y el entorno de acceso limitado que protege el contenedor. El endpoint conecta el entorno aislado del contenedor con la **red** en sí.
- **Network**: en esencia, es la ruta que permite transportar los paquetes

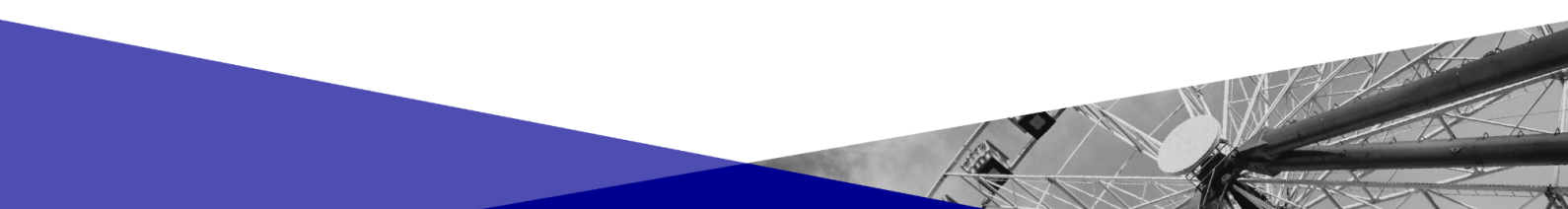
desde un endpoint a otro, es decir, comunica las zonas de acceso limitado de los diferentes contenedores.

Cada componente puede estar relacionado con uno, varios o ningún otro componente, es decir, una sandbox puede tener acceso a uno, varios o ningún endpoint, dando acceso al contenedor a una, varias o ninguna red. Es un modelo flexible que no impone restricciones. Es un modelo de red genérico que no especifica donde deben ejecutarse los contenedores, si de forma local (en el mismo host) o global (en diferentes host).

Pero CNM es solo un modelo de red. Para usarlo necesitamos implementaciones reales del mismo. En la siguiente tabla, presentamos una breve descripción de las implementaciones existentes y sus características principales:

Red	**Empresa**	**Alcance**	**Descripción**
Bridge	Docker	Local	Red simple basada en puentes de Linux para permitir la creación de redes en un solo host.
Macvlan	Docker	Local	Configura múltiples MAC en una sola interfaz de red.
Overlay	Docker	Global	Red de contenedores con capacidad multinodo basada en LAN virtual extensible (VXLan).
Weave Net	Weaveworks	Global	Redes Docker simples y de múltiples hosts.
Contiv Network Plugin	Cisco	Global	Redes de contenedores de código abierto.

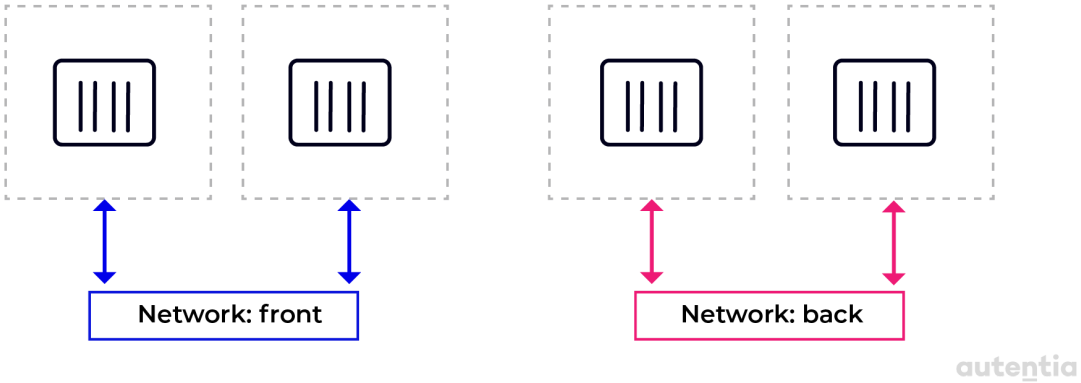
Todos los tipos de red no proporcionados directamente por Docker se pueden agregar a un host Docker como un complemento.



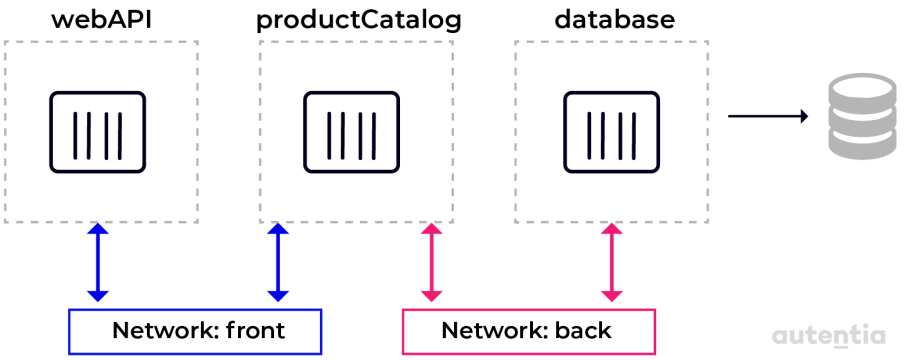
Cortafuegos

Las SDNs (Software Defined Networking) son fáciles y baratas de crear, protegiendo a los contenedores conectados a la red de otros que no lo están y del mundo exterior. Todos los contenedores que pertenecen a la misma red pueden comunicarse libremente entre sí.

La siguiente imagen nos muestra dos redes front (con los contenedores c1 y c2) y back (con los contenedores c3 y c4). Los contenedores de la red front no podrán comunicarse con los de la red back.



En este ejemplo, solo los contenedores que pertenecen a la misma red pueden colaborar entre sí. Un ejemplo más real que conecta varias redes compartiendo un contenedor es el siguiente:



En él podemos ver 3 servicios (**webAPI**, **productCatalog** y **database**) en los que hemos establecido la restricción de que webAPI solo accede a productCatalog y no a la base de datos. Para ello, hemos conectado el contenedor que comparten tanto webAPI como la base de datos a las 2 redes.

Este ejemplo nos muestra una forma de utilizar las SDNs para restringir el acceso a los recursos. Restringiendo la comunicación directa entre webAPI y la base de datos. De esta manera, si sufrimos un ataque informático a webAPI, el atacante tendrá que conseguir acceso también a productCatalog si quiere acceder a la base de datos, ejerciendo el contenedor productCatalog de cortafuegos entre las dos redes.

Implementaciones de red

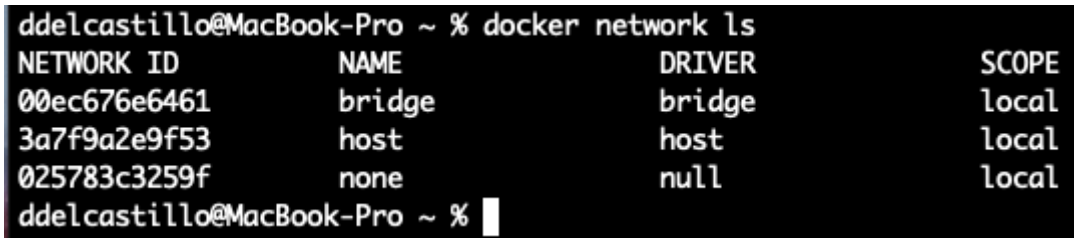
Puente

Es una implementación de red basada en el bridge de Linux. Cuando el Docker Daemon se ejecuta por primera vez, crea un bridge llamado **docker0** junto con otros tipos de redes. Todos los contenedores creados en el host que no estén vinculados a otra red, se conectaran automáticamente a este bridge.

Para listar todas las redes que docker crea por defecto, ejecutaremos el siguiente comando:

```
$ docker network ls
```

Que mostrará la siguiente salida:

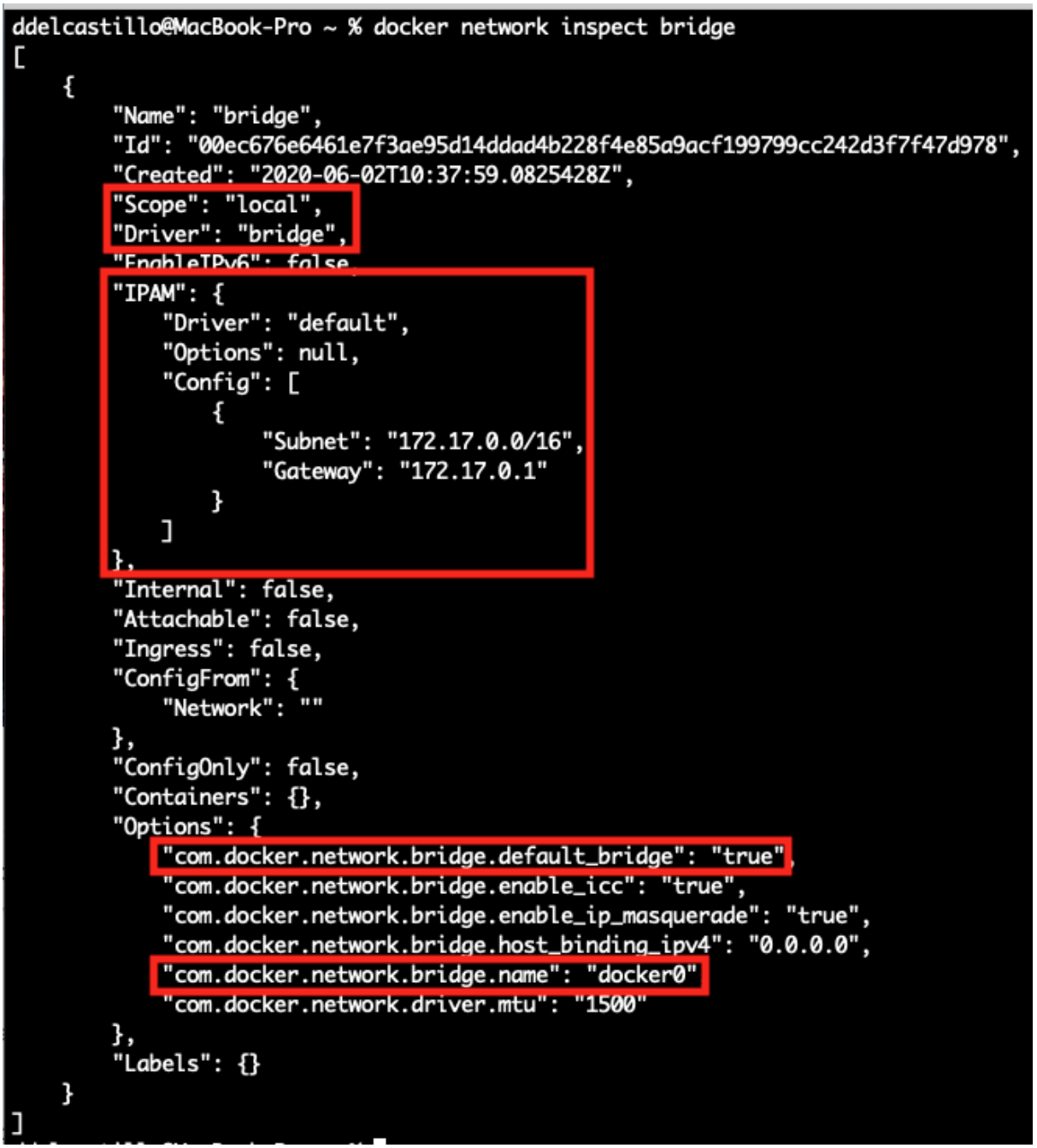


Podemos ver que todas las redes creadas tienen el alcance local y no soportan múltiples hosts. Todas estas redes serán examinadas a lo largo de

los siguientes apartados. Si queremos ver de una forma más detallada la configuración de la red bridge, podemos hacerlo ejecutando el siguiente comando:

```
$ docker network inspect bridge
```

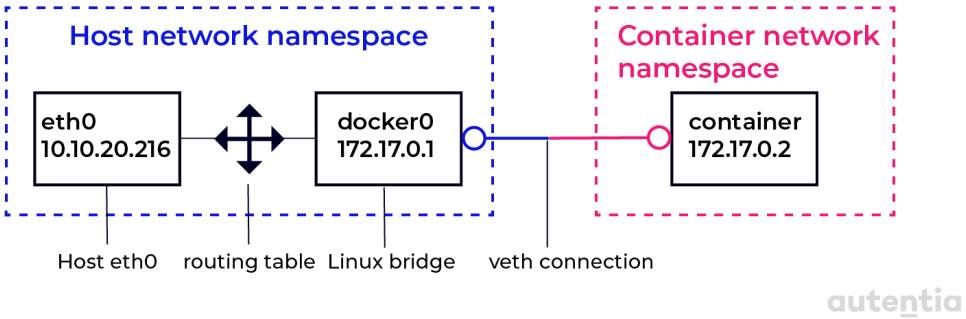
Si examinamos su salida podemos ver lo siguiente:



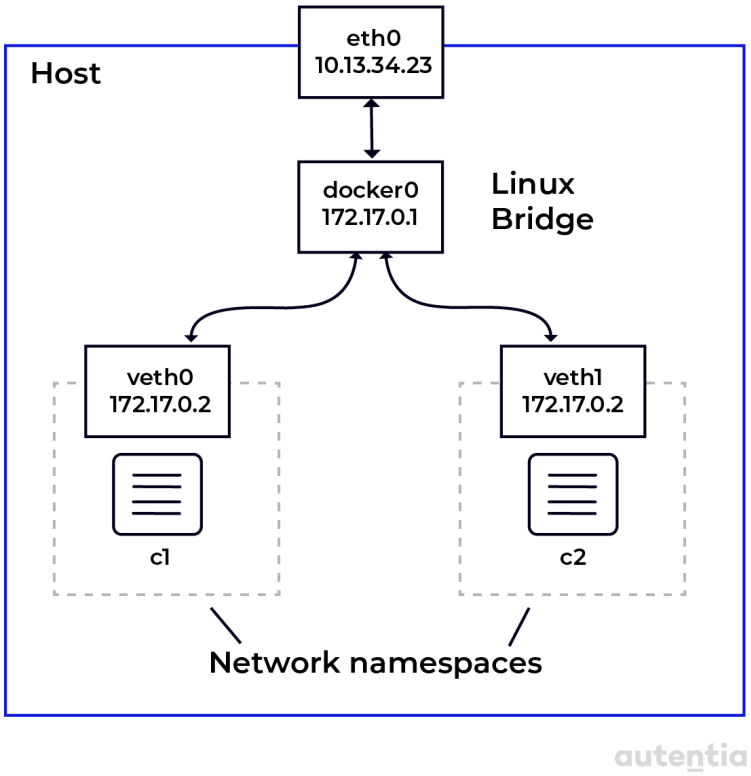
En la salida podemos ver el id de bridge, así como su nombre y su alcance, visto en el comando anterior. También podemos ver la gestión de de direcciones IP mediante IPAM (IP address management). IPAM es un software que se utiliza para rastrear direcciones IP que se usan en el host. Una de las partes más importantes dentro de la configuración IPAM son los valores referidos a subred y puerta de enlace.

La subred se define por defecto como 172.17.0.0/16. Esto significa que todos los contenedores conectados a esta red obtendrán una dirección IP asignada por Docker que se toma del rango dado, que es 172.17.0.2 a 172.17.255.255. La dirección 172.17.0.1 está reservada para el enrutador (función ejercida por el bridge de Linux).

A la vista de estos datos, sabemos que al primer contenedor que se cree se le asignará la dirección 172.17.0.2. Como muestra el siguiente diagrama:



Vemos que el bridge de Linux enruta todo el tráfico de red que viene de la interfaz eth0. Dando la posibilidad al contenedor de salir a internet (tráfico saliente) pero no permitiendo el tráfico entrante. Cada contenedor se conecta mediante una conexión virtual de ethernet (**veth0** y **veth1**) al bridge (**docker0**), como se puede ver en el siguiente diagrama, donde podemos ver las comunicaciones desde el punto de vista del host:



En Docker no solo podemos conectarnos al bridge que se crea al arrancar, también podemos crear nuestras propias redes bridge. Se considera una buena práctica crear redes que nos permitan aislar a los contenedores entre sí y no que estén conectados todos a una misma red.

Para crear una red podemos podemos ejecutar el siguiente comando:

```
$ docker network create --driver bridge sample-net
```

Si inspeccionamos la subred que tiene la red recién creada con el siguiente comando

```
$ docker network inspect bridge
```

podemos ver que se ha creado una nueva subred con el siguiente valor libre a la que tenía asignada la red bridge (172.18.0.0/16). También la podemos especificar nosotros con la opción **--subnet**

```
$ docker network create --driver bridge --subnet "10.1.0.0/16"
test-net
```

Ahora, si levantamos un contenedor con el modo de consola interactiva

(`docker container run --name c1 -it --rm alpine:latest /bin/sh`) y le hacemos un `ip addr` para ver a qué red está conectado, veremos lo siguiente:

```
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
3: ip6tnl0@NONE: <NOARP> mtu 1452 qdisc noop state DOWN qlen 1000
    link/tunnel6 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00 brd 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

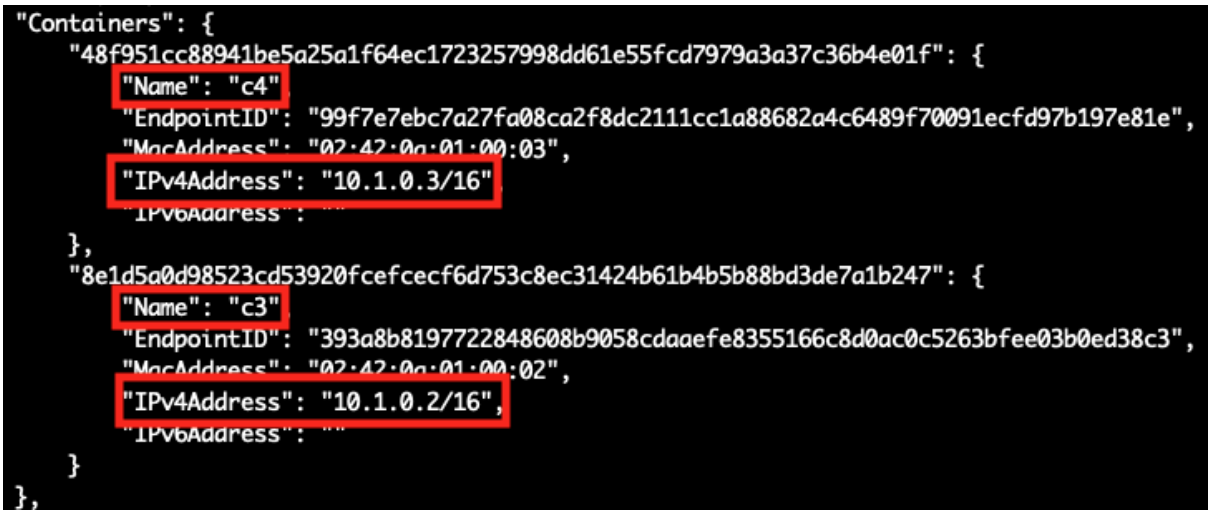
Y si seguimos creando contenedores, se seguirán asignando IP consecutivas del bridge. Como podemos ver en la salida de `docker network inspect bridge`.

```
"Containers": {
    "0f534572aeac7177a9fba41bf6ca70460d7d4011ee990375460d792204740cd9": {
        "Name": "c1",
        "EndpointID": "2c809eaced79af48166dc749218c2d4af7a2e1e804ac24eda656b319d378e93b",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
    },
    "9767936dab0e0cd0280426673147b54eb2a1af2362715b34c7caaf21f86f0afb": {
        "Name": "intelligent_mendel",
        "EndpointID": "6310822e45bc83b682782d51b276d6e90a9116f575f49acb43cd097276fcfd23",
        "MacAddress": "02:42:ac:11:00:03",
        "IPv4Address": "172.17.0.3/16",
        "IPv6Address": ""
    },
    "e1f0b59732840abc9ecfb51b01646b9462d891701538b1b2505c350e80857cab": {
        "Name": "c2",
        "EndpointID": "ba402b0d28a07319c7bab557f93d91674ea1befe317a55e4b4a724e8bf50a0da",
        "MacAddress": "02:42:ac:11:00:04",
        "IPv4Address": "172.17.0.4/16",
        "IPv6Address": ""
    }
```

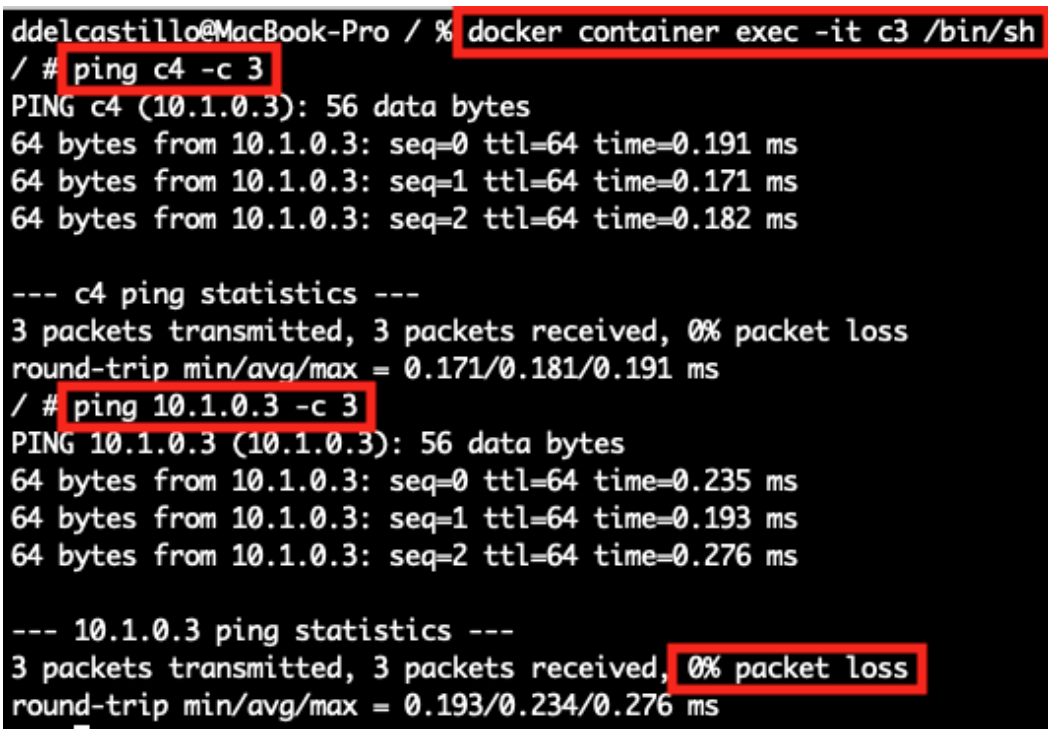
Podemos levantar 2 contenedores (c3 y c4) y conectarlos a la red “test-net” creada en pasos anteriores con los siguientes comandos:

```
$ docker container run --name c3 -d --network test-net \
   alpine:latest ping 127.0.0.1
$ docker container run --name c4 -d --network test-net \
   alpine:latest ping 127.0.0.1
```

Con la opción **--network** podemos especificar a qué redes está conectado el contenedor. Si inspeccionamos ahora la red **test-net** podemos ver que los los dos contenedores están conectados:



Para comprobar la conectividad entre los dos contenedores, vamos a conectarnos al contenedor c3 de forma interactiva y hacer un ping a la IP del otro contenedor. En la siguiente imagen podemos ver que los paquetes ICMP del ping llegan al contenedor c4 al hacer el ping:



Docker no permite establecer múltiples redes en el momento de arrancar el contenedor. Para conectar un contenedor c5 a las redes test-net y

sample-net tenemos que seguir los siguientes pasos:

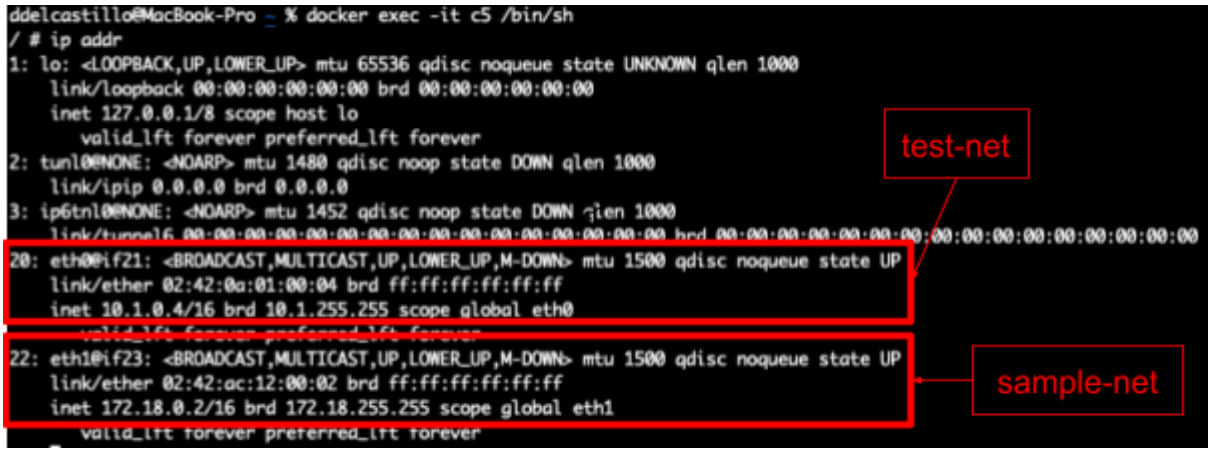
- Levantamos el contenedor c5 conectado a la red test-net. Lo ejecutamos con la opción -d (detach) para ejecutar el contenedor en background:

```
docker container run --name c5 -dit --network test-net
alpine:latest ping 127.0.0.1
```

- Conectamos el contenedor c5 a la red sample-net con el siguiente comando:

```
docker network connect sample-net c5
```

- Si nos conectamos a c5 y hacemos un `ip addr` podemos ver 2 nuevas interfaces de red, cada una con la ip de cada red.



Si quisiéramos borrar la red test-net, tendríamos que ejecutar el comando docker network rm test-net. Pero nos da el siguiente error:

```
Error response from daemon: error while removing network: network test-net id 9c61
c5add58d7ef11b10dbe4d259fd691b1e8ea7eca148935bf78d710132d210 has active endpoints
```

El error nos indica que no se puede borrar una red que tiene contenedores conectados. Para desconectar un contenedor de la red usa el siguiente comando:

```
docker network disconnect test-net c5
```

Ahora ya podríamos borrarla sin problemas ejecutando de nuevo docker network rm test-net:

Otra manera es borrando el contenedor:

```
docker rm -f  c5
```

Una vez que la red no tenga ningún contenedor conectado podemos borrar la red sin problemas.

Otra opción es forzar a Docker a que libere todos las redes con el comando **prune** como hemos visto en el apartado anterior. El comando a ejecutar sería el siguiente:

```
docker network prune --force
```

ADVERTENCIA: este comando no es selectivo y eliminará todas la redes establecidas.

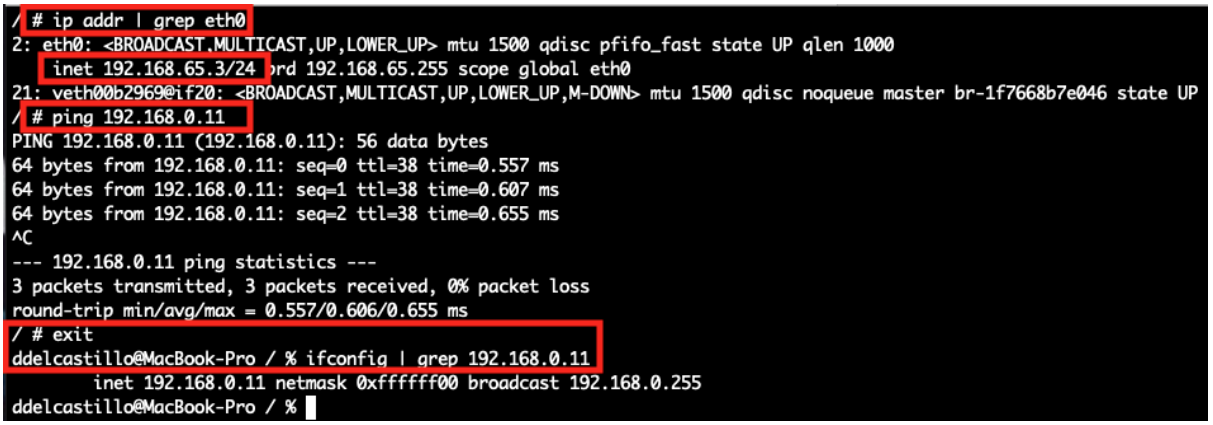
Misma red que el host

En ocasiones, necesitamos que nuestro contenedor comparta la red del host (para tareas de debug o uso de herramientas para analizar el tráfico). Solo se recomienda el uso de este tipo de red en estos escenarios tan específicos, nunca de forma habitual por motivos de seguridad.

La manera de ejecutar un contenedor compartiendo el namespace de red del host es ejecutando el siguiente comando:

```
$ docker container run --rm -it --network host alpine:latest
/bin/sh
```

Si ejecutamos `ip addr` dentro del contenedor, podemos ver que a la interfaz de red eth0 se le ha asignado una IP que pertenece a la misma red que host. Al que incluso le podemos hacer un **ping** al host, tal y como se ve en la imagen.



ADVERTENCIA: el uso de este tipo de network puede causar problemas de seguridad, comprometiendo al host. Utilizar solo en situaciones muy específicas.

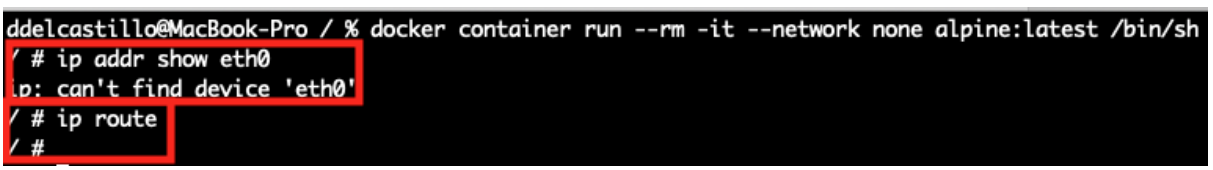
Sin conexión

A veces utilizamos contenedores que no necesitan acceso a internet para hacer su trabajo. Algunos ejemplos podrían ser un contenedor que solo realiza la compilación de un código fuente y deja el binario en un volumen compartido u otro contenedor que reduce el tamaño de un XML grandísimo, aplicando un XSLT y dejando el resultado en otro volumen compartido.

En definitiva, siempre que no se necesite conectividad a la red por parte de un contenedor para realizar su función, se recomienda restringirla. La forma de hacerlo es especificando **none** en la opción **--network**, tal y como se ve en el ejemplo:

```
$ docker container run --rm -it --network none alpine:latest
/bin/sh
```

Si intentamos ver la información de la interfaz de red eth0 o hacer un `ip route` nos muestra lo siguiente:



Con este comando podemos verificar que nuestro contenedor no tiene acceso a la red.

Gestión de puertos

Ahora que sabemos cómo aislar o conectar diferentes contenedores utilizando redes, tenemos que ver cómo permitir el acceso desde el contenedor a los puertos disponibles del host. Ya que existen diferentes maneras.

Una de ellas es utilizando la opción **-P** (en mayúsculas) o **--publish-all,** la cual asigna aleatoriamente un puerto libre del host. Un ejemplo podría ser el siguiente:

```
$ docker container run --name myweb -P -d nginx: alpine
```

Para ver el puerto del host que Docker nos ha asignado tenemos 3 formas:

- Ejecutando docker container port, que en nuestro caso sería:

```
$ docker container port myweb
```

  Indicando myweb, ya que es el nombre de nuestro contenedor.

- Buscar la etiqueta **HostPort** en el nodo **NetworkSettings** al inspeccionar el contenedor myweb, ejecutando el siguiente comando:

```
$ docker container inspect myweb | grep HostPort
```

- Hacer un docker container ls y fijarnos en la columna **port**:

Otra manera es usar la opción **-p** (en minúsculas) o **--publish** para especificar en qué puertos se expondrá el servicio. Utilizada en el siguiente comando:

```
$ docker container run --name web2 -p 8080:80 -d nginx:alpine
```

Los valores que se le pasan como parámetro a la opción -p tienen el siguiente formato **<host port>:<container port>**. Lo que indica que el último comando levantará el servicio de Nginx en el puerto 80 del contenedor, exponiendolo en el 8080 del host. Es decir, podríamos acceder a él si

introducimos localhost:8080 en nuestro navegador.

Con la opción -p también podemos especificar el protocolo del puerto a exponer. Un ejemplo exponiendo un puerto UDP podría ser:

```
$ docker container run --name myweb3 -p 3000:4321/udp -d
nginx:alpine
```

Esta es la manera de utilizar distintos protocolos por el mismo puerto. Si no se le especifica protocolo, por defecto son TCP.

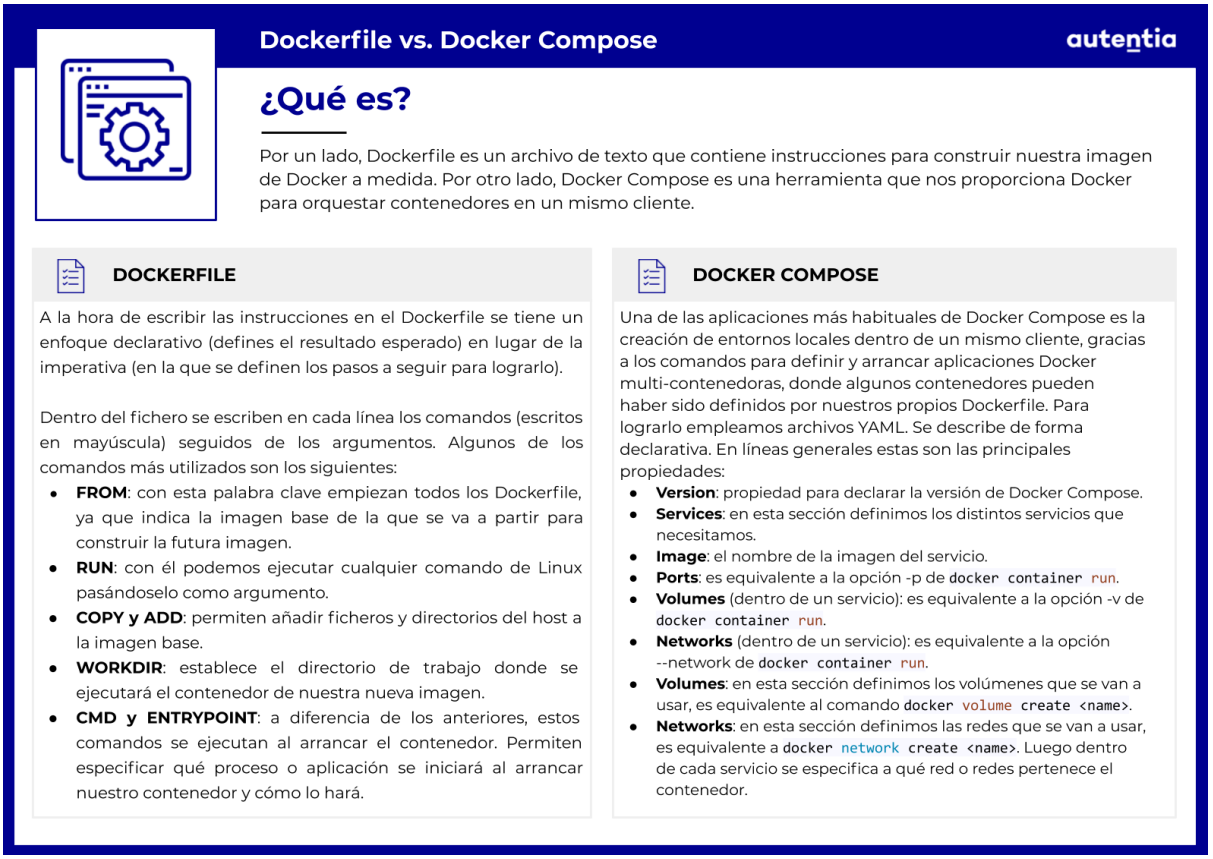
Otras formas de gestionar puertos es definiéndolo en un Dockerfile utilizando la palabra reservada EXPOSE (para usar con la opción -P, ya que si usamos el mapeo de puerto con la opción -p, sobreescribimos el puerto definido en la sentencia EXPOSE) o mapeandolos dentro del fichero yaml para usar docker compose (similar a la opción -p) como veremos en el siguiente apartado.

Docker Compose

Docker Compose es una herramienta que nos proporciona Docker para orquestar contenedores en un mismo cliente. Una de las aplicaciones más habituales es la creación de entornos locales.

Para usarlo, empleamos archivos con el formato YAML (*YAML Ain't Markup Language).* En general el archivo se llamará docker-compose.yml, pero es posible ponerle el nombre que queramos. Dentro del archivo se describe **de forma declarativa** el conjunto de contenedores que componen nuestra aplicación.

Cuando decimos que se describe de forma declarativa nos referimos a que en el archivo se establece el estado que tiene que tener la aplicación y es Docker el que se va a encargar de llegar a ese estado, sin indicar el procedimiento para conseguir esa configuración.



Arrancando un App Multi-Servicio

En la mayoría de los casos, las aplicaciones no consisten en un solo monolito, sino que usan diferentes servicios (bases de datos, colas de mensajería, ftps, etc.). Cuando usamos contenedores con Docker, cada servicio se ejecuta en su propio contenedor. Para ejecutar todos estos servicios, podríamos arrancarlos usando los comandos que hemos visto a lo largo del documento, pero esto nos obligaría a hacer un trabajo manual tedioso (levantado los contenedores, creando redes, volúmenes, etc.) y poco eficiente. Docker Compose nos facilita esta tarea definiendo todas estas operaciones y dependencias en el fichero YAML.

A continuación se muestra un ejemplo de Docker Compose con dos contenedores:

```
version: "3"
services:
  jenkins:
      image: "jenkins/jenkins:lts"
      ports:
        - "8080:8080"
      volumes:
        - jenkins-data:/var/jenkins_home
      networks:
        - mynet
  sonarqube:
      image: "sonarqube"
      ports:
        - "9000:9000"
      networks:
        - mynet
volumes:
   jenkins-data:
networks:
  mynet:
      driver: bridge
```

En líneas generales estas son las principales propiedades:

- **Version**: en esta línea declaramos la versión del formato de Docker Compose que se va a usar, en este ejemplo la 3.
- **Services**: en esta sección definimos los distintos servicios que necesitamos. En el siguiente nivel de indentación tenemos los nombres que le hemos puesto a los contenedores: “jenkins” y “sonarqube”.
- **Image**: el nombre de la imagen del servicio.
- **Ports**: es equivalente a la opción -p de `docker container run`.
- **Volumes** (dentro de un servicio): es equivalente a la opción -v de `docker container run`. Se puede hacer referencia a volúmenes que se definan al final del archivo.
- **Networks** (dentro de un servicio): es equivalente a la opción --network de `docker container run`. También se puede hacer referencia a las redes que se creen al final del archivo.
- **Volumes**: en esta sección definimos los volúmenes que se van a usar, es equivalente al comando `docker volume create <name>`.
- **Networks**: en esta sección definimos las redes que se van a usar, es equivalente a `docker network create <name>`. Luego dentro de cada servicio se especifica a qué red o redes pertenece el contenedor.

Existen más propiedades que soporta Docker Compose pero estas son las más básicas. Para profundizar más lo mejor es ir a la documentación oficial.

Construir imágenes con docker compose

Para construir las imágenes que hemos definido en un *docker-compose.yml* simplemente tenemos que ir al directorio en el que se encuentra el archivo y ejecutar:

```
$ docker-compose build
```

Este comando va a funcionar de igual manera que si ejecutamos un `docker container build` individualmente para cada uno de los servicios que

tenemos en nuestro *docker-compose.yml*.

Si el fichero *docker-compose.yml* se encuentra en un directorio diferente al actual (donde ejecutamos el comando), podemos especificar su ruta con la opción -f, tal y como muestra el ejemplo:

```
$ docker-compose -f ~/my-docker-folder/docker-compose.yml build
```

Arrancar una aplicación con docker compose

Una vez construidas las imágenes se arranca la aplicación con:

```
$ docker-compose up -d
```

La opción -d (detach) sirve para ejecutar los contenedores en segundo plano. Si hacemos cambios en nuestro *docker-compose.yml* podemos volver a ejecutar el comando y Docker se encarga de levantar los contenedores que tienen cambios.

Una vez hemos terminado de usar la aplicación podemos parar y borrar los contenedores (también borra las redes) con:

```
$ docker-compose down
```

Escalar un servicio

Si tenemos una aplicación web y empieza a tener muchos usuarios, vamos a necesitar varias instancias de nuestro servicio web para poder satisfacer la demanda.

Imaginemos que tenemos una aplicación web y un balanceador:

```
version: "3"
services:
  web:
      image: 'dockercloud/hello-world:latest'
      networks:
        - mynet
  loadbalancer:
```

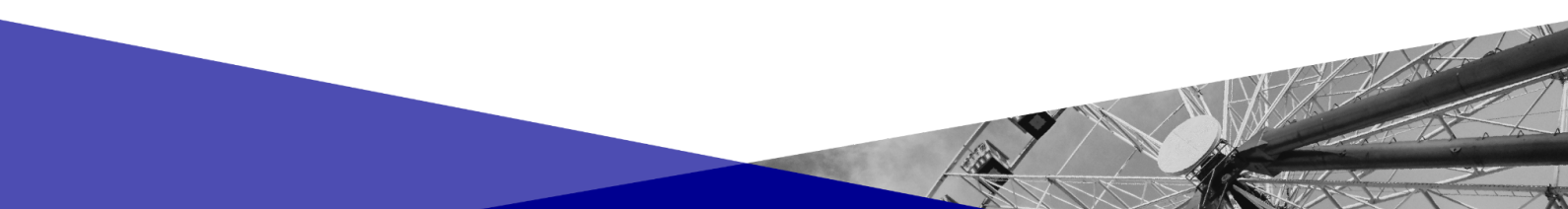
```
      image: 'dockercloud/haproxy:latest'
      ports:
        - '80:80'
      networks:
        - mynet
networks:
  mynet:
      driver: bridge
```

El balanceador va a recibir las peticiones por el puerto 80 y queremos que las distribuya entre las instancias de web. Una posible opción sería copiar la sección del servicio web tantas veces como instancias queramos. Esta opción tiene las desventajas de que es manual y estática, ya que nos obliga a modificar el docker-compose.yml si nuestra carga baja o sube.

Afortunadamente docker-compose nos permite aumentar las instancias de un servicio fácilmente con la opción **--scale** seguida del nombre del servicio con el número de instancias que queremos.

```
$ docker-compose up --scale web=3
```

Cuando arranca los contenedores, se encarga de darles un nombre único a cada una de las instancias (añadiendo un número al final) de manera transparente para nosotros.



Parte 5

Ansible

Introducción

En la actualidad desplegamos las aplicaciones de software juntando distintos servicios que se comunican entre ellos de diversas formas. Hablamos de servidores web, servidores de aplicaciones, colas de mensajería, bases de datos, balanceadores, etc.

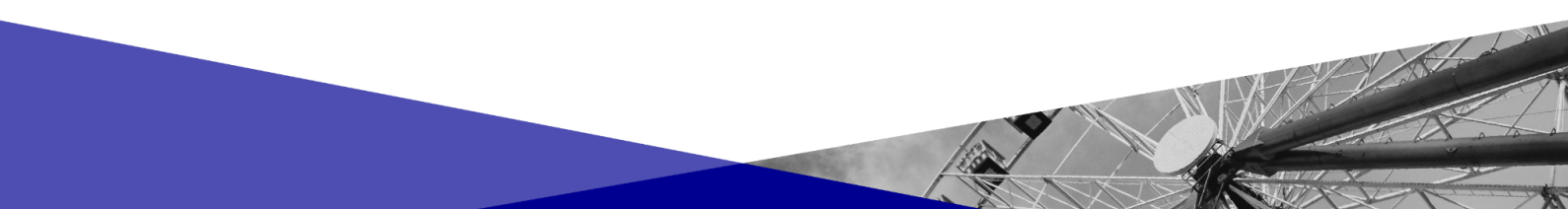
También es necesario controlar los fallos para que cuando sucedan se puedan gestionar y seguir prestando servicio, por lo que aparecen servicios de logs y monitorización que debemos gestionar, además de los servicios de terceros con los que el sistema también tiene que interactuar.

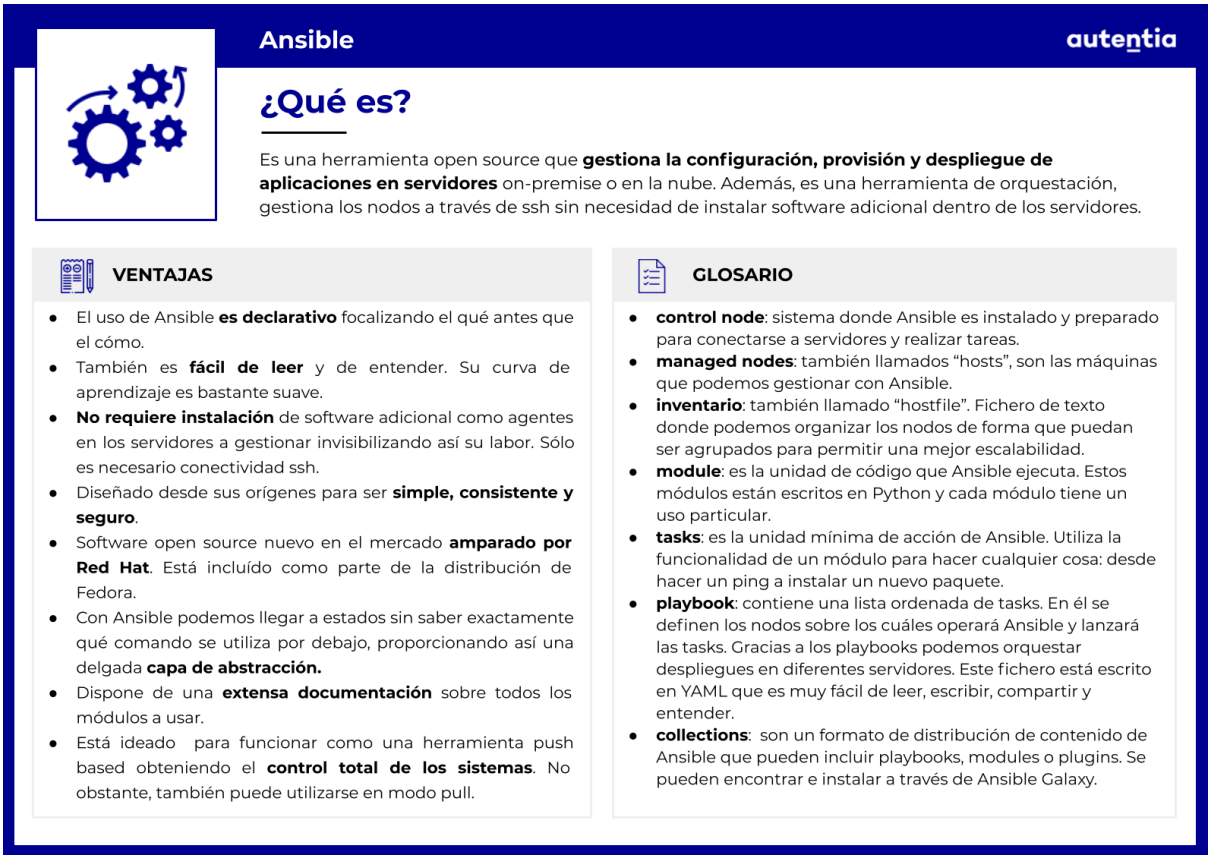
Es posible desplegar todos estos sistemas a mano, arrancando los servidores que se necesitan e instalando las dependencias, archivos de configuración, etc. Todo este proceso manual lleva mucho tiempo y es común cometer errores. Además, se puede dar el caso de tener que operar sobre varios servidores y replicar las acciones cambiando el nombre de la máquina. En este caso, es posible que debamos sincronizar las acciones para seguir prestando servicio al usuario y es muy complicado lograrlo si se hace manualmente.

Es aquí donde entra Ansible como una herramienta para gestionar la configuración, provisión y despliegue de aplicaciones en servidores. En Ansible se describe un servidor y de forma declarativa definimos el estado que queremos alcanzar. Por ejemplo, este estado podría ser la obtención de una configuración, la existencia de ficheros en un directorio, tener permisos de lectura sobre un recurso, un servicio levantado, etc.

Ansible también nos ayuda en las tareas de orquestación. Por ejemplo, cuando necesitamos arrancar una base de datos antes de arrancar los servidores web o quitar los servidores de un balanceador uno a uno para hacer un despliegue sin pérdida de servicio.

También ayuda con el aprovisionamiento de nuevos servidores. Existen módulos para trabajar con las plataformas más usadas como EC2, Azure, Digital Ocean, etc.





Ansible también es una herramienta que permite la Infraestructura como Código (IAC) proporcionándonos una serie de ventajas:

- **La sintaxis es muy fácil de entender**. Los ficheros de configuración de Ansible están escritos en YAML. Este lenguaje de marcado ligero se centra sólo en los datos, potenciando así su simplicidad.

- Al aumentar la complejidad de las tareas, Ansible **se adapta bien** al proveer de mecanismos para descomponerlas en piezas más simples.

- Ansible **es un proyecto open source** bajo el paraguas de Red Hat al que la comunidad ha contribuido con una gran cantidad de módulos.

- Ansible utiliza una **delgada capa de abstracción de recursos** para desplegar paquetes y acciones en diferentes sistemas operativos sin que se tenga que conocer el comando específico.
- Para hacer funcionar Ansible **no es necesario instalar ningún software aparte de Python 2/3 en el servidor**. Ansible no necesita nada más que una contraseña o una clave para conectarse por SSH al servidor.

- Esto es posible gracias a que Ansible es una herramienta de gestión

de configuración basada en Push. Esto quiere decir que el nodo de control o máquina que ejecuta **Ansible tiene el control total del sistema**. Su uso es fácil y de cómoda adopción por ser bastante natural. En caso de tener problemas de escalabilidad, Ansible también nos permite usarlo de modo pull, aunque rompería algunos de los principios enumerados anteriormente.

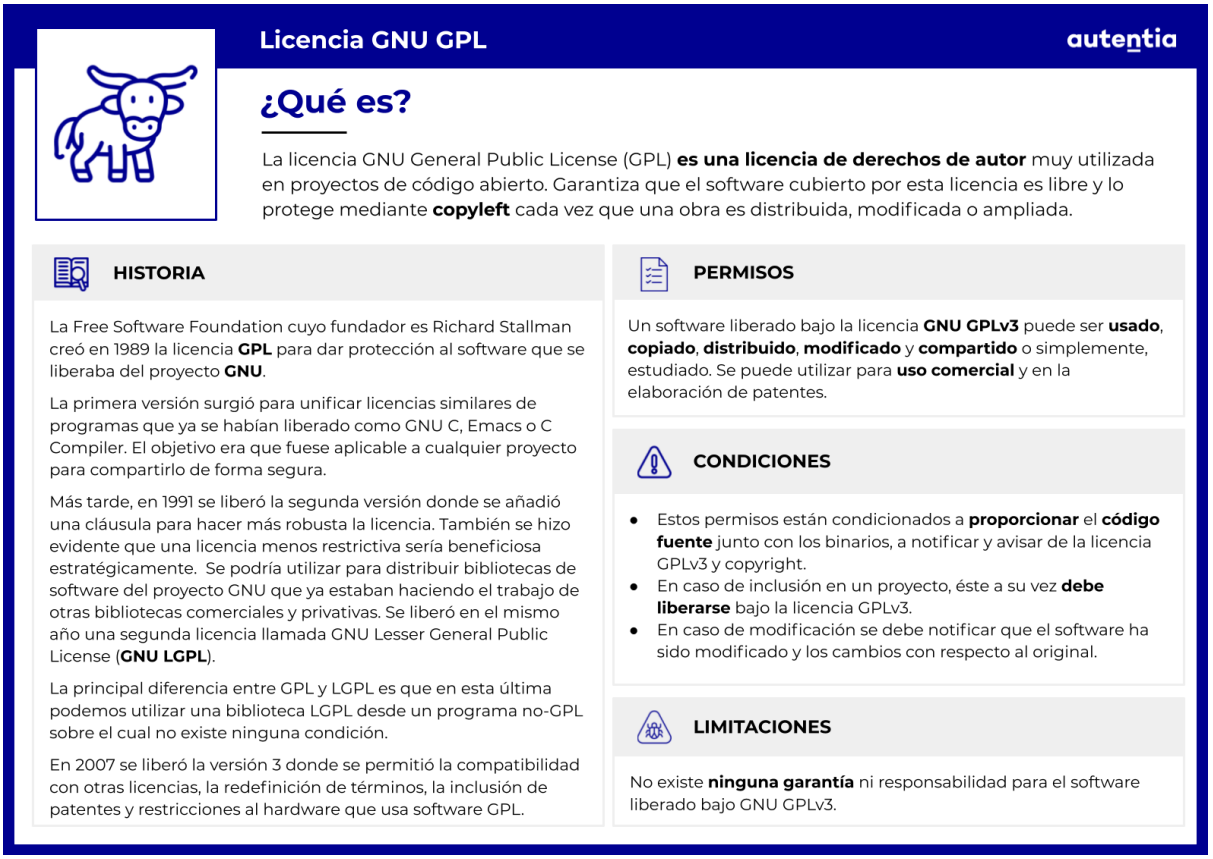
No obstante, también tiene algunos inconvenientes que debemos tener en cuenta:

- Ansible **no tiene estado**: al no realizar un seguimiento de las dependencias, esta herramienta sólo seguirá los pasos que le hemos indicado, informando de cuándo termina o si fallamos y tenemos un error. Esta simplicidad impide que Ansible pueda planificar tareas recurrentes, como podría ser verificar el servicio cada 10 minutos.
- El **soporte a servidores Windows** está creciendo, pero todavía **no está completo**. Pese a que Ansible soporta Windows desde la versión 1.7, aún se requiere que éste se ejecute desde un nodo de control Linux.
- Existen opciones más veteranas como Chef o Puppet para gestionar la configuración. No obstante, desde que Red Hat lo acogió bajo su paraguas, su uso se está extendiendo con celeridad.
- Debido a la necesidad de conexión por SSH, Ansible **es más lento** que otros gestores de configuración que usan agentes dentro de la máquina donde se opera.

Pese a sus inconvenientes, Ansible desde sus orígenes, aboga por ser mínimo. Un gestor de la configuración no tiene por qué añadir dependencias a los sistemas a los que debe servir. Evitar añadir dependencias ayuda a que Ansible sea una herramienta segura para ser utilizada en entornos productivos. Acaba siendo invisible y esa es su mejor baza.

Algo de historia

Ansible fue desarrollada por Michael DeHaan en febrero de 2012. En 2015, la empresa Ansible, Inc (anteriormente AnsibleWorks, Inc.) fue la encargada de patrocinar a Ansible y de respaldarla en la comunidad. RedHat adquirió dicha empresa en el mismo año.



Está escrito en Python y se liberó como GNU GPL v3. El origen de su término es curioso: DeHaan se basó en el término del 'sistema de comunicación instantáneo del hiperespacio' de la novela de *El juego de Ender*, de Orson Scott Card (1985). Sin embargo, el origen real del término fue acuñado por Ursula K. Le Guin en su novela de *El mundo de Rocannon* (1966).

Ansible está incluído como parte de la distribución de Fedora de Linux, cuyo propietario es Red Hat. También está disponible para su descarga a través de EPEL (Extra Packages for Enterprise Linux) para una gran variedad de sistemas operativos.

Distribución Linux

autentia

¿Qué es?

Coloquialmente conocida como "distro de Linux" es una versión personalizada del sistema operativo original, el kernel o núcleo de Linux y suelen ser versiones compuestas de software libre. La licencia GPL permite que los usuarios finales (personas, organizaciones, etc.) tengan libertad de usar, estudiar, compartir (copiar) y modificar el software.

TIPOS

En general se pueden englobar en tres tipos de distribuciones:

- **De escritorio/domésticas:** son las que puede usar cualquier tipo de usuario pero están enfocadas en aplicaciones de uso común como el correo, navegador web, editor de texto, lectura de ficheros, etc. Ubuntu es una de las más conocidas hoy en dia.
- **Servidores:** se enfocan en equipos de tipo servidor que necesitan dar un servicio alto durante todo el día. Estas distribuciones no suelen tener interfaz gráfica.
- **Empresariales:** se enfocan en servicios más concretos y personalizados. Suelen tener un servicio de soporte.

DEBIAN Y RED HAT

La mayoría de las distribuciones de uso comercial se basan en:

Debian (Todas son licencias GPL):

- Kali Linux: se utiliza sobretodo para auditorías de seguridad y pruebas de penetración (Pentesting).
- Ubuntu: popularizó Linux en todo el mundo. Es la beta de su distro principal, Debian.
- Debian y Ubuntu Server.

Red Hat:

- Fedora: El Ubuntu de Redhat.
- CentOS (Licencia GPL).
- Oracle Linux (Licencia GPL).
- AIX (Licencia de Pago de IBM adquirida por RedHat).

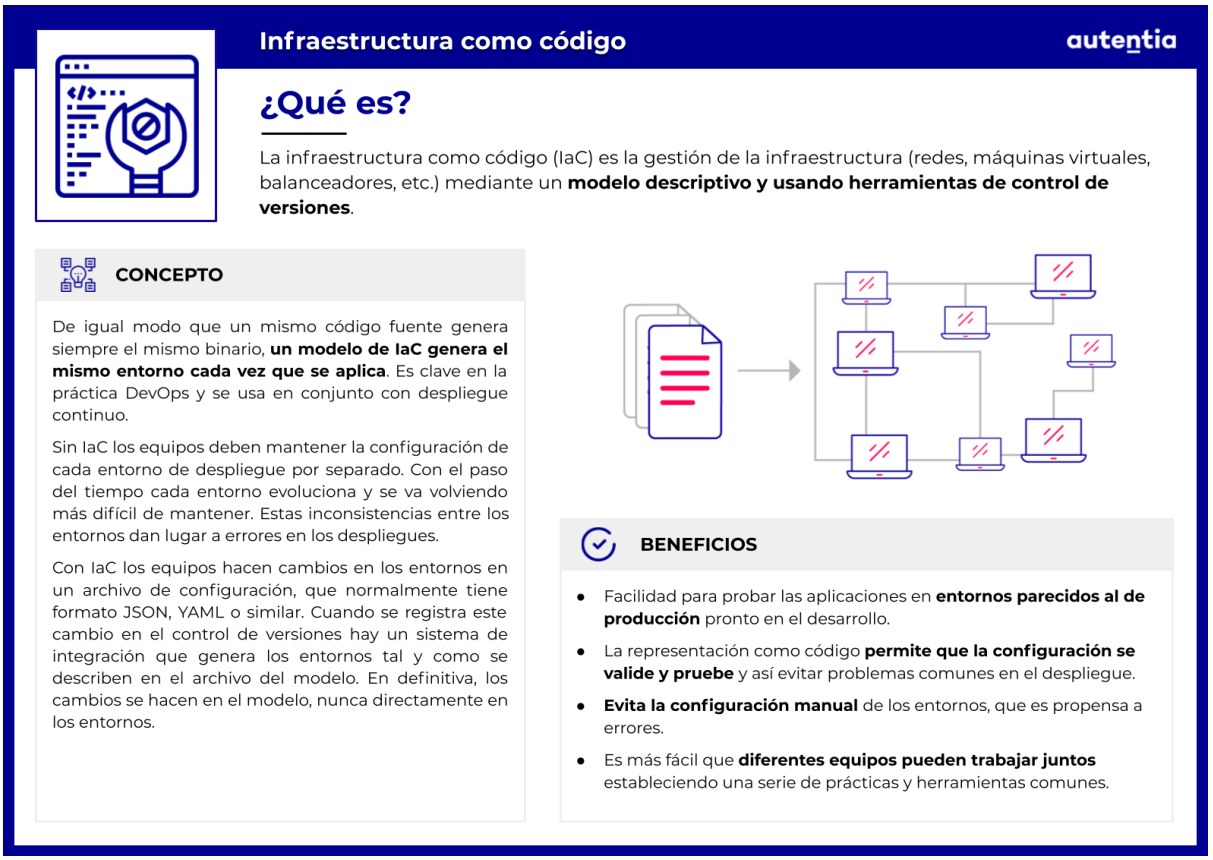
También tiene la capacidad de desplegar a servidores sin sistema operativo, sistemas virtualizados y sistemas en la nube, incluyendo a algunos tan importantes como Amazon Web Services, Google Cloud Platform o Microsoft Azure.

Cada año se organiza el evento AnsibleFest para toda la comunidad de Ansible.

Gestión de la configuración

Infraestructura como Código

Hace algunos años, los administradores de sistemas tenían que gestionar y configurar manualmente todo el software necesario para que las aplicaciones funcionaran correctamente, lo que hacía de esta labor un trabajo bastante costoso. Pero hace relativamente poco, apareció una nueva tendencia que resolvía este problema denominada infraestructura como código (IaC). De este modo, centralizamos toda la gestión y configuración de nuestra infraestructura en ficheros de configuración para que, cuando haya cambios en el modelo, estos se vean reflejados en las máquinas de forma automática.



Otras herramientas

Ansible no es la única herramienta capaz de realizar la gestión y configuración de la infraestructura, existen otras similares como Chef y Puppet. Para entender mejor cuál puede resultar más interesante implementar en nuestro proyecto, debemos tener en cuenta una serie de factores que nos ayudarán a decantarnos por una:

- **Disponibilidad**. En caso de que el controlador o nodo principal falle, ¿cómo gestiona cada herramienta esta problemática? Chef cuenta con un backup del servidor, Puppet tiene un controlador alternativo y Ansible dispone de una instancia secundaria.

- **Lenguaje empleado para la configuración**. En estos casos, hay que tener en cuenta la curva de aprendizaje si el equipo no conoce ninguna de las herramientas. Tanto Chef como Puppet usan Ruby DSL y Puppet DSL respectivamente, que resultan difíciles de aprender. En cambio, Ansible usa YAML, un lenguaje diseñado para ser simple de escribir y de leer, lo que hace que el programador lo aprenda más rápido.

- **Configuración e instalación**. Debido a la arquitectura que tienen las tres, Ansible es una mejor opción. Chef y Puppet son difíciles de configurar debido a su workstation o por tener que configurar certificados entre el master y los agentes.

- **Gestión**. Existen dos tipos de configuraciones: "pull" y "push". La configuración "pull" implica descargarse todas las configuraciones desde un servidor central a los nodos sin ningún comando. Mientras que en una configuración “push”, todas las configuraciones del servidor serán enviadas a los nodos con comandos específicos. Una vez más, Ansible saca ventaja sobre los demás en términos de gestión, ya que soporta el lenguaje YAML y sigue las configuraciones push y pull. Mientras que las otras dos solo soportan una configuración de “pull”.

- **Escalabilidad**. La escalabilidad de las herramientas de configuración es uno de los principales factores. Tanto Chef, como Puppet y Ansible son capaces de gestionar grandes infraestructuras, pero sin embargo, hay una ligera diferencia entre ellas debido a la creación de nuevos nodos. Mientras que en Puppet y Chef es necesario configurar previamente los agentes con una serie de instalaciones para que

puedan formar parte de la infraestructura, en Ansible, con tan solo tener SSH activo en los nodos es suficiente, ya que será el controlador el encargado de instalar lo necesario en ese nodo.

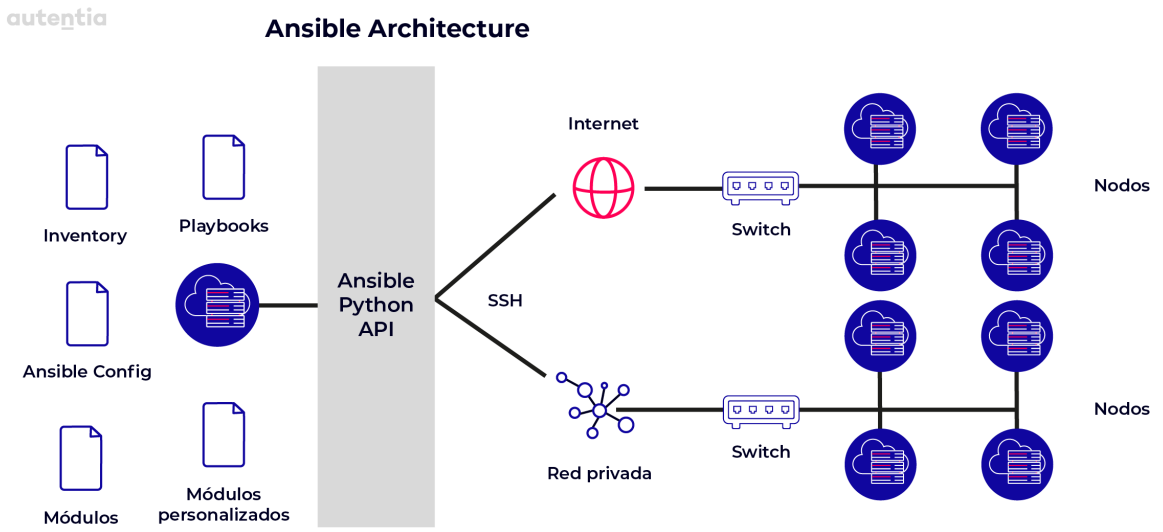
- **Interoperabilidad**. Las tres herramientas permiten lo mismo. El controlador o master se recomienda que sea Linux y para los nodos, ambas herramientas dan soporte a máquinas con otro sistema operativo como Windows.

Arquitectura de Ansible

Para entender correctamente el funcionamiento de Ansible, hay que comprender cuáles son los componentes que interactúan entre sí. Los elementos principales que hay son, por un lado, el **controlador** u **orquestador**, el cual se encarga de comunicar a otros hosts qué es lo que tienen que hacer y por otro lado, los **nodos**, máquinas que contienen los requisitos establecidos en el controlador.

También contamos con otros elementos situados dentro del controlador que permiten crear una infraestructura con las herramientas necesarias para un proyecto.

En primer lugar, contamos con el **inventario** (inventory, en inglés) donde se instancian todas las máquinas que van a formar parte de la infraestructura. También disponemos de **módulos** y **plugins,** los cuales nos van a permitir realizar las tareas necesarias para preparar los entornos. Y por último, a través de un **playbook**, se escribe todo el **workflow** para que cuando se ejecute se disponga de la infraestructura preparada para realizar las operaciones oportunas.



Preparando el entorno de pruebas

En la web adictosaltrabajo.com existe un tutorial donde se explica con gran detalle cómo crear un entorno de integración continua con Ansible.

Antes de explicar cómo configurar Ansible para una prueba básica y entender los distintos elementos que intervienen en la construcción de la arquitectura, vamos a necesitar Vagrant para la creación de nodos en local que en entornos reales, será equivalente a máquinas que estén en data centers con IPs estáticas o en la nube con IPs dinámicas. Anteriormente, se ha explicado cómo configurar Vagrant para crear máquinas virtuales.

La prueba que vamos a explicar dispone de una arquitectura muy sencilla, un controlador y un nodo. El controlador será nuestra máquina real y el nodo será la máquina virtual que hemos creado a través de Vagrant. El nodo tendrá un sistema operativo Linux y usaremos Ansible para instalar un servidor web, como Apache por ejemplo.

```
Vagrant.configure("2") do |config|
 config.vm.box = "ubuntu/jammy64"
```

```
 config.vm.network "private_network", ip: "192.168.1.80"

 config.vm.network "forwarded_port", guest: 80, host:8080, id: "apache",
auto_correct: true

 ssh_pub_key =
File.readlines("#{Dir.home}/.ssh/id_ed25519.pub").first.strip
 config.vm.provision 'shell',
 inline: "echo #{ssh_pub_key} >> /home/vagrant/.ssh/authorized_keys",
privileged: false

 config.vm.provider "virtualbox" do |vb|
   vb.memory = "2048"
 end
end
```

En el controlador debemos tener Python instalado dado que Ansible está desarrollada bajo este lenguaje de programación. Tambíen hay que tener en cuenta que la herramienta se comunica con los nodos a través de SSH: en el fichero Vagrantfile estamos añadiendo a las claves autorizadas para la conexión al nodo, una clave pública (deberás generar el par de claves en la home de tu usuario)

Dicho esto, vamos a generar una estructura de ficheros básica para poder realizar el ejemplo y a explicar el inventory, el playbook y el módulo que usaremos para instalar Apache. Una vez configurados los ficheros ejecutaremos Ansible para tener el servidor web en el nodo.

En primer lugar, vamos a crear el working directory del proyecto en el lugar que se desee. Una vez hecho, creamos los ficheros inventory y playbook.yml y obtendremos la siguiente estructura en el proyecto.

```
└── working_dir/
    ├── inventory
    ├── playbook.yml
```

- Inventory.

```
[webservers]
192.168.1.80
```

Lo que se encuentra entre corchetes “[]” es para definir grupos. Esto se debe a que cuando ejecutemos Ansible podemos decirle en qué grupo queremos que tenga efecto o si queremos que sea en todas las máquinas. Por ejemplo, se podría tener un grupo de webservers, otro de databases y otro de pre producción.

- Playbook.

```
- hosts: webservers
  remote_user: vagrant
  become: yes
  tasks:
    - name: install apache2
      apt: name=apache2 state=latest update_cache=yes
      tags: apache
```

En *hosts* definimos a qué nodos del inventory Ansible va a realizar las tareas especificadas en el playbook. Después, en remote_user estamos diciendo el usuario que debe usar para establecer la conexión con los nodos. En nuestro caso, sería con el nodo *192.168.1.80*. El parámetro *become* se configura si queremos que la tarea se realice siendo superusuario o no.

- Módulo.

En el propio playbook, hacemos uso de los módulos que nos ayudarán a realizar las tareas necesarias para conseguir el resultado buscado. En este caso, hemos usado el módulo *apt* que nos gestionará la instalación de Apache en el nodo.

Con tan solo ejecutar el siguiente comando, se ejecutarán todas las tareas especificadas en el playbook:

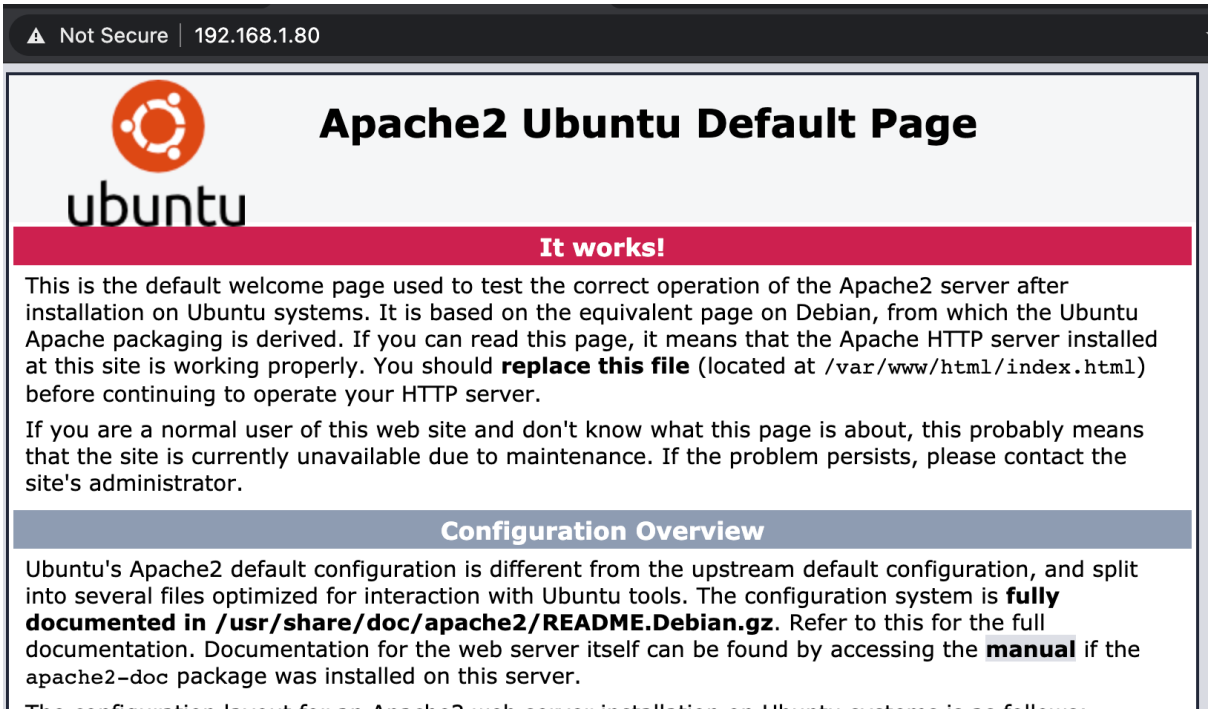
```
ansible-playbook -i inventory playbook.yml -K
```

Con *-K* decimos a Ansible que tenemos que introducir la password de superusuario, ya que la tarea requiere de ese permiso.

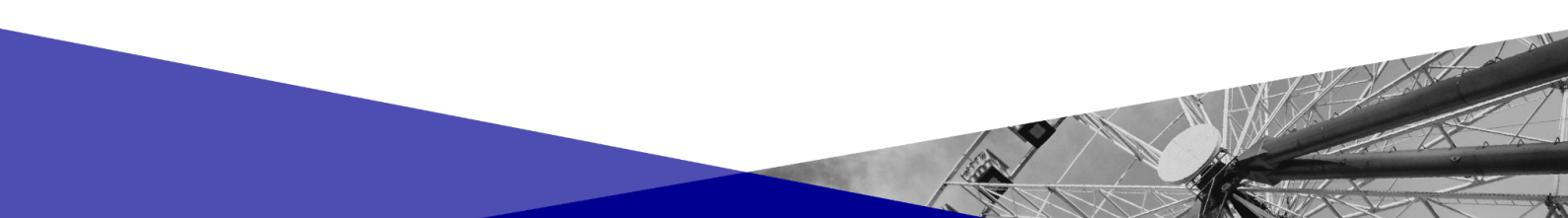
```
// OUTPUT
BECOME password:

PLAY [all]
*********************************************************************
*********************************************
```

```
TASK [Gathering Facts]
*********************************************************************
************************************
ok: [192.168.1.80]

TASK [install apache2]
*********************************************************************
************************************
changed: [192.168.1.80]

PLAY RECAP
*********************************************************************
*********************************************
192.168.1.80               : ok=2    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

Si obtenemos un output similar al de arriba, el nodo ya contiene el servidor web de Apache listo para ser usado. Para poder verlo, abrimos un navegador (Chrome, Firefox, Safari, etc.) y escribimos la dirección IP del nodo. Siguiendo estos pasos, se puede ver que carga en el navegador la página por defecto al instalar el servicio:



Este ejemplo puede ser un caso base de lo que la herramienta Ansible es capaz de ofrecernos. A partir de aquí, solo habría que ir escalando esta lógica y es donde tiene más sentido toda esta práctica de IaC.



Inventario

¿Qué es?

Como ya se ha adelantado en el capítulo anterior, Ansible administra múltiples nodos o hosts en nuestra infraestructura. Esto se hace principalmente, desde un listado de hosts o de grupos de hosts que se definen en el inventario.

Opciones de configuración

La ubicación por defecto del inventario está en `/etc/ansible/hosts` pero podemos definir su ubicación mediante la opción `-i <INVENTORY_FILE_PATH>`. Además, se pueden definir múltiples inventarios al mismo tiempo, así como recuperar hosts en tiempo real. El inventario se suele definir en formato INI ya que este formato es menos redundante cuando definimos hosts y grupos de hosts.

Hosts

Ya hemos definido un inventario anteriormente con un host con la ip 192.168.1.80 al que hemos añadido al grupo de *webservers*:

```
[webservers]
192.168.1.80
```

El nombre entre corchetes es el nombre de un grupo que se suele utilizar para clasificar o categorizar aquellos hosts sobre los que necesitamos distintas operativas. Podemos añadir diferentes grupos y un mismo host puede pertenecer a diferentes grupos. Veamos un ejemplo:

```
[webservers]
192.168.1.80

[virtual]
192.168.1.80
```

```
192.168.1.81
```

Hemos definido el grupo *virtual* con dos hosts donde nuestro primer host está incluído. Vamos a utilizar un comando ad-hoc para probar que estos host son alcanzables mediante un ping. De esta forma, podemos validar inventarios y verificar que hacemos las cosas bien. Veamos el comando:

```
ansible virtual -i inventory -m ping -u vagrant
```

Este comando se estructura de la siguiente manera:

```
ansible <HOST_PATTERN> -i <INVENTORY_FILE> -m <MODULE> -u <REMOTE_USER>
```

Para nuestro ejemplo, estamos virtualizando entornos con vagrant, por tanto, debemos informar a Ansible del usuario de conexión por defecto mediante SSH. Por otra parte, estamos lanzando un ping desde el módulo ping sin ningún tipo de argumento porque no lo necesitamos. Finalmente, ejecutamos Ansible al patrón de host *virtual* que en este caso es un grupo de dos hosts, la 192.168.1.80 y la 192.168.1.81. El resultado es la realización de un ping a los dos hosts en paralelo pertenecientes al grupo *virtual*.

Los inventarios mostrados hasta ahora han sido en formato INI. Este formato es el más adecuado para representar estos inventarios por ser su ilustración muy sencilla. Sin embargo, también podemos representar exactamente el mismo inventario en formato YAML:

```
all:
  children:
    webservers:
      hosts:
        192.168.1.80:
    virtual:
      hosts:
        192.168.1.80:
        192.168.1.81:
```

Para definir el inventario podemos utilizar el tipo de formato que más nos convenga en cada momento. Por ejemplo, en este caso el formato INI es más claro y menos redundante. Vemos cómo se han introducido conceptos nuevos como *all* o *children* que en un formato INI vienen implícitos y que pasamos a comentar a continuación.

Grupos

Profundizando un poco más, vemos que existen dos grupos por defecto: el grupo *all* (todos) y el grupo *ungrouped* (desagrupados). El grupo *all* contiene todos los hosts que declaramos en el inventario. El grupo *ungrouped* agrupa aquellos que no pertenecen a ningún grupo. No hemos mostrado todavía un host desagrupado:

```
192.168.1.82

[webservers]
192.168.1.80

[virtual]
192.168.1.80
192.168.1.81
```

Si lanzamos un ping con el patrón *ungrouped* lo haremos sólo sobre el host 192.168.1.82, sin embargo, si lo hacemos con el patrón *all* lo haremos sobre todas las máquinas.

Podemos agrupar hosts según nuestros criterios, Ansible permite dotar de expresividad y conceptos a tu infraestructura. Algunas de las prácticas más habituales suelen ser:

- **La utilidad del host**: por ejemplo, ¿es una aplicación?, ¿una base de datos?
- **Su ubicación**: podríamos necesitar configurar por ejemplo, un DNS distinto o un timezone diferente.
- **La fase del entorno**: debemos diferenciar entre hosts de producción y de entornos previos.

También podemos definir grupos que hereden unos de otros usando el sufijo *children*. De esta forma, podemos reutilizar código. Veamos un ejemplo simple:

```
[webservers]
192.168.1.80
```

```
[databases]
192.168.1.81

[my-project:children]
webservers
databases

[other-project]
192.168.1.82

[projects:children]
my-project
other-project
```

Podemos utilizar el patrón *projects* para que se aplique sobre los grupos *my-project* y *other-project*, que a su vez se aplicarán en sus hosts.

Ansible también dispone de algunas posibilidades para declarar hosts, como el uso de rangos alfabéticos o numéricos, por ejemplo, podemos definir el siguiente inventario:

```
[webservers]
192.168.1.8[0:3]

[databases]
mydatabase-[a:d].com
```

De esta forma, con cuatro líneas hemos declarado 2 grupos y 8 hosts.

Variables

Las variables son el factor diferencial entre hosts. Cada máquina suele tener sus peculiaridades y Ansible nos permite configurarlas muy fácilmente. Podemos definir estas variables en playbooks, roles, ficheros o desde un comando ad-hoc, pero en esta sección nos centraremos en las variables de un inventario.

Cuando definimos una variable podemos utilizarla en multitud de lugares

dentro de Ansible: como argumentos, usándolas en sentencias “when”, en bucles, etc.

En un inventario, las variables se pueden asignar a un host específico o a un grupo de hosts con el sufijo *vars*. Veamos un ejemplo:

```
[webservers]
192.168.1.80 myvar=3

[webservers:vars]
myvar=100

[virtual]
192.168.1.81

[virtual:vars]
myvar=2
```

Hemos definido la variable *myvar* para el host 192.168.1.80 que tiene más prioridad que la variable declarada para el grupo *webservers*. También hemos definido otra para el grupo virtual. Para poder verificar qué variable es la utilizada, vamos a definir un pequeño playbook que utilice el módulo debug para que imprima por pantalla las variables con las que va a resolver cada host.

```
---
# file debug.yml

- hosts: all
  tasks:
    - name: Imprime variables por pantalla
      debug:
        var: myvar
```

Podemos ejecutar este playbook con el siguiente comando:

```
ansible-playbook debug.yml -i inventory -u vagrant
```

Ansible también permite que utilicemos alias para los hosts para dotar de una mayor legibilidad al configurar nuestros entornos de prueba. Veamos un ejemplo:

```
[webservers]
alfa ansible_host=192.168.1.80 ansible_port=22 myvar=3

[virtual]
beta ansible_host=192.168.1.81 ansible_port=22

[virtual:vars]
myvar=2
```

Ahora podríamos utilizar el patrón *alfa* o *beta* para referirnos a dichos hosts.

Las variables también se pueden declarar en ficheros YAML. En el caso de ser formato INI, las variables declaradas en el host de forma inline son interpretadas como estructuras de literales de Python (strings, números, tuplas, etc.). Sin embargo, si declaramos una sección de variables de un grupo, por ejemplo *[virtual:vars]*, todos los valores serán interpretados como strings. Esta ambigüedad puede ser un problema en el futuro si no te acuerdas de esta peculiaridad, por lo cual y para evitar confusiones, Ansible recomienda que utilicemos el formato YAML cuando definamos variables. Mostramos ahora el ejemplo anterior pero en YAML:

```
---
# file inventory.yml

all:
  children:
    webservers:
      hosts:
        alfa:
          ansible_host: 192.168.1.80
          ansible_port: 22
          myvar: 3
    virtual:
      hosts:
        beta:
          ansible_host: 192.168.1.80
          ansible_port: 22
      myvar: 2
```

Múltiples inventarios

Podemos seleccionar multitud de inventarios con la opción *-i*. Un factor determinante es el orden. Si en dos inventarios definimos una variable para un host o un grupo de hosts, la variable que prevalecerá será la del último inventario. Un ejemplo de ejecución sería el siguiente:

```
ansible-playbook debug.yml -i inventory -i inventory-basic.yml -u
vagrant
```

Generalmente, las variables de los inventarios no se rellenan en un fichero principal como venimos haciendo hasta ahora. Esto es sólo un atajo para mostrar el potencial de Ansible pero no es una buena práctica. Ansible es bastante inteligente y quiere encontrar la información allá donde la tiene que buscar.

Para que esto sea posible, Ansible nos ayuda buscando por carpetas basadas en el nombre. Se considera como el directorio raíz aquel donde hemos definido el inventario y, a partir de éste, se espera encontrar subcarpetas convenidas por nombre de grupo o host donde podemos definir variables.

Supongamos el siguiente inventario principal dentro de una carpeta *mydirectory* que definimos en formato INI:

```
# file mydirectory/inventory

[webservers]
alfa ansible_host=192.168.1.80

[virtual]
beta ansible_host=192.168.1.81

[servers:children]
webservers
virtual
```

Podemos organizar la estructura de carpetas de la siguiente manera:

```
/mydirectory
- inventory
```

```
- /group_vars
-- servers.yml
- /host_vars
-- alfa.yml
```

Los ficheros servers.yml y alfa.yml sólo definirán variables y quedan de esta forma:

```
---
# file mydirectory/group_vars/servers.yml

myvar: 3
```

```
---
# file mydirectory/host_vars/alfa.yml

myvar: 5
```

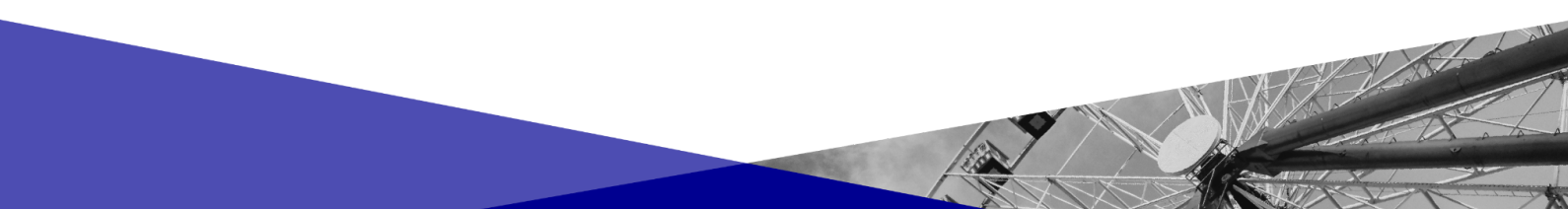
Para comprobar que funciona volvemos a ejecutar `ansible-playbook debug.yml -i mydirectory/inventory -u vagrant`

En caso de necesitar que un grupo de hosts o un host tuviera más de un fichero de configuración de variables, podríamos añadir una carpeta con el nombre y añadir más ficheros YAML dentro de ella. Por ejemplo, para el host alfa, se permitiría lo siguiente:

```
/mydirectory
- inventory
- /group_vars
-- servers.yml
- /host_vars
-- alfa.yml
-- /alfa
--- config.yml
--- other.yml
```

El orden de la lectura es por orden alfabético y en caso de conflicto de variables prevalecerá el último fichero leído. En caso de jerarquías, suele prevalecer lo particular frente a lo general.

La capacidad de separar ficheros es muy útil cuando tenemos algunas variables como secretos que se deban cifrar pero no queramos cifrar el



fichero entero.

Inventario como directorio

Hasta ahora hemos considerado los inventarios como ficheros donde definimos los hosts, pero con Ansible se puede definir directamente un directorio como inventario y que en dicha carpeta podamos encontrar varios inventarios estáticos o dinámicos. Esto ya lo podemos hacer desde el paso anterior si ejecutamos el comando `ansible-playbook debug.yml -i mydirectory -u vagrant` directamente sobre la carpeta *mydirectory*.

En caso de necesitar un cierto orden de lectura de inventarios para poder agrupar inventarios entre sí, podemos hacer uso de los siguientes prefijos:

```
/mydirectory
- 01-inventory
- 02-dynamic-inventory
- 03-other-inventory
- group_vars/
-- all.yml
```

Inventario dinámico

Hemos descrito el inventario para los hosts que conocemos pero, ¿y qué pasa con aquellos que se provisionan y eliminan de forma dinámica?

Ansible dispone de plugins para poder obtener esta información en tiempo real. Para ello se necesitan credenciales de acceso para conocer todo el detalle de los hosts existentes en nuestra infraestructura.

Para poder ilustrar este apartado, hemos provisionado mediante Ansible varias instancias en EC2 mediante el módulo *ec2* como veremos más adelante en detalle y recuperamos el inventario de forma dinámica con el plugin *aws_ec2* de la colección de *amazon.aws*.

Para provisionar, vamos a definir un playbook para crear 3 instancias idénticas en EC2. Este paso lo puedes hacer de forma manual siguiendo este tutorial de creación de instancias de EC2. Además, vamos a recuperar las claves del fichero cifrado secret.yml mediante vault. Más adelante explicaremos todo lo relacionado con los secretos en Ansible.

```
# file create-instance.yml
- hosts: localhost
  vars_files:
      - secret.yml
  tasks:
   - name: Crear instancias en AWS
     ec2:
      key_name: "pardeclaves"
      instance_type: "t2.micro"
      image: "ami-0fc970315c2d38f01"
      wait: yes
      count: 3
      group: launch-wizard-1
      vpc_subnet_id: "subnet-51aba819"
      region: "eu-west-1"
      state: present
      assign_public_ip: yes
      aws_access_key: "{{ AWS_USER }}"
      aws_secret_key: "{{ AWS_PASSWORD }}"
     register: ec2
```

Sobre el fichero anterior debemos pararnos a pensar qué estamos haciendo y el contexto necesario a tener en cuenta:

- El host es localhost, lo que quiere decir que todo esto se va a ejecutar desde la máquina que ejecute ansible.
- Hemos añadido el fichero *secret.yml* para que se añada a las variables de este playbook. En dicho fichero cifrado estamos dando valor a la variable *AWS_USER* y *AWS_PASSWORD*.
- Hemos creado el par de claves pública y privada desde Amazon para este ejemplo. Este par de claves se llama *pardeclaves.pem*. EC2 almacenará la clave pública y la clave privada la utilizaremos para conectar por SSH a las instancias en un futuro. Por simplificación del ejemplo, usaremos la misma clave pública y privada para todas las instancias pero no es lo recomendable para un entorno real.
- Para poder tener visibilidad de nuestras instancias, es necesario crear un grupo de seguridad que abra el puerto 22 (SSH) y 80 (http). Este último puerto lo usaremos más adelante.

Ejecutamos el playbook preguntando por los secretos y ejecutaremos la provisión de tres instancias nuevas en EC2.

```
$ ansible-playbook create-instance.yml --ask-vault-pass
```

Para poder observar este inventario de forma dinámica necesitamos instalar el plugin *aws_ec2*. Para ello introducimos el comando necesario para descargarlo de *ansible-galaxy*:

```
$ ansible-galaxy collection install amazon.aws
```

Definimos el fichero que permitirá obtener el inventario de forma dinámica:

```
---
# file aws_ec2.yml
plugin: amazon.aws.aws_ec2
boto_profile: <AWS_PROFILE>
keyed_groups:
- key: 'security_groups|json_query("[].group_name")'
  prefix: security_group
- key: placement.region
  prefix: region
- key: 'architecture'
  prefix: arch
regions:
- eu-west-1
hostnames:
- tag:Name
- dns-name
- public-ip-address
- private-ip-address
- architecture
```

De este fichero, comentar que debemos introducir el perfil de Amazon en la variable *AWS_PROFILE* para seleccionar las credenciales que tenemos por defecto en `~/.aws/credentials`. Con este plugin también podemos agrupar los hosts dinámicos por grupo de seguridad, región o arquitectura entre otros.

```
$ ansible-inventory -i aws_ec2.yml --list
```

```
{
    "_meta": {
        "hostvars": {
            ...
         }
    },
    "all": {
        "children": [
            "arch_x86_64",
            "aws_ec2",
            "region_eu_west_1",
            "security_group_launch_wizard_1",
            "ungrouped"
        ]
    },
    "arch_x86_64": {
        "hosts": [
            "ec2-18-203-155-35.eu-west-1.compute.amazonaws.com",
            "ec2-3-248-185-39.eu-west-1.compute.amazonaws.com",
            "ec2-54-72-166-231.eu-west-1.compute.amazonaws.com"
        ]
    },
    "aws_ec2": {
        "hosts": [
            "ec2-18-203-155-35.eu-west-1.compute.amazonaws.com",
            "ec2-3-248-185-39.eu-west-1.compute.amazonaws.com",
            "ec2-54-72-166-231.eu-west-1.compute.amazonaws.com"
        ]
    },
    "region_eu_west_1": {
        "hosts": [
            "ec2-18-203-155-35.eu-west-1.compute.amazonaws.com",
            "ec2-3-248-185-39.eu-west-1.compute.amazonaws.com",
            "ec2-54-72-166-231.eu-west-1.compute.amazonaws.com"
        ]
    },
    "security_group_launch_wizard_1": {
        "hosts": [
            "ec2-18-203-155-35.eu-west-1.compute.amazonaws.com",
            "ec2-3-248-185-39.eu-west-1.compute.amazonaws.com",
            "ec2-54-72-166-231.eu-west-1.compute.amazonaws.com"
        ]
    }
```

```
}
```

Todo lo que hemos aprendido sobre inventarios “estáticos” ahora es aplicable a los inventarios dinámicos. Veamos un ejemplo de playbook en el que levantamos un apache y servimos una página web sencilla para todos los nodos. Definimos el siguiente playbook donde el patrón es el grupo *arch_x86_64*:

```
---
# file install-apache.yml

- hosts: arch_x86_64
  become: true
  vars_files:
    - vars.yml
  remote_user: ec2-user
  tasks:
    - name: Instala apache
      yum:
        name: httpd
        state: latest
    - name: Crea una página web por defecto
      shell: echo "<h1>Hello world with Ansible ;)</h1>" >
/var/www/html/index.html
    - name: Activa apache
      service: name=httpd enabled=yes state=started
```

Con el siguiente fichero con variables:

```
---
# file vars.yml

ansible_ssh_private_key_file: ./pardeclaves.pem
ansible_ssh_common_args: '-o StrictHostKeyChecking=no'
```

Nuestra autenticación por SSH será mediante intercambio de clave pública y privada en lugar de por contraseña. Además, como vamos a realizar este playbook sobre tres nodos distintos, vamos a confiar por defecto en la conexión a los servidores para evitar tener que hacerlo manualmente durante la ejecución del playbook. De esta forma sencilla, ejecutamos el playbook referenciando ahora el inventario dinámico:

```
$ ansible-playbook install-apache.yml -i aws_ec2.yml

PLAY [arch_x86_64]
*********************************************************************
*********************************************************************
*************************

TASK [Gathering Facts]
*********************************************************************
*********************************************************************
**********************
ok: [ec2-18-203-155-35.eu-west-1.compute.amazonaws.com]
ok: [ec2-54-72-166-231.eu-west-1.compute.amazonaws.com]
ok: [ec2-3-248-185-39.eu-west-1.compute.amazonaws.com]

TASK [Instala apache]
*********************************************************************
*********************************************************************
***********************
changed: [ec2-54-72-166-231.eu-west-1.compute.amazonaws.com]
changed: [ec2-3-248-185-39.eu-west-1.compute.amazonaws.com]
changed: [ec2-18-203-155-35.eu-west-1.compute.amazonaws.com]

TASK [Crea una página web por defecto]
*********************************************************************
*********************************************************************
*******
changed: [ec2-3-248-185-39.eu-west-1.compute.amazonaws.com]
changed: [ec2-54-72-166-231.eu-west-1.compute.amazonaws.com]
changed: [ec2-18-203-155-35.eu-west-1.compute.amazonaws.com]

TASK [Activa apache]
*********************************************************************
*********************************************************************
************************
changed: [ec2-54-72-166-231.eu-west-1.compute.amazonaws.com]
changed: [ec2-3-248-185-39.eu-west-1.compute.amazonaws.com]
changed: [ec2-18-203-155-35.eu-west-1.compute.amazonaws.com]

PLAY RECAP
*********************************************************************
*********************************************************************
*********************************
```

```
ec2-18-203-155-35.eu-west-1.compute.amazonaws.com : ok=4    changed=3
unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
ec2-3-248-185-39.eu-west-1.compute.amazonaws.com : ok=4    changed=3
unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
ec2-54-72-166-231.eu-west-1.compute.amazonaws.com : ok=4    changed=3
unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Si todo ha ido bien, deberías disponer ya de un Apache a tu disposición para que sirva la siguiente página web en cualquiera de tus hosts:

- http://*<cualquiera_de_tus_hosts_de_ec2>*

Playbooks

Un *playbook* es un archivo escrito en YAML donde se describen las tareas de configuración y administración que queremos realizar en cada uno de los nodos. Un playbook está formado por varias *plays*. *Una play* consiste en varias tareas que vamos a realizar en un nodo. Cada tarea llama a un módulo de Ansible.

Un playbook se ejecuta en orden de arriba a abajo. Dentro de cada play, las tareas también se ejecutan en orden de arriba hacia abajo. Los playbooks con múltiples plays pueden orquestar implementaciones de múltiples máquinas, ejecutando una play en los servidores web, luego otra play en los servidores de base de datos y así sucesivamente. Como mínimo, cada play define dos cosas: *el nodo* donde se van a ejecutar las tareas utilizando los patrones y por lo menos una tarea.

Por ejemplo, este playbook tienes dos plays*: update web servers* y *update db servers*. Cada play tiene sus tareas y también sus palabras claves.

```
---
- name: update web servers
  hosts: webservers
  remote_user: root

  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf

- name: update db servers
  hosts: databases
  remote_user: root

  tasks:
  - name: ensure postgresql is at the latest version
    yum:
      name: postgresql
```

```
      state: latest
  - name: ensure that postgresql is started
    service:
      name: postgresql
      state: started
```

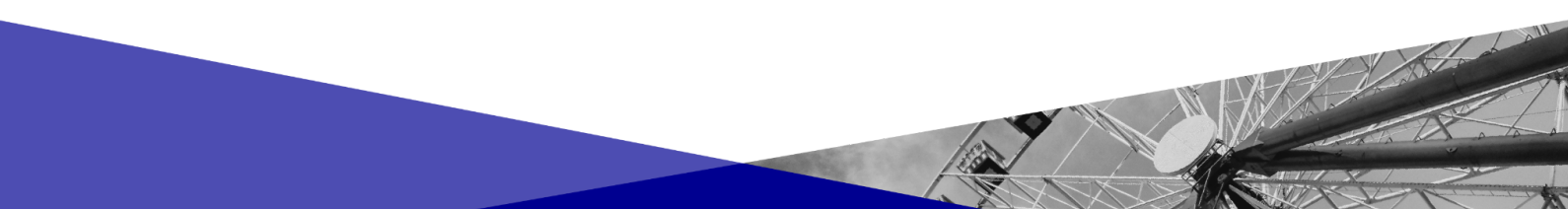
Un playbook puede incluir únicamente una línea de hosts y tareas. Por ejemplo, el playbook del ejemplo establece un usuario remoto para cada play. Esta es la cuenta de usuario para la conexión SSH. Puedes agregar otras palabras claves en el nivel de playbook, play o tarea para influir en el comportamiento de Ansible. Las palabras clave pueden controlar el complemento de conexión, ya sea para usar la escalada de privilegios, cómo manejar los errores y más. Para admitir una variedad de entornos, Ansible permite establecer muchos de estos parámetros como argumentos de línea de comandos, en su configuración de Ansible o en su inventario.

De forma predeterminada, Ansible ejecuta cada tarea de manera ordenada, una a la vez, en todas las máquinas que coinciden con el patrón de host. Cada tarea ejecuta un módulo con argumentos específicos. Cuando una tarea se ha ejecutado en todas las máquinas, Ansible pasa a la siguiente tarea. Se puede cambiar este comportamiento predeterminado con la palabra clave *strategy*. Dentro de cada play, Ansible aplica las mismas directivas a todos los hosts. Si una tarea falla en un host, Ansible saca esa máquina de la rotación para el resto de las tareas del playbook.

Cuando se ejecuta un playbook, Ansible devuelve la información sobre las conexiones, los nombres de todas las plays y tareas, el resultado de la ejecución de cada tarea en cada máquina y si cada tarea ha realizado un cambio en cada máquina. En la parte inferior del resumen de la ejecución de playbook, Ansible proporciona un resumen de todos los nodos que fueron seleccionados.

La mayoría de los módulos de Ansible comprueban si ya se ha alcanzado el estado final deseado y terminan sin realizar ninguna acción si se ha alcanzado ese estado, de modo que la repetición de la tarea no cambia el estado final. Los módulos que se comportan de esta manera se denominan *"idempotentes"*. Ya sea que ejecute un playbook una o varias veces, el resultado debe ser el mismo. Los módulos de Ansible deben ser idempotentes.

Para ejecutar el playbook tienes que lanzar el comando:



```
ansible-playbook playbook.yml
```

Variables y hechos

Ansible permite declarar diferentes variables para gestionar las diferencias entre sistemas. Puedes ejecutar tareas y guías en varios sistemas diferentes con un solo comando. Para representar las variaciones entre esos diferentes sistemas, puedes crear variables con sintaxis YAML estándar, incluidas listas y diccionarios. Puedes definir estas variables en tus playbook, en tu inventario, en archivos y roles reutilizables, o en la línea de comandos. También puedes crear variables durante la ejecución de un playbook registrando el valor o los valores de retorno de una tarea como una nueva variable.

Después de crear variables, ya sea definiéndolas en un archivo, pasándolas en la línea de comandos o registrando el valor o los valores de retorno de una tarea como una nueva variable, puedes usar esas variables en los argumentos del módulo, en declaraciones condicionales "when", en las plantillas y en bucles. Este repo de ansible-examples contiene muchos ejemplos del uso de variables en Ansible.

Puedes definir una variable simple utilizando la sintaxis estándar de YAML. Por ejemplo:

```
base_path: /apps/web-app
```

El nombre de una variable solo puede incluir letras, números y guiones bajos. Las palabras clave de Python o las palabras clave del playbook no son nombres de variable válidos. Un nombre de variable no puede comenzar con un número.

Después de definir una variable, usa la sintaxis *Jinja2* para hacer referencia a ella. Las variables *Jinja2* utilizan llaves dobles. Por ejemplo: *“Mi nombre es {{nombre}}”*. Además puedes utilizar la sintaxis de *Jinja2* en los playbooks. Por ejemplo:

```
- hosts: web_servers
  vars:
       app_path: ‘{{ base_path }}/web’
```

Tienes que usar comillas para crear una sintaxis YAML válida.

Puedes crear variables con múltiples valores usando las listas de YAML.

```
region:
  - north
  - south
  - midwest
  - west
  - earth
```

También puedes crear los diccionarios:

```
foo:
  field1: one
  field2: two
```

Para definir la variable en el playbook tienes que usar la palabra clave *vars*.

```
- hosts: webservers
  vars:
    http_port: 80
```

Puedes definir las variables con la información sensible dentro de los ficheros reusables, así de este modo puedes guardar los playbooks aparte en git sin exponer la información sensible.

```
---

- hosts: all
  remote_user: root
  vars:
    favcolor: blue
  vars_files:
    - /vars/external_vars.yml
```

El contenido del fichero de variables es simplemente un diccionario YAML.

```
---
# vars/external_vars.yml
somevar: somevalue
password: magic
```

Puedes definir variables cuando ejecutas tus playbooks pasando variables en la línea de comandos usando el argumento *--extra-vars (o -e)*.

```
ansible-playbook playbook.yml --extra-vars "somevar=val password=magic"
```

Es posible crear variables a partir de la salida de una tarea de Ansible con la palabra clave *register*. Puedes utilizar variables registradas en cualquier tarea posterior en tu play. Por ejemplo:

```
- hosts: web_servers
  tasks:

     - name: la primera tarea
       ansible.builtin.shell: /usr/bin/foo
       register: foo_result
       ignore_errors: true

     - name: la segunda tarea
       ansible.builtin.shell: /usr/bin/bar
       when: foo_result.rc == 5
```

Con Ansible puedes recuperar o descubrir determinadas variables que contienen información sobre tus sistemas remotos o sobre el propio Ansible. Las variables relacionadas con los sistemas remotos se denominan hechos. Con hechos, puedes utilizar el comportamiento o el estado de un sistema como configuración en otros sistemas. Por ejemplo, puedes utilizar la dirección IP de un sistema como valor de configuración en otro sistema.

Los hechos de Ansible son datos relacionados con tus sistemas remotos, incluidos los sistemas operativos, las direcciones IP, los sistemas de archivos adjuntos y demás. Puedes acceder a estos datos en la variable *ansible_facts*. De forma predeterminada, también puedes acceder a algunos hechos de Ansible como variables con el prefijo *ansible_*. Puedes deshabilitar este comportamiento mediante la configuración *INJECT_FACTS_AS_VARS*. Para ver todos los datos disponibles, agregue esta tarea a una play:

```
- name: Print all available facts
  ansible.builtin.debug:
    var: ansible_facts
```

Luego puedes usar los hechos de Ansible en tu playbook así: '{{ansible_facts['nodename']}}'.

También se puede usar el módulo *ansible.builtin.debug* para imprimir cualquier variable o hecho de Ansible.

```
- name: Print the gateway for each host when defined
  ansible.builtin.debug:
    msg: System {{ inventory_hostname }} has gateway {{
ansible_default_ipv4.gateway }}
```

Aparte de las variables definidas por usuario, Ansible tiene un conjunto de variables que tienen el valor por defecto. Por ejemplo, *ansible_user* que nos provee el usuario que usa Ansible, *ansible_host* que tiene IP/nombre de host, etc. Todo el listado de las variables especiales lo puedes encontrar en la documentación de Ansible.

Plantillas con Jinja

Jinja2 es un motor de plantillas basado en Python potente y fácil de usar que resulta útil en un entorno con varios servidores. Crear archivos de configuración estática para cada uno de estos nodos es tedioso y puede que no sea una opción viable ya que consumirá más tiempo y energía. Y aquí es donde entra la plantilla.

Las plantillas de *Jinja2* son archivos de plantilla simples que almacenan variables que pueden cambiar. Cuando se ejecutan los playbooks, estas variables se reemplazan por valores reales definidos en los playbooks. De esta manera, la creación de plantillas ofrece una solución eficiente y flexible para crear o modificar archivos de configuración con facilidad.

En esta guía, nos centraremos en cómo se pueden configurar y utilizar las plantillas de *Jinja2* en los playbooks de Ansible.

Un archivo de plantilla *Jinja2* es un archivo de texto que contiene variables que se evalúan y reemplazan por valores reales en tiempo de ejecución. En un archivo de plantilla *Jinja2*, encontrarás las siguientes etiquetas:

- {{}}: las llaves dobles son las etiquetas más utilizadas en un archivo de plantilla y se utilizan para incrustar variables y, en última instancia, imprimir su valor durante la ejecución del código. Por

ejemplo, una sintaxis simple que usa las llaves dobles es la que se muestra:

```
{{webserver}} se está ejecutando en {{nginx-version}}
```

- {%%}: se utilizan principalmente para declaraciones de control como bucles y declaraciones como if-else: *{% if %}, {% elif %}, {% else %}, {% endif %}, {% for %}, {% endfor %},* etc.
- {# #}: denotan comentarios que describen una tarea.

En la mayoría de los casos, los archivos de plantilla *Jinja2* se utilizan para crear archivos o reemplazar archivos de configuración en los servidores. Aparte de eso, puedes realizar declaraciones condicionales como bucles y declaraciones if-else, y transformar los datos usando filtros y mucho más.

Los archivos de plantilla tienen la extensión *.j2*. Por ejemplo creamos una plantilla *template.j2* así:

```
El servidor de Apache {{ version_number }} está funcionando en  {{
server }}
```

Aquí, las variables son *{{version_number}}* y *{{server}}*.

Estas variables se definen en un playbook y se reemplazarán por los valores reales en el archivo playbook.yml a continuación.

```
---
- hosts: 127.0.0.1
  vars:
   version_number: '2.9.34'
   server: 'Ubuntu'

  tasks:
  - name: Un ejemplo de la plantilla Jinja 2
    template:
       src: template.j2
       dest: ~/file.txt
```

Cuando se ejecute el playbook, las variables en el archivo de plantilla se reemplazarán por los valores reales y se creará un nuevo archivo o reemplazará uno ya existente en la ruta de destino.

Dentro de las plantillas Jinja2 se pueden utilizar los bucles o las condiciones o incluso filtros. Si creamos una variable *array* dentro de nuestro playbook podremos utilizarla dentro de nuestra plantilla directamente.

```
---
- hosts: 127.0.0.1
  vars:
   array: ['north', 'south', 'east', 'west']

  tasks:
  - name: un ejemplo de bucle
    template:
       src: template.j2
       dest: ~/file.txt
```

Con la plantilla template.j2

```
{% for item in array %}
{{ item}}
{% endfor %}
```

El output sería:

```
north
south
east
west
```

Podemos aplicar un filtro *UPPER* para mayusculizar los strings de nuestro array:

```
{% for item in array %}
{{ item | upper }}
{% endfor %}
```

En este caso el fichero file.txt tendría el siguiente texto:

```
NORTH
SOUTH
EAST
```

```
WEST
```

Existen multitud de filtros para casi cualquier uso. Puedes consultar el listado de todos los filtros de Jinja2 incrustados aquí.

Roles

Los roles son un nivel de abstracción sobre las tareas y playbooks que permiten estructurar la configuración de Ansible en un formato modular y reutilizable, permitiendo dividir un playbook complejo en partes más pequeñas. A medida que se añaden más funcionalidades a los playbooks, éstos pueden volverse poco manejables y difíciles de mantener.

Organizar la configuración de Ansible en roles permite reutilizar pasos de configuración comunes entre diferentes tipos de servidores. Aunque esto también es posible incluyendo múltiples archivos de tareas en un solo playbook, los roles se basan en una estructura de directorios conocida y en convenciones de nombres de archivos para cargar automáticamente los archivos que se utilizarán dentro del playbook.

En conclusión, la idea principal de los roles es permitir compartir y reutilizar las tareas utilizando una estructura consistente, al mismo tiempo que facilita el mantenimiento sin duplicar las tareas para toda su infraestructura.

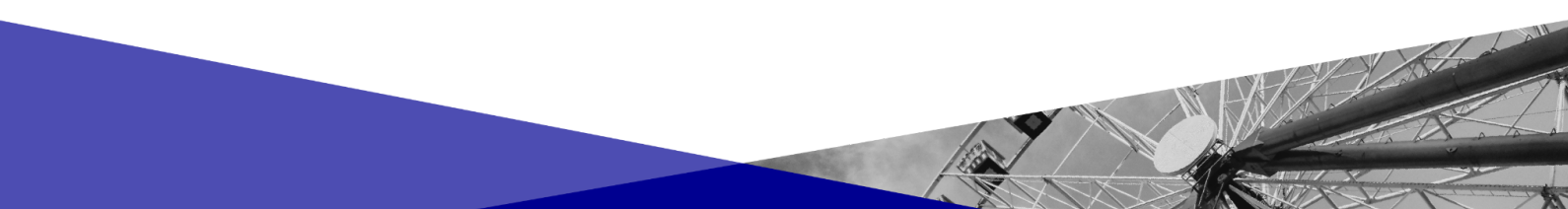
Estructura

Se ha mencionado que los roles necesitan una estructura consistente pero, ¿cómo la definimos? Vamos a explicar esto. Partimos desde un directorio de trabajo que va a ser la raíz de toda la configuración de nuestra infraestructura. En ese directorio, encontraremos los inventarios, playbooks, roles, etc.

Centrándonos en los roles, crearemos en nuestro directorio de trabajo un subdirectorio, llamado roles, donde se alojarán todos los roles necesarios para el proyecto. Por ejemplo, si necesitamos un servidor web como Apache, tendremos un subdirectorio llamado apache con toda la configuración de este.

También hay que tener en cuenta que cada rol tiene una estructura de directorios interna, lo que permite a Ansible saber qué operaciones debe realizar para ese rol. En esta guía se va a definir la estructura completa, pero no quiere decir que todos los roles estén obligados a implementarla, depende de los requisitos de cada rol.

- **defaults**: Este directorio permite establecer variables por defecto



para los roles incluidos o dependientes. Cualquier valor predeterminado que se establezca aquí, puede ser anulado en los playbooks o en los inventories.

- **files**: Contiene archivos que pueden ser copiados o ejecutados en un servidor remoto.
- **handlers**: Almacena todos los handlers referentes a las tareas que se lanzan.
- **meta**: Se reserva para los metadatos de las funciones, normalmente utilizados para la gestión de las dependencias. Por ejemplo, puede definir una lista de roles que deben ser aplicados antes de que el rol actual sea invocado.
- **template**: Contiene las plantillas que generarán archivos en los nodos. Las plantillas suelen utilizar variables definidas en los archivos ubicados en el directorio vars y en la información del controlador que se recoge en tiempo de ejecución.
- **tasks:** Este directorio contiene uno o más archivos con tareas que normalmente se definirían en la sección de tareas de un playbook normal de Ansible. Estas tareas pueden hacer referencia directamente a los archivos y plantillas contenidos en sus respectivos directorios dentro del rol, sin necesidad de proporcionar una ruta completa al archivo.
- **vars:** Las variables de un rol pueden ser especificadas en archivos dentro de este directorio y luego referenciarlas en otra parte.

```
└── working_dir/
    └── roles/
        └── apache/
            └── defaults/
            └── files/
                ├── index.html
            └── handlers/
                ├── main.yml
            └── meta/
            └── template/
                ├── host.conf
            └── tasks/
                ├── main.yml
            └── vars/
```

Si existe un archivo llamado main.yml en un directorio, su contenido se añadirá automáticamente al playbook que llame al rol. Sin embargo, esto no se aplica ni al directorio files ni al templates, ya que su contenido debe ser referenciado explícitamente.

Dependencias entre roles

Haciendo un símil a la programación orientada a objetos, la dependencia entre roles sería algo parecido a la herencia. Resulta ser una buena práctica dado que podemos desacoplar las piezas del puzzle y poder emplearlas en distintos lugares de nuestra infraestructura. Además, permite un mejor mantenimiento de los roles, dado que están claramente segmentados.

La asignación de esos roles dependientes se hará en el subdirectorio *meta* situado en el interior del directorio del rol que depende de ellos. Hay que tener en cuenta también que, aquellos roles "hijos" declarados, se ejecutan siempre antes que el rol "padre" y solo se ejecutan una vez. Esto quiere decir que, en el caso de que dos roles tengan la misma dependencia, solo se ejecutará la primera vez.

Ahora vamos a imaginar que tenemos un rol llamado common, que se encarga de actualizar el SO y otro rol, llamado apache, que levanta un servicio web. Queremos que cada vez que lancemos la tarea de apache también se ejecute la tarea de actualizar el SO del nodo. Para ello, dentro del subdirectorio meta del directorio apache añadimos un fichero con la siguiente información.

```
# working_dir/roles/apache/meta/main.yml
dependencies:
  - role: common
```

Con esta configuración, cuando Ansible procese el playbook con el rol de apache, primero realizará todas las tareas que estén definidas en common.

Como los roles permiten la dependencia entre unos y otros, se puede aplicar el patrón de diseño **wrapper** para "envolver" ciertos roles para cumplir con un propósito específico.

Roles para diferentes plataformas

Hay momentos donde nuestros nodos no tienen todos el mismo sistema operativo. Entonces, ¿creamos el mismo rol pero repetido tantas veces como SO distintos tengamos montados en nuestra infraestructura? Desde un punto de vista de Ansible, no es una buena práctica. Supone un sobreesfuerzo el mantenimiento de ese rol dado que si sufre una modificación tendremos que ir rol a rol realizando ese cambio.

Ansible ya ha contemplado este problema y en vez de escribir roles específicos para cada sistema operativo, lo que haremos será crear un rol y que este apunte a las diferentes plataformas que tengamos definidas.

Por ejemplo, en nuestra infraestructura tenemos dos distros de Linux, una que es Debian y la otra CentOS. Tenemos el rol X pero con pequeños matices para cada plataforma. La estructura de ficheros para este rol es el siguiente:

```
- working_dir
  - roles
    - role_X
      - tasks
        - main.yml
        - debian.yml
        - centos.yml
```

En el fichero *working_dir/roles/roles_X/tasks/main.yml* definiremos todas aquellas tareas comunes a ambos SO e incluiremos tanto el fichero *debian.yml* como *centos.yml* para ejecutar aquellas tareas específicas para cada distro.

```
# working_dir/roles/roles_X/tasks/main.yml
---
- name: general tasks need to be performed to achieve an outcome.
  #some module
  #some tag
...

- include: debian.yml
  when: ansible_os_family == 'Debian'
  #some tag
```

```
- include: centos.yml
  when: ansible_os_family == 'CentOS'
  #some tag
```

```
# working_dir/roles/roles_X/tasks/debian.yml
---
- name: specific tasks need to be performed to achieve an outcome for
Debian
  #some module
  #some tag
...
```

```
# working_dir/roles/roles_X/tasks/centos.yml
---
- name: specific tasks need to be performed to achieve an outcome for
CentOS
  #some module
  #some tag
...
```

De esta forma se facilita el mantenimiento ya que se consigue desacoplar todos los elementos y se permite que la arquitectura pueda escalar sin ninguna dificultad.

Módulos

Un módulo de Ansible es un programa (un script) que se puede utilizar desde el terminal o desde una tarea de un *playbook*. Ansible ejecuta cada módulo, normalmente en el nodo administrado remoto, y recopila valores de retorno. La mayoría de los módulos se aloja en las colecciones. Existen miles de módulos ya creados, pero puedes crear tus propios módulos si no encuentras uno que te sirva.

El módulo puede tener parámetros. Puedes pasar los parámetros a un módulo como *key=value* separando con los espacios. Algunos módulos no tienen parámetros y los módulos *shell* solamente tienen un parámetro que es un String con el comando que se van a ejecutar.

Desde el terminal se ejecuta el módulo utilizando el siguiente patrón,

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

donde *pattern* es un patrón de Ansible que puede referirse a un solo host, una dirección IP, un grupo de inventario, un conjunto de grupos o todos los hosts de inventario. Los patrones de Ansible son muy flexibles. Puedes excluir o añadir subconjuntos de hosts, usar comodines o expresiones regulares, etc. Ansible se ejecuta en todos los hosts de inventario incluidos en el patrón. *module* es el nombre del módulo. Después de la opción -a van los argumentos correspondientes del módulo que se quieran pasar.

Todos los módulos devuelven un JSON, así que se pueden construir con cualquier lenguaje. Los módulos deben ser idempotentes y deben evitar realizar cambios si detectan que el estado actual coincide con el estado final deseado. Por ejemplo:

```
ansible webservers -m service -a "name=httpd state=restarted"
```

Ansible tiene diferentes tipos de módulos. Uno de ellos es el *action plugin*.

Los ***action plugins*** se ejecutan siempre en *el controlador* (máquina donde está instalado Ansible). La mayoría de los *action plugins* configuran algunos valores en el controlador y luego invocan un módulo real en el nodo administrado que hace algo con estos valores. Los plugins aumentan la funcionalidad principal de Ansible y se ejecutan en el nodo de control

dentro del proceso de Ansible. Además, ofrecen opciones y extensiones para las funciones principales de Ansible: transformación de datos, registro de salida, conexión al inventario y más.

Módulos de estilo nuevo (*new style*) son los módulos que vienen con Ansible por defecto. Si bien puedes escribir módulos en cualquier idioma, todos los módulos oficiales (incluidos con Ansible) usan Python o PowerShell. Los módulos de nuevo estilo tienen los argumentos del módulo incrustados dentro de ellos de alguna manera. Los módulos de estilo antiguo copian un archivo separado en el nodo administrado, que es menos eficiente ya que requiere dos conexiones de red en lugar de solo una.

Los módulos JSONARGS son aquellos que contienen una cadena (String), que representa los parámetros de módulo en el formato JSON *<<INCLUDE_ANSIBLE_MODULE_JSON_ARGS>>*.

```
json_arguments = """<<INCLUDE_ANSIBLE_MODULE_JSON_ARGS>>"""
```

Ansible luego parsea este String para obtener los parámetros de módulo.

Si un módulo tiene la cadena WANT_JSON en cualquier lugar, Ansible lo trata como un módulo no nativo que acepta un nombre de archivo como su único parámetro de línea de comando. El nombre de archivo es para un archivo temporal que contiene una cadena JSON que contiene los parámetros del módulo. El módulo necesita abrir el archivo, leer y analizar los parámetros, operar con los datos e imprimir sus datos de retorno como un diccionario JSON en stdout antes de salir.

Desde Ansible 2.2 los módulos también pueden ser binarios. Ansible no realiza ninguna conversión para hacerlos portátiles a diferentes sistemas, por lo que pueden ser específicos del sistema en el que se compilaron o requerir otras dependencias binarias en tiempo de ejecución. Los módulos binarios también tienen sus argumentos y devuelven datos a Ansible de la misma manera que los módulos JSON.

Los módulos de estilo antiguo son similares a los módulos WANT_JSON, excepto que el archivo que toman contiene pares clave=valor para sus parámetros en lugar de JSON. Ansible decide que un módulo es de estilo antiguo cuando no tiene ninguno de los marcadores que mostrarían que es uno de los otros tipos.

Custom modules

Ansible tiene muchos módulos integrados (incrustados) por defecto. El módulo Apt nos permite instalar los paquetes, el módulo Service, lanzar y parar los servicios, el módulo Debug, imprimir las variables durante la ejecución de script, el módulo Copy, copiar los ficheros, el módulo Command, lanzar un comando, etc.

A pesar de que Ansible tiene muchísimos módulos que pueden hacer casi cualquier cosa, a veces tenemos la necesidad de crear nuestro propio módulo.

Para crear un módulo personalizado tienes que crear un fichero *module_name.py* y ponerlo en

- cualquier directorio agregado a la variable de entorno $ANSIBLE_LIBRARY
- ~ /.ansible/plugins/modules /
- /usr/share/ansible/plugins/modules/

La estructura típica de un módulo consiste en la cabecera:

```
#!/usr/bin/python

# Copyright: (c) 2020, Your Name <YourName@example.org>
# GNU General Public License v3.0+ (see COPYING or
https://www.gnu.org/licenses/gpl-3.0.txt)
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type
```

Una variable que contiene la documentación del módulo:

```
DOCUMENTATION = r'''
---
module: my_test_info

short_description: This is my test info module

version_added: "1.0.0"

description: This is my longer description explaining my test info
module.
```

```
options:
    name:
        description: This is the message to send to the test module.
        required: true
        type: str

author:
    - Your Name (@yourGitHubHandle)
'''
```

Un ejemplo de uso que ayudará al usuario a entender el funcionamiento del módulo:

```
EXAMPLES = r'''
#
- name: Test with a message
  my_namespace.my_collection.my_test_info:
    name: hello world
'''
```

Los ejemplos de valores de retorno:

```
RETURN = r'''
# original_message:
    description: The original name param that was passed in.
    type: str
    returned: always
    sample: 'hello world'
message:
    description: The output message that the test module generates.
    type: str
    returned: always
    sample: 'goodbye'
my_useful_info:
    description: The dictionary containing information about your
system.
    type: dict
    returned: always
    sample: {
        'foo': 'bar',
        'answer': 42,
    }
```

```
'''
```

Tenemos que importar el módulo AnsibleModule para construir nuestro módulo.

```
from ansible.module_utils.basic import AnsibleModule
```

El método run_module() contiene toda la lógica de nuestro módulo. Obtiene los parámetros del módulo, define el resultado y lo devuelve.

```
def run_module():
    # los parámetros de nuestro módulo
    module_args = dict(
        name=dict(type='str', required=True),
    )

    # definimos el resultado
    # message - el mensaje que queremos devolver
    # changed=true - si cambiamos el estado de nodo
    # también podemos incluir la información que queremos devolver

    result = dict(
        changed=False,
        original_message='',
        message='',
        my_useful_info={},
    )

    # el objeto AnsibleModule es una abstracción de Ansible
    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )

    # si el usuario solo trabaja con el módulo en el “check mode”
    # (sin hacer ningún cambio)
    # simplemente devolvemos el resultado
    if module.check_mode:
        module.exit_json(**result)

    # manipulamos el estado si hace falta
    result['original_message'] = module.params['name']
```

```
    result['message'] = 'goodbye'
    result['my_useful_info'] = {
        'foo': 'bar',
        'answer': 42,
    }
    #devolvemos el resultado
    module.exit_json(**result)
```

Terminamos definiendo el punto de entrada de nuestro módulo:

```
def main():
    run_module()

if __name__ == '__main__':
    main()
```

Después de crear tu propio módulo puedes invocarlo a través del comando ansible o ansible-doc para ver su documentación.

```
ansible localhost -m my_custom_module
```

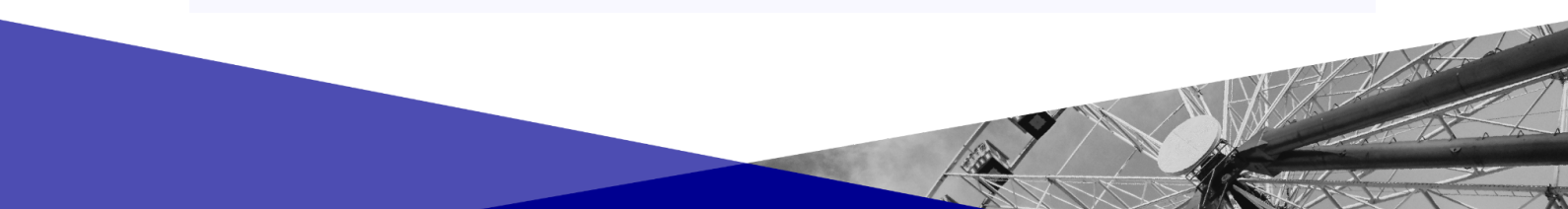
De este modo puedes ver si está definida correctamente la documentación del módulo.

```
ansible-doc -t module my_custom_module
```

Puedes limitar la disponibilidad de tu módulo. Si quieres utilizar un módulo local solo con los playbooks o solo para rol, ponlo en un subdirectorio llamado *library* en el directorio que contiene esos playbooks.

Se pueden crear los módulos con cualquier idioma respetando las mismas reglas que utilizas creando módulos con Python:

```
#!/bin/bash
#
# este script tiene dos parámetros
# 1. app
# 2. appv
# y imprime el resultado
```



```
changed="false"
source $1
display="La app es $app tiene la versión $appv"

if [ "$app" == "bash" ]; then
  changed="true"
fi

printf '{"changed": %s, "msg": "%s"}' "$changed" "$display"

exit 0
```

Depurando los módulos

Para poder depurar los módulos se puede utilizar epdb e insertar dentro del código del módulo:

```
import epdb; epdb.serve()
```

para poder lanzar el servidor telnet en el nodo remoto. Ejecutando epdb.connect() puedes conectarte con el debugger.

Otra posible manera de depurar el código del módulo sería desarchivar los módulos y cambiar el código de módulo para ver el efecto. Los módulos de Ansible están en un archivo zip que contiene el archivo del módulo y los distintos módulos de Python dentro de un script contenedor. Para ver lo que está sucediendo realmente en el módulo, debes extraer el archivo de módulo del contenedor. El script contenedor proporciona métodos auxiliares que permiten hacer eso.

1. Primero tienes que establecer la variable de entorno *ANSIBLE_KEEP_REMOTE_FILES = 1* para que Ansible mantenga los archivos del módulo remoto en lugar de eliminarlos después de que el módulo termine de ejecutarse.

```
$ ANSIBLE_KEEP_REMOTE_FILES=1 ansible localhost -m ping -a
'data=debugging_session' -vvv
```

2. Entrar en el directorio temporal del paso anterior. Si el comando anterior se ejecutó en un host remoto, conéctate primero a ese host

antes de intentar navegar al directorio temporal.

```
$ ssh remotehost  # only if not debugging against localhost
$ cd /home/badger/.ansible/tmp/ansible-tmp-1461434734.35-235318071810595
```

3. Ejecutar el comando explode del contenedor para convertir la cadena en algunos archivos de Python con los que pueda trabajar.

```
$ python AnsiballZ_ping.py explode
```

4. Dentro del directorio temporal vas a ver muchos ficheros auxiliares y tienes que encontrar el fichero ping.py que contenga el código de módulo.
5. Una vez cambies el código lanza el comando *execute* para ejecutar el código cambiado de módulo.

```
$ python AnsiballZ_ping.py execute
```

Puedes continuar ejecutando el módulo de esta manera hasta que comprendas el problema. Luego, puedes copiar los cambios en el archivo del módulo real y probar que funciona a través de los comandos *ansible* o *ansible-playbook*. La guía entera se puede leer en [la documentación de Ansible].

AWS con Ansible

Para trabajar con Amazon Web Services existen multitud de módulos de Ansible. Desde el principio, Ansible ha ofrecido un soporte profundo para AWS. Ansible se puede utilizar para definir, implementar y administrar una amplia variedad de servicios de AWS. Incluso los entornos de AWS más complicados se pueden describir fácilmente en los playbooks de Ansible. Una vez que tus entornos de aplicaciones basados en AWS se describen con Ansible, puedes implementarlos una y otra vez, escalando fácilmente a cientos o miles de instancias en varias regiones, con los mismos resultados.

Por ejemplo, podemos crear un playbook para provisionar las instancias de EC2.

Con la ayuda de *Ansible-Vault*, tenemos que crear el fichero cifrado con la clave de acceso y la clave secreta de AWS para poder usarlas luego en los

playbooks (las variables *ec2_access_key* y *ec2_secret_key*). La parte de los secretos en Ansible la vamos a ver más adelante. Otra manera menos segura es poner la variable *ansible_ssh_private_key_file=~/AWS/mykeys.pem* dentro del fichero de tu inventario.

Definimos las variables para poder utilizarlas luego en el playbook:

```
- hosts: localhost
  connection: local
  gather_facts: false

  vars:
    key_name: my_aws
    region: us-east-2
    image: ami-0f93b5fd8f220e428 #
https://cloud-images.ubuntu.com/locator/ec2/
    id: "web-app"
    sec_group: "{{ id }}-sec"
```

Definimos las tareas que recogen la información sobre las instancias EC2 dentro del bloque "Facts". Hay que instalar los paquetes *boto*, *botocore*, *boto3* y *python v.2.6+* previamente, tanto en el controller como en los nodos manejados. Siempre puedes consultar los requisitos de cada módulo de Ansible en la documentación de cada módulo. Por ejemplo, la documentación del módulo ec2_instance_info.

```
tasks:

    - name: Facts
      block:

      - name: Get instances facts
        ec2_instance_info:
          aws_access_key: "{{ec2_access_key}}"
          aws_secret_key: "{{ec2_secret_key}}"
          region: "{{ region }}"
        register: result

      - name: Instances ID
        debug:
          msg: "ID: {{ item.instance_id }} - State: {{ item.state.name
}} - Public DNS: {{ item.public_dns_name }}"
```

```
        loop: "{{ result.instances }}"

      tags: always
```

Hagamos otro bloque que nos sirve para provisionar las instancias. Dentro de este bloque vamos a crear la tarea de subida de las llaves secretas de AWS.

```
- name: Provisioning EC2 instances
      block:

      - name: Upload public key to AWS
        ec2_key:
          name: "{{ key_name }}"
          key_material: "{{ lookup('file', '~/.ssh/{{key_name }}.pub')
}}"
          region: "{{ region }}"
          aws_access_key: "{{ec2_access_key}}"
          aws_secret_key: "{{ec2_secret_key}}"
```

Otra tarea es crear *un grupo de seguridad de AWS*. Un grupo de seguridad funciona como un firewall virtual para las instancias EC2 para controlar el tráfico entrante y saliente.

```
- name: Create security group
        ec2_group:
          name: "{{ sec_group }}"
          description: "Sec group for app {{ id }}"
          region: "{{ region }}"
          aws_access_key: "{{ec2_access_key}}"
          aws_secret_key: "{{ec2_secret_key}}"
          rules:
            - proto: tcp
              ports:
                - 22
              cidr_ip: 0.0.0.0/0
              rule_desc: allow all on ssh port
        register: result_sec_group
```

Por último, el módulo ec2 de Ansible crea la instancia de EC2 con los ajustes que queramos.

```
- name: Provision instance(s)
        ec2:
          aws_access_key: "{{ec2_access_key}}"
          aws_secret_key: "{{ec2_secret_key}}"
          key_name: "{{ key_name }}"
          id: "{{ id }}"
          group_id: "{{ result_sec_group.group_id }}"
          image: "{{ image }}"
          instance_type: t2.micro
          region: "{{ region }}"
          wait: true
          count: 1
```

Asignamos al último bloque los tags *never* y *create_ec2* para que nunca se ejecute sin que utilicemos el tag *create-ec2* implícitamente.

```
tags: ['never', 'create_ec2']
```

Para ejecutar las tareas tienes que añadir el argumento --ask-vault-pass. De este modo, cada vez que ejecutas el playbook, Ansible te va a pedir la contraseña del fichero cifrado de Ansible-Vault que has creado previamente. El fichero tiene que estar en la carpeta *group_vars/all* dentro de la carpeta con tu playbook.

```
ansible-playbook playbook.yml --ask-vault-pass
```

Para ejecutar el bloque con el tag *create_ec2*, tienes que ponerlo explícitamente.

```
ansible-playbook playbook.yml --ask-vault-pass --tags create_ec2
```

Si usas directamente el certificado *pem* de Amazon AWS dentro de tu inventario, no hace falta crear y usar el fichero cifrado de Ansible-Vault.

Docker con Ansible

Ansible es capaz de conectarse con un contenedor de Docker a través de SSH. Para ello es necesario definir un contenedor con un servicio SSH para

que Ansible pueda conectar y ejecutar los playbooks.

Lo primero que tenemos que hacer es definir un fichero *Dockerfile* con la configuración de la imagen del contenedor que contenga un servidor SSH. Para ello, nos basaremos en esta implementación de OpenSSH.

Un dockerfile que tiene OpenSSH Server:

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo 'root:root' | chpasswd
RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/'
/etc/ssh/sshd_config
# SSH login fix. Otherwise user is kicked off after login
RUN sed 's@session\s*required\s*pam_loginuid.so@session optional
pam_loginuid.so@g' -i /etc/pam.d/sshd
ENV NOTVISIBLE "in users profile"
RUN echo "export VISIBLE=now" >> /etc/profile
EXPOSE 22
CMD ["/usr/sbin/sshd", "-D"]
```

Construimos la imagen:

```
docker image build -t docker-openssh:v1 .
```

Arrancamos el contenedor con la imagen creada anteriormente:

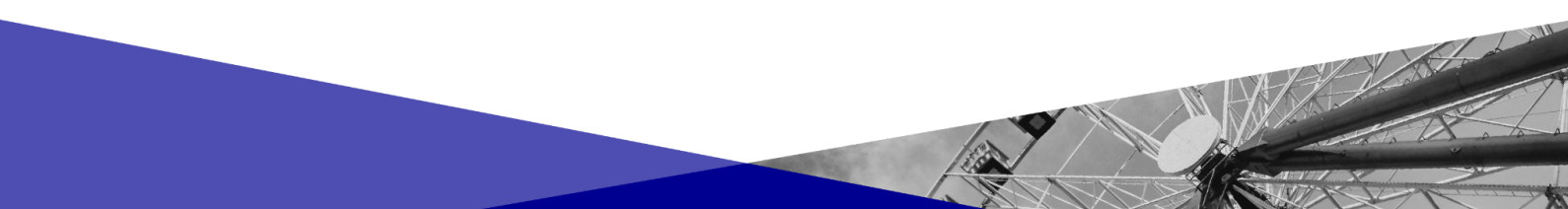
```
docker container run -d --name dockerssh -p 50022:22 docker-openssh:v1
```

Una vez arrancado el contenedor, podemos obtener la IP asignada. Para ello, tenemos que inspeccionar la configuración del contenedor para obtener la ip del Gateway. Esta operación, la realizamos con el siguiente comando:

```
docker container inspect dockerssh | grep Gateway
```

Suponiendo que la ip obtenida es la 172.17.0.1, la conexión SSH la realizamos así:

```
ssh -p 50022 root@172.17.0.1
```



En el caso de macOS tienes que usar *localhost* en lugar de IP de contenedor.

```
ssh -p 50022 root@localhost
```

Ahora nos toca definir el inventario en el fichero */etc/ansible/hosts* (se puede usar otra ubicación utilizando el argumento *-i* con el comando *ansible-playbook*).

```
[all:vars]
ansible_connection=ssh
ansible_user=root
ansible_ssh_pass=root
[dockers]
ubuntu-sshserver ansible_port=50022 ansible_host=172.17.0.1
```

Con estas modificaciones definimos las credenciales de conexión al contenedor Docker y definimos la configuración del contenedor para utilizarla en los playbook.

Definimos un playbook que nos permite instalar, por ejemplo, OpenJDK. El playbook es el siguiente:

```
--- # instalamos openjdk
- hosts: ubuntu-sshserver
  become: yes
  tasks:
   - name: install jdk
     apt:
       name: openjdk-8-jdk
       state: latest
       install_recommends: no
```

Al final ejecutamos nuestro playbook:

```
$ ansible-playbook -i /ansible/hosts ./openjdk-setup.yml
```

Para macOS tenemos que instalar previamente el paquete sshpass.

Para comprobar que OpenJDK está instalado dentro de nuestro contenedor conectamos con el contenedor:

```
$ ssh -p 50022 root@172.17.0.1
# ssh -p 50022 root@localhost para macOS
```

Después comprobamos la versión de java instalada:

```
$ java -version
```

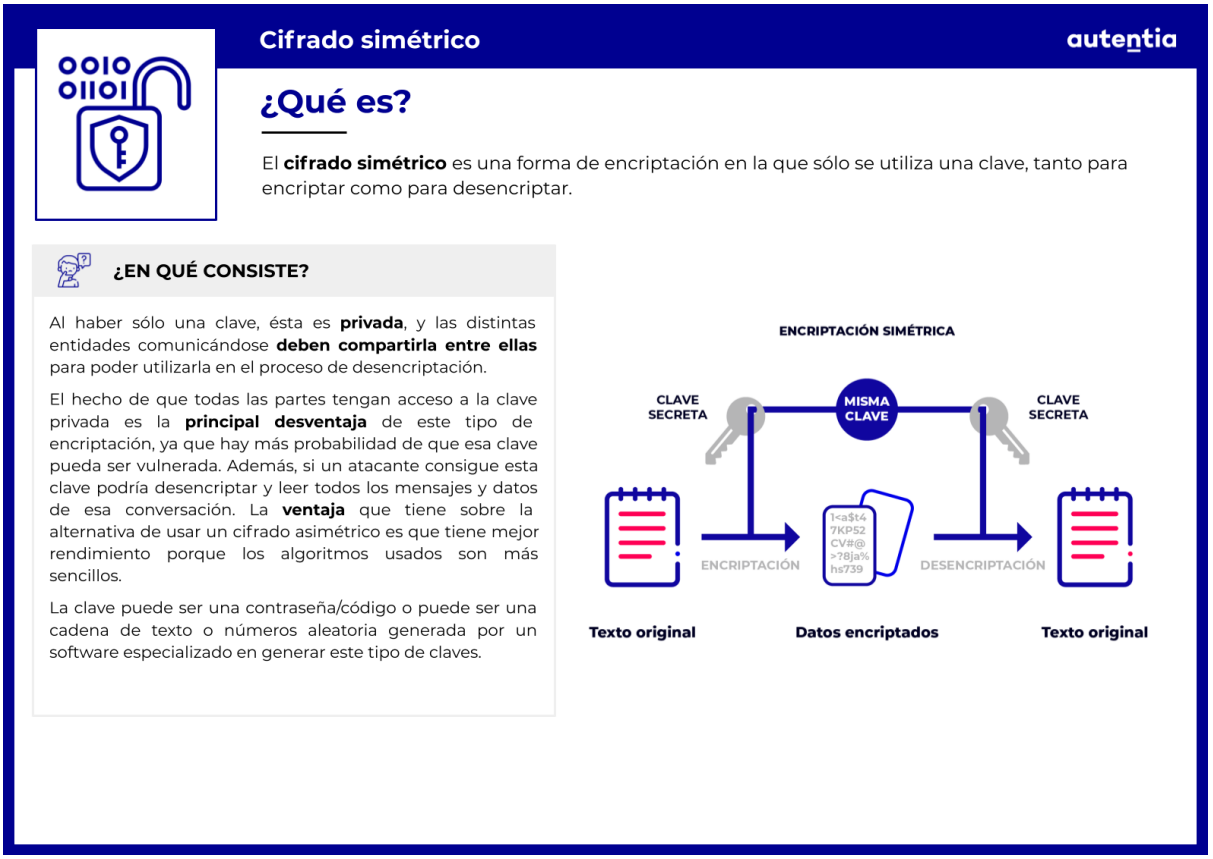
Si no te sale el típico error “command not found” es que tenemos OpenJDK instalado dentro del Docker.

Protegiendo secretos en Ansible

Ansible Vault es una función dentro de Ansible que permite cifrar valores y estructuras de datos dentro de los proyectos de IaC. Esta funcionalidad es capaz de asegurar cualquier dato sensible que no se deba ver públicamente, como las contraseñas o claves privadas, y que sea necesario para ejecutar con éxito los playbooks de Ansible. En tiempo de ejecución, la herramienta descifra el contenido ya cifrado por el Vault a través de la clave que se le proporciona.

Como el lenguaje de configuración es YAML, Ansible Vault puede cifrar cualquier fichero que tenga esta extensión, por lo que se puede cifrar cualquier elemento del proyecto. Entonces, ¿se cifra todo el proyecto? Como buena praxis, se debe cifrar la menor cantidad de datos posibles, solo aquellos que sean sensibles o de gran valor. Esto se debe a que la operación de descifrar conlleva un coste computacional y cuántos más elementos cifrados se tengan, más tiempo llevará realizar el despliegue.

Ansible Vault utiliza el algoritmo *AES-256* para proporcionar un cifrado simétrico con una contraseña proporcionada por el usuario. Esto significa que se utiliza la misma contraseña para cifrar y descifrar el contenido, lo que resulta útil desde el punto de vista de la usabilidad.



Ahora que entendemos un poco lo que es Vault, vamos a ver cómo utilizar *ansible-vault* para cifrar, modificar y descifrar archivos.

Cifrar ficheros

Cuando queremos cifrar el contenido de un fichero, tenemos dos opciones. La primera sería cifrar el contenido de un nuevo fichero y la segunda, cifrar el contenido de un fichero existente.

Para el primer caso, vamos a crear un fichero donde tengamos almacenado algún tipo de dato sensible como puede ser la contraseña de una base de datos. Primero ejecutamos el siguiente comando:

```
$ ansible-vault create vars.yml
```

Nos pedirá que se introduzca la clave Vault con la que va a cifrar el contenido de nuestro archivo. Una vez introducido, automáticamente se abrirá en la terminal un editor de texto para que añadamos la información.

En este ejemplo, hemos añadido la siguiente información:

```
# vars.yml
---
database_pass: "It's our super secret password for database"
```

Si ahora hacemos `cat vars.yml` podremos observar que el fichero está cifrado:

```
$ cat vars.yml
$ANSIBLE_VAULT;1.1;AES256
63343732323565383462666462383564346135653061666538373664303836623639366
1343366353839303135336561323135643135326663373636356534620a34636262636465
6134653635303563303861633335346561663937323838316134313033653162333763616
3832373064323362326230383162376562363936390a363735633239313065366639639
3865363138373435373766653066306433616231313236373837653132356537653832306
6264643936326366326232396433343938633639633161613336613963383537306636656
6439343530346438666461383162663963353365313263643261353562653062386332626
6263656338336230646236353435303638363832396462313536313562663562333764346
6564
```

Ya tenemos nuestros datos sensibles cifrados. Sólo nos queda comprobar que el playbook descifra la información correctamente. Para realizar esta comprobación vamos a depurar en local con el siguiente playbook:

```
# playbook.yml
---
- hosts: localhost
 gather_facts: false
 vars_files:
   - vars.yml
 tasks:
   - name: print database password from vars.yml
     debug:
       var: database_pass
```

Ahora procedemos a ejecutar el playbook recién creado con el comando `ansible-playbook playbook.yml --ask-vault-pass`. Nos pedirá que introduzcamos la clave para que Ansible sea capaz de descifrar la información. El resultado es el siguiente:

```
$ ansible-playbook playbook.yml --ask-vault-pass
```

```
Vault password:

PLAY [localhost]
*********************************************************************
*****************************************************************

TASK [print database password from vars.yml]
*********************************************************************
*********************************
ok: [localhost] => {
    "database_pass": "It's our super secret password for database"
}

PLAY RECAP
*********************************************************************
*********************************************************************
localhost                  : ok=1    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

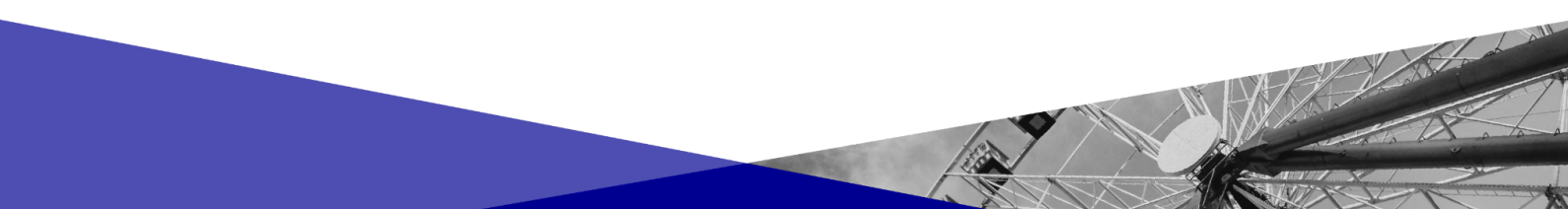
Hemos conseguido descifrar nuestros datos sensibles y los hemos usado en un playbook con total seguridad.

Para el segundo caso, cifrar un fichero que ya tenemos creado y necesitamos añadirle esta capa de seguridad, debemos usar el siguiente comando: `ansible-vault encrypt vars_prod.yml`

En este caso, tenemos un fichero “vars_prod.yml” con todas las claves necesarias para poder levantar un entorno de producción. Como no queremos que esta información sea pública, nos vemos en la obligación de cifrar estos datos. Para comprobar que funciona correctamente, podemos repetir los pasos anteriores a partir de la fase en la que se dispone de un fichero cifrado.

Modificar fichero cifrado

Cuando se necesite realizar algunas modificaciones en los ficheros cifrados, Ansible te ofrece el comando `ansible-vault edit example.yml`. Cuando se ejecute, te pedirá que se inserte la contraseña y se abrirá un editor de texto en la terminal para que se realicen los cambios oportunos. Una vez hechos, cuando se guarde el fichero, se volverá a cifrar con la clave del Vault.



Descifrar fichero

Como estamos viendo, con el comando ansible-vault podemos realizar todas las operaciones. Para el caso de descifrar el fichero, ejecutando el comando `ansible-vault decrypt example.yml` e introduciendo la clave, se obtiene el contenido del fichero en claro. La salida del comando debería ser algo parecido a lo siguiente:

```
$ ansible-vault decrypt example.yml
Vault password:
Decryption successful
```

Podemos ver que el fichero example.yaml ha sido descifrado correctamente.

Cambiar la clave de cifrado

También es posible cambiar la clave de cifrado ejecutando el siguiente comando:

```
$ ansible-vault rekey example.yml
Vault password:
New Vault password:
Confirm New Vault password:
Rekey successful
```

Control de las tareas

Cuando Ansible recibe un código de retorno distinto de cero de un comando o un error de un módulo, por defecto deja de ejecutarse en ese host y continúa en otros hosts. Sin embargo, en algunas circunstancias, es posible que quieras un comportamiento diferente. A veces, un código de retorno distinto de cero indica éxito o deseas que un error en un host detenga la ejecución en todos los hosts. Ansible proporciona herramientas y configuraciones para manejar estas situaciones y ayudar a obtener el comportamiento, los resultados y los informes que deseas.

De forma predeterminada, Ansible deja de ejecutar tareas en un host cuando una tarea falla en ese host. Se puede indicar que se ignoren los errores agregando el atributo *ignore_errors* y teniendolo a *true* para continuar:

```
- name: Ejecutarla incluso si falla
  ansible.builtin.command: /bin/false
  ignore_errors: yes
```

La directiva *ignore_errors* solo funciona cuando la tarea se puede ejecutar y devuelve un valor de "failure". No hace que Ansible ignore errores de variables indefinidas, fallos de conexión, problemas de ejecución (por ejemplo, paquetes faltantes) o errores de sintaxis.

Además, puedes ignorar el fallo de una tarea debido a que la instancia de host sea *"unreachable"* (inalcanzable) con la palabra clave *ignore_unreachable*. Ansible ignora los errores de la tarea, pero continúa ejecutando tareas futuras contra el host inalcanzable.

Si Ansible no puede conectarse a un host, lo marca como *"unreachable"* y lo elimina de la lista de hosts activos para la ejecución. Puedes usar la meta *clear_host_errors* para reactivar todos los hosts, por lo que las tareas posteriores se intentarán ejecutar en todos los hosts nuevamente.

```
- meta: clear_host_errors
```

Además Ansible te permite definir la condición de error en cada tarea usando *fail_when*. Puedes unir múltiples condiciones con un operador *and*, lo que significa que la tarea solo falla cuando se cumplen todas las

condiciones. Si quieres causar un error cuando se cumple alguna de las condiciones, tienes que definir las condiciones en una cadena con un operador explícito *or*:

```
- name: Fail task when both files are identical
  ansible.builtin.raw: diff foo/file1 foo/file2
  register: diff_cmd
  failed_when: diff_cmd.rc == 0 or diff_cmd.rc >= 2
```

También puedes definir el estado “changed” (cambiado) usando *changed_when.* Con esto Ansible te permite usar los códigos de retorno o la salida, para decidir si el cambio tiene que ser reflejado en la estadística de Ansible o si algún *handler* tiene que estar activado.

```
tasks:
  - name: Manda el estado 'changed' cuando el código de retorno sea 2
    ansible.builtin.shell: /usr/bin/sampleapp
    register: result
    changed_when: "result.rc != 2"
  - name: Nunca mandará el estado “changed”
    ansible.builtin.shell: wall 'Hola'
    changed_when: false
```

A veces se desea que un error en un solo host, o los errores en un cierto porcentaje de hosts, cancelen toda la ejecución en todos los hosts. Puedes detener la ejecución de las tareas después de que ocurra el primer error con *any_errors_fatal*. Para un control más detallado, puedes usar *max_fail_percentage* para cancelar la ejecución después de que un determinado porcentaje de hosts haya fallado.

Puedes controlar cómo responde Ansible a los errores de tareas en bloques con los llamados bloques de rescate (*rescue*) y *always*.

Los bloques de rescate especifican las tareas que se ejecutarán cuando falle una tarea anterior de un bloque. Este enfoque es similar al manejo de excepciones en muchos lenguajes de programación. Ansible sólo ejecuta bloques de rescate después de que una tarea devuelva un estado "fallido". Las definiciones de tareas incorrectas y los hosts inalcanzables no activarán el bloqueo de rescate.

```
tasks:
 - name: Gestionar el error
```

```
  block:
   - name: Imprimir el mensaje
     ansible.builtin.debug:
        msg: 'Ejecución normal'
   - name: Forzar el error
     ansible.builtin.command: /bin/false
   - name: Nunca se va a ejecutar
     ansible.builtin.debug:
        msg: 'Nunca se ejecutará'
   rescue:
    - name: Imprimir el error
      ansible.builtin.debug:
        msg: 'S.O.S! ha pasado un error'
```

También puedes agregar una sección *always* a tu bloque. Las tareas de la sección *always* se ejecutan independientemente del estado de la tarea del bloque anterior.

```
- name: La tarea con el siempre
  block:
    - name: Imprimir el mensaje
      ansible.builtin.debug:
        msg: 'Todo ok'
    - name: Forzar el error
      ansible.builtin.command: /bin/false
    - name: Nunca se ejecutará
      ansible.builtin.debug:
        msg: 'Nunca van a ejecutarme :-('
   always:
     - name: Siempre se ejecuta
       ansible.builtin.debug:
         msg: "Siempre me ejecutan :-)"
```

Ansible proporciona un par de variables para tareas en la parte de rescate de un bloque. La primera es *ansible_failed_task* que contiene la tarea fallida y desencadena el rescate. Puedes sacar el nombre de la tarea usando *ansible_failed_task.name*. La otra es *ansible_failed_result* es el resultado de retorno capturado de la tarea fallida.

Handlers

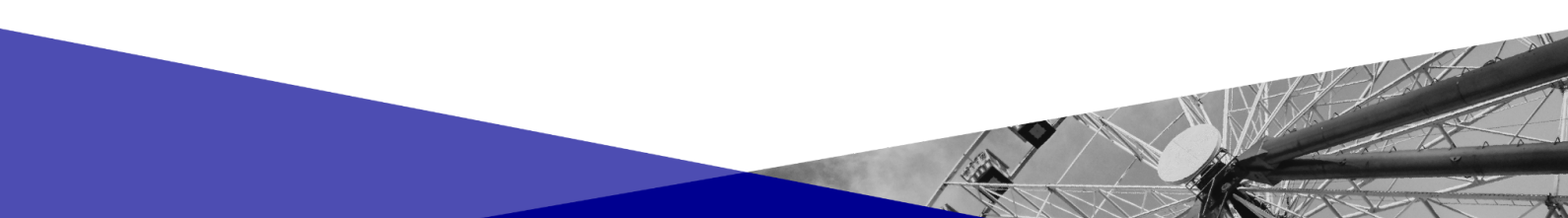
A veces necesitas ejecutar las tareas sólo cuando hay un cambio. Por ejemplo, lanzar un servicio después de que una tarea haya cambiado su configuración. En este caso, es muy útil usar *los handlers*, tareas que se ejecutan sólo cuando, tras la ejecución de una tarea, ésta produce algún cambio. Cada *handler* tiene que tener un nombre único. Se usa la palabra *notify* para indicar qué *handlers* se han de ejecutar.

```
---
- name: Comprobar la instalación de Apache
  hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: Imprimir el fichero de configuración
    ansible.builtin.template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf
    notify:
    - Relaunch Apache

  handlers:
    - name: Relaunch Apache
      ansible.builtin.service:
        name: httpd
        state: restarted
```

Ansible ejecuta los *handlers* al final de cada *play*. Si una tarea notifica a un *handler*, pero otra tarea falla más adelante en el play, de forma predeterminada, el *handler* no se ejecuta en ese host, lo que puede dejar al host en un estado inesperado. Por ejemplo, una tarea podría actualizar un archivo de configuración y notificar a un *handler* que reinicie algún servicio. Si falla una tarea posterior en la misma play, es posible que se cambie el archivo de configuración, pero el servicio no se reiniciará.

Puedes cambiar este comportamiento con la opción de línea de comandos *--force-handlers*, incluyendo *force_handlers: true* en tu playbook, o agregando *force_handlers = true* a *ansible.cfg*. Cuando los *handlers* son forzados, Ansible ejecutará todos los *handlers* notificados en todos los hosts, incluso en los hosts con tareas fallidas. Ten en cuenta que ciertos

errores aún pueden evitar que el *handler* se ejecute, por ejemplo cuando un host se vuelve inaccesible.



Testing

La sección de testing es un apartado clave dentro del ciclo de vida de cualquier producto software. Incluso en el mundo DevOps, es necesario comprobar que los ficheros de configuración que se van a encargar de preparar los entornos funcionan correctamente. Para ello, vamos a hacer uso de Molecule, un framework diseñado para ayudar al desarrollo y prueba de roles en Ansible. Desde septiembre de 2020, Ansible ha anunciado la adopción de Molecule y Ansible-lint como proyectos oficiales. Esto demuestra la confianza que la comunidad tiene en esta herramienta y la cantidad de trabajo que están poniendo en hacerla cada vez mejor.

Para comenzar a usar Molecule debes tener previamente instalados los paquetes necesarios, tales como molecule y python-vagrant, si queremos que Vagrant sea el encargado de levantar las instancias para realizar los tests o python-docker, en el caso de que sea Docker el encargado.

```
$ pip install molecule python-vagrant
```

Una vez instalados los paquetes, hay dos maneras de configurar Molecule en los roles. La primera es si se quiere crear un nuevo rol se generará la estructura de ficheros estándar para un rol de Ansible, y la segunda es si ya se tiene un rol creado.

```
# crear nuevo rol de Ansible con Molecule incluido
molecule init role <role_name> -d vagrant
# añadir Molecule a un rol previamente creado
molecule init scenario -r <role_name> -d vagrant
```

Como se puede observar, para la demostración que vamos a seguir, hemos usado Vagrant. Por eso se añade el argumento `-d vagrant` para que Molecule tenga constancia de cuál es el driver que hemos elegido.

Una vez ejecutados alguno de los dos comandos de arriba, se nos habrá creado un subdirectorio dentro del directorio del role llamado molecule con los siguientes ficheros.

```
└── molecule/
    └── default/
        ├── molecule.yml
```

```
        ├── converge.yml
        ├── verify.yml
        ├── INSTALL.rst
```

En el fichero `molecule.yml`, especificamos toda la configuración de Molecule necesaria para testear los roles.

```
# molecule.yml
---
dependency:
 name: galaxy
driver:
 name: vagrant
 provider:
   name: virtualbox
platforms:
 - name: instance
   box: ubuntu/trusty64
   instance_raw_config_args:
     - "vm.network 'forwarded_port', guest: 80, host: 8088"
provisioner:
 name: ansible
verifier:
 name: ansible
```

- **dependency**. Es el gestor que se encarga de resolver todas las dependencias de los roles. Ansible Galaxy es la dependencia por defecto utilizada por Molecule pero existen otros gestores como Shell y Gilt.
- **driver**. El driver indica a Molecule de dónde queremos que provengan nuestras instancias de prueba. El driver por defecto de Molecule es Docker pero también tiene otras opciones como: AWS, Azure, Google Cloud, Vagrant o Hetzner Cloud.
- **platforms**. Indica qué tipo de instancias queremos lanzar para probar nuestros roles. Esto debe corresponder al driver, por ejemplo, en el fragmento anterior, dice qué tipo de sistema operativo queremos lanzar con Vagrant.
- **provisioner**. El provisioner es la herramienta que ejecuta el archivo converge.yml contra todas las instancias lanzadas (especificadas en

platforms). El único provisioner soportado es Ansible.

- **verifier**. Es la herramienta que valida nuestros roles. Este verificador ejecuta el archivo verify.yml para afirmar que el estado real de nuestra instancia coincide con el estado deseado. Por defecto es Ansible, pero también hay otros verificadores como testinfra, goss e inspec.

El archivo `converge.yml`, se utiliza para convertir el estado de las instancias al estado real declarado en los roles reales que se van a probar. El rol que hemos creado es para levantar un servicio web con Apache:

```
# converge.yml
---
- name: Converge
 hosts: all
 become: yes
 tasks:
   - name: "Include apache"
     include_role:
       name: "apache"
```

El fichero `verify.yml` ejecuta la tarea que llama a los roles de prueba. Estos roles se utilizan para validar que el estado de la instancia coincide con el estado deseado:

```
# verify.yml
---
- name: Verify
 hosts: all
 tasks:
 - name: Gather package facts
   package_facts:
     manager: auto

 - name: Verify Packages
   assert:
     that: "'{{ item }}' in ansible_facts.packages"
   with_items:
     - apache2
```

El archivo `INSTALL.rst` contiene instrucciones para dependencias adicionales necesarias para establecer una interacción exitosa entre Molecule y el driver. Para este caso base, el fichero se va a encontrar con la configuración por defecto.

Los ficheros que acabamos de describir forman parte del ciclo de vida de las pruebas que hace Molecule sobre los roles. Al ciclo de vida se le llama escenario. Es personalizable, ya que los pasos de la secuencia pueden intercambiarse o comentarse para adaptarse a cualquier escenario que se necesite. Cada rol debe tener un escenario por defecto que se llama default. A menos que se indique lo contrario, el nombre del escenario suele ser el nombre del directorio donde se encuentran los archivos de Molecule. A continuación, se muestra el escenario por defecto que se ejecuta cuando ejecutamos la secuencia de comandos correspondiente:

```
scenario:
  create_sequence:
    - dependency
    - create
    - prepare
  check_sequence:
    - dependency
    - cleanup
    - destroy
    - create
    - prepare
    - converge
    - check
    - destroy
  converge_sequence:
    - dependency
    - create
    - prepare
    - converge
  destroy_sequence:
    - dependency
    - cleanup
    - destroy
  test_sequence:
    - dependency
    - lint
    - cleanup
    - destroy
```

```
    - syntax
    - create
    - prepare
    - converge
    - idempotence
    - side_effect
    - verify
    - cleanup
    - destroy
```

A partir del ejemplo anterior, podemos saber lo que sucede cuando ejecutamos un comando de Molecule, por ejemplo, `$ molecule create` ejecutaría entonces create_sequence mientras que `$ molecule check` ejecutaría el check_sequence.

Para el caso que estamos explicando, queremos un escenario que haga las siguientes funciones: eliminar (si existiera) la instancia para crear una "limpia", comprobar que la sintaxis es correcta, crear y preparar la instancia para poder testear los roles, instalar las tareas relacionadas con ese rol, a través de la idempotencia se comprueba que el rol sólo se ha ejecutado una única vez, se realiza el verify para comprobar que se han realizado las tareas y por último, se elimina la instancia para liberar los recursos que se han ocupado.

Antes de ejecutar el comando, debemos realizar una modificación en el fichero molecule.yml para añadir el escenario recién explicado.

```
# molecule.yml
---
dependency:
 name: galaxy
driver:
 name: vagrant
 provider:
   name: virtualbox
platforms:
 - name: instance
   box: ubuntu/trusty64
   instance_raw_config_args:
     - "vm.network 'forwarded_port', guest: 80, host: 8088"
provisioner:
 name: ansible
```

```
verifier:
 name: ansible
scenario:
 test_sequence:
   - destroy
   - syntax
   - create
   - prepare
   - idempotence
   - verify
   - destroy
```

Cuando lo hayamos guardado, ejecutamos el siguiente comando `$ molecule test` para realizar todas las funciones de forma automática. La salida del comando es la que vemos a continuación:

```
INFO     default scenario test matrix: destroy, syntax, create, prepare,
converge, idempotence, verify, destroy
INFO     Running default > destroy

PLAY [Destroy]
*****************************************************************

TASK [Destroy molecule instance(s)]
*******************************************
ok: [localhost] => (item=instance)

TASK [Populate instance config]
************************************************
ok: [localhost]

TASK [Dump instance config]
*************************************************
skipping: [localhost]

PLAY RECAP
*********************************************************************
localhost                  : ok=2    changed=0    unreachable=0
failed=0    skipped=1    rescued=0    ignored=0

INFO     Running default > syntax

playbook:
```

```
/Users/autentia/Downloads/devops05/ansible/roles/apache/molecule/default
/converge.yml
INFO     Running default > create

PLAY [Create]
*************************************************************
.... # En la terminal se verá un output con todas las funciones de test
```

Debugging con Ansible

Comprobar sintaxis

Si necesitamos validar que la sintaxis de los ficheros YAML es correcta, Ansible proporciona el argumento *--syntax-check*. Tan solo se debe ejecutar el siguiente comando y si no da ningún tipo de error quiere decir que la sintaxis está correcta, en caso contrario, se indica el conflicto.

```
$ ansible-playbook playbook.yml --syntax-check
```

Depuración de tareas

Ansible proporciona un debugger sobre las tareas para facilitar la detección y corrección de errores que aparezcan durante la ejecución del playbook. Es útil ya que nos evita estar editando y ejecutando de nuevo para comprobar que los cambios han resuelto el problema.

El debugger ofrece el acceso a todas las funciones que se encuentran en las tareas como pueden ser, definir el valor de las variables o también actualizar los argumentos del módulo, y después volver a ejecutar la tarea con los nuevos cambios. Esto permite al desarrollador que pueda resolver los problemas en tiempo de ejecución.

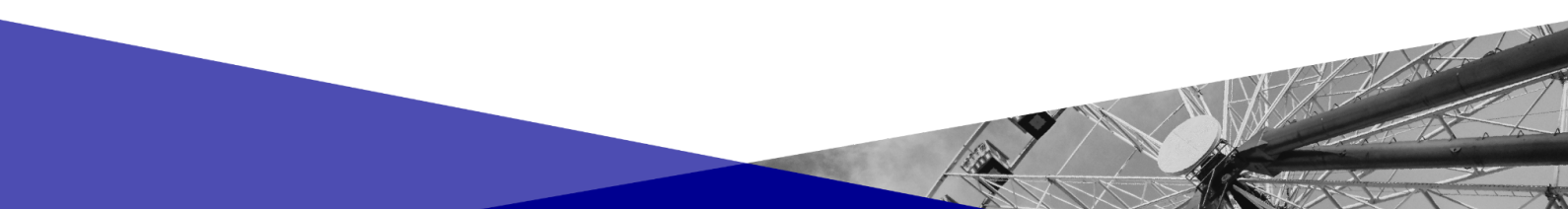
Por defecto está deshabilitado. Existen dos métodos que habilitarán el debugger cuando se ejecute el playbook.

- Con la palabra clave *debugger* en la definición de la tarea.
- Como variable de entorno.

Con la palabra clave *debugger*

Esta keyword se puede usar en todos los niveles del playbook (play, role, block y task). Para que el *debugger* se active, existen cinco valores posibles:

- always: invocar siempre el debug.
- never: no invocarlo nunca.
- on_failed: solo se habilita si una tarea falla.
- on_unreachable: se habilita si no se puede acceder al nodo.



- on_skipped: se invoca cuando la tarea se ha omitido.

Cuando se hace uso de esta keyword, hay que tener en cuenta que Ansible siempre dará preferencia a la definición más específica. Esto quiere decir que si hemos usado *debugger* en un rol y también en las tareas, el debugger se lanzará en función a lo determinado dentro de *task*.

```
---
- hosts: all
  debugger: never
  task:
    - name: Install Apache.
      apt: name=apache2 state=latest
      debugger: on_failed
```

Como variable de entorno

Para habilitar el *debugger* globalmente se dispone de dos opciones. La primera es configurar el fichero *ansible.cfg* añadiendo el siguiente fragmento de código:

```
[defaults]
enable_task_debugger = True
```

También es posible que al lanzar el comando se pueda poner la variable de entorno **ANSIBLE_ENABLE_TASK_DEBUGGER** a True.

```
ANSIBLE_ENABLE_TASK_DEBUGGER=True ansible-playbook -i inventory
playbook.yml
```

En ambos casos, si se ha configurado como True, Ansible lanzará el *debugger* por defecto cuando la tarea que se está procesando tenga algún tipo de error.

Resolver errores con el debugger

Cuando Ansible detecta un error al procesar una tarea y lanza el debugger, pone a nuestra disposición siete comandos que nos pueden ayudar para resolver el problema en tiempo de ejecución.

Primero vamos a explicar cuáles son estos siete comandos y después los

pondremos en práctica con un ejemplo muy sencillo.

- **print (p)**. Imprime la información relacionada con la tarea.
- **task.args[key] = value**. Actualiza los argumentos del módulo.
- **task_vars[key] = value**. Actualiza las variables de la tarea. Si se usa este comando es obligatorio usar update_task.
- **update_task (u)**. Establece los nuevos cambios para cuando se lance de nuevo la tarea.
- **redo (r)**. Relanza la tarea.
- **continue (c)**. Continúa con la ejecución del playbook, ejecutando la siguiente tarea.
- **quit (q)**. Cierra el debugger.

Para el ejemplo práctico, hemos añadido la keyword *debugger* en la tarea que se encarga de instalar Apache. Para forzar el error, se ha añadido la variable *pkg_name* con el valor de *not_exist*:

```
- hosts: all
  gather_facts: false
  become: yes
  vars:
    pkg_name: not_exist
  tasks:
    - name: Install Apache
      apt: name={{ pkg_name }} state=latest
      debugger: on_failed
```

Ejecutamos ansible-playbook para comenzar con el proceso y vemos como al fallar la tarea el debugger, se habilita para que podamos resolver el error mediante los siete comandos mencionados anteriormente.

```
$ ansible-playbook -i inventory playbook.yml

PLAY [all]
*********************************************************************
*********************************************

TASK [apache : Install Apache]
*********************************************************************
```

```
**************************
fatal: [192.168.1.80]: FAILED! => {"ansible_facts":
{"discovered_interpreter_python": "/usr/bin/python"}, "changed": false,
"msg": "No package matching 'not_exist' is available"}
[192.168.1.80] TASK: apache : Install Apache (debug)>
```

Vamos a cambiar el valor de la variable *pkg_name* y volveremos a ejecutar la tarea:

```
[192.168.1.80] TASK: apache : Install Apache (debug)> p
task_vars["pkg_name"]
'not_exist'
[192.168.1.80] TASK: apache : Install Apache (debug)>
task_vars["pkg_name"] = "apache2"
[192.168.1.80] TASK: apache : Install Apache (debug)> p
task_vars["pkg_name"]
'apache2'
[192.168.1.80] TASK: apache : Install Apache (debug)> update_task
[192.168.1.80] TASK: apache : Install Apache (debug)> redo
changed: [192.168.1.80]

PLAY RECAP
*********************************************************************
********************************************
192.168.1.80               : ok=1    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

Como vemos, en tiempo de ejecución se ha conseguido resolver el error y que el playbook siga con su ejecución. Hay que tener en cuenta que los cambios que se hagan no modifican los ficheros, por lo que en este caso, habría que dirigirse al playbook y cambiar el valor de *pkg_name* por *apache2* porque si no la próxima vez que volvamos a lanzar el playbook tendremos que realizar la misma operación cuando el error ya debería estar resuelto.

Logs

Como las trazas pueden llegar a ser muy densas, Ansible por defecto las tiene deshabilitadas. Para habilitarlas, necesitamos definir dónde se quiere que se registren los logs. Se puede hacer mediante la configuración previa de la variable de entorno *ANSIBLE_LOG_PATH* o a través del fichero

ansible.cfg.

Para el primer caso, debemos ejecutar el siguiente comando:

```
export ANSIBLE_LOG_PATH=path/to/logfile
```

Para el segundo caso, se añaden las siguientes líneas al fichero *ansible.cfg*:

```
[defaults]
log_path=path/to/logfile
```

¿Y dónde se pone el fichero *ansible.cfg*? Ansible cuando va a ejecutar un playbook, busca previamente si existe el fichero de configuración en las siguientes rutas:

- ansible.cfg en el directorio donde se ejecuta ansible-playbook.
- ~/.ansible.cfg
- /etc/ansible/ansible.cfg

Una vez que ya se tiene definido dónde se quieren almacenar los logs, tan solo nos queda ejecutar el comando que arranca el playbook y añadir el argumento -v, -vvv o -vvvv (el nivel de trazas aumenta respectivamente)

```
#Log simple
$ ansible-playbook -v playbook.yml -i inventory

#Log verboso
$ ansible-playbook -vvv playbook.yml -i inventory

#Log con la activación del debug de conexión
$ ansible-playbook -vvvv playbook.yml -i inventory
```

Hay que tener en cuenta que cuando se pone -vvvv la salida es bastante grande. La recomendación es sólo usar este argumento en el caso de que se esté teniendo problemas de conexión con los nodos.

Buenas prácticas

Vamos a ver algunos consejos y buenas prácticas que te pueden ayudar a la hora de trabajar con Ansible, para crear un código declarativo más legible y mantenible durante mucho tiempo.

Organización de los archivos

La siguiente estructura es recomendable si tus entornos son muy parecidos y lo único que necesitas es definir los inventarios de cada entorno.

```
├──production          # inventario para los servidores de producción
├──staging             # inventario para los servidores de staging
└──group_vars/
   ├──group1.yml       # las variables para cada grupo
   └──group2.yml
└──host_vars/
   ├──hostname1.yml  # las variables para cada host(máquina)
   └──hostname2.yml
├──library/           # un módulo custom  (opcional)
├──module_utils/      # las utilidades de módulo custom (opcional)
├──filter_plugins/    # los custom plugins para filtros(opcional)
├──site.yml           # el playbook main
├──webservers.yml     # el playbook para webserver
├──dbservers.yml      # el playbook para el servidor de bbdd
└──roles/             # los roles
    └──common/      # el rol “common”
       └──tasks/
          └──main.yml # las tareas del rol “common”
       └──handlers/
          └──main.yml    #los handlers
       └──templates/   #
          └──ntp.conf.j2   #  las plantillas Jinja2
       └──files/            #
          ├──bar.txt  #los ficheros para copiar
          └──foo.sh #los scripts
       └──vars/
           └── main.yml # las variables del rol “common”
       └──defaults/         #
           └──main.yml # los valores por defecto de las variables
```

```
        └──meta/              #
            └──main.yml       # las dependencias del rol
        ├──library/           # los módulos custom
         ├──module_utils/      # las utilidades de los módulos custom
         └──lookup_plugins/   # los plugins de lookup
    ├──webtier/ # la misma estructura como la de common, pero para web
    ├──monitoring/
    └──fooapp/
```

También se puede usar otra estructura de carpetas. Puedes poner los ficheros de los inventarios dentro de las carpetas de cada entorno. Esta estructura tiene más sentido si tus entornos tienen pocas variables/grupos en común.

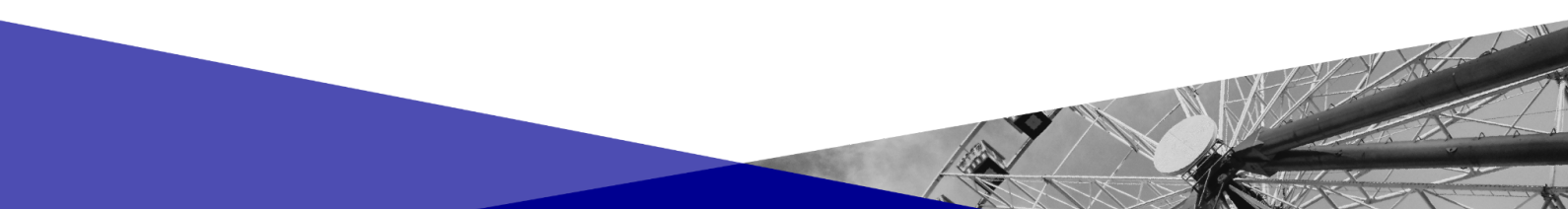
```
└──inventories/
   └──production/
      ├──hosts               # inventario de producción
      └──group_vars/
         ├──group1.yml       # las variables del grupo1 para prod
         ├──group2.yml       # las variables del grupo2 para prod
      └──host_vars/
         ├──hostname1.yml    # las variables del hostname1
         └──hostname2.yml
   └──staging/
      ├──hosts               # inventario de staging
      └──group_vars/
         ├──group1.yml       # las variables del grupo1 para staging
         ├──group2.yml       # las variables del grupo2 para staging
      └──host_vars/
         ├──stagehost1.yml
         └──stagehost2.yml
├──site.yml
├──webservers.yml
├──dbservers.yml
└──roles/
   ├──common/
   ├──webtier/
```

La complejidad es mucho más grande que en la primera estructura, pero te da más flexibilidad para definir las diferencias entre entornos.

Diferencias entre entornos

La mejor manera para distinguir diferentes entornos es usar diferentes ficheros de inventarios para cada entorno. Dentro de cada fichero se pueden crear los grupos dependiendo de su rol o ubicación.

```
# file: producción
[barcelona_webservers]
www-bar-1.example.com
www-bar-2.example.com

[sevilla_webservers]
www-sev-1.example.com
www-sev-2.example.com

[barcelona_dbservers]
db-bar-1.example.com
db-bar-2.example.com

[sevilla_dbservers]
db-sev-1.example.com

# webservers
[webservers:children]
barcelona_webservers
sevilla_webservers

# dbservers
[dbservers:children]
barcelona_dbservers
sevilla_dbservers

#los servidores de barcelona
[barcelona:children]
barcelona_webservers
barcelona_dbservers

#los servidores de sevilla
[boston:children]
sevilla_webservers
sevilla_dbservers
```



Además puedes definir las variables a nivel de cada grupo.

```
---
# file: group_vars/barcelona
ntp: ntp-barcelona.example.com
backup: backup-barcelona.example.com
```

O las variables por defecto para todos los grupos.

```
---
# file: group_vars/all
ntp: ntp-default.example.com
backup: backup-default.example.com
```

Lanzando tareas por lotes

Por defecto, Ansible intenta gestionar todos los hosts de forma paralela pero a veces es necesario definir la cantidad de hosts sobre los que Ansible puede operar al mismo tiempo, esto se hace para poder preservar la estabilidad del servicio y lanzar las actualizaciones paulatinamente.

Es precisamente por eso que existe la palabra clave *serial* que define la cantidad de hosts sobre los que Ansible puede operar a la vez:

```
- name: test play
  hosts: webservers
  serial: 2
  tasks:
  ...
```

También se puede usar el símbolo porcentaje con la palabra *serial* para indicar el porcentaje sobre todos los hosts que están en la play. Por ejemplo, *serial: "30%"*. O puedes especificar la cantidad de hosts en cada lote de ejecución explícitamente:

```
- name: test play
  hosts: webservers
  serial:
  - 1
  - 2
```

```
  - 5
```

Es posible mezclar los porcentajes con los números explícitos definiendo los lotes.

De forma predeterminada, Ansible continuará ejecutando tareas siempre que haya hosts en el lote que aún no hayan fallado. El tamaño del lote para una tarea está determinado por el parámetro *serial*. Si no se establece el número de *serial*, el tamaño del lote son todos los hosts especificados en el campo *hosts*. En algunas situaciones, puede ser deseable cancelar la ejecución cuando se haya alcanzado un cierto umbral de errores. Para lograr esto, puedes establecer un porcentaje máximo de errores en una play de la siguiente manera:

```
- hosts: webservers
  max_fail_percentage: 50
  serial: 10
```

En este caso si fallan más de la mitad de los hosts (por lo menos 6) la play se cancelará.

Mencionar el estado

El parámetro "state" es opcional para muchos módulos. Ya sea "state = present" o "state = absent", siempre es mejor especificar este parámetro explícitamente porque algunos módulos admiten estados adicionales.

Uso de los roles

Este consejo es muy importante. Nuestro sistema puede tener varios grupos. Tener grupos de hosts con nombres como *webservers* y *dbservers* te ayuda especificar claramente el rol de estos hosts. Los nombres de grupos que usan los roles permiten que los playbooks apunten a las máquinas según el rol y asignen variables específicas de cada rol utilizando el sistema de variables de grupo.

Diferentes sistemas operativos

Cuando se trata de un parámetro que es diferente para sistemas operativos

diferentes, una excelente manera de manejar esto es usando el módulo *group_by*.

```
---

 - name: recopilamos la info sobre los hosts
   hosts: all
   tasks:
     - name: Clasificando los host según su SO
       group_by:
         key: os_{{ ansible_facts['distribution'] }}

 # solo para Ubuntu...
 - hosts: os_Ubuntu
   gather_facts: False
   tasks:
     - # las tareas para Ubuntu
```

Así podemos crear un grupo dinámico de hosts que coincide con ciertos criterios, incluso si ese grupo no está definido en el archivo de inventario. Además podemos definir las variables según el grupo.

```
---
# file: group_vars/all
var1: 10
---
# file: group_vars/os_Ubuntu
var1: 42
```

Algunos consejos generales

Si un playbook tiene un directorio *./library* relativo a su archivo YAML, este directorio se puede usar para agregar módulos customizados de Ansible que estarán automáticamente en la ruta de módulos de Ansible. Es una manera de mantener juntos los módulos customizados que van con un playbook. Un ejemplo de una estructura de directorios así está al principio de esta sección.

Se recomienda el uso generoso de espacios para dividir las cosas y el uso de comentarios (que comienzan con "#").

Es posible omitir el 'name' de una tarea determinada, aunque se recomienda proporcionar una descripción de por qué se está haciendo algo. Este nombre se muestra cuando se ejecuta el playbook.

Intenta hacer las cosas lo más sencillas posible. No intentes utilizar todas las funciones de Ansible juntas. Utiliza solo lo que necesites. Si algo se ve complicado, probablemente lo sea, y siempre hay posibilidad de simplificar las tareas.

Utiliza el **control de versiones**. Mantén tus playbooks y los inventarios en git (u otro sistema de control de versiones). De esta manera, siempre tienes la historia de los cambios que describe cuándo y por qué cambiaste las tareas que automatizan tu infraestructura.

Por lo general, es más fácil usar *grep,* o herramientas similares, para encontrar variables en una configuración de Ansible. Dado que *Ansible-Vault* cifra sus variables, es mejor trabajar con un fichero que tenga todas las variables. Al ejecutar un playbook, Ansible encuentra las variables en el archivo no cifrado y todas las variables sensibles provienen del archivo cifrado. Una manera de hacerlo podría ser creando una carpeta *group_vars/nombre_del_grupo*. Dentro de esta carpeta, crear dos archivos *vars* y *vault*. Dentro del archivo *vars*, definir todas las variables necesarias, incluidas las sensibles. Después, copiar todas las variables sensibles en el archivo *vault* y agregar el prefijo *vault_* a estas variables. Debes asignar las variables en el archivo *vars* para que apunten a las variables *vault_* utilizando la sintaxis de *Jinja2* y asegurarte de que el archivo *vault* esté cifrado. De este modo, puedes tener las variables sensibles y no sensibles en el mismo lugar sin tener ningún límite.

Parte 6

Kubernetes

Introducción

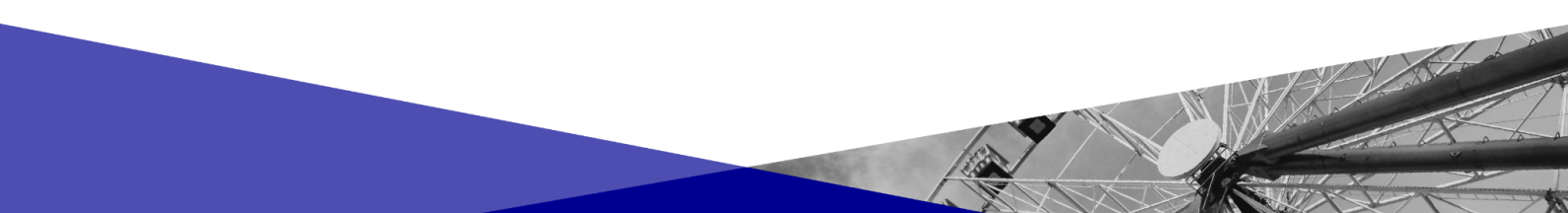
Los contenedores se han puesto muy de moda últimamente. Y se entiende su popularidad si vemos las ventajas que aportan frente a máquinas virtuales o servidores tradicionales. Los contenedores no tienen el sistema operativo completo, sino sólo las partes que necesitan, por lo tanto ocupan menos espacio, y el rendimiento es mayor ya que no ejecutan procesos innecesarios, sino únicamente aquellos que necesita nuestra aplicación.

Sin embargo, cuando se ejecutan distintas aplicaciones en un número elevado de contenedores, su gestión se complica. Para ello existen las herramientas de gestión y orquestación de contenedores, que son esenciales para reducir la complejidad que supone administrarlas manualmente.

La orquestación de contenedores consiste en la automatización de la supervisión, el desarrollo, el escalado y la configuración de los contenedores. Dentro de estas funciones, los orquestadores realizan tareas de afinidad, control del estado de los contenedores y reprogramación de estos, conmutación de nodos cuando se producen errores, proporción de redes para coordinar la comunicación de los contenedores, etc.

En esta línea, existen varias herramientas de gestión de contenedores. En esta sección vamos a profundizar en el software de Kubernetes, también conocido por su abreviatura K8s. Hemos seleccionado este sistema por las ventajas que supone respecto al resto, algunas de las cuales veremos a lo largo de esta guía, y el uso extendido que ha adquirido actualmente.

K8s permite ejecutar aplicaciones de forma escalable, extensible y portátil. Esto hace que se pueda mantener la coherencia entre entornos cuando se mueven las aplicaciones de máquinas locales a producción y garantizar el crecimiento controlado de los contenedores agregando servicios de seguridad y administración.



¿Qué es Kubernetes?

K8s es una plataforma de código abierto, extensible y portable, diseñada para la gestión de la carga de trabajo y servicios de una aplicación que se ejecute sobre contenedores. Se caracteriza por una amplia gama de herramientas y servicios, la posibilidad de autoescalar de forma rápida y la facilidad para automatizar y configurar declarativamente nuestros proyectos.

K8s podría encajar en varias definiciones. Algunas son:

- Plataforma de contenedores.
- Plataforma de microservicios.
- Plataforma portable de nube.

Sin embargo, la definición que más ajustada nos parece es la que define a K8s como una **plataforma de orquestación de contenedores** de código abierto.

A menudo, nos enfrentamos a aplicaciones con una arquitectura basada en

microservicios, repartidos entre varios hosts, no necesariamente en la misma red. Aquí es donde K8s destaca, facilitando el despliegue, la explotación y la actualización de aplicaciones. Es decir, orquesta la infraestructura, redes y almacenamiento de forma automática, lo que evita tener que hacerlo manualmente y los errores que eso conlleva.

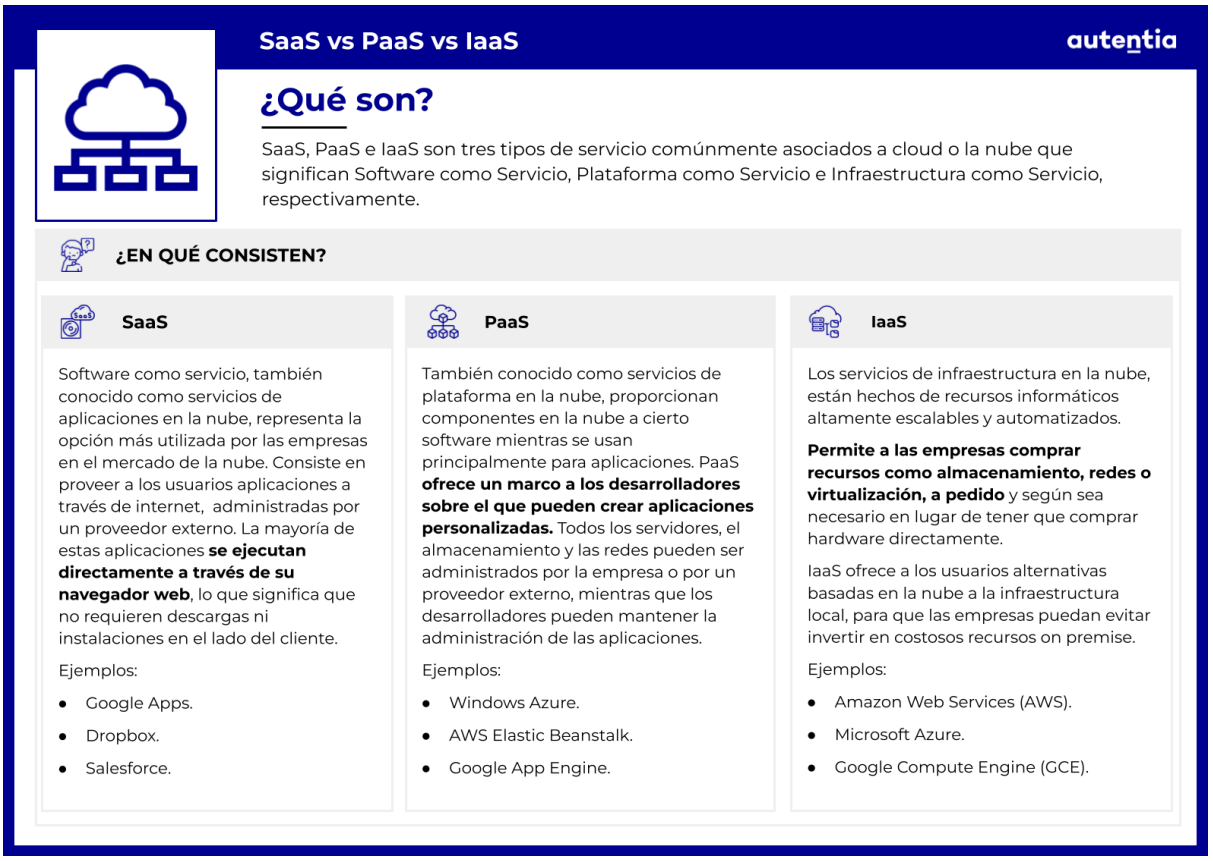
Para hacer esto, entra en juego la configuración declarativa. Se crea una capa de abstracción sobre los hosts involucrados en la aplicación, donde los equipos de desarrollo pueden declarar qué objetivos tiene la aplicación y la tecnología necesaria:

- Consumo de recursos por aplicación/host y límites de consumo de cada uno.
- Salud de los hosts y contenedores, reiniciando los mismos o repartiendo la carga según las necesidades de cómputo.
- Reparto uniforme de la carga de aplicación a la infraestructura de hosts.
- Equilibrio automático de las solicitudes de carga entre diferentes instancias de una aplicación.

Esta capa de abstracción, y la facilidad de uso que conlleva, ha provocado que K8s se haya establecido como el **estándar *de facto*** para la orquestación de contenedores.

¿Qué no es Kubernetes?

Es importante dejar claro que K8s no funciona como una Plataforma como Servicio (PaaS) convencional, ya que estas operan a nivel de hardware, mientras que K8s lo hace a nivel de contenedor. Por otro lado, también ofrece características comunes a las PaaS, como deployments, escalado, balanceo de carga y monitorización.



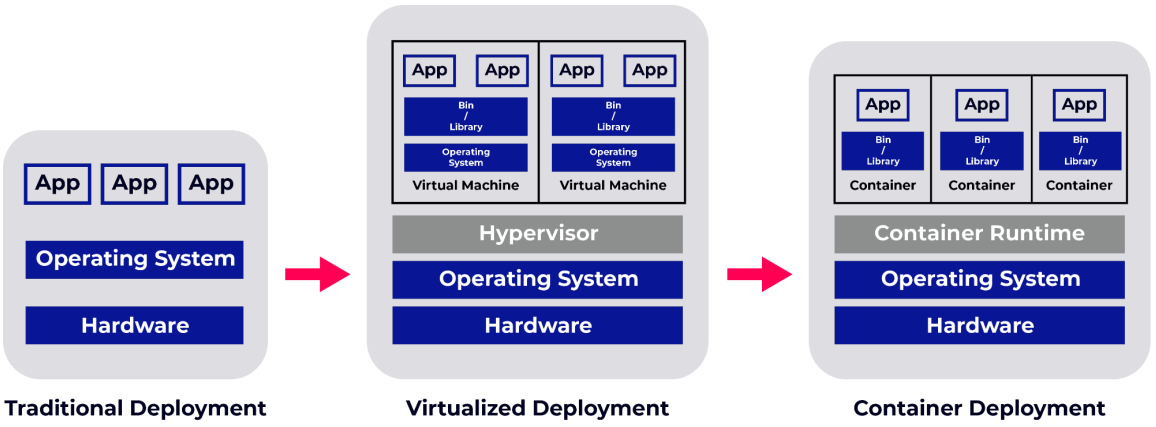
Dicho esto, podemos afirmar que K8s:

- No limita el tipo de aplicaciones que soporta. De hecho, si una aplicación puede correr en un contenedor la debería soportar.
- No interfiere en los flujos de integración, entrega y despliegue continuo.
- No provee servicios en capa de aplicación. Se pueden lanzar middlewares, bases de datos, cachés o sistemas de almacenamiento o acceder por medio de un mecanismo portable, pero no los provee K8s directamente.
- No obliga a usar ningún sistema o lenguaje de configuración. De hecho, ofrece una API declarativa que podemos usar con cualquier forma de especificación declarativa.
- No ofrece un sistema de mantenimiento exhaustivo, administración o corrección automática de errores.
- No es sólamente un sistema de orquestación. De hecho, elimina la necesidad de orquestar, ya que está compuesto por un conjunto de procesos de control independientes y combinables entre sí, que

permiten transferir el estado actual al deseado sin dar importancia al camino.

Como hemos visto, K8s simplifica mucho nuestro trabajo. No es una plataforma restrictiva e intenta darle mayor importancia a la elección del usuario y la flexibilidad del proyecto. No es monolítico y no requiere un control centralizado, lo que resulta en un **sistema robusto, poderoso, extensible y resiliente,** donde no se imponen sino que se proponen soluciones.

Algo de historia



autentia

Tradicionalmente, las aplicaciones se desplegaban instalándolas en los servidores de manera manual. De esta forma, solían surgir problemas de asignación de recursos, puesto que no había manera de definir el límite de algunos recursos que las aplicaciones podían utilizar, desaprovechando así los mismos. Una posible solución para este problema podría ser la utilización de un servidor diferente para cada aplicación. Pero esto supondría un coste bastante elevado, tanto económico como de mantenimiento.

En consecuencia, surgió la virtualización. Gracias a esto, podemos tener varias máquinas virtuales ejecutándose en un mismo servidor, aislando unas aplicaciones de otras y permitiendo, además, una mejor utilización de recursos. El problema con esta solución es que las máquinas virtuales son bastante pesadas, ya que se trata de máquinas completas que contienen su propio sistema operativo.

Por ello, aparecieron los contenedores. Son parecidos a las máquinas virtuales, pero relajan sus propiedades de aislamiento para compartir el sistema operativo. Por lo tanto, los contenedores se consideran ligeros, permitiendo así una mayor eficiencia y facilidad en el despliegue de aplicaciones. Sin embargo, cuando se desarrollan aplicaciones con un gran número de contenedores o con la necesidad de ser altamente escalables debido a la gran cantidad de usuarios, su gestión se hace más compleja.

Es aquí donde entra en juego K8s, que empezó siendo una solución de orquestación de contenedores internos de Google. Su nombre en clave original fue “Project Seven of Nine”, haciendo referencia al dron con el mismo nombre de la serie de ciencia ficción *Star Trek* (1966).

En 2014, Google liberó K8s como proyecto *open-source*, uniéndose a la comunidad grandes compañías como Microsoft, IBM, RedHat, Docker, etc. El término Kubernetes proviene del griego y su significado es timonel o piloto. La abreviatura que mencionamos antes se obtiene de reemplazar las ocho letras que contiene "ubernete" por el número 8.

K8s se basa en los más de 15 años de experiencia de Google corriendo aplicaciones en producción a gran escala, juntando además las mejores ideas y prácticas de la comunidad.

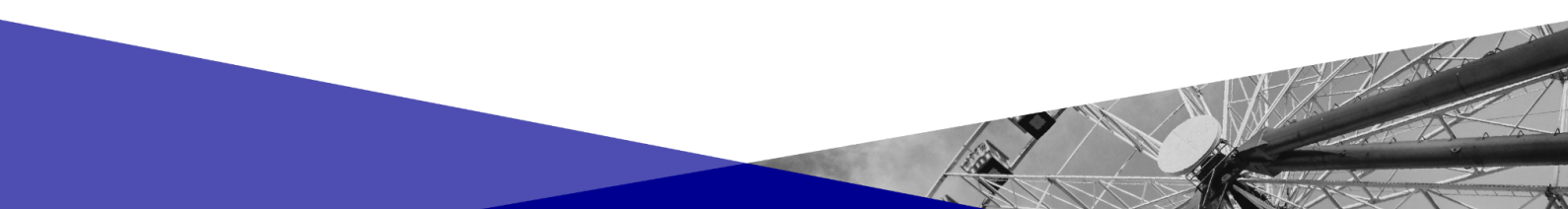
Clúster de Kubernetes

En esta sección veremos qué es un clúster de K8s, así como definiremos todos y cada uno de sus componentes y explicaremos los pasos a seguir para poder crear un clúster.

Cuando ejecutamos K8s, estamos ejecutando un clúster. Un clúster consta de un conjunto de máquinas denominadas **nodos** que ejecutan aplicaciones dentro de contenedores. Cada clúster debe tener al menos un **plano de control** y uno o varios nodos.

El plano de control es el encargado de mantener el estado que se desea alcanzar dentro del clúster. Además, se encarga de controlar las aplicaciones que se ejecutan en cada nodo.

Un nodo contiene una serie de **pods**, que son la unidad mínima de gestión dentro de K8s. Estos componen la carga de trabajo de la aplicación. Asimismo, dentro de un pod podemos encontrar un grupo de uno o más contenedores.



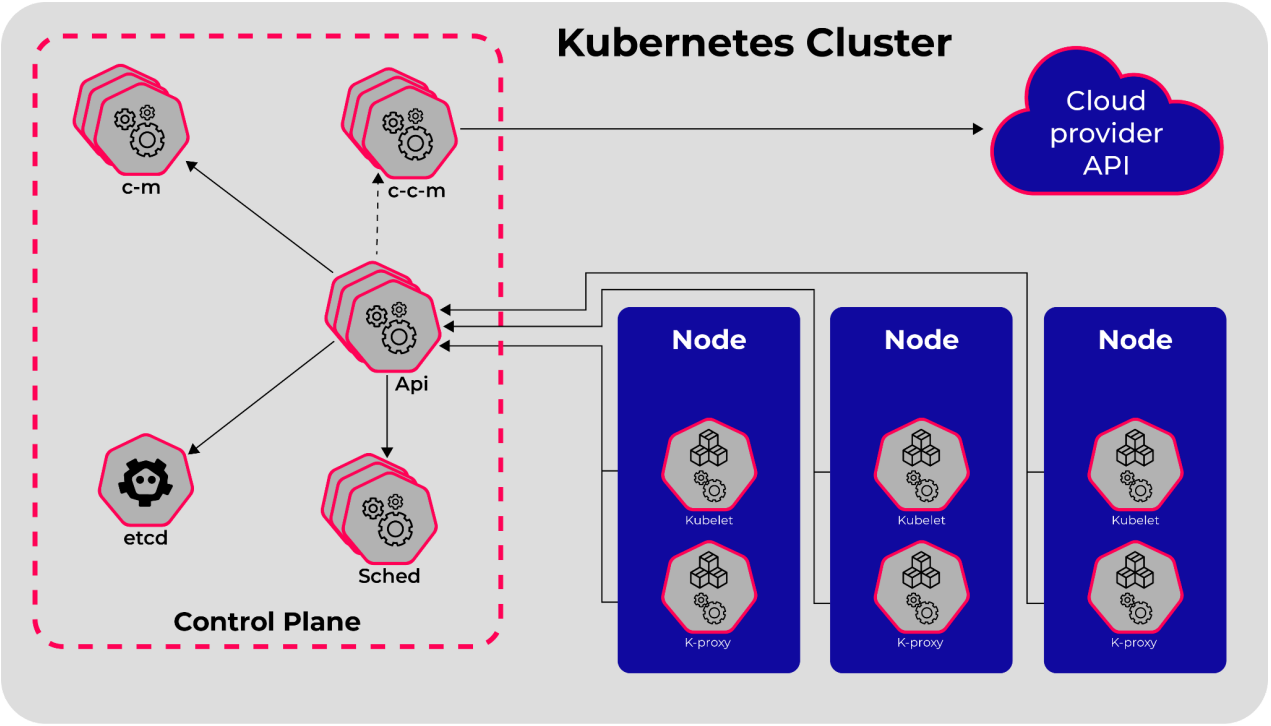
Los clústeres tienen un estado deseado que define las aplicaciones y/o las cargas de trabajo que deben ejecutarse, las imágenes que se usan para crear los contenedores necesarios, los recursos y **volúmenes** que deben estar disponibles y otras especificaciones de la configuración.

El estado deseado se define a través de archivos de configuración compuestos por manifiestos escritos en JSON o YAML. Estos indican el tipo de aplicación que se ejecutará y el número de réplicas necesarias para que el sistema funcione como debería.

Para interactuar con el clúster y establecer o modificar el estado deseado, podemos llevarlo a cabo desde la línea de comandos, usando ***kubectl***, o llamando a la **API** de K8s.

La gestión del clúster se realiza de forma automática para que coincida con dicho estado, aumentando, en caso de que fallara alguna, el número de réplicas de cada aplicación. De esta manera, se consigue una alta disponibilidad y tolerancia a fallos.

Componentes



A continuación describiremos los distintos componentes que son necesarios para operar con los clústeres de K8s.

Componentes del plano de control

Los componentes que forman parte del plano de control toman decisiones globales sobre el clúster y responden a eventos, como podría ser la creación de un nuevo pod cuando se incumple el número de réplicas deseado.

El plano de control está compuesto por:

- **Kube-apiserver**: El servidor de la API es el componente que se encarga de exponer la API de K8s. Este recibe las peticiones y actualiza apropiadamente el estado en etcd.
- **Etcd**: Almacén de datos persistente, consistente y distribuido de clave-valor que guarda toda la información del clúster.
- **Kube-scheduler**: Encargado de encontrar los pods que no están asignados en ningún nodo, y asignarlos a uno para poder ser ejecutados dentro de ese nodo.

- **Kube-controller-manager**: Componente que ejecuta los controladores de K8s. A pesar de que cada controlador sea un proceso independiente, todos se compilan y ejecutan en un mismo proceso, para así reducir la complejidad. Estos controladores incluyen:
    - **Controlador de nodos**: Responsable de detectar cuándo un nodo deja de funcionar y actuar en consecuencia.
    - **Controlador de replicación**: Encargado de mantener el número de pods correspondiente para cada controlador de replicación del sistema.
    - **Controlador de endpoints**: Construye la unión entre los servicios y los pods.
    - **Controladores de tokens y cuentas de servicio**: Crea las cuentas y tokens de acceso a la API necesarios para los nuevos namespaces.
- **Cloud-controller-manager**: Ejecuta los controladores que interactúan con proveedores de servicio de internet. Permite que el código de K8s y el del proveedor de la nube evolucionen por separado. Los controladores que pertenecen a este grupo son:
    - **Controlador de rutas**: Configura las rutas en la infraestructura cloud subyacente.
    - **Controlador de servicios**: Crea, actualiza y elimina los balanceadores de carga en la nube
    - **Controlador de volúmenes**: Orquesta la creación, conexión y montaje de volúmenes interactuando con el proveedor de la nube.

Componentes de nodo

Los componentes de nodo se ejecutan en cada nodo, manteniendo a los pods en funcionamiento. Estos son los siguientes:

- **Kubelet**: Agente que se encarga de asegurar que los contenedores están ejecutándose en un pod. Además, garantiza que dichos contenedores estén funcionando y en buen estado.

- **Kube-proxy**: Es un proxy de red que implementa parte del concepto de servicio de K8s. Mantiene reglas de red en los nodos que permiten la comunicación entre los pods desde sesiones de red dentro o fuera del clúster.

- **Runtime de contenedores**: Es el software responsable de ejecutar los contenedores, como puede ser Docker, containerd, cri-o, etc.

Addons

Los addons (complementos) usan recursos de K8s para implementar funciones de clúster. Algunos de estos complementos son:

- **DNS**: Todos los clústeres deben tener un DNS interno. El DNS de clúster es un servidor DNS que sirve registros DNS a los servicios de K8s.
- **Dashboard**: Es una interfaz web de propósito general para clústeres de K8s. Permite administrar y resolver posibles problemas que puedan ocurrir tanto con las aplicaciones como con el clúster.
- **Monitor de recursos de contenedores**: Almacena métricas generales sobre los contenedores en una base de datos centralizada, pudiendo acceder a dicha información a través de una interfaz.
- **Registros de clúster**: Almacena los logs de los contenedores de forma centralizada, proporcionando una interfaz para buscar y visualizarlos.

API de Kubernetes

En el punto anterior se han descrito los componentes que forman parte de un clúster, pero estos componentes interactúan entre sí a través de la API. Lanzamos **llamadas a la API** a través de kubectl, mediante comandos que nos permiten crear, consultar, actualizar y eliminar objetos.

La API actúa como una base para la configuración declarativa del sistema, permitiendo a su vez almacenar el estado de los recursos que la componen por medio de etcd.

Definiciones de API según la versión

K8s hace uso de OpenAPI para documentar los detalles de especificación de su API. OpenAPI Specification (OAS) define una descripción estándar, agnóstica del lenguaje de programación, de interfaces diseñadas para APIs

HTTP.

Sin embargo, OpenAPI no se empezó a utilizar hasta la versión 1.10 de K8s. Por tanto, debemos ser conscientes de qué versión utilizamos en nuestro proyecto y cómo afecta la misma a las llamadas a la API.

Versionado de la API

K8s intenta facilitar la eliminación de propiedades de un recurso y dar soporte a varias representaciones de recursos. Para lograr esto, soporta múltiples versiones de la API en diferentes rutas.

Para asegurar una visión clara y consistente de los recursos y el comportamiento del sistema, el versionado se produce a nivel de API y no a nivel de recursos o propiedades. Esto nos permite controlar el acceso a APIs experimentales o aquellas en proceso de extinción.

Cada versión de la API implica distintos niveles de estabilidad y soporte. Los niveles de versionado más comunes son:

- **alpha**: Recoge versiones que pueden contener errores, inestables y cuyo soporte podría ser eliminado sin previo aviso. Por ello, sólo es recomendable su uso en clústers efímeros y de prueba.
- **beta**: Recoge versiones más estables que el nivel alpha, en las que el código ha sido probado y está libre de errores. Sin embargo, estando en fase beta, la implementación, el esquema y/o la semántica de un objeto podría llegar a cambiar y volverse incompatible en el futuro. Por ello, **no es recomendable montar aplicaciones críticas de negocio sobre estos niveles** ya que, aunque son más estables, pueden presentar cambios incompatibles, sobre todo en aplicaciones con clústeres muy dependientes.
- **stable**: Recoge las versiones estables de la API, que no presentan futuras incompatibilidades y están libres de errores.

Para identificar cada versión podemos fijarnos en el nombre, donde las versiones alpha se representan como **vXalphaY**, las versiones beta como **vXbetaY** y las versiones estables como **vX**, donde X representa el número de la versión e Y representa la versión del nivel alpha o beta.

Pasos para crear un clúster

Para crear un clúster tenemos dos opciones:

- Podemos hacer uso de **kubeadm** para automatizar la instalación y configuración de componentes en los diferentes servidores o máquinas virtuales. Como mínimo necesitaremos una máquina que contendrá el plano de control (nodo maestro) y otra máquina que realizará la función de nodo trabajador, en la cual se encontrarán los pods ejecutando los contenedores necesarios. Además, como kubeadm no crea usuarios ni maneja la instalación de dependencias al nivel del sistema operativo ni la configuración de las distintas máquinas, necesitaremos de una herramienta de administración de configuración como es el caso de Ansible.
- O bien podemos usar **Minikube**, que es una herramienta que crea una máquina virtual y despliega un clúster sencillo que solo contiene un nodo.

En nuestro caso, para simplificar el proceso instalaremos Minikube para crear el clúster con el que realizaremos todas nuestras pruebas.

Para comprobar que lo hemos instalado correctamente, verificamos la versión:

```
minikube version
```

A continuación, vas a necesitar instalar la herramienta de línea de comandos **kubectl**. Esta herramienta permite ejecutar comandos contra K8s. Comprueba que la instalación es correcta con el siguiente comando:

```
kubectl version -o json
```

El siguiente paso es arrancar Minikube:

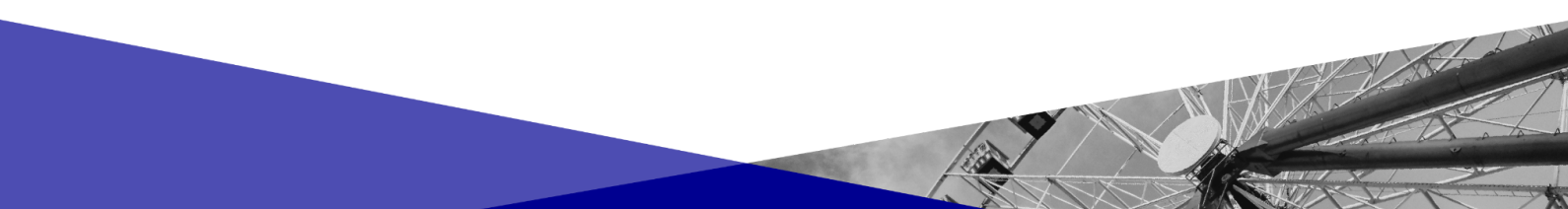
```
minikube start
```

Si es la primera vez que ejecutamos este comando, puede tardar unos minutos.

Los comandos más importantes de Minikube son:

- **minikube ssh**: para acceder a la máquina virtual.
- **minikube stop**: para parar el servidor.

- **minikube delete**: para borrar la máquina virtual.
- **minikube dashboard:** para abrir el panel de control de K8s en el navegador. Nosotros vamos a continuar por terminal, pero esta también es una buena opción.



Contenedores

El método de **contenerización** consiste en agrupar el código de una aplicación con sus archivos de configuración y dependencias necesarias para ejecutarse dentro de un paquete de software aislado. Su finalidad principal es poder implementar estas aplicaciones sin problemas en cualquier entorno debido a que se ejecutan dentro de un contenedor que esté donde esté, siempre es el mismo.

Los **contenedores** son una **forma de virtualización** diferente a las máquinas virtuales. Un contenedor virtualiza el sistema operativo de la máquina host y hace que la aplicación que se implementa en su interior perciba que tiene el sistema operativo incluido.

Los contenedores se utilizan en numerosos entornos, algunos de ellos son:

- Microservicios
- DevOps
- Cloud
- Migración de aplicaciones monolíticas

Lo que vamos a hacer con K8s es gestionar estos contenedores de forma conjunta. Cuando trabajamos con aplicaciones contenerizadas, hay que controlar varios contenedores funcionando a la vez y cuando el número de contenedores es alto, esto es una tarea compleja.

El uso más cotidiano de contenedores es a través de **Docker** y por supuesto que K8s puede orquestar contenedores de este tipo. Sin embargo, permite otras opciones:

- **CRI-O**: Es una implementación de CRI compatible con OCI. Se creó como alternativa a Docker Engine y permite iniciar pods de K8s y extraer imágenes.
- **Containerd**: Es una solución creada por Docker. Se encarga de gestionar la transferencia y almacenamiento de imágenes. En cuanto a los contenedores, puede crearlos, ejecutarlos y supervisarlos.

K8s permite declarar el estado de un entorno de contenedores mediante archivos YAML y posteriormente se encarga de mantener el estado, reiniciar la aplicación si falla, equilibrar la carga, escalado, etc.

Imágenes

Como vimos en el capítulo de Docker de esta guía, las imágenes son plantillas a partir de las cuales se crean contenedores.

Nombrado de imágenes

Las imágenes de contenedores normalmente reciben nombres como **pause**, **example/mycontainer** o **kube-apiserver**, según el alcance y la función de la imagen. Adicionalmente, los nombres de imagen pueden incluir un nombre de host de registro, **fictional.registry.example/myimage**, y cuando sea necesario un número de puerto asociado, **fictional.registry.example:5555/myimage**.

En caso de no especificar los campos adicionales mencionados, se asumen los valores por defecto, en el caso del registro se asume el registro público de Docker.

Hemos mencionado que las imágenes son inmutables y versionadas, por lo tanto, existirán varias imágenes referentes al mismo contenedor, pero representando **diferentes versiones** del mismo. Para diferenciar estas

imágenes podemos añadir una **etiqueta** de forma similar a cómo lo hacemos con Docker. Una etiqueta nos permite identificar diferentes versiones de una misma serie de imágenes asociadas a un contenedor.

En el caso de no especificar una etiqueta, K8s asume que la imagen es la última versión y le asigna la etiqueta **latest**. La convención de etiquetas nos permite utilizar letras, mayúsculas y minúsculas, dígitos, guiones, guiones bajos y puntos para designar la etiqueta de una imagen.

Actualización de imágenes

Como hemos explicado, un contenedor cambia en función de la imagen que contiene, y sabemos que las imágenes reciben actualizaciones y cambian su versión. Por ello, cuando kubelet lanza un contenedor, tiene que decidir qué imagen utilizar para lanzarlo, y para ello hace uso de la política de extracción de imágenes que el contenedor tenga asociada.

Para modificar la política de extracción de imágenes debemos cambiar el campo **imagePullPolicy** en el archivo **images.yaml**, al que podemos asignar los siguientes valores:

- **IfNotPresent**: Sólo se realiza la descarga de una imagen cuando no se encuentra en local.
- **Always**: Cada vez que se lanza el contenedor, kubelet busca en el registro del contenedor una **image digest** asociada al nombre del contenedor. Una *image digest* identifica una única versión de una imagen, de modo que K8s ejecuta el mismo código cada vez que ejecuta un contenedor con un nombre y un *digest* específico. Sin embargo, si kubelet ya tiene cacheada la imagen localmente, la usa sin volver a descargarla.
- **Never**: kubelet no descarga la imagen, sino que trata de ejecutar el contenedor con la imagen ya presente en él. Este concepto se conoce como **pre-pulled images** y es útil cuando necesitamos velocidad de ejecución o queremos evitar la autenticación contra un registro privado. Sin embargo, si la imagen no existe en el contenedor, el lanzamiento del mismo falla.

En muchas situaciones necesitamos que un **pod** (conjunto de contenedores activos, explicado en detalle más adelante) siempre utilice la misma imagen. Para esto, la documentación de K8s nos recomienda utilizar el *digest* de la imagen como etiqueta. Esta recomendación prevé situaciones

en las que el registro de imágenes cambia la versión asociada a una etiqueta, resultando en una mezcla de pods activos y recién desplegados, ejecutando diferentes versiones. La unicidad y el carácter descriptivo del *image digest* asegura que la etiqueta únicamente cambiará cuando el contenido de la imagen cambie, es decir, no habrá imágenes con la misma etiqueta que ejecuten código distinto.

Como dato de interés, existen controladores de admisión de terceros que "mutan" los pods creados para definir la carga de trabajo en función del *digest* en lugar de la etiqueta de la imagen.

Política de extracción de imágenes por defecto

Cuando se lanza un nuevo pod, el clúster le da valor al campo **imagePullPolicy** en función de ciertas condiciones:

- Si la etiqueta del contenedor es **:latest**, se le asigna un valor **Always**.
- Si no especificamos ninguna etiqueta, se le asigna **:latest** por defecto y caemos en el caso anterior.
- Si la etiqueta del contenedor es distinta de **:latest**, el campo toma el valor **IfNotPresent**.

Es importante mencionar que esta asignación automática del valor por defecto de **imagePullPolicy** sólo tiene lugar cuando se crea un objeto. Esto quiere decir que si cambiamos la etiqueta una vez creado, debemos modificar el campo **imagePullPolicy** manualmente.

Estado ImagePullBackOff

Hemos mencionado que existen casos en los que kubelet no puede descargar la imagen del contenedor, algunos motivos pueden ser:

- Un nombre inválido.
- Acceso a un registro privado sin autorización.
- Políticas de extracción de imágenes.

Cuando esto ocurre, el contenedor pasa a un estado de espera (Waiting), y como causa podremos identificar el estado **ImagePullBackOff**. Este estado indica un error en la descarga de la imagen y "BackOff" indica que K8s intentará seguir descargando la imagen en intervalos crecientes de espera hasta alcanzar un límite definido en compilación, 300 segundos.

Cargas de trabajo

La carga de trabajo o ***workload*** se define como la cantidad de procesamiento que tiene asignado un equipo informático en un momento concreto. Se utiliza como referencia para evaluar el rendimiento de un sistema informático. Para medirla, se tiene en cuenta el tiempo de respuesta del sistema y la cantidad de trabajo que se realiza en un periodo de tiempo.

Para nosotros, una carga de trabajo es una aplicación que se ejecuta en K8s. Puede estar formada por un único contenedor o varios funcionando juntos. Este conjunto de contenedores se ejecuta en un pod.

La gestión de los pods es compleja. Si se produce un fallo en un nodo donde se está ejecutando un pod, todos los pods de ese nodo van a fallar. K8s resuelve ese fallo creando un nuevo pod, aunque el nodo se recupere después.

Para agilizar esta gestión de los pods y que se pueda realizar en conjunto en lugar de individualmente, se pueden configurar controladores que permiten gestionar los pods de forma global.

Pods

Los pods son las **unidades más pequeñas** de K8s. Están compuestos por un grupo de uno o más contenedores que comparten almacenamiento, red y especificaciones de ejecución. Además, todos los contenedores que hay dentro de un mismo pod comparten contexto.

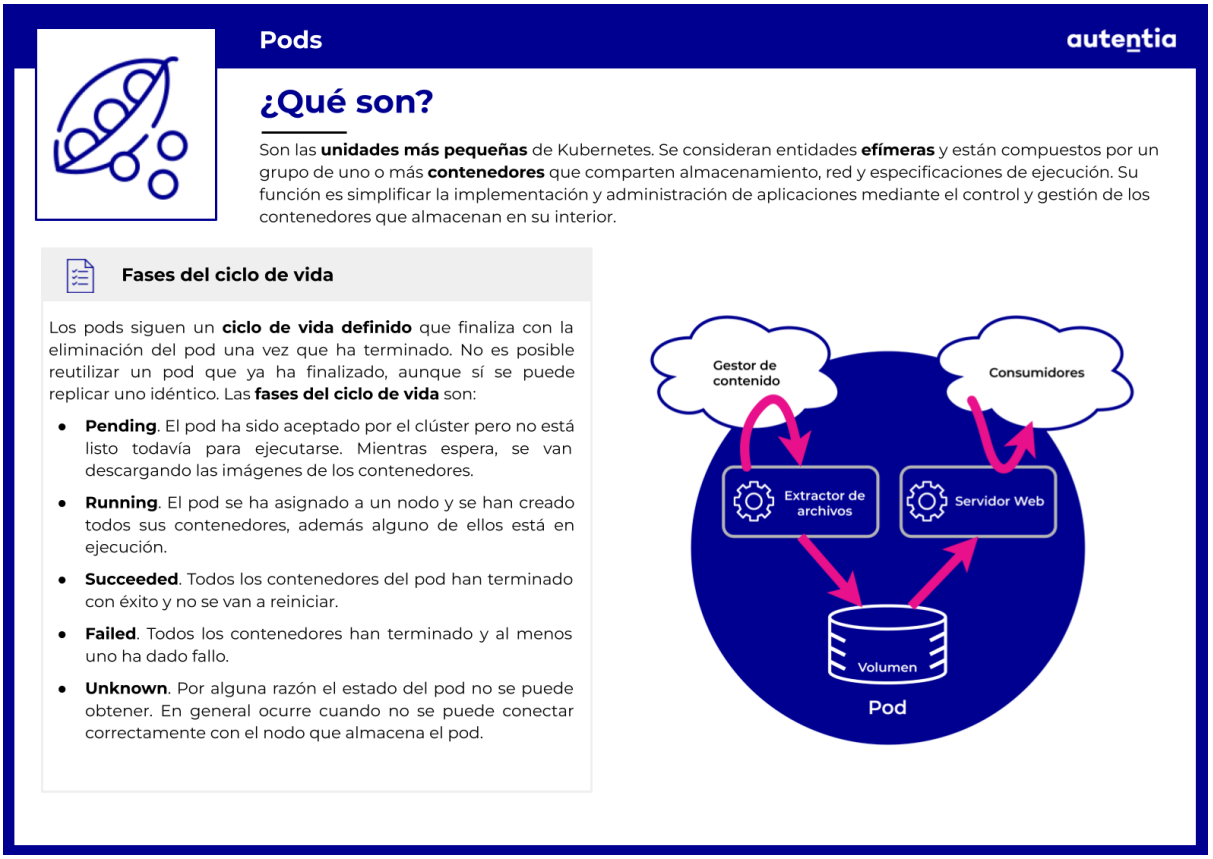
Un pod se puede definir como un “host lógico”, que es equivalente a ejecutar contenedores en una misma máquina física o virtual. De esta forma, los contenedores de un pod podrían comunicarse a través de localhost, ya que comparten dirección IP y puerto. Por el contrario, los contenedores que se ejecutan en pods distintos tienen direcciones IP distintas.

Los pods simplifican la implementación y la administración de aplicaciones consiguiendo un mayor nivel de abstracción. Sirven para desplegar, escalar horizontalmente y replicar contenedores.

Además de definir contenedores dentro de un pod, en el pod se especifican los volúmenes de almacenamiento compartido. Estos volúmenes permiten que los datos se mantengan a pesar de reinicios de contenedores y que se compartan entre los elementos internos del pod.

Los pods **no se tratan como entidades duraderas**, no sobreviven a errores de planificación, caídas de nodo y otros errores. En general, los **controladores** (que veremos más adelante) se encargan de proporcionar la **recuperación automática** del clúster, creando de nuevo los pods necesarios. Sin embargo, si la persistencia es imprescindible, existen otros controladores como *StatefulSet* que proporciona soporte para persistir el estado de los pods.

El uso principal de los pods es la **replicación**, que a su vez es una de las funciones por las que se conoce K8s. La replicación consiste en controlar la sobrecarga de los pods y, cuando sea demasiado alta, K8s lo replica y lo implementa en un clúster de forma automática.



Crear un pod

En este punto vamos a crear un pod. Para ello, tenemos que tener primero un clúster de K8s funcionando. Si has seguido los pasos anteriores en los que creamos un clúster, deberías tener Minikube instalado y funcionando.

En primer lugar, debemos definir en un manifiesto YAML la especificación del pod que queremos crear:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

En este archivo hemos definido un pod simple que tiene un solo contenedor ejecutando la imagen “nginx:1.14.2”. Los campos necesarios para conseguir esto son:

- **.kind**: Indicamos que es un pod (Pod).
- **.metadata.name**: Nombre que recibirá el pod al ser creado.
- **.spec.containers**: Aquí especificamos los contenedores y sus características que se ejecutarán dentro del pod:
    - **.name**: Nombre del contenedor.
    - **.image**: Imagen que ejecutará el contenedor.
    - **.ports.contianerPort**: Puerto en el que estará corriendo el contenedor.

Una vez definido el archivo YAML, vamos a crear el pod con el siguiente comando:

```
kubectl apply -f https://k8s.io/examples/pods/simple-pod.yaml
```

Nota: Al tratarse de un ejemplo oficial de K8s, el comando anterior descarga el manifiesto de internet, por lo que no es necesario crear el

archivo y guardarlo en nuestro dispositivo.

Para comprobar que se ha creado correctamente, podemos listar todos los pods con el comando:

```
kubectl get pods
```

```
jcmoreno@localhost:~$ kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          29s
```

También podemos obtener información más detallada del pod con el comando ***describe***:

```
kubectl describe pod nginx
```

Si bien hemos visto cómo definir y crear un pod de forma manual, esto no se recomienda hacer así, sino a través de un controlador que gestione los pods, lo cual veremos más adelante. Esto es así ya que, en caso de pérdida del pod o caída del nodo en el que se encontraba, el contenedor o contenedores que estuvieran ejecutándose dentro del pod serían destruidos también. Sin un controlador que se encargue de crear de nuevo el pod, los recursos que están haciendo uso de él se quedarán sin servicio.

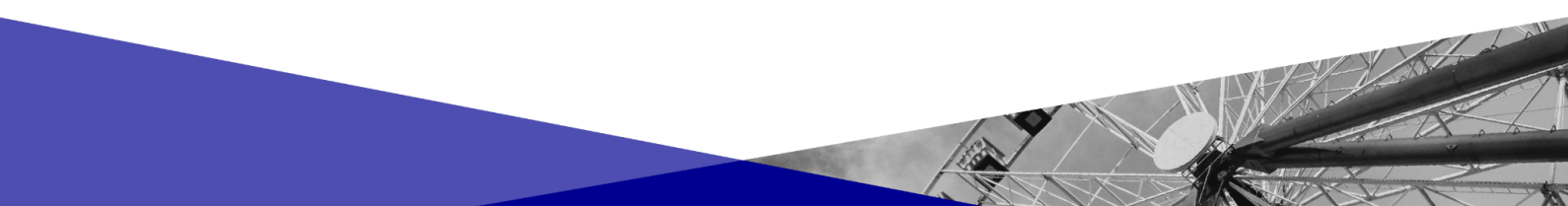
Ciclo de vida y estados de un pod

Los pods siguen un ciclo de vida definido que comienza en la fase de ***pending***, continúa como ***running*** y después puede pasar por las fases de ***succeeded*** o ***failed***. Cuando no se logra obtener el pod porque hay algún error de comunicación con el nodo donde debería estar el pod, se produce la fase de ***unknown***.

Cuando un Pod está en la fase de *running*, kubelet puede reiniciar sus contenedores si fallan para recuperar el pod. El estado del pod depende de ciertas condiciones que se dan durante su ejecución. Además, se pueden personalizar más condiciones en su configuración.

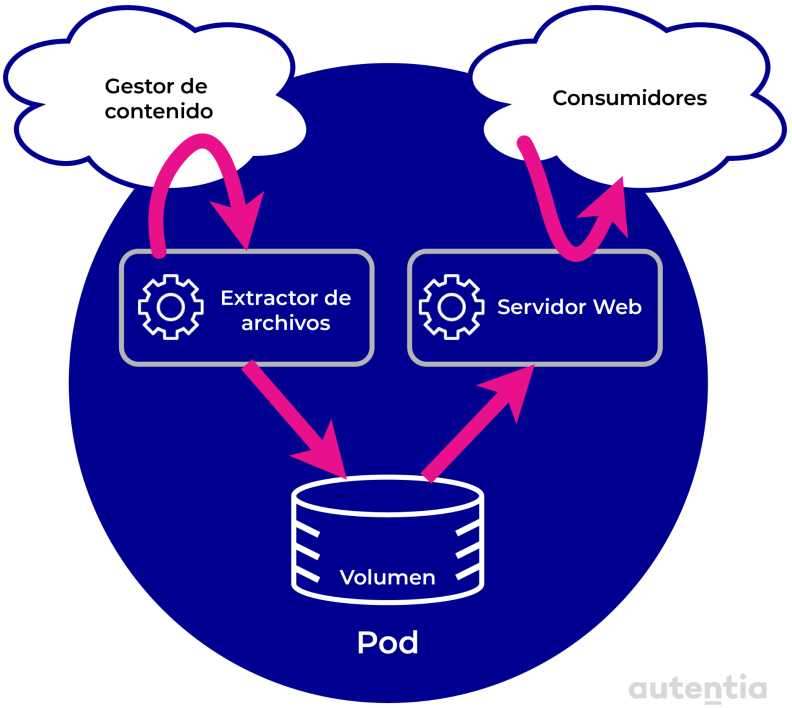
Los pods se consideran **entidades efímeras**. Se crean, se les asigna un identificador único y se asignan a nodos donde permanecen hasta que el nodo se detiene o termina. Si un nodo muere, los pods que tiene asignados se eliminan después de un periodo de tiempo.

Es importante recordar que los pods solo pueden pertenecer a un nodo durante su ciclo de vida. Se puede crear un pod idéntico pero con un



identificador distinto y asignarlo a otro nodo.

Este es un ejemplo de un pod con más de un contenedor que contiene un extractor de archivos y un servidor web. Además, los contenedores utilizan un volumen persistente de datos compartidos.



Aparte del ciclo de vida y las fases, los pods mantienen el estado de los contenedores que tienen dentro. Cuando se asigna un pod a un nodo, automáticamente se crean los contenedores de dicho pod y estos contenedores pueden tener tres estados:

1. **Waiting**: este estado se da cuando el contenedor está iniciándose. Por ejemplo, puede estar cargando la imagen, datos secretos, etc.
2. **Running**: indica que el contenedor está ejecutándose sin problemas.
3. **Terminated**: se produce cuando el contenedor ha terminado su ejecución o por algún motivo ha fallado. Con kubectl se puede comprobar por qué motivo ha terminado, el código de salida y el tiempo de ejecución.

En cuanto a la fase final del ciclo de vida de un pod, es importante tener en cuenta la forma en la que finaliza. Hay que configurar de forma adecuada que los pods terminen un tiempo después de que dejen de ser útiles en lugar de pararlos con una señal de *KILL* de forma forzosa.

Normalmente, el contenedor manda una señal de *TERM* para indicar que ha terminado y cuando ya no queda ningún contenedor dentro del pod en ejecución, el API server se encarga de borrar el pod.

Para los pods que han terminado, la API dispone de un **recolector de basura** que **elimina todos los que estén en fase *succeeded o failed*** cuando estos superan un número determinado, para evitar así la sobrecarga que supone tener almacenados pods que no van a usarse más.

Controladores

Los controladores de K8s son bucles de control que vigilan el estado de tu clúster, haciendo o solicitando cambios cuando sea necesario. Un bucle de control se entiende como un bucle infinito que regula el estado de un sistema. Esto quiere decir que cada controlador tratará de avanzar el estado actual del clúster hacia el estado objetivo o deseado.

Un controlador rastrea al menos un tipo de recurso, siendo el responsable de hacer que el estado actual se acerque al deseado, ya sea por cambios directos o por interacciones con la API.

Los controladores **built-in** se encargan de gestionar el estado interactuando con el servidor API del clúster. Pongamos como ejemplo un *Job Controller*, que explicaremos en detalle más adelante. Cuando el controlador observa una nueva tarea, se asegura de que exista el número necesario de pods para que esa tarea pueda ser completada. En este instante, el estado deseado es completar la tarea, por lo que el controlador envía una petición a la API para acercar el estado actual al deseado, creando los pods necesarios.

Por otro lado, existen controladores que necesitan hacer cambios externos al clúster. En este caso, los controladores hacen uso de la API para conocer el estado deseado y para reportar el estado actual, pero los cambios en el estado del sistema los hace mediante interacción directa con otros sistemas externos. Un ejemplo puede ser un bucle de control que asegure la existencia de suficientes nodos en nuestro clúster, en el que el controlador se comunica con un sistema externo al clúster, que le asigna nuevos nodos.

K8s ha sido diseñado para usar múltiples controladores, cada uno encargado de un aspecto específico del estado del clúster. El diseño más común consiste en un controlador que usa un tipo de recurso concreto

como su estado deseado, mientras gestiona varios tipos distintos de recursos para alcanzar ese estado. Siguiendo el ejemplo anterior, un *Job controller* monitoriza objetos de tipo *Job* para alcanzar un estado de tarea terminada, y para ello gestiona objetos de tipo *Job* y *Pod*.

Si nos paramos a pensar en cómo funciona K8s, podemos plantear una situación en la que los controladores entren en conflicto. Por ejemplo, dos controladores que crean y liberan pods podrían pisar su trabajo mutuamente. Sin embargo, esta posibilidad está contemplada y controlada gracias a etiquetas de los pods que los controladores pueden usar para diferenciarlos, aplicando cambios sólamente sobre los pods que tengan asignados.

ReplicaSet

ReplicaSet es un controlador que mantiene un conjunto estable de réplicas de pods que siempre están en ejecución. Esto garantiza la disponibilidad de un número concreto de pods iguales en todo momento.

Cuando definimos un ***ReplicaSet controller*** debemos tener en cuenta tres

campos principales:

1. **Selector**, que identifica a los pods que puede adquirir.
2. **Número de réplicas objetivo**, que representa el número de pods que debe gestionar.
3. **Plantilla de pod**, que define los datos de los nuevos pods a crear.

De este modo, queda claro que el objetivo de un *ReplicaSet* es crear y eliminar pods hasta alcanzar un objetivo, usando la plantilla para generar los nuevos pods.

El selector hace uso del enlace que existe entre un *ReplicaSet* y sus pods, el campo **metadata.ownerReferences** de cada pod. Este campo indica qué recurso es su propietario, y permite al selector adquirir aquellos pods que no tengan *OwnerReference* o donde el valor de este campo no se corresponde con un controlador. Además, el enlace de propiedad entre *ReplicaSet* y sus pods, le permite conocer el estado de los mismos y actuar en consecuencia.

Cuándo usar ReplicaSet y alternativas

ReplicaSet no es la única opción que nos ofrece K8s para gestionar la creación de réplicas estables de pods. Por eso, es importante saber cuándo merece la pena usar un *ReplicaSet* y cuándo es mejor buscar una alternativa.

En la práctica, no necesitarás manipular los objetos *ReplicaSet*. En su lugar, puedes hacer uso del controlador *Deployment*, que es un controlador de más alto nivel cuyo objetivo es gestionar *ReplicaSets* y llevar a cabo actualizaciones de los pods de forma declarativa. La única situación en la que es recomendable usar *ReplicaSet* es cuando se necesite una orquestación personalizada de actualización o no se necesite actualizar. En cualquier otro caso, usa *Deployment* y define tus requisitos en las sección *spec*.

Además del controlador mencionado, existen casos especiales en los que otros controladores se adaptan mejor a las necesidades de la aplicación:

- **Job** resulta muy útil para pods involucrados en trabajos por lotes, donde se espera que terminen por sí mismos.
- **DaemonSet** es útil para pods asociados a funcionalidad a nivel de

servidor, como monitorización o logging. El ciclo de vida de estos pods es distinto a los creados con *ReplicaSet*, su ciclo de vida comienzan antes que los pods dentro del servidor y es seguro que terminen cuando el servidor esté listo para reiniciar y apagar.

Sin embargo, también existen alternativas que son menos recomendadas en las que podamos disponer de, al menos, un *ReplicaSet*:

- **Pods simples**: Se debe evitar su uso siempre que tengamos acceso a *ReplicaSets*, ya que estos se encargan de mantener la salud de los pods que adquieren. Su función es reiniciarlos o sustituirlos en caso de fallo.
- **ReplicationController**: Es el antecesor de *ReplicaSet*, los dos tienen el mismo objetivo y trabajan de forma similar. Sin embargo, *ReplicationController* no soporta el selector basado en conjuntos, una funcionalidad muy útil de *ReplicaSet*.

Escribir un manifiesto de ReplicaSet

Si consideramos necesario utilizar *ReplicaSet* en lugar de *Deployment*, debemos saber cómo definir uno. Veamos un ejemplo de manifiesto de *ReplicaSet*:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modifica las réplicas según tu caso de uso
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
```

```
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```

Supongamos que el archivo anterior se llama **frontend.yaml** y está contenido en el paquete **controllers**. Podemos observar que tienen los mismos campos básicos que todos los objetos de la API de K8s: **apiVersion**, **kind**, y **metadata**. Analizando estos campos podemos diferenciar manifiestos según el campo **kind**, que tendrá valor *ReplicaSet* o *Pod* en función de si la creación se hace mediante plantilla o no, y un campo **spec**, en el que definiremos las características del *ReplicaSet*:

- **Réplicas**: El *ReplicaSet* creará o eliminará sus pods hasta alcanzar el número especificado en el campo **.spec.replicas**.
- **Selector de Pod**: Como hemos explicado, se trata de un selector de etiquetas. En este campo podemos especificar las etiquetas que se usan para identificar pods potenciales a adquirir. Las etiquetas de **.spec.selector.matchLabels** deben coincidir con **.spec.template.metadata.labels**, de lo contrario, la API rechazará el objeto.
- **Plantilla de Pod**: La plantilla de las réplicas se define en el campo **.spec.template**, y es necesario definir obligatoriamente etiquetas en el campo **.spec.template.metadata.labels**. Una vez cumplido ese requisito, podemos especificar los contenedores del pod en el campo **.spec.containers**.

Cabe destacar que hay campos con valores por defecto, como **.spec.replicas**, inicializado a 1, y **.spec.template.spec.restartPolicy**, que indica la política de reinicio y solo puede tener valor Always. Por último, es importante comprender que 2 *ReplicaSets* que especifican el mismo **.spec.selector** pero tienen **.spec.template.metadata.labels** y **.spec.template.labels** distintos, ignoran los pods creados por el otro.

Ejemplos de uso

Si entramos a la parte práctica de *ReplicaSet*, podemos trabajar con ellos a través de los siguientes comandos de **kubectl**:

Creación de un *ReplicaSet*

Una vez definido el manifiesto, en nuestro caso el archivo **frontend.yaml**, podemos lanzar el siguiente comando para crear el *ReplicaSet* y todos sus

Pods correspondientes:

```
kubectl apply -f {file-path}/frontend.yaml
```

Cabe destacar que el comando ***apply*** sirve tanto para crear el recurso, en caso de que no exista, como para actualizarlo si se produce algún cambio en el manifiesto. También podemos usar el comando ***create*** para crear el objeto. Sin embargo, este lanzaría un error en caso de que ya existiera. Además, si usamos *create* para crear el recurso a partir de un archivo YAML, tendremos que especificar el flag "--save-config" para poder actualizar el recurso en un futuro con el comando *apply*. En caso contrario, dará error al intentar actualizarlo.

Listado de ReplicaSets

Una vez creado nuestro *ReplicaSet* necesitamos una forma de ver un resumen de su estado y todos los ReplicaSet del cluster. Para ello ejecutamos:

```
kubectl get replicaset
```

Estado de un *ReplicaSet*

Para comprobar el estado del *ReplicaSet* con mayor detalle, ejecutamos:

```
kubectl describe replicaset/frontend
```

Eliminación de un *ReplicaSet*

Para eliminar un *ReplicaSet* y todos sus pods escribiremos en la consola:

```
kubectl delete replicaset {nombreDelReplicaSet}
```

Por último, existen situaciones en las que queremos eliminar un *ReplicaSet* sin afectar a sus nodos, en ese caso escribiremos el parámetro **--cascade=false** en consola:

```
kubectl delete replicaset --cascade=false {nombreDelReplicaSet}
```

Este caso de uso es útil si queremos reemplazar un *ReplicaSet*. Mientras el viejo y el nuevo **.spec.selector** coincidan, el nuevo *ReplicaSet* adoptará los Pods que han quedado huérfanos.

Escalado de *ReplicaSet*

Para escalar un *ReplicaSet* basta con actualizar el valor del campo **.spec.replicas**, ya que el controlador se encarga de crear y eliminar los pods necesarios para alcanzar dicho número.

Alternativamente, podemos declarar el *ReplicaSet* como blanco de un **Horizontal Pod Autoscaler** (HPA) de modo que el *ReplicaSet* se escala automáticamente dependiendo del uso de CPU de los pods implicados.

Para configurar un HPA debemos crear un manifiesto parecido al siguiente:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: frontend-scaler
spec:
  scaleTargetRef:
    kind: ReplicaSet
    name: frontend
  minReplicas: 3
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Al que denominaremos **hpa-rs.yaml** para futuros ejemplos de comandos. En este manifiesto podemos especificar números mínimos y máximos de réplicas que permitimos escalar, así como el porcentaje de uso de CPU que deseamos.

Además de los campos mencionados, en el campo **.spec.scaleTargetRef** debemos especificar el tipo y el nombre del objeto a escalar, en nuestro caso se llama frontend y es un *ReplicaSet*.

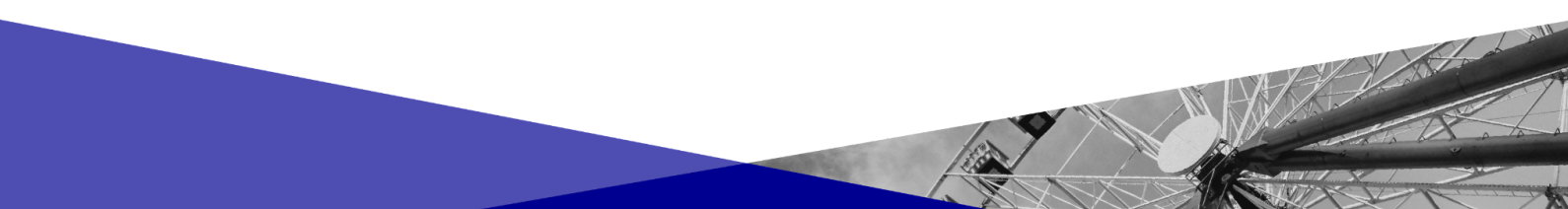
Para crear un HPA a partir del manifiesto anterior, ejecutaremos:

```
kubectl apply -f {file-path}/hpa-rs.yaml
```

Por otro lado, podemos alcanzar el mismo objetivo con el comando:

```
kubectl autoscale rs frontend --min=3 --max=10
```

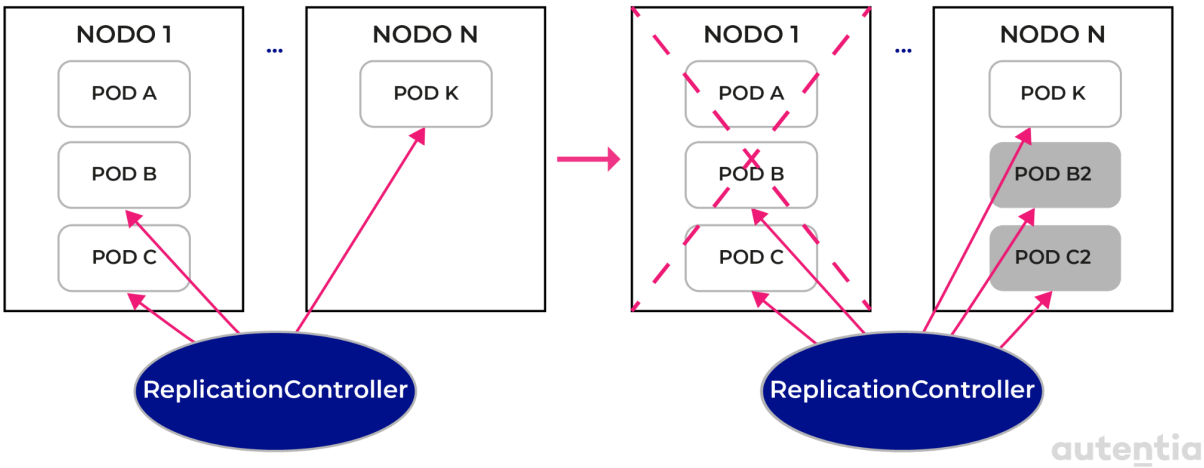
Donde **rs** es una abreviatura de **replicaset**, que podemos usar en cualquiera de los ejemplos de ejecución anteriores.



ReplicationController

Este controlador se encarga del manejo del **ciclo de vida** de los pods. Su función principal es asegurar que el número de réplicas de un pod que están en ejecución en un momento determinado es el deseado. Además, en caso de que este número no sea el correcto, tiene la capacidad de levantar o parar los pods en otros nodos.

Un ejemplo de uso sería el siguiente:



Como podemos ver en el diagrama, tenemos un conjunto de nodos. En el Nodo 1 están los pods A, B y C. Los pods B y C están controlados por el *ReplicationController*, mientras que el A no lo está.

Posteriormente se produce un error en el Nodo 1 que hace que deje de funcionar. En este momento el *ReplicationController* se encarga de levantar en otro nodo (Nodo N) los pods que tenía bajo su control, en este caso el B y el C. Gracias a esto, tenemos una nueva instancia de los pods que estaban vigilados por el controlador. Sin embargo, el pod que estaba creado manualmente muere junto al nodo que lo contenía.

Como se ha mencionado en el punto anterior donde se habla del controlador ReplicaSet, este controlador es su antecesor y por tanto, sus características son más limitadas.

La **diferencia** principal reside en que el *ReplicationController* no soporta el selector basado en conjuntos. Para entender mejor a qué nos referimos con esto, vamos a ver un ejemplo:

- ***ReplicationController***: Podemos seleccionar todos los recursos con el mismo valor para un solo identificador. Por ejemplo: “Entorno =

producción”.

- ***ReplicaSet***: Sin embargo, con este controlador podemos seleccionar todos los recursos con el mismo identificador que tengan varios valores concretos. Por ejemplo: “Entorno in (producción, test)”

Hasta ahora hemos visto que el uso de un controlador de replicación resulta necesario evitando la creación de pods directamente. De entre los controladores de replicación conviene elegir ReplicaSets sobre ReplicationController, ya que el primero viene a sustituir al segundo y aporta mayor versatilidad a la hora de seleccionar pods. Sin embargo, los controladores de replicación siguen teniendo algunas limitaciones que podemos superar con un controlador de más alto nivel, el controlador *Deployment*.

Deployment

Como ya hemos mencionado, el controlador *Deployment* abstrae al desarrollo de los *ReplicaSets* proporcionando actualizaciones declarativas tanto para *ReplicaSets* como para pods. De esta forma, disponemos de un controlador que gestiona los *ReplicaSets* cambiando el estado actual de forma controlada, creando o eliminando *ReplicaSets* y adoptando recursos en nuevos *Deployments*.

Estado de un *Deployment*

Los estados de un *Deployment* son los siguientes:

- **Progresando:** El *Deployment* está desplegando un nuevo *ReplicaSet*. Se alcanza cuando:
    - Crea un nuevo *ReplicaSet*.
    - Escala el *ReplicaSet* más reciente.
    - Reduce el *ReplicaSet* más antiguo.
    - Hay nuevos pods disponibles para ser adquiridos.
- **Completo:** El *ReplicaSet* se ha desplegado con éxito. Se alcanza sólo cuando todas las siguientes características se cumplen:
    - Todas las réplicas asociadas han sido actualizadas a la versión indicada.

    - Todas las réplicas asociadas están disponibles.
    - El *Deployment* no tiene réplicas viejas en ejecución.
- **Fallido:** Ha ocurrido un error al desplegar el *ReplicaSet*, dejando el *Deployment* bloqueado si se cumplen algunas de estas condiciones:
    - Cuota insuficiente.
    - Fallos en la prueba de despliegue listo.
    - Errores en la descarga de imágenes.
    - Permisos insuficientes.
    - Rangos de límites de recursos.
    - Mala configuración del motor de ejecución.

El estado fallido se puede detectar mediante la especificación de un parámetro **.spec.progressDeadlineSeconds**, que denota el número de segundos que el controlador de *Deployment* debe esperar antes de cambiar el estado a fallido. Cambiando este parámetro podemos ampliar o reducir, según nuestras necesidades, el rango de detección de fallo del *Deployment*. Presentamos un ejemplo en el que asignaremos un intervalo de 5 minutos al archivo **nginx-deployment**:

```
kubectl patch {filePath}/nginx-deployment -p \
'{"spec":{"progressDeadlineSeconds":300}}'
```

Es importante destacar que usar este parámetro de vencimiento no afecta al pausar *Deployments* en medio de un despliegue y reanudarlos tiempo más tarde. Una vez se ha excedido el vencimiento, el controlador añade un objeto *DeploymentCondition* al campo **.status.conditions** del *Deployment*. Ese objeto tendrá un aspecto parecido a este:

```
{
  "Type": "Progressing",
  "Status": False,
  "Reason": "ProgressDeadlineExceeded"
}
```

Que podemos analizar mediante el comando:

```
kubectl describe deployment nginx-deployment
```

Obteniendo una salida como la siguiente:

```
<...>
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     False   ProgressDeadlineExceeded
<...>
```

Ejemplos de uso

Creación de un *Deployment*

Para crear un *Deployment* utilizaremos el comando **apply**, para el cual tendremos que definir el objeto en un archivo YAML, al que nos referiremos como **nginx-deployment.yaml** en este ejemplo:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Como podemos observar en el campo **.metadata.name**, vamos a crear un deployment llamado "nginx-deployment". Explicaremos los campos más relevantes. Presta atención en lo similar que es este objeto a un objeto *ReplicaSet*.

La diferencia más importante entre la especificación de estos objetos es el campo **.kind**. Este campo permite a **kubectl** reconocer qué tipo de objeto debe crear. El resto de campos típicos de un controlador (réplicas, selector y plantilla de pod) son idénticos a los objetos que abstrae.

Por último, para llevar a cabo la creación del objeto, ejecutamos el siguiente comando:

```
kubectl apply -f {file-path}/nginx-deployment.yaml
```

Cuyo éxito podemos comprobar listando los deployments.

Listado de *Deployments*

A menudo necesitaremos listar los deployments existentes y sus detalles en nuestro sistema, para ello ejecutamos el comando:

```
kubectl get deployments
```

Recordando el ejemplo anterior, si la creación ha sido exitosa y listamos el deployment "nginx-deployment", vemos una salida parecida a esta:

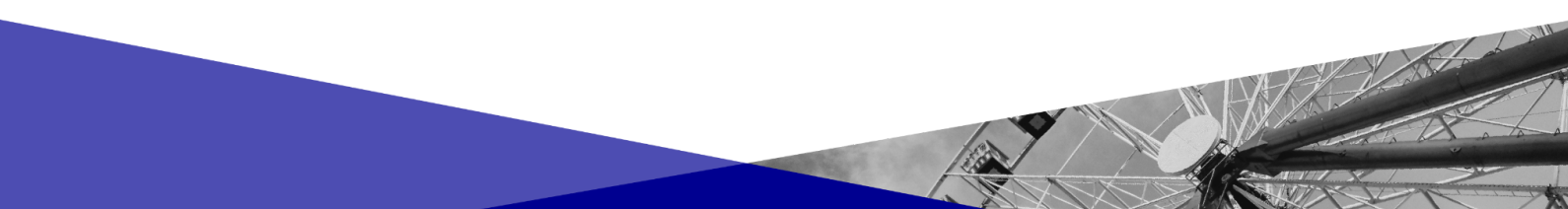
```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           1s
```

Donde podemos consultar el estado de las réplicas listas frente a las deseadas, actualizadas y disponibles, además del tiempo que el deployment lleva creado. Sin embargo, esta información puede ser insuficiente cuando queremos consultar la imagen que utiliza un deployment, o el selector que tiene definido. En estos casos, aplicaremos el parámetro "-o wide" al comando anterior:

```
kubectl get deployments -o wide
```

Obteniendo más información en la salida:

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES         SELECTOR
nginx-deployment   3/3     3            3           10s   nginx        nginx:1.7.9    app=nginx
```



Los *ReplicaSet* generados siguen el patrón **{nombreDeplyoment}-{CadenaAleatoria}**, por lo que podemos listar los *rs* existentes y filtrar por aquellos que contengan, en nuestro caso, “nginx” como etiqueta en .spec.metadata.labels:

```
kubectl get rs -l app=nginx
```

Salida:

```
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-75675f5897   3         3         3       18s
```

De forma similar, podemos listar todos los pods pertenecientes a un *Deployment* o su *ReplicaSet*, ya que estos se crean siguiendo el patrón **{nombreReplicaSet}-{CadenaAleatoria}**:

```
kubectl get pods -l app=nginx
```

Que en nuestro ejemplo tendría la siguiente salida:

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-75675f5897-7ci7o   1/1     Running   0          18s
nginx-deployment-75675f5897-kzszj   1/1     Running   0          18s
nginx-deployment-75675f5897-qqcnn   1/1     Running   0          18s
```

Los pods tienen una etiqueta **pod-template-hash**, que se añade automáticamente mediante el controlador de *Deployment* a cada *ReplicaSet* creado o adoptado. Se genera mediante una función hash aplicada a la plantilla de pod y no debes cambiarla, ya que esta etiqueta garantiza que todos los *ReplicaSet* hijos de un *Deployment* no mezclen sus cargas de trabajo. Esta etiqueta se puede mostrar aplicando el parámetro **--show-labels** al comando anterior.

Actualización de un *Deployment*

Existen dos formas de actualizar la imagen que usa un *Deployment*, pero ambas se basan en el mismo principio: un *Deployment* sólo se actualiza cuando se presenta un cambio en la plantilla del pod (campo **.spec.template**).

Actualizar un pod significa activar el lanzamiento de *Deployment* para cambiar las propiedades del objeto actual. Otras actualizaciones, como el escalado, se pueden llevar a cabo sin activar un lanzamiento de despliegue.

Las dos maneras que existen para actualizar *Deployments* son:

- **Actualizar la imagen**: Hacemos uso del comando ***set image*** para actualizar la imagen que utilizan todos los objetos hijos del *Deployment*. Si además usamos el parámetro **--record**, la actualización se guardará en el historial de versiones con el comando que hemos utilizado como valor del campo **CHANGE-CAUSE**:

```
kubectl set image deployment/nginx-deployment \
nginx=nginx:{newVersionOfNginx} --record
```

- **Editar el *Deployment***: También podemos editar directamente el objeto *Deployment* y cambiar el valor de la imagen en el campo **.spec.template.spec.containers**. Para ello usaremos el comando:

```
kubectl edit deployment/nginx-deployment
```

Para comprobar el estado del *Deployment* y su actualización, hacemos uso de ***rollout status*** y ***describe deployments***. Lanzando el comando:

```
kubectl rollout status deployment/nginx-deployment
```

Podemos consultar el estado de la actualización y el número de réplicas actualizadas. Sin embargo, si queremos una visión más detallada del proceso, ejecutamos el comando:

```
kubectl describe deployments
```

Y en la sección “Events” de la descripción de “nginx-deployment” observaremos una salida como esta:

```
Name:                   nginx-deployment
...
StrategyType:           RollingUpdate
...
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  2m    deployment-controller  Scaled up
replica set nginx-deployment-2035384211 to 3
  Normal  ScalingReplicaSet  24s   deployment-controller  Scaled up
replica set nginx-deployment-1564180365 to 1
  Normal  ScalingReplicaSet  22s   deployment-controller  Scaled down
```

```
replica set nginx-deployment-2035384211 to 2
  Normal  ScalingReplicaSet  22s   deployment-controller  Scaled up
replica set nginx-deployment-1564180365 to 2
  Normal  ScalingReplicaSet  19s   deployment-controller  Scaled down
replica set nginx-deployment-2035384211 to 1
  Normal  ScalingReplicaSet  19s   deployment-controller  Scaled up
replica set nginx-deployment-1564180365 to 3
  Normal  ScalingReplicaSet  14s   deployment-controller  Scaled down
replica set nginx-deployment-2035384211 to 0
```

Esta salida puede ser útil cuando queramos depurar los cambios que ha sufrido nuestro deployment tras una actualización.. En particular, se puede ver cómo no se elimina una réplica antigua hasta que no se ha creado correctamente la nueva. Esto asegura que en ningún momento, durante la actualización de la imagen de los pods, la aplicación se quede sin servicio por lo que estará disponible para usarse en todo momento.

Existen formas más complejas de actualización, llevando a cabo múltiples actualizaciones a la vez y utilizando el selector de etiquetas. Si estás interesado en seguir aprendiendo al respecto, te recomendamos leer sobre actualizaciones por sobrescritura y selector de etiquetas.

Rollback de un *Deployment*

Al igual que necesitamos actualizar *Deployments*, habrá situaciones en las que necesitaremos revertirlos. Cada vez que se actualiza un *Deployment*, modificando su plantilla, se crea una "revisión": una versión que se almacena en un historial. Este historial, por defecto, almacena todas las revisiones del *Deployment*, pero este número puede ser modificado.

Es importante darse cuenta de que, según la definición anterior, sólo las actualizaciones que implican un cambio en la plantilla del pod generan nuevas revisiones, los cambios de escalado no. Esto significa que cuando revertimos a una versión anterior, tan sólo revertimos la plantilla del pod.

A continuación presentamos un ejemplo que nos muestra cómo detectar errores en actualizaciones y cómo revertir a una revisión anterior:

Supongamos que a la hora de actualizar el *Deployment*, nos equivocamos en la versión de la imagen, lo que provoca que la actualización fracase, dejando un pod intentando descargar una imagen que no existe en bucle. Como siempre, comprobamos el estado de la actualización con:

```
kubectl rollout status deployment/nginx-deployment
```

Y la salida se atasca en el siguiente punto:

```
Waiting for rollout to finish: 1 out of 3 new replicas have been
updated...
```

En este momento podemos suponer un fallo en la actualización. Para finalizar la monitorización pulsamos **Ctrl+C** y ejecutamos el comando:

```
kubectl get pods
```

En cuya salida podemos ver que, entre todos los pods, el creado por el nuevo *ReplicaSet* no consigue descargar la imagen asociada:

```
NAME                                READY   STATUS             RESTARTS   AGE
nginx-deployment-1564180365-70iae   1/1     Running            0          25s
nginx-deployment-1564180365-jbqqo   1/1     Running            0          25s
nginx-deployment-1564180365-hysrc   1/1     Running            0          25s
nginx-deployment-3066724191-08mng   0/1     ImagePullBackOff   0          6s
```

Llegados a este punto existen diferentes situaciones en función del parámetro **maxUnavailable** del controlador de *Deployment*. Este parámetro define el porcentaje de pods erróneos que se permiten antes de que el *Deployment* se pare. Por tanto, en este punto podríamos tener un deployment corriendo en condiciones peores a la óptima o tenerlo parado.

Sea cual sea el caso, necesitamos volver a una versión estable. Para ello revisamos el historial del *Deployment* con el comando **rollout history**:

```
kubectl rollout history deployment/nginx-deployment
```

Que, en nuestro ejemplo, tendría la salida:

```
deployments "nginx-deployment"
REVISION    CHANGE-CAUSE
1           kubectl apply
--filename=https://k8s.io/examples/controllers/nginx-deployment.yaml
--record=true
2           kubectl set image deployment.v1.apps/nginx-deployment
nginx=nginx:1.16.1 --record=true
3           kubectl set image deployment.v1.apps/nginx-deployment
nginx=nginx:1.161 --record=true
```

Volviendo al caso que nos concierne, la última revisión estable es la segunda, pero no siempre querremos revertir a la última revisión. Por ello, veremos cómo revertir a la más reciente:

```
kubectl rollout undo deployment/nginx-deployment
```

O a otra más antigua, indicando su número de revisión:

```
kubectl rollout undo deployment/nginx-deployment --to-revision=1
```

Estos son los casos más comunes al usar **rollout**, pero si te interesa indagar más al respecto te recomendamos visitar la documentación.

Con esto hemos revertido el *Deployment* a una versión estable y podemos volver a trabajar con él. Es importante mencionar que, aunque el proceso es similar en la mayoría de controladores, no todos se pueden revertir. Además, debemos asegurarnos de reanudar los *Deployments* antes de revertirlos, de lo contrario, la reversión fracasará.

Escalado de un *Deployment*

El escalado de este controlador es similar al resto, hacemos uso del comando **scale** para escalar manualmente a un número de réplicas, o el comando **autoscale**, para definir un rango de réplicas y una carga objetivo:

```
kubectl scale deployment/nginx-deployment --replicas=10

kubectl autoscale deployment/nginx-deployment --min=10 --max=20
--cpu-percent=70
```

Este último necesita que el clúster tenga habilitado el **escalado horizontal de pods**, de lo contrario fallará.

A diferencia de otros controladores, *Deployment* presenta un escalado proporcional a las actualizaciones que sufre. Cuando se produce un escalado con actualización continua que interfiere con otro despliegue, activo o pausado, el controlador balancea las réplicas adicionales para minimizar el riesgo. Una muestra de esto ocurre en el ejemplo de actualización explicado anteriormente.

StatefulSet

StatefulSet es el controlador que se utiliza para gestionar las aplicaciones con estado. Al igual que los anteriores controladores, este gestiona el despliegue y escalado de los pods que contienen dichas aplicaciones. Para ello, se define el estado deseado y el controlador efectúa las actualizaciones necesarias para alcanzarlo.

De la misma forma que un *Deployment*, el *StatefulSet* gestiona los pods que se crean a partir de una misma especificación de contenedor. Sin embargo, un *StatefulSet* mantiene una identidad asociada a los pods, es decir, se generan a raíz de una especificación idéntica pero no pueden intercambiarse, ya que cada uno tiene su propio identificador que mantiene durante cualquier reprogramación.

Los *StatefulSets* cobran sentido cuando las aplicaciones necesitan de alguna de los siguientes características:

- **Identificadores de red** únicos y estables.
- **Almacenamiento** estable y persistente.
- **Despliegue** y **escalado** ordenado y controlado.
- **Actualizaciones** continuas ordenadas y automatizadas.

Con el término estable nos referimos a la necesidad de que se mantengan sus propiedades entre distintas reprogramaciones de pods. En caso de que tu aplicación no necesite de ninguna de estas características, deberías desplegarla haciendo uso de un controlador como *Deployment* o *ReplicaSet*, ya que los *StatefulSets* están pensados para utilizarlos con aplicaciones con estado.

Limitaciones

- El **almacenamiento** de un determinado pod debe provisionarse por un provisionador de volúmenes persistentes, de los que hablaremos más adelante.
- **Eliminar** un *StatefulSet* no eliminará sus volúmenes asociados. Esto se hace así de manera intencional para garantizar la seguridad de los datos.
- Los *StatefulSets* necesitan un **servicio *Headless*** como responsable de la identidad de red de los pods, el cual no se crea automáticamente, sino que es nuestra responsabilidad crear este servicio.

- Al eliminar un *StatefulSet*, no se garantiza que los pods que creó se eliminen también. Para conseguir esto, existe la posibilidad de reducir el número de **réplicas a 0 antes de eliminarlo**.

Ejemplos de uso

Creación de un *StatefulSet*

Para crear un *StatefulSet* necesitaremos especificar los siguientes componentes en nuestro archivo YAML:

- Un **servicio *Headless*** llamado nginx, para controlar el dominio de red.
- El propio *StatefulSet* llamado web, que creará 2 réplicas del contenedor nginx en pods únicos.
- Un ***volumeClaimTemplate***, que proporciona almacenamiento estable a través de volúmenes persistentes.

Teniendo todo esto en cuenta, creamos el siguiente archivo YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
```

```
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

El valor del campo ***.spec.selector.matchLabels.app*** debe coincidir con el del campo ***.spec.template.metada.labels.app***. En caso contrario, se producirá un error en la creación del *StatefulSet*.

Guardamos el archivo como web.yaml y ejecutamos el siguiente comando:

```
kubectl apply -f {filePath}/web.yaml
```

Para comprobar que el servicio se creó correctamente ejecutamos lo siguiente:

```
kubectl get service nginx
```

Obteniendo una salida parecida a esta:

```
NAME      TYPE         CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx     ClusterIP    None         <none>        80/TCP    12s
```

Y para comprobar que el *StatefulSet* está correctamente creado, ejecutamos el siguiente comando:

```
kubectl get statefulset web
```

Obteniendo lo siguiente:

```
NAME      DESIRED   CURRENT   AGE
web       2         1         20s
```

También podemos observar los volúmenes persistentes que se crearon con el comando:

```
kubectl get pvc -l app=nginx
```

Obteniendo algo parecido a esto:

```
NAME        STATUS  VOLUME                                     CAPACITY  ACCESSMODES  AGE
www-web-0   Bound   pvc-15c268c7-b507-11e6-932f-42010a800002   1Gi       RWO          48s
www-web-1   Bound   pvc-15c79307-b507-11e6-932f-42010a800002   1Gi       RWO          48s
```

Creación ordenada de pods

Para un *StatefulSet* con n réplicas, a la hora de crear los pods, estos se crearán en orden secuencial de 0 a n-1. Además, como hemos mencionado anteriormente, cada pod recibirá un identificador único. Este identificador se obtiene a partir del nombre del controlador y del índice ordinal de creación del pod. Por lo tanto, el nombre del pod tendrá la forma {nombre statefulset}-{índice ordinal}. De esta manera, como nuestro controlador tiene 2 réplicas, este creará dos pods con los nombres web-0 y web-1.

Para ver el estado de los pods que se están creando, podemos ejecutar el siguiente comando en otra terminal:

```
kubectl get pods -w -l app=nginx
```

```
NAME      READY     STATUS              RESTARTS   AGE
web-0     0/1       Pending             0          0s
web-0     0/1       Pending             0          0s
web-0     0/1       ContainerCreating   0          0s
web-0     1/1       Running             0          19s
web-1     0/1       Pending             0          0s
web-1     0/1       Pending             0          0s
web-1     0/1       ContainerCreating   0          0s
web-1     1/1       Running             0          18s
```

Como podemos observar, el pod web-1 no se crea hasta que el pod web-0 está preparado y ejecutándose. Esto se debe a que ***OrderedReady*** es la estrategia predeterminada para la creación de los pods de un *StatefulSet*.

Sin embargo, también existe la estrategia ***Parallel***, con la que el controlador creará y eliminará los pods en paralelo, sin esperar a que estos estén preparados y ejecutándose.

Para ello, dentro de nuestro archivo web.yaml, añadiríamos al campo ***.spec.podManegementPolicy*** del *StatefulSet* web el valor "Parallel".

Escalado de un *StatefulSet*

Para escalar nuestro controlador basta con ejecutar el siguiente comando, indicando el nuevo número de réplicas que queremos:

```
kubectl scale sts web --replicas=5
```

Actualización de un *StatefulSet*

Existen dos tipos de estrategias a la hora de actualizar el controlador *StatefulSet*:

- ***RollingUpdate***: Es la estrategia por defecto, la cual actualizará automáticamente todos los pods en orden inverso, es decir, del número ordinal más grande al más pequeño.
- ***OnDelete***: Con esta estrategia, el controlador no actualizará los pods de manera automática, sino que deberemos eliminar manualmente los pods para que el controlador los cree de nuevo teniendo en cuenta las nuevas modificaciones.

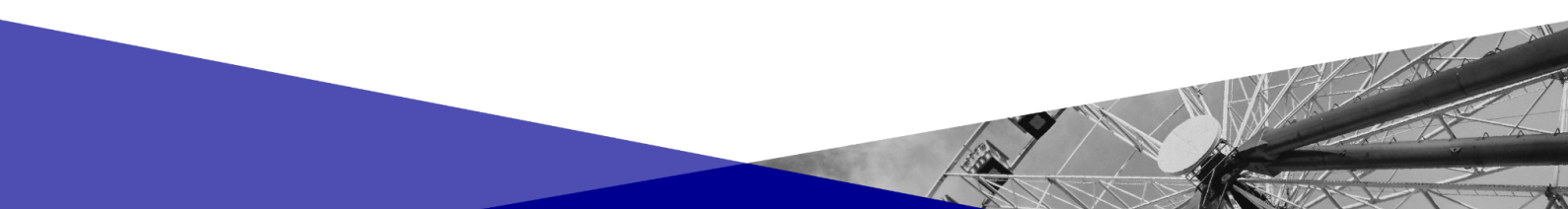
Podemos usar una u otra dependiendo del valor que tengamos en el campo ***.spec.updateStrategy.type*** de nuestro archivo YAML.

Para actualizar nuestro controlador, ejecutamos el siguiente comando, indicando la parte que nos interesa cambiar y su nuevo valor:

```
kubectl patch statefulset web --type='json' -p='[{"op": "replace",
"path": "/spec/template/spec/containers/0/image",
"value":"gcr.io/google_containers/nginx-slim:0.8"}]'
```

Eliminación de un *StatefulSet*

Los *StatefulSets* soportan ambos tipos de eliminación: **normal**, que solo eliminará el controlador; y **en cascada**, que eliminará a su vez todos los



pods que controlase.

Para eliminar el controlador pero no sus pods, utilizaremos el comando:

```
kubectl delete statefulset web --cascade=orphan
```

Sin embargo, para eliminar el controlador con todos sus pods, ejecutamos el comando anterior sin la opción --cascade=orphan:

```
kubectl delete statefulset web
```

Si el campo ***.spec.podManegementPolicy*** del *StatefulSet* está omitido o su valor es *OrderedReady*, entonces los pods se eliminarán de manera secuencial en orden inverso a su creación.

A pesar de haber eliminado el *StatefulSet* junto con todos sus pods, el comando anterior no elimina el servicio *Headless* asociado al controlador. Para eliminarlo también, ejecutamos lo siguiente:

```
kubectl delete service nginx
```

DaemonSet

El controlador ***DaemonSet*** se emplea para asegurar la ejecución en los nodos disponibles de todos los pods asociados a él. Esto se consigue ejecutando copias del pod en todos los nodos.

El **proceso** es el siguiente: cuando se añade un nodo a un clúster, el controlador añade un pod nuevo a ese nodo. De la misma forma, cuando se elimina un nodo, el controlador se asegura de que el pod que contenía sea recolectado por la basura. Cuando eliminamos un *DaemonSet*, se limpian todos los pods que haya creado.

La **importancia** de este controlador reside en la facilidad que supone para los administradores de sistemas la **configuración de los pods** en el conjunto de nodos.

Algunos de los **usos más comunes del *DaemonSet*** son: ejecutar procesos de **almacenamiento** en el clúster, **recolectar logs** en los nodos o ejecutar procesos de **monitorización** de nodos. Con estas aplicaciones del controlador, se consigue mejorar el rendimiento de los clúster de K8s distribuyendo las tareas de mantenimiento de pods entre todos los nodos.

En sistemas sencillos, se suele utilizar un solo *DaemonSet* para todos los nodos, pero en otros más complejos se pueden configurar varios controladores de este tipo con características diferentes.

Escribir una especificación de DaemonSet

La configuración de este controlador en un archivo .yaml requiere una serie de campos que hemos visto para el resto de controladores que son: **apiVersion, kind y metadata.** Por otro lado, contiene la sección **spec**. Vamos a ver un ejemplo de configuración antes de explicar cada uno de los campos que lo componen.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: gcr.io/fluentd-elasticsearch/fluentd:v2.5.1
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
```

```
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

La sección **spec** tiene dos campos obligatorios:

- **Template:** es una plantilla del pod que va a copiarse en todos los nodos. Tiene el mismo esquema que un pod, la diferencia es que no tiene los campos apiVersion o Kind y está anidado. En el campo **spec** de un template, podemos especificar mediante **nodeSelector** (**.spec.template.spec.nodeSelector**) los nodos concretos donde el DaemonSet creará los pods. De forma similar, con **nodeAffinity** (**.spec.template.spec.nodeAffinity**) se crearán pods en los nodos que cumplan con la afinidad del nodo que indique el campo. Si no se configura ni uno ni el otro, se crearán en todos los nodos.

- **Selector:** el selector de un *DaemonSet* es fijo y no puede cambiar una vez que se ha creado el controlador por primera vez. Si se cambia el selector una vez iniciado el controlador, los pods que se habían creado quedarán "huérfanos". Los campos del selector no tienen valores por defecto, tienen que coincidir con las etiquetas del *template*. El campo **.spec.selector** a su vez, consiste en:

    - **matchLabels**: Es exactamente igual que en el resto de controladores.

    - **matchExpressions**: Permite construir selectores más específicos indicando claves y valores, así como un operador que los relaciona.

Si configuramos los dos campos del selector, el controlador resultante es el de la conjunción de ambos. Además, si configuramos el campo **.spec.selector**, debemos asegurarnos de que coincide con el campo **.spec.template.metadata.labels** porque si no, la API rechazará la

configuración. En el ejemplo que acabamos de ver, coinciden. Una configuración de este tipo sería rechazada:

```
...
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearchspec
  template:
    metadata:
...
```

Ejemplos de uso

Creación de un *DaemonSet*

Mediante **kubectl** podemos crear un controlador *DaemonSet* con el comando **apply** y el nombre del archivo de configuración que hayamos creado.

```
kubectl apply -f {file-path}/config-DaemonSet.yaml
```

Listado de *DaemonSets*

Para obtener la información de los controladores de este tipo ejecutamos el comando get:

```
kubectl get daemonset
```

Y si queremos obtener más información:

```
kubectl get pod -o wide
```

Comunicación con los pods de un *DaemonSet*

Hay varias formas de establecer comunicación con los pods de un controlador *DaemonSet*:

- **Push**: Los pods no tienen clientes, es decir, envían información a otros servicios pero no reciben nada del exterior.
- **NodeIP y Known Port**: Se puede conectar con los pods mediante la IP de los nodos que los contienen.
- **DNS**: Se puede configurar un servicio con el mismo selector de pod y se descubren los *DaemonSet* usando los *endpoints*.
- **Service**: Sirve para seleccionar un nodo aleatorio del *DaemonSet* que se puede utilizar después para crear un servicio con el mismo selector de pod.

Actualización de un *DaemonSet*

Si cambiamos las etiquetas de nodo en el campo .**spec.template**, el *DaemonSet* comenzará a añadir Pods a los nodos coincidentes con esas etiquetas y eliminará los pods de aquellos nodos que ya no coincidan con las etiquetas. Como hemos visto antes, el campo .**spec.selector** no debe modificarse porque esto provocará que los pods con la antigua configuración queden huérfanos.

El comando para aplicar los cambios es:

```
kubectl apply -f {file-path}/config-daemonSet.yaml
```

Eliminar un *DaemonSet*

Como en el resto de controladores, para eliminar un DaemonSet ejecutamos:

```
kubectl delete <nombreControlador>
```

Y si queremos conservar los pods, añadimos el parámetro cascade=false:

```
kubectl delete <nombreControlador> --cascade=false
```

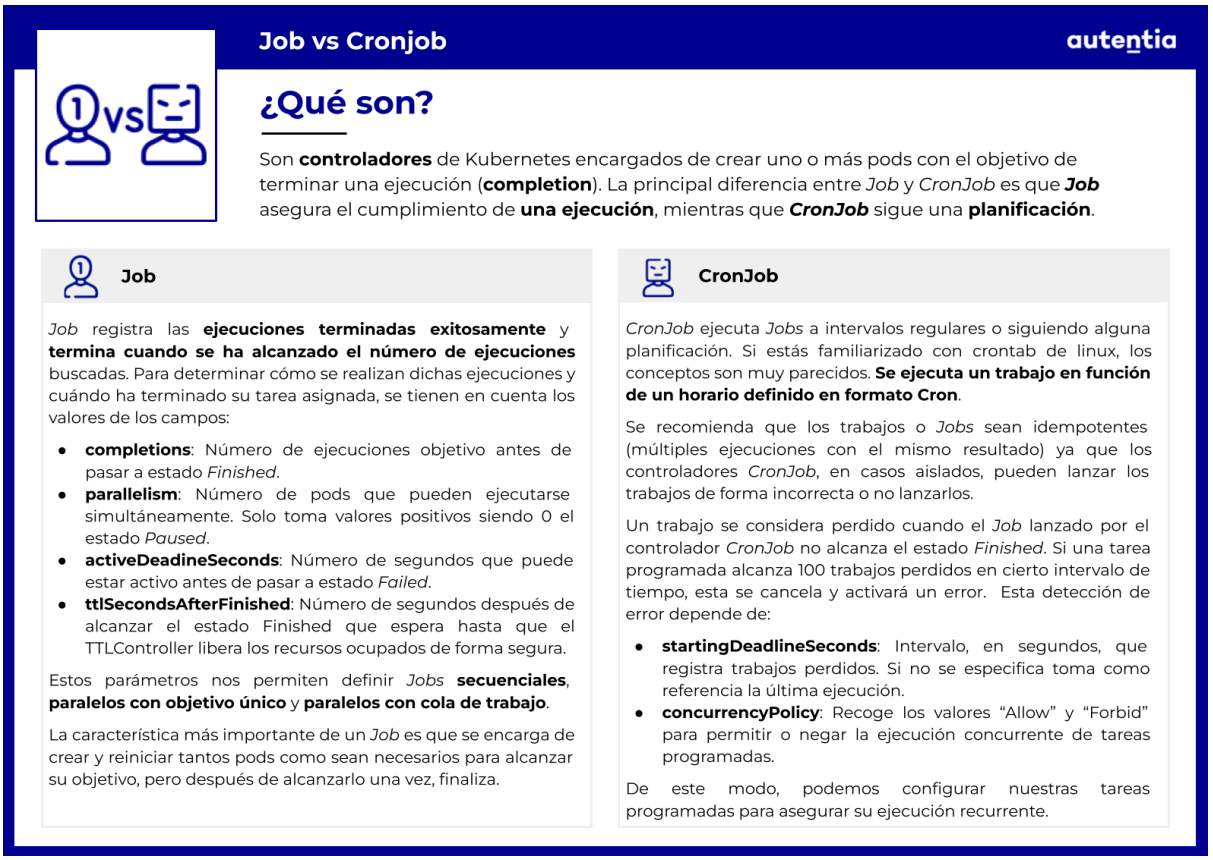
El efecto será que los pods seguirán ejecutándose en los nodos y se podrá crear un nuevo *DaemonSet* con otra plantilla distinta que reconocerá los pods existentes con etiquetas coincidentes y no modificará ni eliminará ninguno de los pods que ya existen aunque la plantilla no coincida. Para que se sustituyan los pods antiguos por los nuevos definidos en la plantilla del controlador, será necesario **forzar la creación de los pods** mediante la eliminación manual de los antiguos en cada nodo o el propio nodo.

Jobs

Un *Job* o tarea se encarga de crear uno o más pods con el objetivo de terminar una ejecución. En algunos casos no es suficiente con una sola ejecución (**completion**). Por ello, el Job registra las ejecuciones terminadas exitosamente y termina cuando se ha alcanzado el número deseado de ejecuciones.

Podemos eliminar o suspender *Jobs*, lo que provocará que, en ambos casos, se eliminen los pods que ha creado, ya sea de forma temporal (hasta que se reanude el *Job*) o definitiva. Además, podemos ejecutar múltiples pods en paralelo definiendo el parámetro **parallelism**, como veremos más adelante.

Un *Job* está diseñado para ejecutarse una sola vez, asegurar el cumplimiento del objetivo, y terminar. Sin embargo, existen ocasiones en las que nos interesa que el *Job* siga un cierto programa o calendario. En estos casos haremos uso de ***CronJob***, diseñado para llevar a cabo tareas programadas o recurrentes en el tiempo.



Escribir una especificación de Job

Al igual que todos los objetos de K8s, *Job* requiere una especificación en un archivo YAML, que para futuros ejemplos nombraremos **job.yaml**.

Como el resto de objetos, es necesario que definamos los campos **apiVersion**, **kind** y **metadata**, cuidando que el campo **name** sea un nombre de subdominio válido.

Para definir el objeto también necesitamos campos que no son comunes a todos los demás, como:

- **.spec.template**: Es la plantilla del pod, y recoge exactamente las mismas características que los demás controladores que hemos visto. Sin embargo, por la naturaleza de un *Job* (siempre busca terminar), el campo **.spec.template.RestartPolicy** sólo puede tener los valores **Never** y **OnFailure**.
- **.spec.selector**: Este es un campo opcional, con las mismas características que el resto de controladores.
- **.spec.completions**: Es el número de pods que deben terminar su ejecución exitosamente para que el *Job* se considere finalizado.
- **.spec.parallelism**: Define el número de pods que pueden estar ejecutándose a la vez en un instante de tiempo. Puede tomar únicamente valores positivos, siendo el valor por defecto 1 (Si le damos valor 0, el Job queda pausado hasta que incrementemos este campo).
- **.spec.template.spec.restartPolicy**: Campo que define cuándo se reiniciará un pod contenido por el *Job*. Como ya se ha mencionado, nunca puede tener valor “Always”. La definición de este campo, junto con los campos *parallelism* y *completions*, nos permiten gestionar la manera en la que el *Job* responde a errores, volviendo a iniciar un pod, o desechando y sustituyéndolo.
- **.spec.backoffLimit**: Es el número de veces que un *Job* permite que se produzcan fallos en sus pods. Si el número de pods fallidos alcanza el valor de este campo, el propio *Job* terminará en fracaso. Para detectar esos pods fallidos, contaremos el número de pods con fase *Failed* o, si la política de reinicio es *OnFailure*, el número de reintentos de los pods en fase *Pending* o *Running*.
- **.spec.activeDeadlineSeconds:** Igual que en otros controladores,

define el número de segundos que un pod puede estar activo. Si se sobrepasa ese límite, el pod pasa a un estado fallido.

- **.spec.ttlSecondsAfterFinished:** El número de segundos que deben pasar desde que finaliza un *Job* hasta que el *TTLController* limpia y libera los recursos del *Job* y sus hijos, en cascada. Es muy recomendable especificar un valor para este campo, ya que, por defecto, los *Jobs* tienen una política de eliminación *orphanDependents*. Esto provoca que, si un pod aún no se ha eliminado pero el *Job* sí, ese pod quede huérfano. Aunque lo recogerá el recolector de basura, puede provocar degradación del rendimiento o fallos más graves.

Al trabajar con *parallelism* y *completions*, existen varias combinaciones de estos campos que definen cómo actúan los pods:

- ***Jobs* secuenciales**: Un único pod activo a la vez, que es reemplazado si falla. El *Job* finaliza cuando se alcanzan las *completions* deseadas.
- ***Jobs* paralelos con objetivo fijo**: Se especifica un número fijo de *completions*, donde el *Job* representa la tarea genérica, y los pods reciben subtareas o ejecuciones más concretas. Finaliza cuando se alcanzan las *completions* deseadas.
- ***Jobs* paralelos con cola de trabajo**: Caso más especial donde no especificamos el campo *.spec.completions*. Los propios pods se coordinan entre sí o con un servicio externo para decidir quién debería trabajar en qué tarea. Los pods son conscientes del estado de sus compañeros, y finalizan su ejecución cuando al menos un pod ha finalizado con éxito.

Veamos un ejemplo, en el que definimos un *Job* para calcular el número PI:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  backoffLimit: 5
  activeDeadlineSeconds: 100
  ttlSecondsAfterFinished: 100
  selector:
    matchLabels:
      app: pi
```

```
  template:
    metadata:
      labels:
        app: pi
    spec:
      containers:
      - name: pi
        image: perl:5.36.0
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

Este archivo, al que antes llamamos **job.yaml**, se ha definido para calcular el número PI mediante el lenguaje de programación Perl 5.36.0. Hemos especificado un número máximo de 5 pods fallidos, donde cada pod puede estar activo como máximo 100 segundos, y cuando el *Job* termine, será elegible para su eliminación pasados 100 segundos de TTL.

Si queremos planificar y repetir la ejecución del *Job*, haríamos uso de un *CronJob*, la diferencia entre definiciones es mínima, quedando:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: pi
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
     spec:
        backoffLimit: 5
        activeDeadlineSeconds: 100
        ttlSecondsAfterFinished: 100
        selector:
           matchLabels:
           app: pi
        template:
           metadata:
             labels:
               app: pi
           spec:
             containers:
               - name: pi
                 image: perl:5.36.0
                 command: ["perl",  "-Mbignum=bpi", "-wle", "print
bpi(2000)"]
```

```
            restartPolicy: Never
```

Como podemos observar, *CronJob* engloba el *Job* que utilizamos para calcular Pi dentro de su campo **.spec.jobTemplate.spec**. Si prestamos atención al campo **.spec.schedule** vemos que toma valor "*/1 * * * *", lo que significa una ejecución cada minuto.

Ejemplos de uso

Creación de un *Job*

Para crear un *Job* primero debemos definir su archivo YAML, para nuestro ejemplo utilizaremos el archivo **job.yaml** explicado anteriormente. Una vez tenemos definido nuestro objeto, podemos crear el *Job* mediante el comando:

```
kubectl apply -f {file-path}/job.yaml
```

Listado de un *Job*

Para listar los *Jobs* existentes podemos utilizar el comando:

```
kubectl get jobs
```

O si queremos conseguir información más detallada:

```
kubectl describe job pi
```

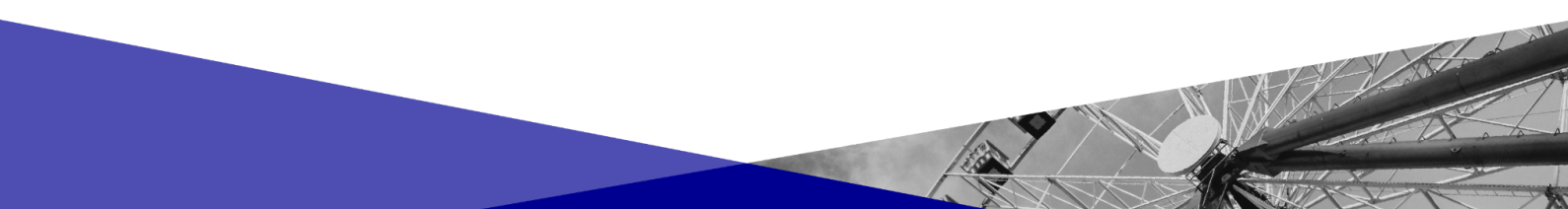
(Observar que pi es el nombre que tiene el *Job* en **job.yaml**)

Actualización de un *Job*

En ocasiones necesitamos actualizar la definición de los *Jobs*. El ejemplo más claro es cuando suspendemos un *Job* por un tiempo. Para suspenderlo, modificamos el campo **.spec.suspend** con el siguiente comando:

```
kubectl patch job/pi --type=strategic --patch
'{"spec":{"suspend":true}}'
```

Si lo queremos reanudar, basta con poner valor *false*.



Nodos

Los **nodos** son máquinas de trabajo que pueden ser físicas o virtuales en función del tipo de clúster al que pertenezcan. Al principio, los nodos se conocían como **minions.**

Un clúster puede estar formado por un gran número de nodos en función de los recursos disponibles o puede existir un único nodo en un clúster sin ningún tipo de problema.

Cada nodo se encarga de ejecutar los pods que tiene dentro y está gestionado por la herramienta de **plano de control**.

Los nodos, a diferencia de otros recursos que hemos visto en esta guía, no se crean desde la plataforma de K8s de forma directa, sino que se crean de forma externa (desde proveedores de servicios en la nube o en máquinas físicas o virtuales del usuario). Cuando “creamos” un nodo, lo que se hace realmente es crear un objeto que representa a este nodo.

Estado de un nodo

El estado de un nodo está compuesto por la información sobre direcciones, estados, capacidad e información general de dicho nodo. Cada uno de estos puntos contiene una serie de campos de especificación del nodo que se detallan a continuación.

Direcciones

Estos campos dependen de la configuración del proveedor de servicios en la nube o la configuración en máquinas locales o virtuales de los nodos.

- **HostName**: Es el nombre de la máquina *host* en el kernel del nodo. Se puede reconfigurar con **kubelet** mediante el parámetro --hostname-override.
- **ExternalIP**: Es la dirección IP del nodo a la que se puede acceder desde fuera del clúster del nodo.
- **InternalIP**: Es la dirección IP del nodo a la que solo se puede acceder desde dentro del clúster que lo contiene.

Estados

El estado de los nodos se describe en el campo **conditions** de todos los nodos que están en ejecución. Los posibles estados de un nodo pueden tomar los valores **true** o **false** y son:

- **OutOfDisk**: Es true cuando **no hay espacio** en el nodo para añadir más pods.
- **MemoryPressure**: True cuando el **consumo de memoria** en el nodo es muy alto.
- **PIDPressure**: True si hay **muchos procesos** en el nodo. Se mide con el recuento de PIDs.
- **DiskPressure**: True si la **capacidad** del disco es **baja**.
- **NetworkUnavailable**: True si la **configuración de red** no es correcta.
- **Ready**: True cuando el nodo está en **buen estado** y admite nuevos pods. En este caso, false indica que no se pueden aceptar nuevos pods y hay una tercera posibilidad, que el estado sea **Unknown.** Este estado indica que el controlador aún no tiene constancia del nodo porque todavía no se ha producido el **node-monitor-grace-period** que, por defecto, se ejecuta cada cuarenta segundos.

  Además, si el estado es False o Unknown durante más tiempo que el **pod-eviction-timeout,** el **kube-controller-manager** se encarga de que todos los pods de ese nodo se marquen para eliminarse. Este tiempo es por defecto **5 minutos**.

Capacidad

Se refiere a los recursos disponibles en el nodo: CPU, memoria y pods que pueden planificarse dentro del nodo. Estas características se utilizan para determinar los estados que acabamos de ver en el punto anterior.

La capacidad se declara en el momento de crear el nodo. El planificador de K8s se encarga de comprobar que en cada nodo existen recursos suficientes para ejecutar los pods que hay dentro de él.

Información

Almacena la información general del nodo: versión del kernel, versión de K8s (kubelet y kube-proxy), nombre del sistema operativo, etc.

Controlador de nodos

El controlador de nodos es un componente maestro de K8s. Su función, como indica su nombre, es gestionar los nodos durante todo su ciclo de vida.

Cuando el nodo se registra, el controlador le asigna un **bloque CIDR (Class Inter-Domain Routing)** donde se almacenan las IPs disponibles para asignar a los objetos que habrá en ejecución dentro de ese nodo posteriormente.

Si el sistema administrado desde K8s utiliza servicios de un proveedor en la nube, el controlador de nodos tiene una **lista interna de máquinas disponibles** en el proveedor de servicios. Si un nodo deja de responder, el controlador se encarga de pedir al proveedor que compruebe si la máquina de ese nodo está disponible. Si no lo está, el controlador borra el nodo de la lista.

En el caso de que la salud del nodo cambie, el controlador de nodos es quien cambia el estado de **NodeReady** a **ConditionUnknown** cuando deja de estar accesible. Los nodos envían unas señales de vida cuando están disponibles que se conocen como **latidos o *heartbeats*** y, si estas señales no llegan, el controlador de nodos tiene que desalojar todos los pods del nodo. Sin embargo, esto no es inmediato. El controlador intenta conectarse con el nodo durante cuarenta segundos desde que deja de recibir latidos hasta que cambia el estado a ConditionUnknown y si el nodo no vuelve a emitir señales durante cinco minutos más, tiene que enviar los pods del nodo a otro nodo que esté disponible.

Cuando se produce **desalojo de nodos** porque uno o varios nodos están en estado ConditionUnknown. Existe una tasa de desalojo **--node-eviction-rate** cuyo valor por defecto es 0,1. Esto significa que no se desalojan pods de más de un nodo cada diez segundos. Si el controlador detecta que muchos nodos de una misma zona están fallando a la vez, la política de desalojo depende de dos factores:

- **Cantidad de nodos con problemas.** Si el número de nodos con problema es mayor al 55% de nodos de esa zona, el ratio de

desalojos se reduce. Este valor se establece en 55% por defecto, pero puede cambiarse en **--unhealthy-zone-threshold.**

- **Cantidad de nodos del clúster.** Si el clúster tiene menos de 50 nodos, entonces se paran los desalojos. Si no, se reduce por defecto a 0,01 (un nodo por segundo). Este valor se puede cambiar en **--secondary-node-eviction-rate.**

Es una buena práctica la distribución de nodos en **zonas de disponibilidad** porque así el volumen de trabajo puede distribuirse a nodos de una zona en buen estado cuando alguna de ellas se caiga. De esta forma, si una zona entera está en mal estado, el ritmo de desalojo es más alto para que los pods se distribuyan entre los nodos de las zonas en buen estado. Sin embargo, en el caso de que todas las zonas estén en mal estado, los desalojos se deben paralizar hasta que los nodos vuelvan a estar conectados.

Si no hay zonas de disponibilidad definidas, se considera que todos los nodos pertenecen a la misma zona.

Registro de nodos

Existen dos formas de crear nodos en un clúster:

- **Auto-Registro** de nodos: En kubelet está habilitado por defecto el atributo --register-node. Esto hará que kubelet intente auto-registrarse con la API de K8s. Este es el patrón de diseño más común. Los atributos necesarios para iniciar kubelet para auto-registro de nodos son:
    - **--kubeconfig**: Indica la ruta a las credenciales para autenticarse con la API.
    - **--cloud-provider**: En caso de utilizar un proveedor de servicios en la nube, almacena los metadatos necesarios para comunicarse con él.
    - **--register-node**: Como hemos mencionado anteriormente, sirve para habilitar el auto-registro de nodos.
    - **--register-with-taints**: Si el atributo anterior está habilitado, este atributo guarda una lista de taints. Un taint es lo contrario a una afinidad, define una característica que hará que un nodo

no permita crear en su interior pods que cumplan esa característica o condición.

- **--node-ip**: Es la dirección IP del nodo.
- **--node-labels**: Son etiquetas que se añadirán al nodo cuando se registre en un clúster.
- **--node-status-update-frequency**: Es la frecuencia con la que el nodo manda información de estado al clúster, los latidos de los que hablábamos en este punto.

- **Administración manual** de nodos: En este caso, hay que levantar **kubelet** con el atributo --register-node en *false* para poder crear nodos manualmente. Lo que sí se puede hacer de forma manual sin cambiar este atributo es modificarlos (crear etiquetas en el nodo o marcarlos como no planificables para que no se puedan planificar pods en ese nodo). En este punto, cabe destacar que los pods creados por el controlador *DaemonSet* ignoran el atributo no planificable de los nodos.

Comunicación Nodo-Maestro

Hay distintos modos de comunicación entre el nodo maestro y el clúster de K8s. El **nodo maestro es el apiserver** y la intención es que el usuario pueda proteger sus configuraciones de red personalizando sus instalaciones para que el clúster sea seguro aunque la red no sea todo lo segura que debería.

Si pensamos en el ámbito local o en un servidor propio, generalmente no tiene sentido imaginar una red poco segura. Sin embargo, si extrapolamos esto a la nube, donde las IP pueden ser públicas, esto empieza a cobrar más sentido.

A continuación vamos a ver la comunicación en ambos sentidos, de clúster a maestro y viceversa.

Comunicación Clúster - Maestro

Cualquier canal de comunicación entre el clúster y el nodo maestro termina en el apiserver porque es el único componente que puede exponer servicios remotos. Si no cambiamos la configuración por defecto, el apiserver escucha peticiones remotas a través de HTTPS y además, habilita la **autenticación** de los clientes que realizan las peticiones. Además, se puede

configurar la **autorización** que es muy recomendada si se aceptan peticiones anónimas o tokens de cuenta de servicio.

Para que un nodo pueda conectarse con el apiserver de forma segura, debe tener el **certificado raíz público** del clúster y unas credenciales válidas. Los pods del interior de cada nodo se conectan con el apiserver a través de una cuenta de servicio. Así, K8s inserta en el pod el certificado y un **bearer token** válido para que se autentique correctamente.

El resultado de estas configuraciones es que las conexiones desde el clúster al nodo maestro sean seguras aunque las redes sean públicas o inseguras.

Comunicación Maestro - Clúster

El Nodo Maestro o apiserver se puede conectar al clúster por dos caminos. Por un lado, a través de cada proceso de kubelet que se ejecuta en los nodos del clúster y por otro lado, a través del proxy del apiserver.

Nodo Maestro - kubelet

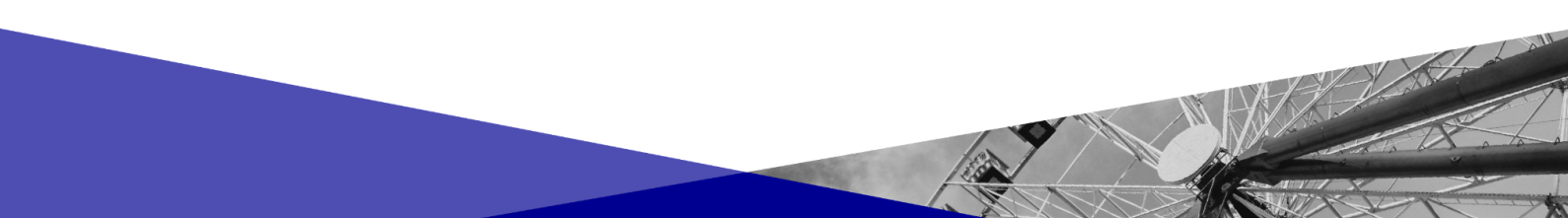
Cuando el apiserver conecta con el proceso de kubelet puede ser por tres motivos: recoger entradas del log de los pods, conectar con **kubectl** con pods en ejecución de algún nodo, facilitar la funcionalidad del **port-forwarding** del kubelet.

El port-forwarding consiste en redirigir conexiones para permitir que los dispositivos remotos se conecten de forma directa con el nodo, designando puertos específicos para esa conexión.

Por defecto, el nodo maestro no comprueba el certificado del kubelet, esto hace que la conexión sea vulnerable a **ataques** del tipo **man-in-the-middle.** Este tipo de ataque consiste en interceptar los mensajes entre los dispositivos que se comunican. Es un ataque difícil de detectar y además, el atacante puede retener mensajes para que no lleguen al receptor, modificarlos o simplemente recoger información.

Para proteger nuestros nodos de este tipo de ataque se utiliza el atributo **--kubelet-certificate-authority**, que realiza el cifrado raíz que se menciona en el punto anterior.

Cuando no es posible establecer un certificado raíz, se utiliza un túnel SSH para protegerse en redes inseguras.



Nodo Maestro - nodos, pods, servicios

Si usamos la configuración por defecto, las conexiones entre el apiserver y los nodos, pods o servicios se realizan con el protocolo **HTTP**. Si queremos que la **conexión** sea **segura**, debemos utilizar **HTTPS**. Esto aporta algo más de seguridad porque la conexión está encriptada pero **no garantiza la integridad**, ya que los receptores no comprueban los certificados ni dan credenciales de cliente.

Almacenamiento

Los **pods** de K8s son **efímeros**. Por ello, la información que procesan o guardan no es persistente. Sin embargo, existen multitud de aplicaciones que no solo se benefician, sino que necesitan persistencia para poder alcanzar sus metas.

Volúmenes

Aquí es donde entran en juego los **volúmenes de K8s**. Recordemos el concepto de volúmen, visto en el capítulo de Docker: mecanismo que permite mantener el estado de los contenedores, pudiendo consumir o generar ficheros para persistir el estado del contenedor.

K8s proporciona su propia implementación de volúmenes, que se pueden agrupar en tres categorías.

Efímero

Estos volúmenes comparten el tiempo de vida del pod que los contiene. Es decir, el volúmen se crea con el pod, y cuando éste deja de existir, K8s destruye el volumen con él.

Son muy útiles para aplicaciones que no tienen necesidad de persistencia. Por ejemplo, aplicaciones de cómputo donde los datos almacenados sólo son útiles para la ejecución actual y de éste único pod.

Sin embargo, este tipo de volúmenes es incompatible con pods que necesiten almacenar un estado a prueba de fallos. Sabemos que los pods fallidos se reinician, y en el caso de los volúmenes efímeros, estos se reinician con el pod, **perdiendo los datos almacenados**.

emptyDir

El tipo de volumen más básico de K8s se caracteriza por cumplir estrictamente la definición de volúmen efímero: nace y muere con el pod al que ha sido asignado. Como su nombre indica, se asigna vacío al contenedor, que tiene permisos de lectura y escritura sobre el volúmen.

El volúmen ***emptyDir* se utiliza como espacio temporal**, para operaciones sobre archivos recuperados de un servidor o para operaciones basadas en disco, y para marcar cálculos útiles para la recuperación de fallos.

Aunque hemos mencionado que los volúmenes efímeros mueren con el pod, debemos recordar que la **eliminación se lleva a cabo cuando el pod se elimina del nodo**. Es decir, cuando un contenedor del pod colapsa, los datos de un volumen *emptyDir* siguen estando disponibles y seguros para facilitar una recuperación o la continuación del trabajo. En el siguiente punto veremos un ejemplo de este tipo de volumen.

Dependiendo del entorno, podemos encontrar estos volúmenes montados sobre cualquier medio de almacenamiento que respalde el nodo, ya sean discos duros, almacenamiento en red o incluso en memoria, en cuyo caso se monta un **tmpfs** (sistema de ficheros respaldado por RAM). Este es el medio de almacenamiento más rápido, pero debemos tener en cuenta que está **limitado por la memoria del contenedor** y que, además, pierde todo su contenido cuando el nodo se reinicia.

Persistente

Este tipo de volumen es un recurso del clúster que, a diferencia de los volúmenes efímeros, no se destruye cuando el pod es eliminado del nodo. Es decir, independientemente del tipo de volumen persistente implementado, **los datos se conservan** cuando se producen reinicios de pods.

Los volúmenes persistentes o ***PV* (*PersistentVolume*),** son provistos a los nodos de forma manual, por un administrador, o automática, mediante clases de almacenamiento **(*StorageClasses*).** Un *PV* es un recurso de la misma naturaleza que un nodo dentro del clúster, pudiendo implementar almacenamiento NFS, iSCSI o basado en la nube, entre otros.

Al igual que el resto de recursos de un clúster, K8s provee un mecanismo para que otro recurso solicite almacenamiento persistente, el ***PersistentVolumeClaim (PVC).*** Para entenderlo mejor, haremos una analogía con los pods. Un pod consume recursos de un nodo, de la misma forma que un PVC consume recursos de un PV. De esta forma, los pods obtienen la capacidad de solicitar diferentes tipos de *Claims*, que varían en tamaño y modos de acceso:

- **ReadWriteOnce (RWO):** El volumen se monta en un único nodo con permisos de lectura y escritura. Permite a otros pods del mismo nodo acceder al mismo volumen.
- **ReadOnlyMany (ROX):** Se crea un volumen de solo lectura que puede ser montado en varios nodos.
- **ReadWriteMany (RWX):** Se crea un volumen de lectura y escritura que puede ser montado en varios nodos.
- **ReadWriteOncePod (RWOP):** El volumen creado se monta con permisos de lectura y escritura en un único pod. Es decir, ningún otro pod, independientemente del nodo, puede acceder a dicho PVC.

Es importante recordar que, aunque un volumen soporta varios modos de acceso, estos **no pueden ser simultáneos**. Si estás interesado en conocer los tipos de volúmenes persistentes te recomendamos consultar esta página.

Proyectado

Un volumen proyectado **mapea** diferentes fuentes de volumen en el mismo directorio. Sin embargo, no todas las fuentes se pueden mapear, por lo que debemos mantener una lista actualizada de fuentes mapeables. En nuestro caso, para la versión 1.25 estable de K8s:

- **secret**: Un volumen utilizado para la transferencia de información sensible, como contraseñas o API tokens, a los pods, sin guardarlos en claro en la definición del objeto.
- **downwardAPI**: Es un mecanismo que expone la información del pod o de un contenedor al código que está ejecutando. Montando un volumen *downwardAPI*, ofrecemos a los pods del nodo acceso en modo sólo lectura a los datos expuestos.
- **configMap**: *ConfigMap* provee un mecanismo para inyectar datos de configuración en los pods. Para acceder a ellos se referencia un volumen del tipo *configMap*, y puede ser consumido por las aplicaciones contenerizadas.
- **serviceAccountToken**: Los *ServiceAccountToken* son tokens de autenticación de tipo *Bearer* que se crean automáticamente y ofrecen identidades válidas tanto dentro como fuera del clúster, que permiten interactuar con la API. El volumen *serviceAccountToken* nos permite inyectar esta información en una ruta específica del pod.

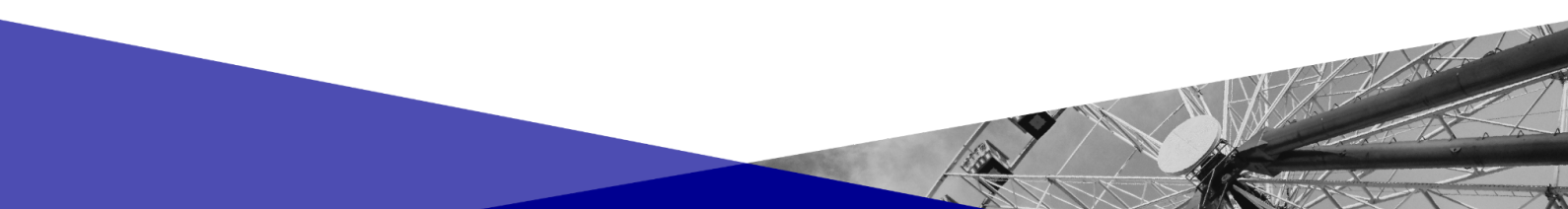
Todas las fuentes necesitan compartir el espacio de nombres del pod al que se inyectan, además de cumplir las restricciones del *all-in-one volume design*.

Configuración de volúmenes

Una vez que hemos comprendido la parte teórica de los volúmenes, es el momento de **configurar un volumen** como almacenamiento de un pod.

Volumen efímero

A continuación, vamos a iniciar la creación de un **volumen efímero** de tipo **EmptyDir** para un pod. En primer lugar, necesitas tener configurado un clúster.



Partiendo del clúster de Minikube, vamos a ejecutar este comando para crear un pod con este contenido:

```
kubectl apply -f https://k8s.io/examples/pods/storage/redis.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
```

El pod que hemos creado tiene un único contenedor llamado “redis”. Dentro de él creamos el volumen “**redis-storage**” de tipo **emptyDir** (directorio vacío). Este volumen existirá mientras exista el pod, aunque el contenedor se destruya.

Ahora que el volumen está creado, lo que vamos a hacer es comprobar que aunque el contenedor se destruya, mientras el pod permanezca, el volumen se mantiene.

Para comprobar que el pod se está ejecutando utilizamos el comando **get** que ya hemos usado en otras ocasiones en esta guía.

```
kubectl get pod redis --watch
```

Ahora, abre una nueva terminal, sin cerrar el que estás usando, y conéctate con el contenedor que has creado:

```
kubectl exec -it redis -- /bin/bash
```

Dentro del terminal, ve a la carpeta /data/redis y crea un fichero similar a este:

```
root@redis:/data# cd /data/redis
root@redis:/data/redis# echo Hello > test-file
```

Y ahora vamos a listar los procesos en ejecución desde esta misma terminal:

```
root@redis:/data/redis# apt-get update
root@redis:/data/redis# apt-get install procps
root@redis:/data/redis# ps aux
```

```
root@redis:/data/redis# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
redis          1  0.2  0.3  53608  8088 ?        Ssl  10:21   0:01 redis-server *:6379
root          21  0.0  0.1   4096  3432 pts/0    Ss   10:31   0:00 /bin/bash
root         351  0.0  0.1   6696  2832 pts/0    R+   10:33   0:00 ps aux
```

Finalmente, matamos el proceso de redis que en este caso tiene como PID 1:

```
root@redis:/data/redis# kill 1
```

Y en la otra terminal veremos lo siguiente:

```
vagrant@localhost:~$ kubectl get pod redis --watch
NAME    READY   STATUS    RESTARTS   AGE
redis   1/1     Running   0          2m23s
redis   0/1     Completed   0          12m
redis   1/1     Running     1 (3s ago)   12m
```

Por la política **RestartPolicy** establecida en *Always*, el contenedor se ha vuelto a crear cuando lo hemos destruido. Lo que nos interesa en este punto es comprobar que el archivo que habíamos creado en el volumen del pod, antes de destruir el contenedor, aún existe. Para ello, volvemos a entrar desde la otra terminal igual que antes y comprobamos si el archivo está en la carpeta.

```
kubectl exec -it redis -- /bin/bash
root@redis:/data# cd /data/redis
root@redis:/data/redis# ls
```

```
root@redis:/data/redis# ls
test-file
root@redis:/data/redis#
```

Los resultados son los esperados, ya que el archivo “test-file” sigue existiendo. Si lo hubiésemos creado en el contenedor en lugar de hacerlo en el volúmen, habría desaparecido al matar el proceso del contenedor.

Volumen persistente

Ahora, vamos a configurar un volumen persistente para el almacenamiento de un pod. Para este tipo de volumen, necesitamos crear un archivo de configuración del volumen, un **PersistentVolumeClaim** para configurar el almacenamiento del volumen, y por último configurar el pod para que utilice este volumen.

Comenzamos abriendo una terminal dentro del clúster y creamos el siguiente directorio y archivo dentro de él. Después salimos de la terminal:

```
minikube ssh
sudo mkdir /mnt/data
sudo sh -c “echo ‘Testing a Persistent Volume’ > /mnt/data/index.html”
exit
```

Para este ejemplo, vamos a crear un volumen persistente de tipo *hostPath*. Esto solo se permite en K8s para un clúster de un solo nodo donde emulamos un disco persistente de algún proveedor de servicios en la nube, recursos de NFS, etc. Como estamos haciendo pruebas de configuración, para crear los archivos necesarios es suficiente este ejemplo.

Ahora vamos a configurar el volumen con un archivo similar a este, al que llamaremos “pv-volume.yaml”:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Este archivo de configuración indica el directorio (“/mnt/data”) donde está el volumen persistente del nodo del clúster. Además, indica que el tamaño son 10 Gi y que el modo de acceso es ReadWriteOnce. El StorageClassName se define como “Manual” para que podamos vincular las solicitudes del PersistentVolumeClaim con este volumen.

Ahora vamos a crear el volumen con este archivo. Si no quieres copiarlo, como es un ejemplo de la documentación de K8s, puedes ejecutar el siguiente comando:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-volume.yaml
```

Si no, basta con sustituir la url por la ruta y el nombre que hayas utilizado para crear el archivo.

Para comprobar que lo hemos creado correctamente, ejecutamos:

```
kubectl get pv task-pv-volume
```

```
vagrant@localhost:~$ kubectl get pv task-pv-volume
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE
task-pv-volume   10Gi       RWO            Retain           Available           manual                  58s
```

El siguiente paso es configurar el archivo del PersistentVolumeClaim, al que llamaremos “pv-claim.yaml”.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Este Claim solicita 3 Gi de almacenamiento físico que permita lectura y escritura solo para un nodo.

Lo creamos igual que antes:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-claim.yaml
```

Y ahora volvemos a comprobar el estado del volumen persistente:

```
kubectl get pv task-pv-volume
```

```
vagrant@localhost:~$ kubectl get pv task-pv-volume
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                   STORAGECLASS   REASON   AGE
task-pv-volume   10Gi       RWO            Retain           Bound    default/task-pv-claim   manual                  7m12s
```

En el campo “CLAIM” vemos como se ha vinculado correctamente el PersistentVolumeClaim al volumen que habíamos creado previamente.

El último paso es crear un Pod para comprobar el funcionamiento de todo en conjunto. Para crear el pod vamos a copiar este archivo de configuración “pv-pod.yaml”:

```
apiVersion: v1
kind: Pod
metadata:
 name: task-pv-pod
spec:
 volumes:
   - name: task-pv-storage
     persistentVolumeClaim:
       claimName: task-pv-claim
 containers:
   - name: task-pv-container
     image: nginx
     ports:
       - containerPort: 80
         name: "http-server"
     volumeMounts:
       - mountPath: "/usr/share/nginx/html"
         name: task-pv-storage
```

Desde el pod no hacemos referencia al volumen persistente en ningún momento, sino al *Claim*. El pod trata al *Claim* como si fuese el volumen, puesto que este es el que solicita al volumen el acceso de lectura y escritura para que el pod lo utilice.

Creamos el pod y comprobamos que se está ejecutando su contenedor:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-pod.yaml
kubectl get pod task-pv-pod
```

Por último, vamos a abrir una terminal dentro de este contenedor que hemos creado en el pod para comprobar que está utilizando el volumen

persistente que le hemos asignado:

```
kubectl exec -it task-pv-pod -- /bin/bash
apt update
apt install curl
curl http://localhost/
```

```
root@task-pv-pod:/# curl http://localhost/
Testing a persistent volume
root@task-pv-pod:/#
```

La salida es el contenido del archivo “index.html” que hemos creado al inicio de este ejemplo, por lo tanto, hemos configurado correctamente el volumen persistente.

Volumen proyectado

Por último, vamos a ver un ejemplo del archivo de configuración de un pod con un **volumen proyectado** que contiene dos **secretos.** En este caso, no vamos a realizar el ejemplo completo porque es similar al del volumen efímero que hemos realizado previamente. A continuación se muestra el archivo de configuración de un pod para utilizar este tipo de volumen.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox:1.28
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - secret:
```

```
          name: mysecret2
          items:
            - key: password
              path: my-group/my-password
              mode: 511
```

Si analizamos el archivo que hemos creado, tenemos un pod que se llama “volume-test” que tiene un contenedor “container-test”. Este contenedor utiliza el volumen “all-in-one” que está definido debajo como volumen proyectado. Este volumen tiene dos fuentes de tipo **secret**, aunque podría tener más tipos, como un **downwardAPI** o un **configMap**.

En el siguiente ejemplo vemos cómo se incluirían en el archivo de configuración del pod estas fuentes dentro de un volumen proyectado:

```
...
...
  volumes:
  - name: all-in-one
    projected:
      sources:
      - downwardAPI:
          items:
            - path: "labels"
              fieldRef:
                fieldPath: metadata.labels
            - path: "cpu_limit"
              resourceFieldRef:
                containerName: container-test
                resource: limits.cpu
      - configMap:
          name: myconfigmap
          items:
            - key: config
              path: my-group/my-config
```

Límites

Aunque K8s permite varios tipos de volúmenes, los distintos proveedores de almacenamiento en la nube tienen sus propias restricciones acerca de cuántos volúmenes como máximo puede montar un nodo. No ser consciente de estos límites podría conllevar fallos de diseño en nuestras aplicaciones, ya que los pods de un nodo podrían quedarse atascados esperando una PVC.

K8s puede determinar automáticamente el tipo de nodo administrado siempre que se utilicen plugins nativos de volumen, y aplicar la cantidad máxima adecuada para las necesidades de ese nodo, ampliando o reduciendo el máximo de volúmenes según el proveedor y los requisitos de nuestros nodos.

Servicios, balanceo de carga y redes

Modelo de red de Kubernetes

En K8s, cada pod tiene su propia IP única en todo el clúster. Esto significa que no es necesario definir explícitamente vínculos entre los pods, y casi nunca tendremos que mapear los puertos de los contenedores con los del host.

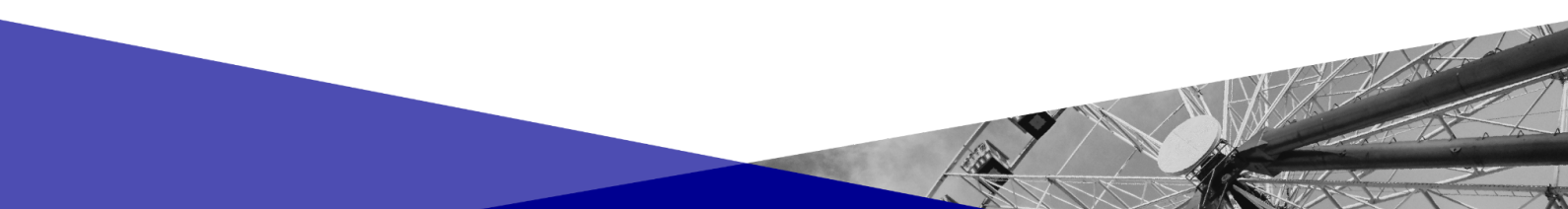
Además, que cada pod tenga su propia dirección IP implica que los contenedores que se encuentren dentro de un mismo pod comparten su espacio de nombres, incluida su dirección IP y MAC. Por lo tanto, dichos contenedores pueden comunicarse entre sí a través de localhost. A su vez, también hay que tener en cuenta que estos contenedores tendrán que coordinarse para el uso de puertos, nada diferente de los procesos en una máquina virtual. A esto se lo conoce como **modelo “IP por pod”**.

Esta característica de K8s crea un modelo limpio, donde los pods son tratados más como máquinas virtuales o físicas desde la perspectiva de asignación de puertos, nombrado, descubrimiento de servicios, balanceo de carga, configuración de aplicaciones y migración.

Este modelo no es solo menos complejo en general, sino que principalmente es compatible con el objetivo de K8s de conseguir la **portabilidad de aplicaciones** que corrían en máquinas virtuales a contenedores.

Teniendo esto en cuenta, cualquier implementación de la red de K8s debe cumplir con las siguientes características:

- Los contenedores dentro de un pod usan la **red para comunicarse** entre sí a través de **localhost**.
- Los pods pueden comunicarse entre sí, independientemente de si se encuentran en el mismo nodo o no.
- Los agentes de nodos, como **kubelet**, pueden comunicarse con todos los pods del nodo en el que se encuentren.



- Los **servicios** permiten exponer una aplicación que se ejecuta en un pod para que sea accesible desde fuera del clúster.
- También se pueden crear **servicios solo para consumo interno** del clúster.

Servicios de red

Como sabemos, los pods no son duraderos. Cuando un nodo muere, sus pods terminan con él. En caso de que hubiéramos creado un controlador de réplicas, este creará nuevos pods con IPs diferentes para que las aplicaciones que estuvieran ejecutándose en los otros pods, lo sigan haciendo en los nuevos.

Imagina que tenemos un conjunto de pods ejecutando una aplicación backend, los cuales sirven ciertas funcionalidades a otro conjunto de pods ejecutando el frontend. Si alguno de los pods del backend muere, aunque este sea recreado por el controlador, el frontend no tendrá manera de saber cuál es la nueva IP del pod que se acaba de crear, impidiendo así la comunicación con el backend.

Es aquí donde entran en juego los servicios para solucionar este tipo de problemas. Un servicio es una abstracción que define un conjunto lógico de pods que ejecutan la misma aplicación y una política para poder acceder a ellos. Los pods a los que apunta el servicio son normalmente determinados por un **selector de etiquetas** (***LabelSelector***). Las etiquetas son un par clave/valor adjunto a los objetos.

Cuando creamos un servicio, K8s le asigna una dirección IP única, también conocida como ***clusterIP***, la cual no cambiará mientras el servicio siga existiendo. Con K8s no necesitamos modificar las aplicaciones para que usen un sistema de descubrimiento de servicios ajeno, ya que los pods tienen su propia dirección IP y se les asigna un nombre DNS para cada conjunto de réplicas de pods, pudiendo balancear la carga entre ellos.

Definiendo un servicio

Al igual que el resto de objetos de la API, para definir un servicio tenemos que crear un archivo YAML que contenga todas las especificaciones. En este caso, se creará un servicio llamado “mi-servicio” que apunta al puerto 9376

de cualquier pod que tenga la etiqueta “app=MyApp”:

```
apiVersion: v1
kind: Service
metadata:
  name: mi-servicio
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Los parámetros necesarios para especificar un servicio son:

- **.kind**: Indicamos que es un servicio (*Service*).
- **.spec.selector**: En él indicamos la etiqueta a través de la cual encontrar los pods que queremos incluir en el servicio.
- **.spec.ports.port**: Es el puerto abstracto que se le asigna al servicio para que otros pods puedan acceder al él.
- **.spec.port.targetPort**: Es el puerto en el que los contenedores aceptan el tráfico.

Para crear el servicio, basta con ponerle un nombre, por ejemplo servicio.yaml, y ejecutar el siguiente comando:

```
kubectl apply -f {filePath}/servicio.yaml
```

También podemos crear un servicio utilizando el comando *expose* y un *deployment* que hayamos creado previamente:

```
kubectl expose deployment hello-node
```

Servicios sin selectores

Los servicios normalmente abstraen el acceso al conjunto de pods que engloban, los cuales han sido seleccionados por un selector. Sin embargo, también puede haber ocasiones en las que realicen otras funciones, para las que no necesitaríamos definir el selector, por ejemplo:

- Tenemos un clúster de base de datos en producción, pero para el entorno de pruebas queremos tener nuestras propias bases de datos.

- Queremos apuntar nuestro servicio a un servicio externo de otro clúster diferente.
- Estamos migrando nuestras aplicaciones a K8s y, mientras lo llevamos a cabo, solo estamos ejecutando una parte en K8s.

Un ejemplo de definición de un servicio sin selector sería el siguiente:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Como este servicio no tiene selector, el correspondiente objeto *Endpoint* no se crea de manera automática. Podemos crearlo de forma manual, mapeando el servicio con la dirección de red y puerto en el que se está ejecutando:

```
apiVersion: v1
kind: Endpoints
metadata:
  # El nombre debería coincidir con el nombre del servicio
  name: my-service
subsets:
  - addresses:
      - ip: 192.0.2.42
    ports:
      - port: 9376
```

Servicio *Headless*

En algunas ocasiones es posible que no necesitemos balancear las cargas entre los diferentes pods o tener una IP única para todos ellos. En este caso, podemos crear un servicio *Headless*.

Podemos usar este servicio para crear una interfaz con otros sistemas de descubrimiento de servicios, sin necesidad de estar sujetos a la implementación de K8s.

Para crear este tipo de servicios, basta con especificar el valor "None" para el campo ***.spec.clusterIP***. De esta manera, el kube-proxy no se hará cargo del servicio, por lo que no se le asignará una *clusterIP* y no habrá balanceo de cargas.

Publicar un servicio

Por defecto, los pods son solo accesibles por su dirección IP interna dentro del clúster. Pero en determinadas ocasiones necesitaremos exponer algunas partes de nuestra aplicación (como es el caso del frontend) a una IP externa fuera del clúster. Es por ello que existen diferentes tipos de servicios. Para utilizar uno u otro, a la hora de definir la especificación del servicio en un archivo YAML, debemos cambiar el valor del campo ***.spec.type*** utilizando uno de los siguientes valores:

- **ClusterIP**: Es el valor por defecto. Expone el servicio en una IP interna del clúster. Por lo tanto, este es solo accesible desde dentro del clúster.
- **NodePort**: Expone el servicio en un puerto estático en cada IP del nodo (*NodePort*). Además, se crea automáticamente un servicio de tipo *ClusterIP*, al cual enruta el *NodePort*. Por lo tanto, podremos acceder al servicio desde fuera del clúster haciendo una petición a {ClusterIP}:{NodePort}.
- **LoadBalancer**: Expone el servicio externamente usando un balanceador de carga del proveedor en la nube. También se crean los servicios de tipo *ClusterIP* y *NodePort* de manera automática, a los cuales apuntará el balanceador.
- **ExternalName**: Mapea el servicio con el valor del campo **.spec.*externalName*,** que contiene un *CNAME*.

Exponer un pod como servicio

En este ejemplo vamos a exponer públicamente un pod como un servicio de tipo *NodePort*. Para ello, creamos el *Deployment* que crea el pod y ejecutamos el siguiente comando:

```
kubectl create deployment hello-node --image=k8s.gcr.io/echoserver:1.4
kubectl expose deployment hello-node --type=NodePort --port=8080
```

Para comprobar que se ha creado correctamente, obtenemos los servicios con el comando:

```
kubectl get services
```

Obteniendo una respuesta similar a la siguiente:

```
jcmoreno@localhost:~$ kubectl get services
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
hello-node   NodePort    10.103.15.83   <none>        8080:30236/TCP   9s
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP          32d
```

Podemos observar que actualmente hay dos servicios: el que acabamos de crear (hello-node), de tipo *NodePort,* y uno que crea automáticamente Minikube al iniciarse (Kubernetes), de tipo *ClusterIP*.

También podemos ejecutar el comando *describe* para obtener todos los detalles del servicio:

```
kubectl describe service hello-node
```

A continuación, para comprobar que efectivamente podemos acceder al pod desde fuera del clúster, ejecutaremos el siguiente comando:

```
curl $(minikube service hello-node --url)
```

Y como podemos observar, obtenemos una respuesta, aunque esta no tenga ningún tipo de mensaje o contenido:

```
jcmoreno@localhost:~$ curl $(minikube service hello-node --url)
CLIENT VALUES:
client_address=172.17.0.1
command=GET
real path=/
query=nil
request_version=1.1
request_uri=http://192.168.49.2:8080/

SERVER VALUES:
server_version=nginx: 1.10.0 - lua: 10001

HEADERS RECEIVED:
accept=*/*
host=192.168.49.2:30236
user-agent=curl/7.58.0
BODY:
```

Ahora vamos a proceder a eliminar el servicio que hemos creado:

```
kubectl delete service hello-node
```

Si ahora intentamos conectarnos de nuevo desde fuera del clúster:

```
curl $(minikube service hello-node --url)
```

Veremos que no podemos conectarnos, puesto que hemos eliminado el servicio que exponía el pod públicamente:

```
jcmoreno@localhost:~$ curl $(minikube service hello-node --url)
❌  Exiting due to SVC_NOT_FOUND: Service 'hello-node' was not found in 'default' namespace.
You may select another namespace by using 'minikube service hello-node -n <namespace>'. Or list out
 all the services using 'minikube service list'
curl: try 'curl --help' or 'curl --manual' for more information
```

Sin embargo, si accedemos desde dentro del clúster con los siguientes comandos, sí que obtendremos respuesta:

```
export POD_NAME=$(kubectl get pods -o go-template --template \
'{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')
echo POD_NAME: $POD_NAME
kubectl exec -ti $POD_NAME -- curl localhost:8080
```

```
jcmoreno@localhost:~$ kubectl exec -ti $POD_NAME -- curl localhost:8080
CLIENT VALUES:
client_address=127.0.0.1
command=GET
real path=/
query=nil
request_version=1.1
request_uri=http://localhost:8080/

SERVER VALUES:
server_version=nginx: 1.10.0 - lua: 10001

HEADERS RECEIVED:
accept=*/*
host=localhost:8080
user-agent=curl/7.47.0
BODY:
```

Enrutamiento

Como se ha mencionado, los pods de un clúster tienen su propia dirección IP y MAC. Es decir, pueden comunicarse entre sí a través de la red de K8s. Aquí es donde entra en juego el **enrutamiento basado en topología**, que dirige el tráfico basándose en la topología que forman los nodos del clúster. Anteriormente, se utilizó el *Topology Aware Traffic Routing with Topology Keys*, que por medio de un ***Service Topology***, hacía de puerta de enlace, enrutando el tráfico según las *Topology Keys* de cada paquete.

A partir de la versión 1.21 de K8s, se usan los ***Topology Aware Hints*** para llevar a cabo el **enrutamiento dinámico**, incluyendo sugerencias sobre cómo los clientes deberían consumir los *endpoints*. Este enrutamiento nos permite reducir costes y mejorar el rendimiento de la red gracias al mecanismo que nos permite mantener el tráfico cerca de la zona que lo generó.

La administración de *endpoints* y el control de tráfico pasan a manos de dos componentes: el controlador ***EndpointSlice***, que, según la topología de los nodos, crea *hints* para ubicar cada *endpoint* en una zona; y el componente ***kube-proxy***, que consume esas "pistas" o *hints* para manipular el enrutamiento del tráfico, favoreciendo los *endpoints* más cercanos.

Controlador EndpointSlice

Como se ha comentado, este controlador es el responsable de poblar la sección de *hints* en los objetos *EndpointSlice*. De manera que aloja una cantidad de *endpoints* a cada zona, proporcional al tráfico que circula por ella.

Esta proporcionalidad se basa en la cantidad de núcleos CPU disponibles para los nodos de dicha zona, teniendo preferencia las zonas con mayor disponibilidad. A continuación podemos observar un ejemplo de objeto *EndpointSlice* poblado en función de la topología de nodos de la aplicación que creamos en el apartado anterior:

```
apiVersion: v1
kind: EndpointSlice
metadata:
  name: example-hints
  labels:
    service-name: example-svc
ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
endpoints:
  - addresses:
      - "10.1.2.3"
    conditions:
      ready: true
    hostname: pod-1
```

```
    zone: zone-a
  hints:
    forZones:
      - name: "zone-a"
```

Observa que el objeto adquiere un campo ***.endpoints*** que se rellena con diferentes hints. En nuestro ejemplo, el tráfico dirigido al *pod-1* tendrá favorecido su enrutamiento en la zona *zone-a*.

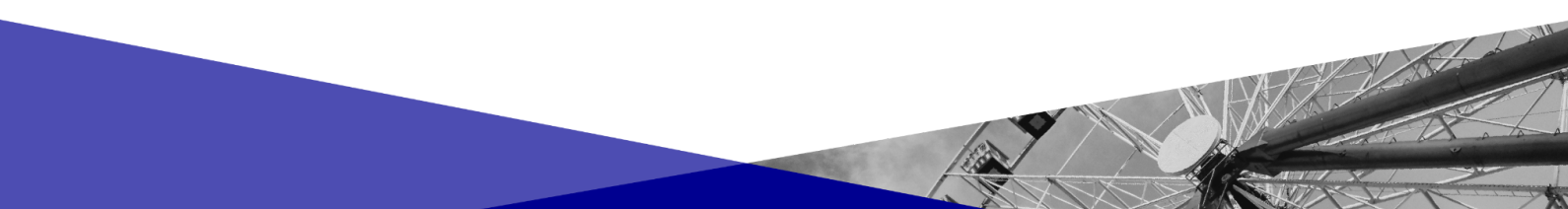
kube-proxy

Este componente es el responsable de enrutar el tráfico basándose en las *hints* definidas por el controlador *EndpointSlice*. De este modo, *kube-proxy* distribuye los *endpoints* en zonas más cercanas o lejanas al origen de su tráfico, en función de la topología y una distribución lo suficientemente equitativa.

Excepciones y limitaciones

En situaciones ideales, los componentes mencionados harán uso de las *hints* para optimizar el tráfico y el coste del mismo. Sin embargo, existen situaciones en las que *kube-proxy* elegirá *endpoints* de cualquier zona del cluster, ignorando las *hints*. Estas situaciones son:

- **Endpoints insuficientes**: Si existen menos endpoints que zonas en un cluster.
- **No se alcanza la distribución equitativa**: Si no se puede garantizar una distribución balanceada de *endpoints*, el uso de *hints* podría dar lugar a sobrecarga de trabajo en ciertos endpoints y defecto de carga en otros.
- **Uno o más nodos tienen información incompleta**: Para poder configurar *hints* basados en la topología de los nodos, estos deben tener completos los datos sobre su zona y los núcleos disponibles para alojar.
- **Uno o más endpoints no tienen hints asignados**: En estos casos, *kube-proxy* asume que la aplicación está pasando por una transición hacia o desde *Topology Aware Hints*. Es decir, si alguno de los nodos no tiene *hint*, *kube-proxy* ignora este enrutamiento.



- **Una zona no aparece en las hints**: Si añadimos una nueva zona a un clúster en funcionamiento, y ninguno de sus *endpoints* tiene un *hint* apuntando a esa zona, *kube-proxy* ignora las *hints*.

Para activar este enrutamiento en un *Service*, basta con darle al campo ***topology-aware-hints*** el valor ***auto***. Esto lo podemos hacer en la definición del objeto o mediante el comando:

```
kubectl annotate svc mi-servicio \
service.kubernetes.io/topology-aware-hints=auto
```

O también podemos activarlo mediante "feature gates", set de pares clave-valor que describen las características de objetos K8s. En este caso necesitaríamos establecer:

```
--feature-gates=...,TopologyAwareHints=true
```

Sin embargo, no siempre obtenemos el resultado esperado con *Topology Aware Hints*. Por ejemplo, algunas limitaciones son:

- Cuando ***externalTrafficPolicy*** o ***internalTrafficPolicy*** estén configuradas como ***Local***. Esta restricción sólo se cumple para el mismo servicio, las políticas de otros servicios no afectan a este.
- Si diseñamos servicios que tienen un tráfico desproporcionado en ciertas zonas.
- El controlador *EndpointSlice* ignora los nodos que no están listos para llevar a cabo el cálculo de proporciones, y no tiene en cuenta tolerancias durante ese cálculo. Esto puede provocar consecuencias no intencionadas en el enrutamiento y la planificación.
- Si el tráfico se origina en una única zona, sólo esos endpoints lo deberían tratar. Dificultando el autoescalado o provocando que éste cree nuevos pods en otras zonas, a las que el tráfico no llegará.

DNS

El Sistema de Nombres de Dominio (DNS) permite asociar las direcciones IP con nombres de dominio sencillos de recordar. K8s establece el uso de registros DNS para servicios y pods. Gracias a esto, se puede conectar con los Servicios con nombres específicos en lugar de con direcciones IP.

La mayor parte de clústeres implementa automáticamente un servicio DNS para descubrir servicios debido a que a cada servicio de un clúster se le asigna un nombre.

Espacios de nombres

Las consultas DNS dependen del espacio de nombres que indique el pod que realiza dicha consulta. Cuando no se especifica un espacio de nombres, las consultas están limitadas al propio espacio de nombres del pod.

Si queremos hacer una consulta desde un pod a un servicio que está en su mismo *namespace* o espacio de nombres, con poner el nombre del servicio en la consulta será suficiente para que lo encuentre. Sin embargo, si el servicio está en un *namespace* diferente, será necesario indicar de forma explícita dónde se encuentra. La forma de hacerlo es separando el nombre del *namespace* y el del servicio con un punto. Por ejemplo, el servicio "test-service" que está en el espacio de nombres "tests" se consulta así: "tests.test-service".

El DNS de K8s se controla con un archivo de configuración donde se define su comportamiento. A lo largo de las versiones de K8s, los servidores DNS han ido cambiando. Hasta la versión 1.11 usaban "**kube-dns**" y a partir de ahí surge "**coreDNS**".

El servicio de **kube-dns** está formado por tres contenedores que se ejecutan en un pod **kube-dns** y tienen un espacio de nombres **kube-system.** Los contenedores se encargan de realizar las consultas DNS, almacenar las respuestas y generar informes y métricas sobre la salud del servicio.

CoreDNS es una mejora del anterior, ya que puede utilizarse de forma predeterminada como servicio DNS. Tiene la misma funcionalidad que kube-dns con algunas mejoras. En este caso, es un único contenedor que resuelve y almacena las consultas DNS, y realiza métricas y análisis de salud. Además, añade mejoras de seguridad y corrige algunos errores de la

versión anterior.

Registros DNS

Hay dos tipos de objetos de K8s que implementan registros DNS, se trata de los servicios y los pods. A continuación, vamos a ver cómo se configuran los registros DNS para estos objetos.

Servicios

Para los servicios existen dos tipos de registros DNS:

- **Registros A/AAAA.** Los servicios normales reciben un registro DNS o AAAA en función de la IP de su clúster. Sin embargo, los ***servicios headless*** también reciben un registro DNS o AAAA pero no se resuelve en la IP del clúster (porque no tienen), sino en el conjunto de IPs de los pods seleccionados por el servicio en cuestión. Esto es así porque los clientes de este tipo de servicios normalmente consumen todo el conjunto.

  Los nombres de este tipo se definen según el siguiente ejemplo: “mi-servicio.espacio-de-nombres.servicio.dominio-cluster.ejemplo”

- **Registros SRV.** Estos registros se crean para los puertos con nombre de los servicios normales o *headless*. Para servicios normales, se resuelve con el número de puerto y el nombre del dominio. Para servicios *headless* se resuelven varias respuestas, una por cada pod del servicio. En estos casos cada respuesta contiene el número de puerto y el nombre de dominio que se autogenera para cada caso.

  En este caso, para servicios normales, los nombres son de esta forma:
  “nombre-puerto.protocolo-puerto.mi-servicio.mi-espacio-de-nombres.servicio.dominio-cluster.ejemplo” y los de servicios *headless* son así: “nombre-autogenerado.mi-servicio.mi-espacio-de-nombres.servicio.dominio-cluster.ejemplo”

Pods

Los nombres DNS de los pods se definen de forma similar a los servicios. En primer lugar se indica la dirección IP del pod, después el espacio de nombres, el pod y el dominio del clúster. La plantilla sería así:

“direccion-ip-pod.pod.dominio-cluster.ejemplo”. Los servicios exponen los pods con un nombre similar a este “direccion-ip-pod.nombre-servicio. Espacio-de-nombres.svc.dominio-cluster.nombre”

El nombre de host de un pod se define en su archivo de configuración. Si no se especifica algo distinto en el atributo **spec.hostname**, toma el valor de **metadata.name**.

Políticas DNS

En el archivo de configuración del pod también podemos configurar la política DNS que queremos aplicar. Esto se hace en el atributo **spec.dnsPolicy**. Puede tomar los valores:

- **Default**: El pod hereda la configuración del nodo donde está ejecutándose.
- **ClusterFirst**: Si se envía una consulta que no coincide con el dominio del clúster configurado, se reenvía al servidor de nombres heredado del nodo por si hay otro servidor DNS adicional.
- **None**: En este caso el pod ignorará la configuración DNS de K8s y tomará la configuración que se establezca en el atributo **dnsConfig**.

Configuración DNS

La configuración DNS del pod es opcional, se establece en el campo **spec.dnsConfig.** Solamente es obligatorio definirlo si se establece la política **none**. Las propiedades que se pueden definir en este atributo son:

- **nameservers**: Es una lista de hasta tres direcciones IP que se utilizarán como servidores DNS del pod. Esta propiedad debe tener al menos una dirección si la política es **none**, en cualquier otro caso es opcional.
- **searches**: Lista de dominios de búsqueda DNS para nombres de hosts en el pod. Es una propiedad opcional en todos los casos. Si se especifica se fusionará con los nombres de dominio de búsqueda que se hayan generado a partir de la política seleccionada. Se permite un máximo de seis dominios.
- **options**: Es una lista opcional de objetos que tendrán un atributo obligatorio **name** y uno opcional **value**. Estas opciones se añaden a las establecidas en la política seleccionada.

Un ejemplo de configuración del pod sería el siguiente:

```
apiVersion: v1
kind: Pod
```

```
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
   - name: test
     image: nginx
  dnsPolicy: "none"
  dnsConfig:
    nameservers:
      - 1.2.3.4
    searches:
      - ns1.svc.cluster-domain.example
      - my.dns.search.suffix
    options:
        - name: ndots
          value: "2"
        - name: ends0
```

Ingress

Como alternativa para publicar un servicio fuera del clúster de K8s, en vez de indicar el tipo de servicio que queremos crear como *NodePort* o *LoadBalancer*, podemos utilizar el recurso *Ingress*. Es un objeto de la API de K8s que se utiliza para acceder de forma externa a los servicios del clúster, generalmente por HTTP o HTTPS. Además, puede proporcionar balanceo de carga y hosting virtual basado en nombres.

Para poder hacer uso de un *Ingress*, necesitamos que el clúster esté ejecutando un *Ingress controller*. Crear solamente un recurso *Ingress* no tendrá efecto alguno. Es por ello que debemos seleccionar uno, como es el caso del *NGINX Ingress controller*, aunque existe un gran número de controladores.

Los diferentes tipos de *Ingress* que podemos crear son:

- ***Ingress* respaldado por un solo servicio**: Expone un único servicio fuera del clúster. Esto equivale a especificar el tipo de servicio como *NodePort* o *LoadBalance*.
- ***Fanout* simple**: Dirige el tráfico desde una única dirección IP a varios servicios, dependiendo de la URI requerida. Se podría considerar

como un API Gateway, el cual establece un único punto de entrada para diferentes servicios, a la vez que balancea la carga entre sus pods.

- **Hosting virtual basado en nombres**: Permite dirigir el tráfico a varios hosts desde una misma IP. Es parecido al anterior, pero este dirige el tráfico en función del nombre del host y no de la URI requerida.

Definiendo un *Ingress*

Para definir un recurso *Ingress* simple que expone un servicio llamado "test" en la ruta "/testpath", debemos especificar las siguientes propiedades en un archivo YAML:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx-example
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

Los parámetros necesarios para especificar un recurso *Ingress* son:

- **.kind**: Indicamos que es un *Ingress*.
- **.spec.ingressClassName**: Nombre de la *Ingress class*, que contiene información adicional sobre la configuración, incluyendo el nombre del controlador que implementa la clase. Si se omite este campo, se debería definir una clase por defecto.
- **.spec.rules**: Cada regla HTTP contiene la siguiente información:

- **host**: En este ejemplo no se especifica ningún host, por lo que la regla aplica a todo el tráfico HTTP entrante a través de la IP especificada. En caso de que sí se proporcionase un host, la regla aplicaría solo a dicho host.
- **paths**: Lista de rutas, cada una de las cuales tiene un servicio asociado, especificado por los campos ***.service.name***, para el nombre del servicio, y ***.service.port.number***, para indicar el puerto. Hay que indicar obligatoriamente el tipo de ruta que estamos definiendo con el campo ***.path.pathType***. En nuestro caso es de tipo *Prefix*, lo que indica que aceptará URLs con un prefijo de ruta seguido de la ruta especificada, separados por "/". Existen otros tipos, como *Exact*, cuya ruta debe coincidir exactamente, o *ImplementationSpecific*, que deja su definición en manos de la *Ingress class*.

Ejemplo práctico con Ingress

En este ejemplo, vamos a crear un objeto simple *Ingress* que redirige las peticiones a un servicio u otro dependiendo de la URI especificada. Para ello, haremos uso de Minikube como en los apartados anteriores.

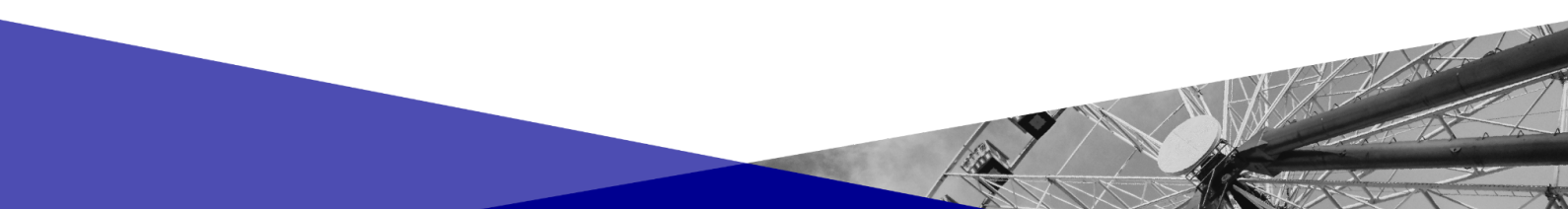
El primer paso será crear un par de servicios que expongan dos aplicaciones hello-world, cada una ejecutándose en pods diferentes. Para ello, creamos lo siguientes deployments y a continuación, sus respectivos servicios:

```
kubectl create deployment web --image=gcr.io/google-samples/hello-app:1.0
kubectl expose deployment web --type=NodePort --port=8080
```

```
kubectl create deployment web2 \
--image=gcr.io/google-samples/hello-app:2.0
kubectl expose deployment web2 --type=NodePort --port=8080
```

Para comprobar que ambos servicios están funcionando de manera correcta, ejecutamos el siguiente comando y, acto seguido, realizamos una petición con el comando *curl* a la url que nos devuelva:

```
minikube service web --url
```



```
jcmoreno@localhost:~$ minikube service web --url
http://192.168.49.2:30209
jcmoreno@localhost:~$ curl http://192.168.49.2:30209
Hello, world!
Version: 1.0.0
Hostname: web-6bf786c76b-qts5q
jcmoreno@localhost:~$ minikube service web2 --url
http://192.168.49.2:30411
jcmoreno@localhost:~$ curl http://192.168.49.2:30411
Hello, world!
Version: 2.0.0
Hostname: web2-6c9455cbb9-hfn6q
```

A continuación, vamos a habilitar el *NGINX Ingress controller*:

```
minikube addons enable ingress
```

Para verificar que el controlador se está ejecutando, lanzamos el siguiente comando:

```
kubectl get pods -n ingress-nginx
```

Deberíamos obtener una respuesta similar a esta:

```
jcmoreno@localhost:~$ kubectl get pods -n ingress-nginx
NAME                                        READY   STATUS      RESTARTS   AGE
ingress-nginx-admission-create-vgsrw        0/1     Completed   0          52s
ingress-nginx-admission-patch-48w9l         0/1     Completed   1          52s
ingress-nginx-controller-755dfbfc65-wt99f   1/1     Running     0          52s
```

Una vez hecho esto, vamos a definir nuestro *Ingress*, que definirá dos *endpoints*, "/" para el servicio web y "/v2" para el servicio web2, haciendo uso del siguiente archivo YAML:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
    - host: hello-world.info
      http:
        paths:
          - path: /
            pathType: Prefix
```

```
            backend:
              service:
                name: web
                port:
                  number: 8080
          - path: /v2
            pathType: Prefix
            backend:
              service:
                name: web2
                port:
                  number: 8080
```

Para crear el recurso, guardamos el archivo como example-ingress.yaml y ejecutamos el siguiente comando:

```
kubectl apply -f example-ingress.yaml
```

Verificamos que la dirección IP ha sido asignada ejecutando lo siguiente y comprobando que la columna *ADDRESS* contiene la IP externa del clúster (debería ser la misma IP que se obtiene al ejecutar el comando $minikube ip):

```
kubectl get ingress
```

```
jcmoreno@localhost:~$ kubectl get ingress
NAME              CLASS   HOSTS              ADDRESS        PORTS   AGE
example-ingress   nginx   hello-world.info   192.168.49.2   80      17m
jcmoreno@localhost:~$ minikube ip
192.168.49.2
```

Añadimos la siguiente línea con la dirección IP obtenida anteriormente al final del archivo /etc/hosts (necesitaremos permisos de administrador):

```
192.168.49.2 hello-world.info
```

Por último, para comprobar que todo funciona como debería, hacemos las peticiones correspondientes haciendo uso del comando curl:

```
curl hello-world.info
```

```
[jcmoreno@localhost:~$ curl hello-world.info
Hello, world!
Version: 1.0.0
Hostname: web-6bf786c76b-qts5q
[jcmoreno@localhost:~$ curl hello-world.info/v2
Hello, world!
Version: 2.0.0
Hostname: web2-6c9455cbb9-hfn6q
```

Como podemos observar, si hacemos una petición a la URI normal, obtenemos la versión 1.0.0, y si lo hacemos a la URI "/v2", obtenemos la versión 2.0.0.

Política de tráfico interno

La política de tráfico interno de los servicios permite aplicar restricciones al tráfico interno. Se entiende por tráfico interno todo aquel tráfico cuyo origen sean los pods del clúster actual. Estas restricciones se pueden aplicar para **reducir costes y mejorar el rendimiento**.

Para ello, el componente *kube-proxy* enruta el tráfico basado en el atributo ***.spec.internalTrafficPolicy***. Si ese campo toma valor *Local*, sólo se tendrán en cuenta *endpoints* locales para calcular la ruta del tráfico. Sin embargo, si no se especifica ningún valor o tiene valor *Cluster*, se tienen en cuenta todos los *endpoints* disponibles.

Por lo tanto, para usar la política de tráfico interno, necesitamos especificar el valor *Local* en los servicios que queramos que sigan esta política. Un ejemplo de servicio local es:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  internalTrafficPolicy: Local
```

Recuerda que para poder utilizar esta configuración necesitamos habilitar su respectivo *feature gate*, ***ServiceInternalTrafficPolicy=true***.

Políticas de red

Cuando trabajamos en la red de K8s, lo hacemos a nivel de direcciones IP y puertos. K8s dispone de ***NetworkPolicies*** para aplicaciones alojadas en sus clústers. Estas políticas se aplican a uno o ambos extremos de la conexión a un pod, sin afectar a otras conexiones, para regular y supervisar la comunicación entre pods y otras entidades (*Endpoint*, *Service*, etc).

Podemos identificar las entidades con las que un pod tiene visibilidad y, por tanto, capacidad para comunicarse, comprobando si esas entidades son otros pods a los que tiene acceso, namespaces a los que tiene acceso o cualquier entidad con la que no presente un bloqueo IP.

Mediante políticas de red podemos aplicar restricciones o normas, por ejemplo, usar un selector para filtrar el tráfico permitido hacia un pod, de modo que sólo el tráfico que cumpla la expresión buscada acceda al pod.

Las políticas de red no están implementadas por defecto en K8s. Si queremos usarlas en nuestra aplicación necesitamos utilizar el ***network-plugin*** de K8s que soporte objetos de tipo *NetworkPolicy*. Sin embargo, el plugin necesita que definamos un controlador *NetworkPolicy* que implemente los recursos de este tipo, de lo contrario, ninguna configuración tendrá efecto.

Aislamiento de pods

Existen dos tipos de aislamiento de pods que trataremos en K8s: aislamiento para la salida (*Egress*) y aislamiento para la entrada (*Ingress*). Estos términos se refieren a las conexiones entrantes y salientes que puede establecer un pod.

Se entiende por aislamiento la aplicación de una serie de restricciones que limitan y regulan las conexiones (entrantes y salientes), no un aislamiento absoluto del pod. Observa que cada vez que hablamos de conexiones, son entrantes **y** salientes, no entrantes **o** salientes. Ambos tipos de conexión tienen su configuración de aislamiento independiente de la otra, lo que significa que un pod puede estar más o menos aislado hacia dentro o hacia

fuera.

Una vez comprendidos estos conceptos, podemos explicar las políticas de aislamiento de un pod. Por defecto, un pod no se encuentra aislado ni para la entrada ni para la salida.

En el caso de la **salida**, decimos que una *NetworkPolicy* se aplica a la salida de un pod cuando asignamos el valor "Egress" a su campo ***policyTypes***. En esta situación, las únicas conexiones permitidas son las contenidas en la lista *egress* contenida en la definición de los objetos *NetworkPolicy*.

El caso de la **entrada** es análogo, se dice que un pod está aislado para la entrada cuando su campo ***policyTypes*** contiene el valor "Ingress". En este caso, las únicas conexiones permitidas son las que se encuentran en la lista *ingress* y las propias del nodo.

Siguiendo estas políticas, para que se permita una conexión desde un origen a un pod destino, las políticas aplicadas en ambos pods deben permitir la conexión, de salida en el origen y de entrada en el destino. Las políticas dentro de un mismo pod no entran en conflicto, son aditivas y su orden de evaluación no altera el comportamiento de la política.

Implementación de políticas de red

Para incluir políticas de red en nuestra aplicación debemos, en primer lugar, disponer de un recurso ***NetworkPolicy*** donde definiremos las restricciones y políticas de red aplicadas al tráfico del clúster. Un ejemplo de definición de *NetworkPolicy* es:

```
apiVersion: v1
kind: NetworkPolicy
metadata:
  name: example-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: networkingExample
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
```

```
        - ipBlock:
            cidr: 172.17.0.0/16
            except:
              - 172.17.1.0/24
        - namespaceSelector:
            matchLabels:
              project: networkingExample
        - podSelector:
            matchLabes:
              role: frontend
      ports:
        - protocol: TCP
          port: 6379
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/24
      ports:
        - protocol: TCP
          port: 5978
```

Utilizaremos el nombre ***networkpolicy.yaml*** para referirnos a este archivo en futuros ejemplos. Este es un ejemplo de *NetworkPolicy* que define políticas tanto de entrada como salida sobre todos los pods que cumplan con el rol **networkingExample**. Por eso vemos dos listas diferentes: **ingress**, para definir las políticas de entrada, y **egress** para las de salida.

En el ejemplo mostrado, se aplica sobre las peticiones entrantes un bloqueo sobre todos los nodos de la red 172.17.0.0/16, a excepción de los que se encuentren en el rango 172.17.1.0/24, pero se permiten aquellos pods cuyo namespace sea *networkingExample* y cuyo rol sea *frontend*. Observa en el campo **.spec.ingress.ports** cómo el único puerto por el que se puede establecer una conexión entrante con el pod es el definido en el campo **port**, 6379/tcp en este caso.

En cuanto al tráfico de salida, se bloquean todas las IP de la red 10.0.0.0/24, sin excepción, y sólo se permiten conexiones al puerto 5978/tcp.

Recuerda que lanzar este archivo contra la API no tendrá ningún efecto a menos que instalemos el plugin de red y la solución de red correspondiente a *NetworkPolicy*.

Hablemos de cómo crear nuestras propias definiciones de *NetworkPolicy*. Como todos los objetos de K8s, debemos especificar los campos

apiVersion, *kind* y *metadata*. En este caso, es importante que *kind* obtenga el valor *NetworkPolicy* para que la API entienda el tipo de objeto que está leyendo. En cuanto al resto de campos:

- **.spec:** Es la sección en la que definiremos toda la configuración de las políticas de red aplicadas a un determinado namespace.
- **.spec.podSelector:** Es el selector que indica a qué grupo de pods se aplican las políticas definidas a continuación. La etiqueta de selección se especifica en el campo **.spec.podSelector.matchLabels.role**. Si no tiene valor, se seleccionan todos los pods del namespace **.metadata.namespace**.
- **.spec.policyTypes:** Lista de políticas a aplicar que puede contener los valores *Ingress*, para políticas de entrada, *Egress*, para las salidas, o ambas. En caso de omisión se tomará *Ingress* como valor por defecto y se añadirá *Egress* cuando se encuentren elementos en la lista *egress*.
- **.spec.ingress:** Lista de reglas de entrada (*ingress*) permitidas. Está compuesta por los subcampos **from** y **ports**. En el campo *from* podemos definir qué pods pueden establecer conexiones al pod actual. Podemos definirlos mediante bloqueos IP o mediante selectores de pods. En el campo *ports* especificamos los puertos, en formato: número de puerto y protocolo, que aceptan conexiones entrantes.
- **.spec.egress:** Lista de reglas de salida (*egress*) permitidas. Está compuesta por los subcampos **to** y **ports**. Tiene la misma estructura que la lista de *ingress*, la diferencia es que los elementos de esta lista definen las conexiones de salida aceptadas.

En caso de duda, siempre es recomendable utilizar **kubelet describe** para ver qué políticas hemos definido.

Políticas por defecto

En muchos proyectos puede ser muy beneficioso definir políticas por defecto que se sobreescriben por las políticas más concretas. Te presentamos cómo definir algunas de las políticas por defecto más básicas:

Denegar todo el tráfico de entrada

Para evitar dejar accesos no intencionados a ciertos pods, podemos definir una política de denegación de tráfico de entrada. Para hacer esto basta con dejar el campo **.spec.podSelector** vacío (afecta a todos los pods) y especificar el valor *Ingress* en **.spec.policyTypes**, se incluye por mejorar la legibilidad, *Ingress* es un valor por defecto. Esta política es la que se aplica si no se define ninguna política de entrada.

```
apiVersion: v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

Permitir todo el tráfico de entrada

Similar al anterior pero más peligroso. Si aplicamos esta política por defecto, ninguna otra política definida puede negar una conexión de entrada. Para definirla basta con incluir un elemento **{}** en la lista *ingress*:

```
apiVersion: v1
kind: NetworkPolicy
metadata:
  name: default-allow-all-ingress
spec:
  podSelector: {}
  policyTypes:
    - Ingress
  ingress:
    - {}
```

Denegar todo el tráfico de salida

Análogo a la entrada, pero cambiando *Ingress* por *Egress*.

```
apiVersion: v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress
```

```
spec:
  podSelector: {}
  policyTypes:
    - Egress
```

Permitir todo el tráfico de salida

Análogo a la entrada pero cambiando *Ingress* por *Egress*. Esta política es la que se aplica si no se define ninguna política de salida.

```
apiVersion: v1
kind: NetworkPolicy
metadata:
  name: default-allow-all-egress
spec:
  podSelector: {}
  policyTypes:
    - Egress
  egress:
    - {}
```

La combinación de las políticas por defecto anteriores puede resultar útil en las diferentes fases de un proyecto, tanto para facilitar el desarrollo como para mejorar la seguridad de la aplicación.

Límites

A pesar de la amplia funcionalidad que nos permite la API de NetworkPolicy, existen casos de uso que aún no están contemplados en K8s.

Configuración

Una vez que hemos realizado un recorrido por todos los aspectos básicos de K8s, vamos a hablar sobre algunos puntos importantes a la hora de establecer una buena configuración donde podamos sacar el máximo partido a esta herramienta.

Consejos de configuración y buenas prácticas

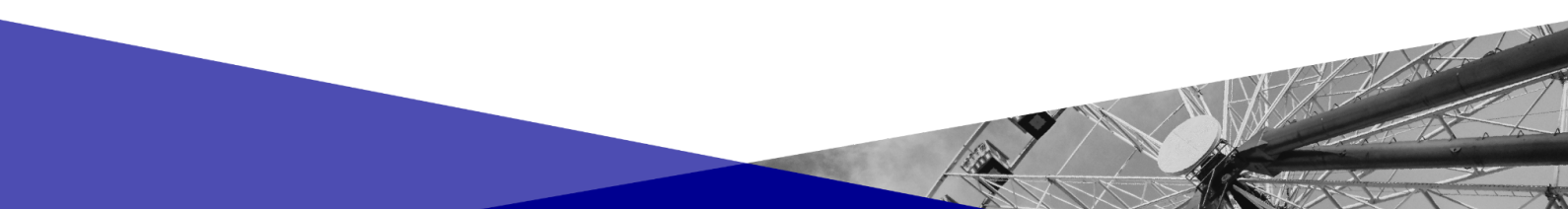
K8s permite una amplia variedad de opciones de configuración, esto hace que sea una **herramienta muy potente** si está bien configurada, pero también puede ser poco eficiente en caso contrario. En este punto, vamos a hablar de algunos consejos sencillos a la hora de configurar nuestros clústeres.

En cuanto a los archivos de configuración, hemos visto a lo largo de esta guía que se utilizan **archivos YAML**. K8s permite configuraciones con archivos de tipo JSON pero es más sencillo y común hacerlo con los primeros. Además, hay que tener en cuenta las **configuraciones por defecto** que se indican en la documentación para no repetir atributos que ya están definidos si no vamos a cambiar los valores preestablecidos. Esto hará que los archivos sean más cortos y legibles.

A lo largo de estas páginas, hemos utilizado en ocasiones atributos ***label* o etiquetas**. Es un atributo opcional pero facilita el proceso de identificación de objetos en el sistema y cuando tratamos con sistemas muy grandes, agiliza el trabajo.

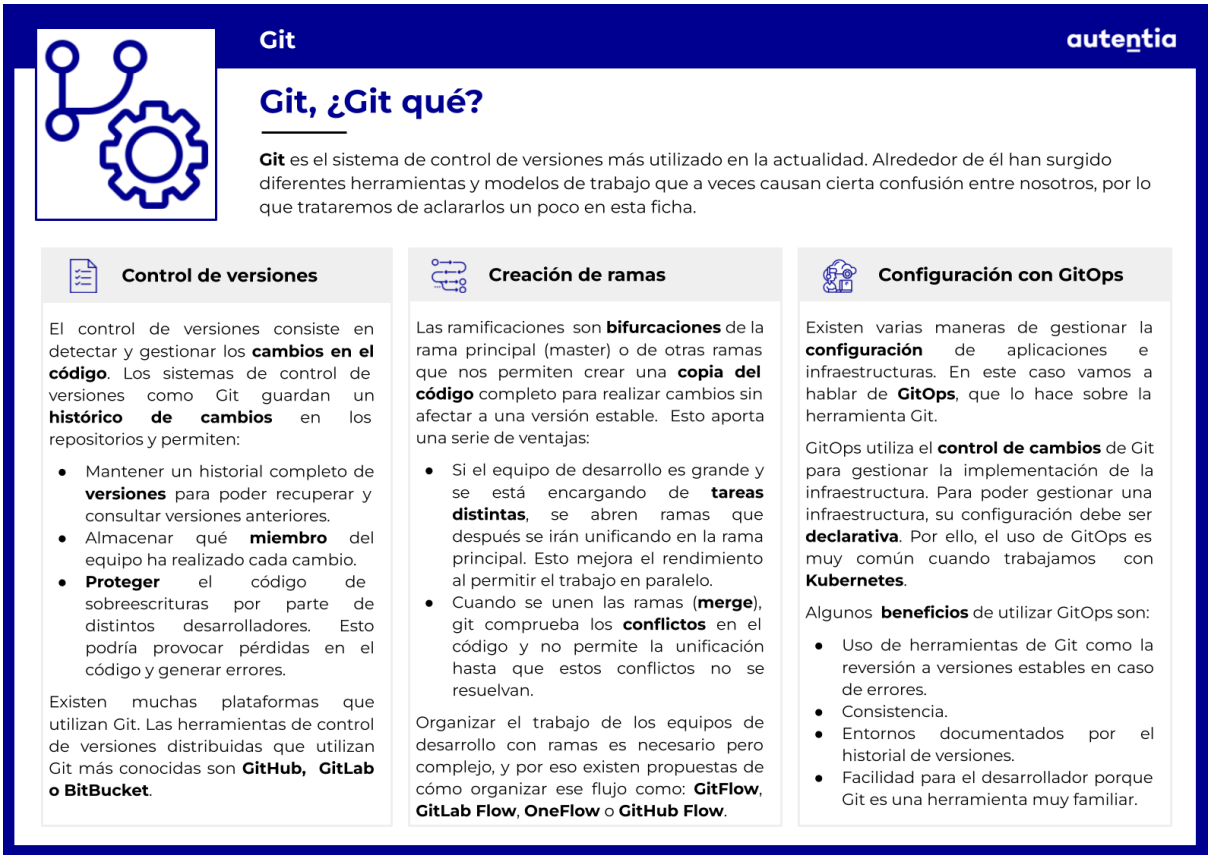
Una de las cosas más importantes que evaluamos a la hora de analizar el rendimiento de un sistema es el **tiempo de construcción y carga**. En un sistema donde utilizamos contenedores con imágenes, si estas son pequeñas, la velocidad es mayor. Por lo tanto, se recomienda hacer uso de **imágenes** lo más **pequeñas** posibles.

Siguiendo el hilo del rendimiento, es muy bueno aplicar configuraciones de **escalado automático** de servicios del clúster, así como las políticas de red que permiten restringir el acceso a algunos servicios al interior del clúster, a los metadatos de los usuarios, etc.



Por otro lado, hemos hablado de **crear pods** de forma manual o a través de controladores como ReplicaSet, Deployment o Job. Si puedes evitarlo, es mejor no crear pods de forma manual, ya que estos pods no se volverán a crear si fallan, y cuando tenemos muchos, es muy difícil gestionar esto a mano.

Por último, cabe mencionar que, como buena práctica, la manera más adecuada de trabajar con K8s sería a través de un repositorio y haciendo uso de **GitOps**, de lo cual hablaremos más adelante.



Estas son algunas recomendaciones para empezar a utilizar la herramienta. Cuanto más complejos sean los sistemas que se construyen, habrá más instrucciones y consejos que incrementarán el rendimiento y beneficios de K8s.

Kubectl

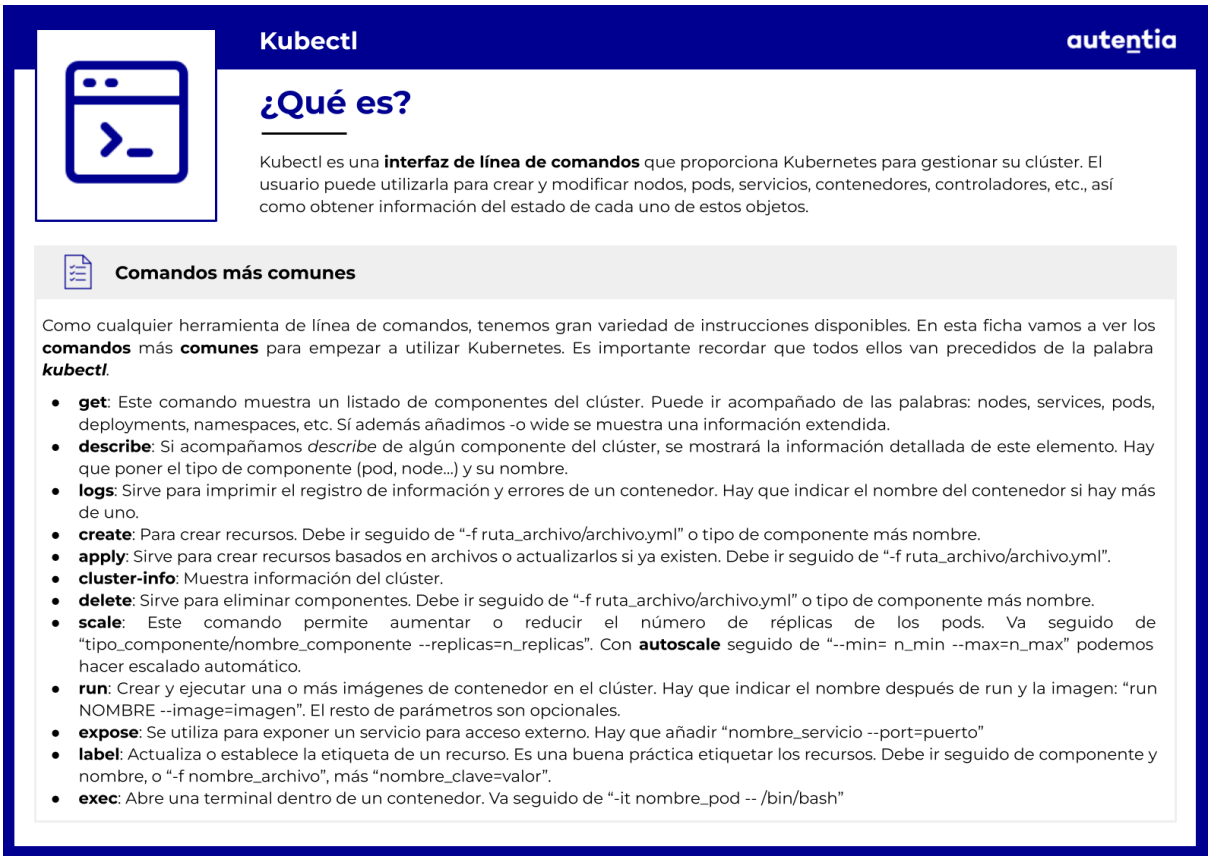
Kubectl es la herramienta de línea de comandos que utilizamos para trabajar con K8s. A lo largo de esta guía, hemos utilizado los comandos más importantes, así que vamos a hablar sobre ellos y sobre cómo usarlos

correctamente.

Cuando creamos un recurso, podemos utilizar el comando ***create***, indicando el tipo de recurso y el nombre, o el comando ***apply***, seguido de la ruta y el nombre del archivo YAML que contiene su configuración. La principal diferencia entre ellos es que *create* es un comando que se usa a través de consola y que sigue el paradigma imperativo, mientras que *apply* es declarativo y trabaja con manifiestos, en los que sólo indicamos el estado que se desea alcanzar. Por esta razón, siguiendo los principios de K8s, es recomendable crear los recursos de manera declarativa con *apply*. Además, de esta forma podremos guardar los archivos de configuración en un repositorio de Git como recomienda GitOps para la automatización y versionado de los cambios.

Unido a lo que mencionamos anteriormente sobre el uso de etiquetas, se recomienda que cuando usemos los comandos ***get*** o ***delete***, utilicemos selectores de etiqueta en lugar de nombres específicos de los objetos.

Por lo demás, a base de usar los comandos de kubectl para obtener información del estado de los objetos, registros de logs, estado de los controladores, etc., irás descubriendo nuevas funcionalidades y parámetros para filtrar las salidas de forma que obtengas la información que realmente necesitas consultar. Por ejemplo, es mejor consultar el estado de un servicio en concreto en lugar de consultarlos todos para ver el estado de uno solo.



ConfigMaps

Un *ConfigMap* es un objeto de la API de K8s que se utiliza para almacenar datos no confidenciales en pares clave-valor. Un *ConfigMap* nos permite desacoplar la configuración específica del entorno, de las imágenes de contenedor, creando así aplicaciones fácilmente portables.

Imagina que estamos desarrollando una aplicación para que pueda ejecutarse en nuestro equipo local para desarrollo y en la nube para mantener el tráfico real. Ahora escribimos el código necesario para configurar una variable llamada "DATABASE_HOST", asignándole el valor "localhost" cuando estamos trabajando en local, y la referencia al servicio de K8s que expone la base de datos cuando estemos en el entorno de producción. De esta manera, podemos tener un contenedor ejecutándose en la nube y tener exactamente el mismo código en local para depurarlo si fuera necesario.

Los *ConfigMaps* no están diseñados para almacenar grandes cantidades de información. La información almacenada en ellos no puede superar 1 MiB. Si necesitas guardar datos de configuración que ocupen más que este límite,

deberías usar un volumen específico o una base de datos separada.

Podemos utilizar los *ConfigMaps* para configurar los contenedores de un pod de las siguientes maneras:

- A través de **comandos y argumentos** de un contenedor.
- Usando **variables de entorno** de un contenedor.
- Como **fichero** en un volumen de solo lectura, al cual tiene acceso la aplicación.
- **Ejecutando código** dentro de un pod que utilice la API para leer el *ConfigMap*.

En el caso de elegir alguna de las 3 primeras opciones, kubelet será el encargado de utilizar la información del *ConfigMap* cuando se lancen los contenedores de un pod.

Definiendo y usando un *ConfigMap*

Para definir un *ConfigMap* tenemos que hacer uso de archivos de especificación YAML al igual que pasa con el resto de objetos y componentes. Sin embargo, este recurso no necesita especificar el campo ***.spec***, si no que tiene en su lugar un campo ***.data*** donde se almacenan todos los pares clave-valor. Aquí tenemos un ejemplo:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys
  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
```

Como podemos ver, en el campo ***.kind*** indicamos que se trata de un *ConfigMap*, y dentro del campo ***.data*** distinguimos los siguientes casos:

- **Property-like keys**: Son las claves que referencian a un único valor, actuando como propiedades.
- **File-like keys**: Son las claves que referencian a varios pares clave-valor, por lo que en realidad definen archivos de configuración.

Cabe destacar que podemos hacer un *ConfigMap* **inmutable** añadiendo al final el campo "inmutable: true".

Por otro lado, este sería un ejemplo de pod que utiliza el anterior *ConfigMap*:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: game.example/demo-game
      env:
        - name: PLAYER_INITIAL_LIVES
          valueFrom:
            configMapKeyRef:
              name: game-demo
              key: player_initial_lives
        - name: UI_PROPERTIES_FILE_NAME
          valueFrom:
            configMapKeyRef:
              name: game-demo
              key: ui_properties_file_name
      volumeMounts:
      - name: config
        mountPath: "/config"
        readOnly: true
  volumes:
    - name: config
      configMap:
        name: game-demo
        items:
        - key: "game.properties"
```

```
      path: "game.properties"
    - key: "user-interface.properties"
      path: "user-interface.properties"
```

Para conectar ambos archivos, necesitamos especificar los siguientes campos:

- **.spec.containers.env**: Define las variables de entorno. Para ello, por cada variable hacemos uso de los campos:
    - **.name**: Nombre de la variable de entorno.
    - **.valueFrom.configMapKeyRef.name**: Nombre del *ConfigMap* del que estamos cogiendo los valores.
    - **.valueFrom.configMapKeyRef.key**: Nombre de la clave del *ConfigMap* con la que queremos que se corresponda la variable de entorno.
- **.spec.volumes.configMap**: En este campo indicamos los valores del *ConfigMap* a partir del cual queremos definir un volumen para montarlo dentro del contenedor:
    - **.name**: Nombre del *ConfigMap* que queremos montar.
    - **.items.key**: Nombre de la clave del *ConfigMap* de tipo *file-like key* que queremos crear como archivo.
    - **.items.path**: Nombre del archivo en el que guardaremos los pares clave-valor que contiene la clave especificada en el punto anterior.

Como nota, mencionar que varios pods pueden hacer uso del mismo *ConfigMap*.

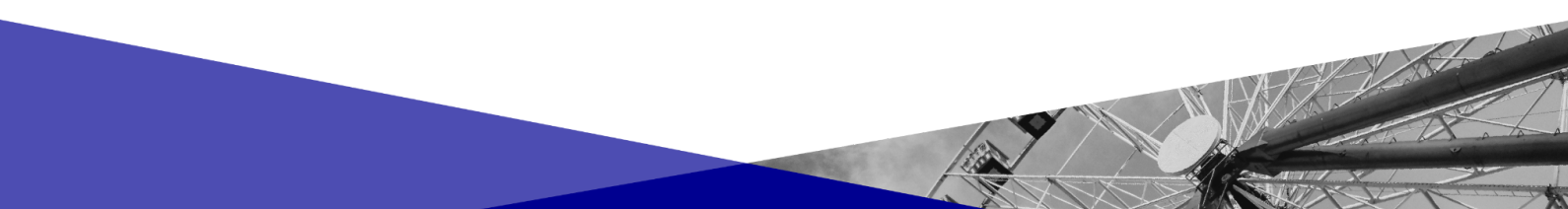
Secrets

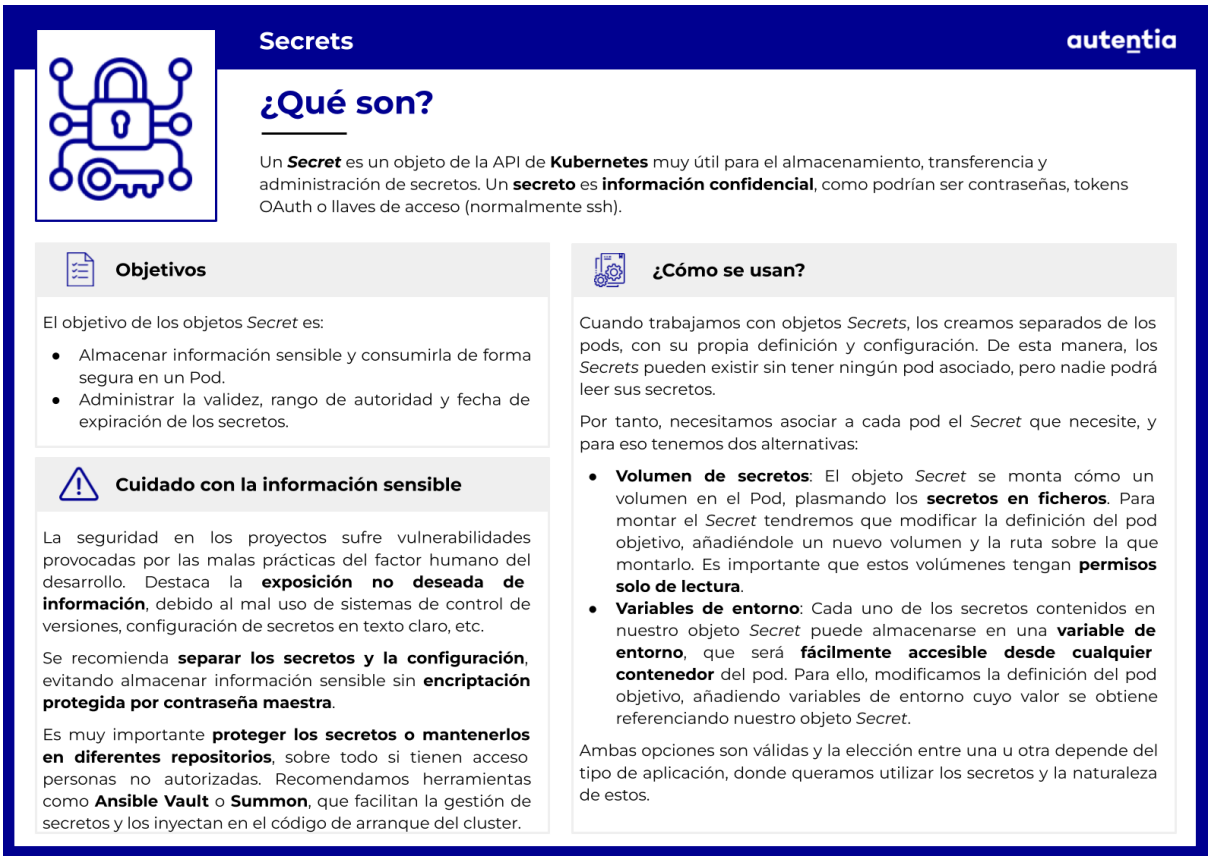
Un *Secret* es un objeto de la API de K8s muy útil para el almacenamiento, transferencia y administración de secretos. Un secreto es información confidencial, como podrían ser contraseñas, tokens OAuth o llaves de acceso (normalmente ssh).

Los secretos surgen debido a las siguientes necesidades de los usuarios de K8s:

- **Almacenar secretos** para mis aplicaciones, que pueden **consumir de forma segura**. De esta forma, separamos la configuración de la aplicación, de las imágenes que la forman.
- Permitir a los **contenedores** de un pod **consumir información confidencial** sobre los servicios que corren. Por ejemplo, token de autenticación de una API.
- Asociar un pod a un ***ServiceAccount*** específico, que permita consumir secretos, definiendo su **validez**, rango de **autoridad** y **fecha de expiración**. Esto nos permitirá actualizar los secretos de forma automática, haciendo que sean dinámicos y proporcionando un mayor nivel de seguridad.

Por lo tanto, los *Secrets* surgen para almacenar información confidencial en un entorno más seguro y flexible que escribiéndola en la definición de un pod o una imagen de contenedor. Esta última aproximación conlleva el riesgo de exposición accidental, ya sea mediante la API de K8s o mediante el sistema de control de versiones que utilicemos (cuidado con hardcodear credenciales, los commits se pueden recuperar).





Crear un Secret

Para crear nuestro propio *Secret* usaremos la API de K8s.

Supongamos que tenemos una API REST a la que necesitamos acceder con unas credenciales “admin:1234”, siendo “admin” el nombre de usuario y “1234” la contraseña.

El comando que utilizaremos para crear el secreto es:

```
kubectl create secret generic rest-user-pass {args}
```

A partir de este punto podemos crear el secreto con el argumento **--from-literal={value}**, al que le daremos los valores de la credencial, o mediante el argumento **--from-file={filePath}**, donde *filePath* será la ruta de los archivos secretos *username.txt* y *password.txt*. Así se verían los ejemplos mencionados:

Valores literales:

```
kubectl create secret generic rest-user-pass \
```

```
--from-literal=username='admin' --from-literal=password='1234'
```

Recuerda que estamos creando objetos desde la línea de comandos. Es importante que utilices las comillas simples para que, en caso de tener caracteres especiales como *, tu línea de comandos no los interprete. Siempre que algún usuario o contraseña te de error de interpretación, escapa el carácter que creas erróneo con el carácter \.

Archivos de secreto:

```
kubectl create secret generic rest-user-pass \
--from-file={filePath}/username.txt --from-file={filePath}/password.txt
```

En este caso vamos a recoger las credenciales de los archivos *username.txt* y *password.txt*. Recuerda que si guardamos los valores en archivos, no es necesario envolverlos en comillas simples ni escapar los caracteres especiales, pues no existe el riesgo de que la línea de comando los interprete.

Si profundizamos en la seguridad de nuestra aplicación, debemos recordar que una buena práctica es guardar la información confidencial, utilizada en los comandos de terminal, en archivos de secreto con permisos correctamente definidos.

Recuerda que la terminal guarda un histórico de los comandos ejecutados, y los diferentes procesos en sistemas UNIX o similares guardan una copia del comando ejecutado para invocarlos. Por tanto, si alguien consigue acceso a este tipo de información, podría obtener credenciales en texto claro fácilmente.

Para crear un *Secret* de forma declarativa, basta con crear una definición de *Secret* y utilizar *apply* para crearlo. Veamos un ejemplo con las credenciales anteriores: “admin:1234”.

En primer lugar, codificamos las credenciales en base64. Para ello, podéis utilizar cualquier herramienta a vuestro gusto, nosotros lo haremos mediante consola:

```
echo -n 'admin' | base64
echo -n '1234' | base64
```

Obteniendo la siguiente salida:

```
❯ echo -n 'admin' | base64
YWRtaW4=
❯ echo -n '1234' | base64
MTIzNA==
```

Ahora podemos insertar estos datos en un archivo YAML que defina el *Secret*:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: rest-user-pass
type: Opaque
data:
  username: YWRtaW4=
  password: MTIzNA==
```

Al que llamaremos ***secret.yaml*** desde ahora. Para verificar y crear el objeto, ejecutamos:

```
kubectl apply -f {filePath}/secret.yaml
```

Usar un Secret

Ahora que ya tenemos nuestro secreto, para usarlo necesitamos crear una referencia en el pod, ya sea montándolo como archivos en un volumen o como argumento de *kubelet* al extraer imágenes del pod.

Secrets como archivos

Una vez creado el *Secret*, debemos seguir los siguientes pasos para montarlo en un pod objetivo:

1. **Crear el *Secret*** si no existe aún, no importa si ya lo está referenciando otro pod.
2. **Agregar un volumen** al campo .**spec.volumes[]**, asignando como valor del campo **.spec.volumes[].secret.secretName** el nombre del secreto a referenciar.
3. Agregar las necesidades de clave en **.spec.volumes[].secret.items**, definiendo qué clave queremos obtener, la ruta donde guardarla y

qué permisos tendrá ese archivo (por defecto 0644).

4. **Agregar un .spec.containers[].volumeMounts[]** a cada contenedor que necesite un *Secret*. *VolumeMount* es una estructura en la que podemos definir sobre qué ruta se montará el volumen, **mountPath**, y darle permisos de solo lectura (ideal para secretos), **readOnly=true**.
5. **Modificar la implementación de la imagen** para que las credenciales se busquen en estos directorios.

Este es un ejemplo básico de como quedaría:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: plain-rest-user-pass
      items:
      - key: username
        path: my-group/my-username
        defaultMode: 256
```

En el que modificamos el pod “mypod” para montar en la ruta “/etc/foo” el nombre de usuario en un archivo con permisos 0400 (sólo el dueño tiene permisos, el resto no puede ni leerlo).

Secrets como variables de entorno

Otra manera de utilizar *Secrets* desde un contenedor es transferir estos como variables de entorno del pod. Una variable de entorno es una **par clave=valor** a la que pueden acceder todos los contenedores de un pod.

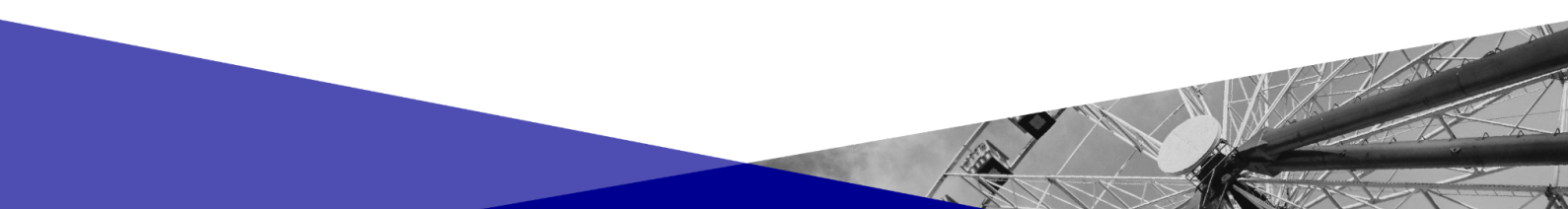
Para volcar secretos en una variable de entorno debemos:

1. **Crear el *Secret*** si no existe aún, no importa si ya lo está referenciando otro pod.
2. **Modificar la definición del pod**, añadiendo en el campo **.spec.containers[].env[]** una nueva variable de entorno, que referencia al *Secret* mediante el campo **.spec.containers[].env[].valueFrom.secretKeyRef**. En este campo rellenamos los campos *name* y *key* con los valores correspondientes a nuestro *Secret*.
3. **Modificar la implementación de la imagen** para que las credenciales se busquen en estas variables de entorno.

Un ejemplo de *Secret* como variable de entorno es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: env-pod
spec:
  containers:
  - name: env-container
    image: redis
    env:
    - name: SECRET_USERNAME
      valueFrom:
          secretKeyRef:
            name: plain-rest-user-pass
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: plain-rest-user-pass
            key: password
  restartPolicy: Never
```

En este caso hemos transferido el usuario y contraseña de nuestra API REST a los contenedores *redis* de nuestra aplicación como variables de entorno.



Restricciones

Aunque los *Secret* son muy útiles para la administración y almacenamiento de información confidencial en K8s, no siempre están disponibles. Presentan una serie de restricciones a tener en cuenta si los queremos implementar:

- Los *Secrets* no encriptan los secretos que almacenan, por lo que alguien con acceso a la API puede acceder a su valor.
- Los *Secrets* montados como volúmenes se deben crear antes de ser referenciados por cualquier objeto.
- Los objetos *API Secret* sólo pueden ser referenciados por pods de su mismo namespace.
- Cada *Secret* individual está limitado a 1MiB de tamaño, para evitar inanición del *apiserver* y *kubelet*.
- Sólo los pods obtenidos por medio de la API de K8s pueden referenciar a los *Secrets*. No soporta pods creados por medio de flags de *kubelet*, como *--manifest-url* o *--config*, ni de la API REST.
- Los *Secrets* consumidos como variables de entorno deben crearse antes de ser consumidos, de lo contrario, el pod no iniciará. Existe una excepción, configurar la variable de entorno como “Optional”.
- No se permite poblar variables de entorno desde un *Secret* a través del campo *envFrom*. Estas variables serán omitidas, pero pueden dar lugar a errores futuros.

Parte 7

Kubernetes Avanzado

Configuración

Gestión de recursos de pod y contenedores

A la hora de crear contenedores en una infraestructura de K8s, sobre todo en aplicaciones grandes, es importante **gestionar los recursos** que se asigna a cada contenedor. Cuando un contenedor llega al límite de los recursos que se le han asignado, cambiará de estado *Running* a *Terminated*. Esto provoca que se reinicie para restablecerse. Por otro lado, si todos los contenedores de un nodo sobrepasan los recursos del nodo, algunos contenedores se reiniciarán, e incluso eliminarán para liberar recursos y que no se caiga el nodo completo.

En este punto vamos a ver cómo configurar estos límites en pods y contenedores para que las aplicaciones que despleguemos en K8s sean robustas y no se paren por falta de recursos.

Recursos en Kubernetes

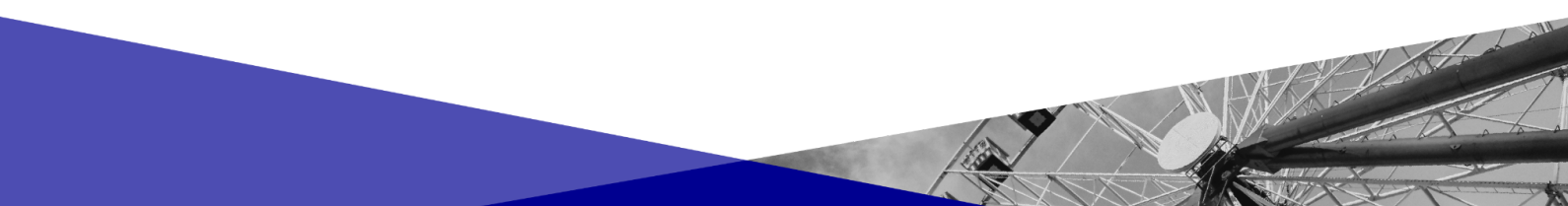
Cuando hablamos de recursos, nos referimos a recursos de computación. En este caso, podemos configurar CPU y memoria.

- **CPU**: Estos recursos se miden en “unidades de CPU” y la equivalencia de una unidad en K8s es 1 vCPU/Core. Aunque se mide en unidades, la especificación permite peticiones fraccionadas, pero la petición mínima es 0.1, que se convierte a 100m por la API y se traduce como “cien milicpus o cien milicores”.
- **Memoria**: La memoria se mide en bytes. Se pueden usar valores enteros en bytes o un número decimal con los sufijos que conocemos: E, P, T, G, M, k, m; o sus equivalentes en potencias de dos: Ei, Pi, Ti, Gi, Mi, ki.

Request y limit

Para configurar los recursos disponibles de un nodo de forma que los pods y los contenedores los utilicen, se deben configurar ***requests*** y ***limits***.

Los recursos que necesitan los contenedores se especifican en ***request.*** En este caso, cuando el nodo donde corre un contenedor tiene suficientes recursos disponibles, el contenedor puede exceder los límites establecidos



en el *request*. Sin embargo, los recursos especificados en ***limit*** no se pueden exceder en ningún caso.

Según lo indicado en el párrafo anterior, si configuramos una request de memoria de 500 MiB para un contenedor que está corriendo en un pod ubicado en un nodo de 10 GiB de memoria y no hay otros pods, o los que hay no están utilizando toda la memoria RAM del nodo, el contenedor podrá usar más memoria.

Por el contrario, si configuramos un límite (*limit)* de memoria para el contenedor de 4GiB, independientemente de que haya más memoria disponible en el nodo que 4 GiB, el contenedor no podrá consumir más de la establecida en el límite.

Ambas especificaciones (requests y limits) son compatibles. De hecho, si solo se establece el límite pero no la petición, por defecto K8s establecerá como *request* el mismo valor que se haya establecido en el límite.

Para entender mejor esto último, vamos a analizar este archivo de configuración de un pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: log-aggregator
    image: images.my-company.example/log-aggregator:v6
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

El pod que acabamos de configurar está formado por **dos contenedores**. El primero de ellos, llamado "app"**,** tiene una petición de 64 MiB de memoria y 500m de CPU (0.5); el límite se establece en 128 MiB de memoria y 500m de CPU. El segundo, "log-aggregator"**,** tiene una petición de 64MiB de memoria y 250m de CPU. El límite es de 128 MiB de memoria y 500m de CPU. Recuerda que cada contenedor pedirá al pod, y el pod a su vez pedirá al nodo la cantidad de memoria y cpu especificada en *request*, pero utilizará como máximo la especificada en *limit* si están disponibles en el nodo los recursos necesarios.

Si te da miedo realizar este tipo de configuraciones porque no quieres quedarte sin recursos suficientes para todos los pods de un nodo, no te preocupes, el planificador de K8s se encargará de comprobar que la suma de todos los recursos solicitados en los *request* de los contenedores de los distintos pods no superan los recursos disponibles en el nodo. Si la comprobación de capacidad falla, el pod no se programará.

Una vez que hemos programado nuestros contenedores con las peticiones y límites deseados, si durante la ejecución del contenedor excede su límite de memoria y es reiniciable, *kubelet* lo reiniciará. Sin embargo, si excede su petición de memoria, probablemente el pod que lo contiene sea desalojado en cuanto el nodo se quede sin memoria.

Almacenamiento local efímero

Otra posibilidad es configurar el almacenamiento local efímero de los nodos. Normalmente está respaldado por la memoria RAM y, como indica su nombre, no se garantiza su durabilidad a largo plazo. Los pods lo pueden utilizar para añadir espacio, caché o registrar logs. Este tipo de almacenamiento se utiliza para crear volúmenes efímeros de tipo *emptyDir*.

Las solicitudes de almacenamiento efímero en un pod se programan de la siguiente forma:

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
```

```
    resources:
      requests:
        ephemeral-storage: "2Gi"
      limits:
        ephemeral-storage: "4Gi"
    volumeMounts:
      - name: ephemeral
        mountPath: "/tmp"
  volumes:
  - name: ephemeral
    emptyDir: {}
```

Recursos extendidos

Los recursos de K8s son la memoria y la CPU. Sin embargo, se pueden consumir otros que no están integrados en K8s. Para ello, el operador del clúster debe "anunciar" el recurso extendido y se debe configurar la solicitud de uso en el archivo de configuración del pod.

Para que el clúster anuncie un recurso extendido, debe enviar una solicitud HTTP PATCH al servidor API especificando la petición del recurso. Por ejemplo:

```
curl --header "Content-Type: application/json-patch+json" --request PATCH
--data '[{"op": "add", "path": "/status/capacity/example.com~foo",
"value": "5"}]' http://k8s-master:8080/api/v1/nodes/k8s-node-1/status
```

Ahora solo falta configurar adecuadamente los pods que queramos que lo utilicen. Esto se hace añadiendo en *requests* y *limits* los recursos extendidos. Vamos a ver un ejemplo donde el pod solicita 2 CPUs y 1 "example.com/foo", que es el recurso que hemos anunciado en el clúster.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    resources:
      requests:
        cpu: 2
```

```
        example.com/foo: "1"
      limits:
        example.com/foo: "1"
```

Problemas de configuración de recursos

Como hemos mencionado antes, puede ocurrir que por un fallo de configuración de los pods, intentemos asignar más recursos a los contenedores de los disponibles en el nodo donde se encuentra el pod de los contenedores. Si esto ocurre, quiere decir que el planificador no encuentra un lugar donde colocar el pod y se producirá un evento.

Estos eventos indicarán un mensaje que podrá ser de dos tipos:

- **“PodExceedsFreeCPU”**: Indica que no hay CPUs suficientes para cubrir las peticiones de los contenedores del pod.
- **“PorExceedsFreeMemory”**: Indica que no hay memoria suficiente para cubrir las peticiones de los contenedores del pod.

En este caso, podemos resolverlo de varias formas:

- Reduciendo las peticiones de los contenedores.
- Añadiendo más nodos al clúster.
- Finalizando los pods que no son necesarios.
- Comprobando que el pod no es más grande que todos los nodos disponibles (en ese caso nunca se podrá programar).

Cuando un contenedor se reinicia varias veces por falta de recursos y aun así no los consigue, pasa a estado ***Terminated***. Si escribimos el siguiente comando por terminal, veremos el código de salida y la razón por la que se ha finalizado el contenedor.

```
kubectl get pod -o go-template
'{{range.status.containerStatuses}}{{"Container Name:
"}}{{.name}}{{"\r\nLastState: "}}{{.lastState}}{{end}}' nombre-pod
```

Acceso al clúster con archivos *kubeconfig*

Supongamos que tenemos dos clústers, uno para desarrollo (“development”) y otro para pruebas (“scratch”). En el clúster de desarrollo, los desarrolladores de frontend trabajan en un espacio de nombres llamado “frontend”, y los de almacenamiento en uno llamado “storage”. En el clúster para pruebas, se trabaja en el espacio de nombres por defecto. Para acceder al clúster de desarrollo, se requiere autenticación por certificado, mientras que en el de pruebas solo se requiere autenticación a través de usuario y contraseña.

Como podemos imaginar, cambiar de un espacio de trabajo a otro puede resultar una tarea tediosa. Es aquí donde entran en juego los **archivos *kubeconfig***.

Los archivos de configuración de K8s, conocidos como archivos *Kubeconfig*, nos permiten **organizar la información** acerca de los clústers, usuarios, espacios de nombres y métodos de autenticación. La herramienta de línea de comandos *kubectl* utiliza estos archivos para encontrar la información necesaria para establecer la conexión con el servidor de la API del clúster indicado.

Por defecto, *kubectl* busca el archivo “config” en el directorio “$HOME/.kube”. No obstante, se pueden definir otros archivos *kubeconfig* indicándolos a través de la variable de entorno “KUBECONFIG” o mediante el flag “--kubeconfig”.

Para que resulte más sencillo realizar cambios entre clústers y espacios de nombres, existen los contextos. Un contexto es un elemento dentro de un archivo *kubeconfig* que se utiliza para agrupar los parámetros de acceso con un único nombre. Aunque *kubectl* está configurada para comunicarse con el clúster utilizando los parámetros del contexto actual, podemos cambiar en cualquier momento de contexto usando el comando:

```
kubectl config use-context
```

Ejemplo práctico de uso

Nota: Como en esta guía estamos haciendo uso de Minikube, no es posible realizar estos pasos, ya que solo tenemos acceso a un único clúster.

Para solventar el problema planteado al inicio de este apartado, vamos a definir el siguiente archivo de configuración YAML al que llamaremos

“config-demo”:

```
apiVersion: v1
kind: Config
preferences: {}

clusters:
- cluster:
  name: development
- cluster:
  name: scratch

users:
- name: developer
- name: experimenter

contexts:
- context:
  name: dev-frontend
- context:
  name: dev-storage
- context:
  name: exp-scratch
```

Con este archivo, estamos definiendo los dos clústers (el de desarrollo y el de pruebas), dos usuarios (desarrollador y experimentador) y 3 contextos. Una vez creado el archivo, debemos ejecutar los siguientes comandos para añadir algunos detalles de configuración:

```
kubectl config --kubeconfig=config-demo set-cluster development \
--server=https://1.2.3.4 --certificate-authority=fake-ca-file
kubectl config --kubeconfig=config-demo set-cluster scratch \
--server=https://5.6.7.8 --insecure-skip-tls-verify
```

Añadimos los detalles de los usuarios:

```
kubectl config --kubeconfig=config-demo set-credentials \
developer --client-certificate=fake-cert-file \
--client-key=fake-key-seefile
kubectl config --kubeconfig=config-demo set-credentials experimenter \
--username=exp --password=some-password
```

Y añadimos los detalles de los contextos:

```
kubectl config --kubeconfig=config-demo set-context \
dev-frontend --cluster=development --namespace=frontend \
--user=developer
kubectl config --kubeconfig=config-demo set-context dev-storage \
--cluster=development --namespace=storage --user=developer
kubectl config --kubeconfig=config-demo set-context exp-scratch \
--cluster=scratch --namespace=default --user=experimenter
```

Si ahora abrimos nuestro archivo de configuración “config-demo” con el siguiente comando:

```
kubectl config --kubeconfig=config-demo view
```

Veremos algo similar a lo siguiente:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: fake-ca-file
    server: https://1.2.3.4
  name: development
- cluster:
    insecure-skip-tls-verify: true
    server: https://5.6.7.8
  name: scratch
contexts:
- context:
    cluster: development
    namespace: frontend
    user: developer
  name: dev-frontend
- context:
    cluster: development
    namespace: storage
    user: developer
  name: dev-storage
- context:
    cluster: scratch
    namespace: default
    user: experimenter
  name: exp-scratch
current-context: ""
kind: Config
preferences: {}
```

```
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
- name: experimenter
  user:
    password: some-password
    username: exp
```

Una vez realizados todos estos pasos, podemos cambiar fácilmente de contexto, por ejemplo al contexto “dev-frontend”, con el comando:

```
kubectl config --kubeconfig=config-demo use-context dev-frontend
```

Si queremos visualizar únicamente la información relativa al contexto actual, ejecutamos:

```
kubectl config --kubeconfig=config-demo view --minify
```

Y veremos algo similar:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: fake-ca-file
    server: https://1.2.3.4
  name: development
contexts:
- context:
    cluster: development
    namespace: frontend
    user: developer
  name: dev-frontend
current-context: dev-frontend
kind: Config
preferences: {}
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
```

Seguridad

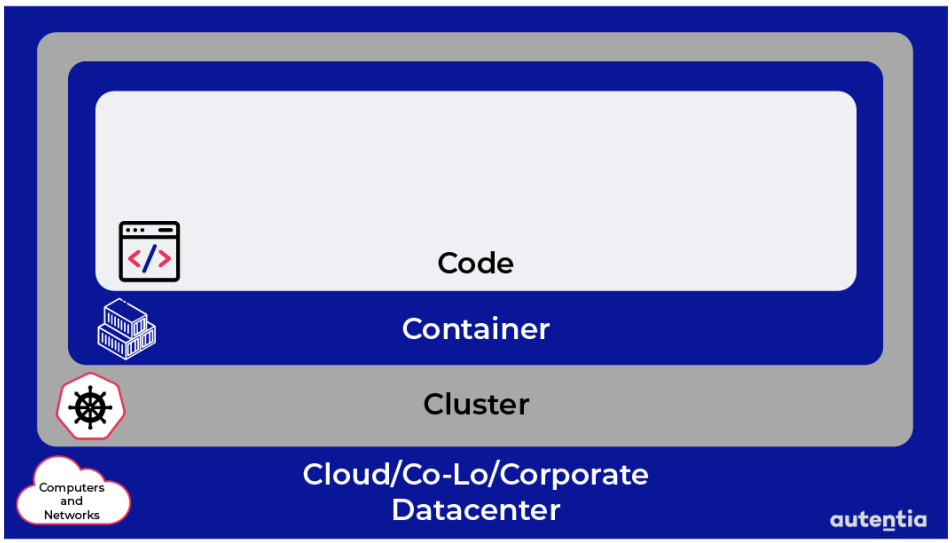
La seguridad es imprescindible en cualquier proyecto. Cuando hablamos de K8s, la seguridad comienza a tomar importancia en la fase de diseño y debemos tenerla en cuenta durante todo el ciclo de vida del proyecto.

Las 4C's de Cloud Native Security

Vamos a definir una aproximación de modelo de seguridad en el contexto de *Cloud Native Security*. Esta aproximación consiste en un enfoque en capas, compuesto por las capas:

- **Cloud**: La nube, conjunto de servidores o centro de datos corporativo, donde se aloja la infraestructura de K8s.
- **Cluster**: Capa de nodos que ejecutan las aplicaciones de K8s.
- **Container**: Capa de contenedores que forman parte de un nodo y llevan a cabo la ejecución de las aplicaciones.
- **Code**: Capa de código que administra la seguridad del código fuente de la aplicación.

Cada capa enumerada está basada en la siguiente capa más externa, como podemos ver en la siguiente imagen:



Por tanto, está claro que no podemos garantizar la seguridad de nuestra aplicación sólo aplicando seguridad a nivel de código. La capa de código se basa en la seguridad del contenedor, la cual se basa en su clúster, y ésta a

su vez, se basa en la seguridad de la nube que lo aloja.

Nube (Cloud)

La nube es la capa que engloba toda la aplicación y que gestiona el clúster de K8s. Si esta capa sufre alguna vulnerabilidad, se pierden todas las garantías de seguridad del resto de capas.

La seguridad en la nube abarca desde la configuración de los componentes que la forman, hasta la propia infraestructura subyacente. Cada proveedor *cloud* ofrece a sus clientes recomendaciones de seguridad para securizar las cargas de trabajo en sus entornos.

Si hablamos de infraestructura, nos centramos en aplicar las siguientes medidas de seguridad al clúster de K8s:

- Controlar el **acceso al plano de control** de K8s, mediante listas de control de acceso y bloqueos IP.
- Administrar las **conexiones de los nodos**, evitando exponerlos públicamente a Internet. Es recomendable configurar los nodos para aceptar conexiones sólo en los puertos designados y sólo a servicios de K8s.
- Aplicar el **principio de mínimo privilegio** para administrar el acceso a los recursos por parte del plano de control y los nodos de K8s. Es recomendable que sea el propio clúster quién controle el permiso de acceso.
- Limitar el **acceso al banco de datos**, *etcd*, al plano de control para minimizar el número de conexiones a *etcd*.
- Siempre que sea posible, debemos **encriptar todos los mecanismos de almacenamiento.** Por ejemplo, etcd mantiene el estado de todo el clúster, por lo que es imprescindible encriptar su disco.

Clúster (Cluster)

Para securizar la capa de clúster hablaremos de medidas de protección de configuración de componentes y protección de las aplicaciones que son ejecutadas.

Para proteger la configuración del clúster debemos adoptar buenas prácticas de seguridad, limitando el acceso al clúster, asegurando que la

comunicación entre componentes es segura y que la carga de trabajo no vulnera la integridad del clúster. Más adelante hablaremos en más detalle de estos aspectos.

Por otro lado, para proteger el clúster debemos aplicar ciertas medidas de seguridad en el diseño de nuestras aplicaciones. En capítulos anteriores ya hemos explicado algunas de estas medidas, como las políticas de red o la aplicación de TLS (Transport Layer Security). Las medidas de seguridad recomendadas por K8s sobre un clúster son:

- Acceso a la API de K8s sólo mediante **autorización RBAC**, basada en roles. En un apartado posterior de este capítulo definiremos los roles con mayor detalle.
- Autenticación de usuarios contra la API mediante **módulos Authenticator**.
- **Administración de secretos** de aplicación. En la guía oficial de K8s puedes encontrar detalles sobre cómo encriptar el etcd y securizar la información confidencial de nuestra aplicación.
- Aplicar las **políticas de seguridad del pod**. A partir de la versión 1.21 de K8s, *PodSecurityPolicy* ha quedado obsoleto. Para versiones posteriores a esa, recomendamos el uso de *PodSecurityAdmission* o algún plugin de terceros que cumpla con los mismos objetivos.
- División en clases **QoS (Quality of Service)** para aplicar estándares de calidad de servicio y gestión de recursos del clúster.
- Aplicación de las **políticas de red** explicadas en esta guía para evitar amenazas procedentes de Internet.
- **Uso de TLS** para las operaciones *Ingress*, con el objetivo de realizar comunicaciones seguras con componentes ajenos a la red K8s.

Contenedor (Container)

La seguridad de un contenedor depende de la imagen del mismo y la configuración que se le aplica. Normalmente las garantías de los contenedores estarán basadas en las garantías que dan la nube y el clúster sobre el que se aloja. Sin embargo, existen una serie de recomendaciones a tener en cuenta a la hora de crear y configurar un contenedor:

- Se recomienda utilizar un **escáner de vulnerabilidades** para poder identificar las vulnerabilidades que amenazan un contenedor, desde

su configuración de usuarios y permisos hasta los programas instalados.

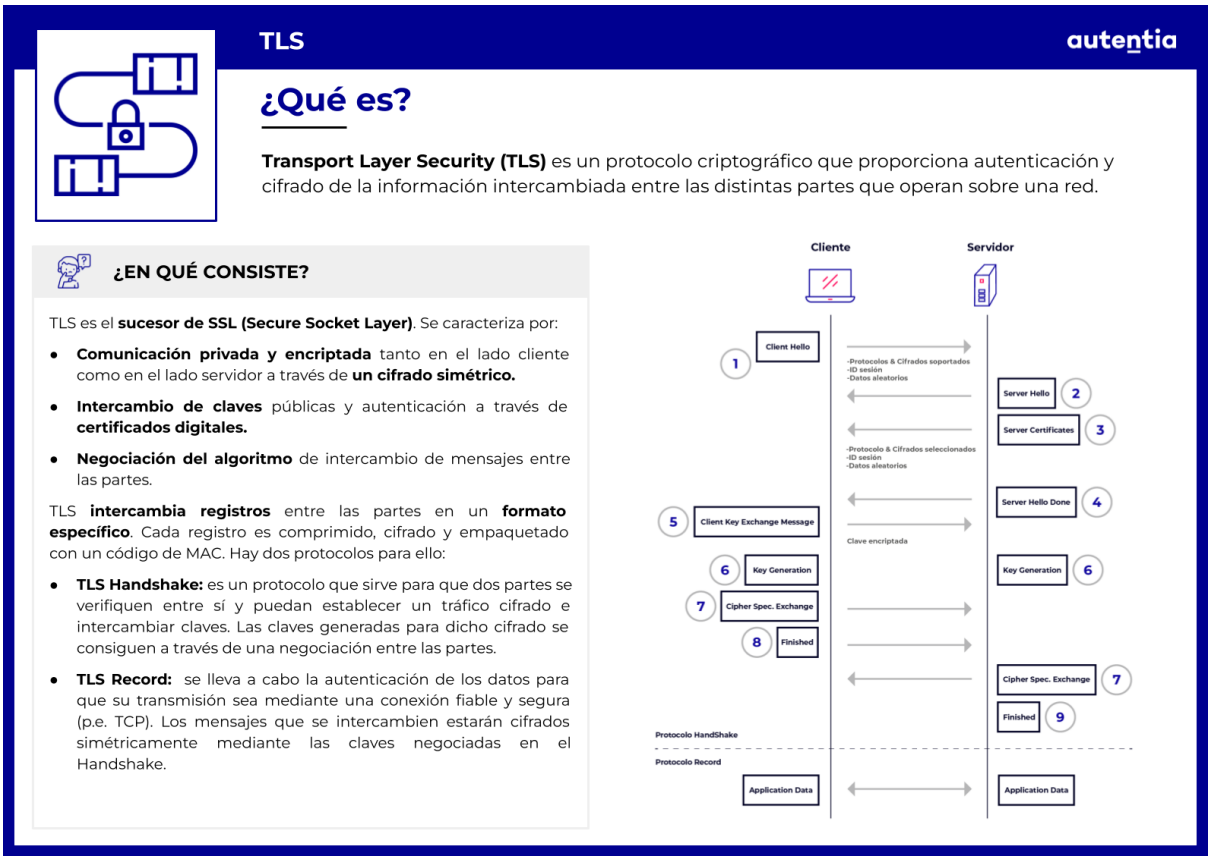
- La **firma de imágenes de contenedores** es muy útil para garantizar la integridad del contenido de los contenedores. Evitando el riesgo de lanzar un contenedor manipulado que supone una amenaza para nuestra aplicación.
- Aplicar el **principio de mínimo privilegio**. Esto implica el uso de imágenes mínimas, sin más paquetes o software que los estrictamente necesarios, privilegios mínimos para los usuarios del contenedor y la eliminación de componentes innecesarios. Existen herramientas de depuración, como CURL, con las que un atacante podría llegar a obtener control remoto del contenedor.
- **Automatizar** los **procesos de seguridad**. Los procesos manuales tienen riesgo de sufrir errores, por eso es recomendable automatizar las medidas mencionadas.

Código (Code)

De manera similar al contenedor, la seguridad del código se encuentra fuera de las políticas de seguridad de K8s. Sin embargo, es responsabilidad del equipo de desarrollo asegurar el código de la aplicación, ya que es uno de los principales focos de ataque y la capa sobre la que más control tenemos.

Aquí tienes una lista de recomendaciones de seguridad para el código fuente de tus aplicaciones, independientemente de si está montado sobre K8s o no:

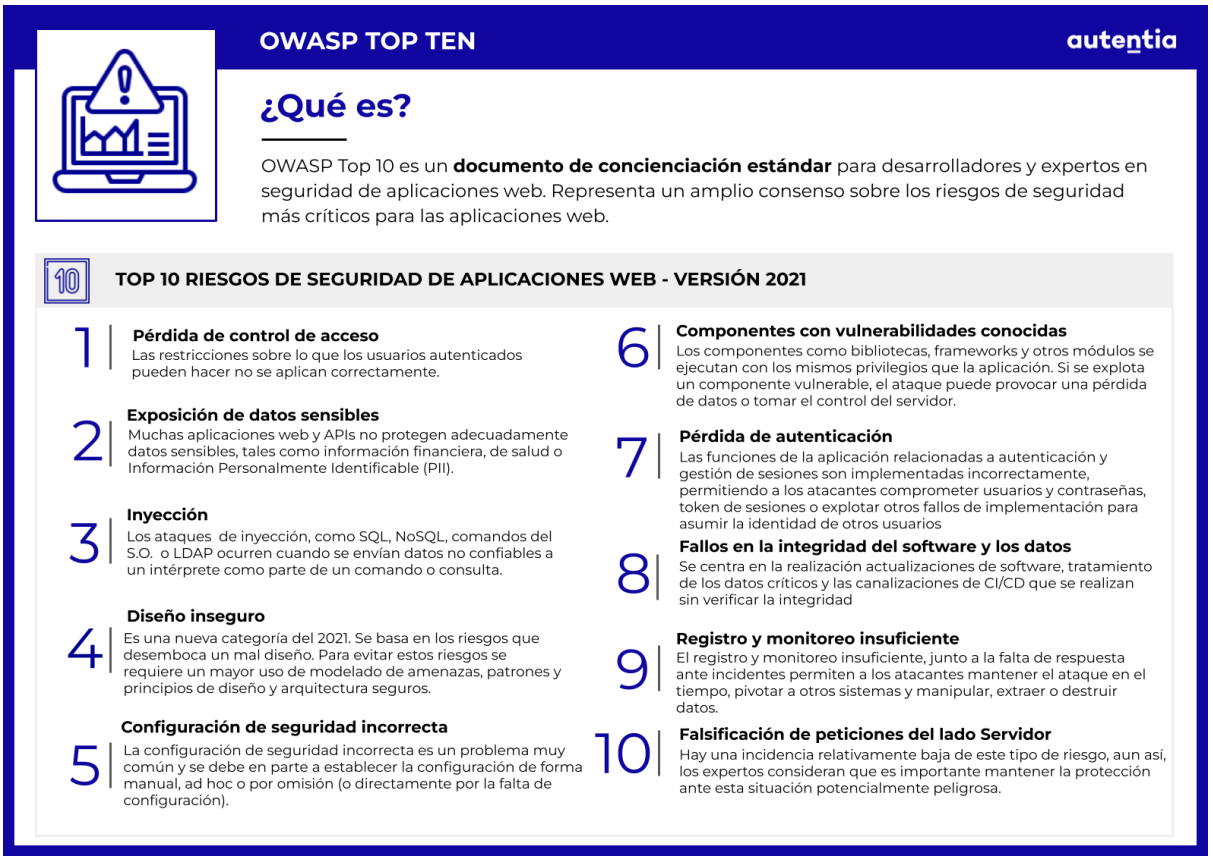
- Asegurar la **comunicación segura mediante TLS**. Si el código necesita comunicarse por la red, en este caso TCP, siempre es recomendable encriptar todos los datos que formen el tráfico de red. A menudo, es recomendable encriptar el tráfico entre los propios servicios de la aplicación, mediante un proceso de autenticación mutua o mTLS (mutual TLS).



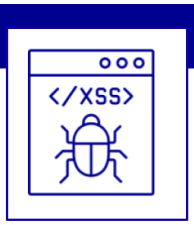
- **Exponer únicamente los puertos necesarios** para la actividad de la aplicación o, en caso necesario, para la recopilación de métricas.

- La gran mayoría de las aplicaciones hacen uso de **dependencias de terceros** para apoyar su funcionalidad y mejorar su productividad. Es importante hacer comprobaciones periódicas que aseguren la integridad de esas dependencias para poder reaccionar a tiempo en caso de nuevas vulnerabilidades.

- Es importante **asegurar el código fuente** que utiliza nuestra aplicación. Para ello, existen multitud de herramientas de análisis de código estático que nos permiten automatizar la búsqueda de piezas de código inseguras o vulnerables. Para decidir qué herramienta se ajusta mejor a nuestras necesidades, recomendamos visitar la página de OWASP.



- El código fuente no sólo contiene amenazas potenciales en tiempo de compilación, sino también en tiempo de ejecución. Algunos de los ataques más conocidos en este ámbito son la **inyección SQL**, **Cross Site Request Forgery (CSRF)** y **Cross Site Scripting (XSS)**. Estas vulnerabilidades tienen que ver con una mala configuración y/o implementación de servicios, repositorios o componentes de nuestra aplicación, y se detectan mediante **ataques de sondeo dinámico**. De nuevo, recomendamos visitar OWASP para escoger entre las herramientas de sondeo dinámico más populares.



XSS (Cross-Site Scripting)

autentia

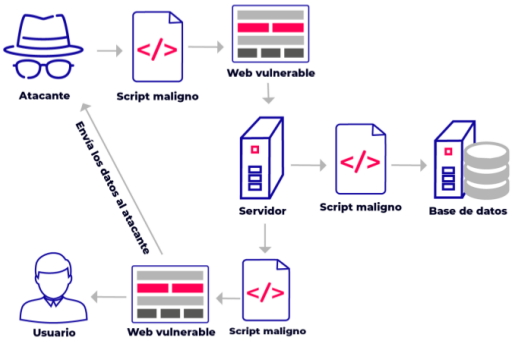
¿Qué es?

Consiste en conseguir **ejecutar código JavaScript en una página web** para tratar de acceder a datos confidenciales como las credenciales o las cookies, acceder a la cámara, etc.

VULNERABILIDADES

Existen **dos tipos** de vulnerabilidades:

- **Persistente:** el código JavaScript inyectado se queda almacenado en el servidor, por ejemplo formando parte de un comentario que aparece en un foro. Un navegador que cargue ese comentario ejecutará el código JavaScript persistido.
- **Reflejado:** el código JavaScript no se queda almacenado en el servidor, sino que el atacante de alguna manera consigue hacer que el usuario ejecute código JavaScript para robar datos sensibles.



SOLUCIÓN

Cualquier valor que pueda ser introducido por el usuario en la aplicación debe considerase no fiable y ser procesado antes de utilizarlo.

La mayoría de los frameworks se protegen contra este ataque escapando automáticamente cualquier texto que se utilice dentro del HTML.

Es importante seguir las recomendaciones del framework que se utiliza para prevenir los ataques XSS.

CSRF (Cross-Site Request Forgery)

autentia

¿Qué es?

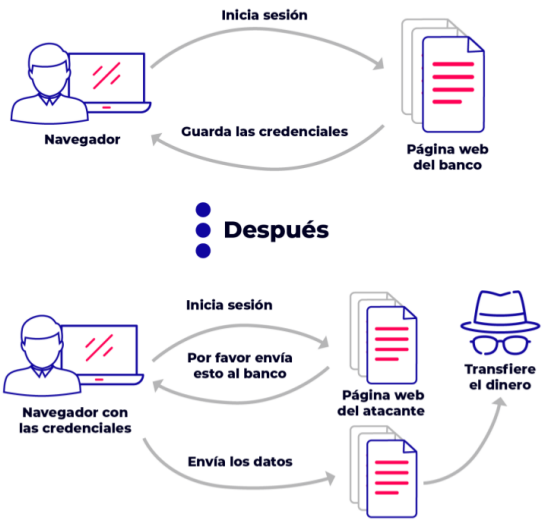
Consiste en hacer que **el usuario, sin saberlo, envíe una petición a algún servidor de algún sistema** en el que esté autenticado para hacer una transacción, modificar una contraseña, etc.

VULNERABILIDAD

Cuando el navegador envía una petición al servidor, envía también todas las cookies asociadas a ese dominio en la petición. Por este motivo no es necesario que el atacante tenga acceso a las cookies del usuario, solo necesita ser capaz de enviar la petición al servidor con la acción que desea y las cookies serán enviadas automáticamente por el navegador.

SOLUCIÓN

La técnica más utilizada para protegerse contra este ataque es generar un token CSRF en el lado del servidor que se envía al cliente. En cada petición HTTP que el cliente envié, el servidor espera recibir el token enviado. Si el token falta o el valor es incorrecto, la petición se rechaza.



Seguridad de un pod

Como ya hemos mencionado, las políticas de seguridad de pod tradicionales han quedado obsoletas desde la versión 1.21 de K8s. Por ello, en la versión 1.25 utilizaremos la aproximación *Pod Security Admission*.

Niveles de aislamiento

Los estándares de seguridad de pods en K8s establecen niveles de aislamiento diferentes, dependiendo de cuánto y cómo necesitamos restringir el comportamiento de los pods. Estos niveles se definen como perfiles de seguridad del pod, resumidos como:

- **Privileged:** Es una política o perfil completamente abierto, sin restricciones. Típicamente se encuentra en cargas de trabajo a nivel de sistema y/o infraestructura, en las que los usuarios que las administran son confiables y tienen privilegios. Si se aplica esta configuración a pods no preparados para ello, puede dar lugar a escaladas de privilegios en caso de ataque.
- **Baseline:** Está dirigida a restringir la estructura mínima necesaria del pod para prevenir ataques de escalada de privilegios conocidos. Aplica el principio del privilegio mínimo para la configuración del pod.
- **Restricted:** Su objeto es reforzar las medidas de seguridad del pod actual, aplicando las mejores prácticas y medidas muy restrictivas, a costa de compatibilidad. Común en operadores o desarrolladores de aplicaciones con una seguridad crítica, o en usuarios no autorizados en cualquier tipo de aplicación.

Estas políticas son acumulativas, aplicando desde la configuración menos restrictiva a la más restrictiva.

Seguridad de namespaces

Por otro lado, en K8s también se definen etiquetas de seguridad características de *namespaces* concretos. Es decir, mediante etiquetas podemos definir qué modo de control de admisión queremos aplicar en cada pod, siendo esa etiqueta equivalente a uno de los siguientes modos:

- **Enforce:** La violación de alguna política de seguridad provoca el fallo o rechazo del pod.

- **Audit:** La violación de alguna política de seguridad activa la adición de una etiqueta de auditoría, que quedará registrada en el *audit log*.
- **Warn:** La violación de alguna política de privacidad activa una alerta, orientada a usuarios, con información sobre dicha alerta y su casua.

Si se produce una violación en los modos *Audit* y *Warn*, los pods seguirán activos. Esto quiere decir que es **muy importante llevar a cabo auditorías periódicas y revisar y corregir los *warnings*** que lancen nuestros pods.

Pongamos un ejemplo en el que lanzamos una carga de trabajo consistente en un grupo de *Deployment* y *Jobs*. Los pods solicitados por estos objetos se crean a partir de un controlador que utiliza una plantilla definida en su llamante. Por defecto, los objetos de la carga de trabajo adquieren los modos *audit* y *warn*, mientras que crean pods a los que pueden aplicar el modo *enforce*.

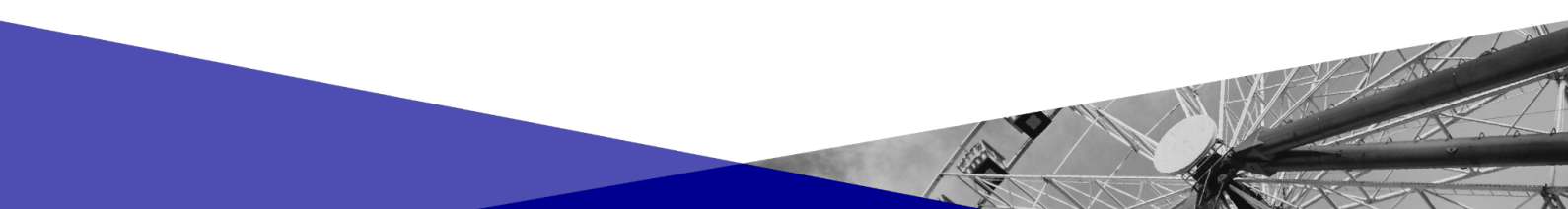
Excepciones

Al igual que podemos elegir qué modos se aplican a cada nodo, también podemos definir excepciones para algunos de ellos. Una excepción se encarga de permitir la creación de pods que, de otra forma, habrían sido rechazados debido a violaciones en la política de su namespace.

Estas se pueden definir en la configuración del *AdmissionController* y deben estar listadas explícitamente. Una excepción puede actuar en base a:

- **Nombres de usuario**: Las peticiones por parte de usuarios, con un nombre de usuario recogido en la definición de una excepción, serán ignoradas.
- **Nombres de clases Runtime**: Se ignoran pods y recursos de carga de trabajo definidos en base a un *runtime class name* recogido en una excepción.
- **Espacios de nombres**: Se ignoran pods y recursos de carga de trabajo recogidos en un espacio de nombres exento.

Volviendo al ejemplo del apartado anterior, recuerda que las excepciones se producen sobre los pods creados directamente, no sobre los recursos que los crean.



Seguridad y control de la API

En este capítulo hemos definido la aproximación por capas de *Cloud Native Security* y definido las políticas principales de la seguridad de un pod. Sin embargo, estas no son las únicas medidas de seguridad que debemos tener en cuenta. Cuando desarrollamos un proyecto debemos seguir ciertas pautas o buenas prácticas que, en este caso, están enfocadas a K8s pero pueden extrapolarse a cualquier otro proyecto de desarrollo donde apliquen.

Comunicación segura

Cualquier clúster típico de K8s dispone de una API que sirve las peticiones que recibe en el puerto 443, **HTTPS**. Es decir, la comunicación con la API está protegida por TLS.

Debemos asegurar que todo nuestro tráfico está cifrado y se utilizan protocolos seguros para su transporte. Es imprescindible el uso de un certificado, ya sea de autoridad privada o basado en alguna autoridad de certificación reconocida.

De este modo, los clientes que interactúan con la API deben tener configurado el certificado para realizar cualquier acción, permitiendo al *API Server* confiar en la conexión y en la integridad y autoridad de los datos transferidos.

Autenticación

La etapa de autenticación toma lugar una vez que se ha establecido una conexión TLS exitosa. K8s provee diferentes mecanismos de autenticación para el *API Server*, entre los que podemos elegir aquellos que encajan mejor con los patrones de acceso a nuestro clúster. Un clúster puede utilizar uno o más módulos de autenticación a la vez, en función de los patrones de acceso que existan.

Todos los clientes de API necesitan autenticación, incluso los componentes internos al clúster. En este punto diferenciamos dos tipos de cuentas: cuentas de usuario y cuentas de servicio.

Las **cuentas de usuario** son cuentas destinadas a clientes, utilizados por humanos para interactuar con la API, suelen tener un ámbito global, con acceso a todos los espacios de nombres disponibles. Generalmente son

cuentas sincronizadas con la base de datos de la aplicación, donde la creación de un nuevo usuario implica la actualización de procesos de negocio complejos y le otorga al mismo ciertos privilegios en función de su rol dentro de la aplicación.

Por otro lado, las **cuentas de servicio** están destinadas a los procesos del clúster, ya sean administrativos de K8s o procesos que corren dentro de cada pod. Las cuentas de servicio son dependientes de su espacio de nombres, creadas según el principio del mínimo privilegio con el objetivo de ser temporales y desechables cuando el objeto que las utiliza finaliza su ciclo de vida. El objetivo de estas cuentas es ofrecer un método de autenticación ligero y portable para tareas específicas que no necesitan sincronización con la aplicación.

Volviendo al punto que nos concierne, los componentes internos de K8s típicamente utilizan cuentas de servicios, creadas junto con el clúster o como parte del proceso de instalación. De esta manera, si la autenticación tiene éxito, el usuario se valida y se registra su nombre de usuario para pasos posteriores.

Se recomienda utilizar módulos de autenticación consistentes, basados en **certificados o tokens de autenticación**, para que, junto con TLS, la identidad del cliente se pueda verificar.

Autorización

Cuando llegamos a esta fase la petición del usuario ya ha sido autenticada contra la API y tiene una cuenta de usuario o de servicio. Pero el siguiente paso es autorizar la petición. Esta autorización tiene lugar siempre que haya una política que confirme que el usuario de la petición tiene permisos suficientes para llevar a cabo esa acción.

Las peticiones de autorización deben incluir el nombre de usuario del solicitante, asociado a su autenticación, la acción solicitada, generalmente un verbo HTTP, y el objeto afectado por la petición.

Si hemos seguido los consejos de configuración explicados hasta el momento, los usuarios tendrán el privilegio mínimo para poder interactuar con la API. De no ser así, recomendamos aplicar las buenas prácticas mencionadas y **otorgar a cada cuenta, de usuario o servicio, los privilegios mínimos para poder realizar los casos de uso que fueron diseñados para ellos**.

Las peticiones de autorización pueden ser aceptadas, si la cuenta tiene suficientes privilegios, o denegadas, en caso contrario. Es importante que los atributos de esta petición se definan en formatos REST ya que los sistemas de control pueden interactuar con otras APIs ajenas a la de K8s.

Los encargados de determinar si los usuarios tienen los privilegios necesarios para lanzar la petición son los módulos de autorización, entre los que destacamos: Control de Acceso Basado en Atributos (ABAC), Control de Acceso Basado en Roles (RBAC) y el modo *Webhook*.

Control de Acceso Basado en Atributos (ABAC)

Define un paradigma de control de acceso en el que las peticiones se aprueban mediante el uso de políticas que declaran ciertos atributos. Pongamos este ejemplo de política:

```
apiVersion: v1
kind: Policy
spec:
   user: "bob"
   namespace: "myProject"
   resource: "services"
   Readonly: true
```

Esta política le permite al usuario de nombre “bob” acceder a los servicios del espacio de nombres “myProject” con permisos de sólo lectura. Aquí te dejamos una serie de ejemplos útiles para la definición de políticas basadas en atributos.

Control de Acceso Basado en Roles (RBAC)

Define un paradigma de control de acceso basado en los roles de los usuarios de la aplicación. Permite controlar el acceso a los recursos de red haciendo uso del *API group*, un conjunto de rutas relativas a la API de K8s, donde podemos configurar políticas basadas en roles de forma dinámica.

Este tipo de control de acceso necesita activarse, ya que no viene activado por defecto. Para ello haremos uso del parámetro --authorization-mode aplicado a **kube-apiserver**.

Estas políticas pueden definirse a nivel de espacio de nombres, gracias a los objetos *Role* y *RoleBinding*, o a nivel de clúster, por medio de *ClusterRole* y *ClusterRoleBinding*.

Un *Role* o *ClusterRole* está compuesto por normas que representan los permisos de cada espacio de nombres o clúster. Estos permisos son puramente aditivos, si no existe una norma definida, ese permiso no será concedido. Para seguir con el ejemplo mostrado en ABAC, veamos la siguiente definición de *Role* y *ClusterRole*:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
   namespace: myProject
   name: service-reader
rules:
 - apiGroups: [""]
   resource: ["services"]
   verbs: ["get", "watch", "list"]
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
   name: service-reader
rules:
 - apiGroups: [""]
   resource: ["services"]
   verbs: ["get", "watch", "list"]
```

Estos ejemplos definen permisos de lectura (verbos get, watch y list) sobre servicios a nivel de espacio de nombres o de clúster respectivamente. Sólo aplican a aquellos usuarios con el rol “service-reader”.

Un *RoleBinding* o *ClusterRoleBinding* es el encargado de relacionar los permisos, definidos en un rol, con un conjunto de usuarios. También es el encargado de conceder esos permisos en función de la pertenencia, o no, de un usuario a un rol. La diferencia entre *RoleBinding* y *ClusterRoleBinding* reside en que *RoleBinding* define la relación de roles en un espacio de nombres específico, mientras que *ClusterRoleBinding* lo hace sobre todo el clúster. Sin embargo, un *RoleBinding* puede referenciar a un *ClusterRole* ajeno a su espacio de nombres, de hecho un *ClusterRoleBinding* no es más que un *ClusterRole* referenciado en todos los espacios de nombres del clúster. Veamos un ejemplo de cada uno:

```
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: RoleBinding
metadata:
   namespace: myProject
   name: service-reader
subjects:
 - kind: User
   name: bob
   apiGroup: rbac.authorization.k8s.io
 roleRef:
   kind: Role
   name: service-reader
   apiGroup: rbac.authorization.k8s.io
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
   namespace: myProject
   name: service-reader
subjects:
 - kind: User
   name: bob
   apiGroup: rbac.authorization.k8s.io
 roleRef:
   kind: ClusterRole
   name: service-reader
   apiGroup: rbac.authorization.k8s.io
```

Los ejemplos anteriores asocian al usuario bob a los roles definidos previamente. Los cuadros de código muestran los archivos YAML que usaremos para definir un *RoleBinding* y un *ClusterRoleBinding*, respectivamente. Al igual que ocurría con los roles, la principal diferencia es el ámbito de acción. Sin embargo, a diferencia de los roles, en la definición de *ClusterRoleBinding* no se omite el campo **.metadata.namespace** ya que, en lugar de indicar cuál es el espacio de nombres de este objeto, indica a cual afecta.

Si estás trabajando con roles en un proyecto, te recomendamos la lectura de esta recopilación de utilidades básicas de consola de la guía oficial de K8s.

WebHook

Define un paradigma de control de acceso basado en autorización externa a K8s. Cuando se produce un nuevo evento, en nuestro caso una petición de autorización, *WebHook* envía un mensaje de tipo POST a un servicio REST que conoce, evalúa y concede o niega el acceso a una cuenta, de usuario o servicio, en función de sus privilegios.

Para utilizar *WebHook* necesitamos definirlo como en el siguiente ejemplo:

```
apiVersion: v1
kind: Config
clusters:
 - name: externalAuthService
   cluster:
     certificate-authority: {CAPath}/ca.pem
     server: https://authz.example.com/authorize
users:
 - name: private-ca-server
   user:
     client-certificate: {clientPath}/cert.pem
     client-key: {keyPath}/key.pem
current-context: webhook
contexts:
 - context:
      cluster: externalAuthService
      user: private-ca-server
   name: webhook
```

Mediante este ejemplo definimos el uso de un servicio REST externo, **externalAuthService** al que referenciamos como **private-ca-server**, que actúa como autoridad de certificación para autorizar cuentas por medio de webhook.

Para ello debemos indicar el certificado CA y la URL de certificación en la sección **.clusters.cluster.***, observa que podemos definir más de un clúster en esa lista. El siguiente paso es crear un **.users.user** que contenga un certificado propio y la clave para validar su autoridad, de nuevo, podemos definir varios. Por último, este archivo de configuración está definido según *kubeconfig*, por tanto, debemos indicar un contexto. En este caso indicamos que usaremos webhook y asociaremos en **.contexts.context.*** el

servicio remoto y la configuración de usuario que le hemos dado.

Auditoría

Define un conjunto de registros con orden cronológico cuyo objetivo es documentar una serie de actividades relacionadas con la seguridad. Los usuarios objetivo de esta auditoría son tanto usuarios humanos, individuales y administradores, como cuentas de servicio.

El componente encargado de llevar a cabo la auditoría es **kube-apiserver**. Para ello sigue un reglamento que define qué se audita qué no, y persiste los resultados en un "backend" que funciona como banco de registros. Cada petición genera un evento, dependiendo de su fase de ejecución, lo que activa el pre-procesamiento según ese reglamento.

Dicho reglamento define reglas concretas sobre qué eventos deben registrarse y cuántos de sus datos se deben mostrar. Sigue la estructura **audit.k8s.io** y realiza una búsqueda secuencial en la lista de reglas cada vez que recibe un evento. Al igual que un *Log*, tenemos distintos niveles de auditoría:

- **None:** No se registra ningún evento correspondiente a esa regla.
- **Metadata:** Registro de metadatos de la petición: usuario, marca de tiempo, verbo, recurso, permisos...)
- **Request**: Registro de metadatos y además del cuerpo de la petición. (Sólo aplica a peticiones de recurso)
- **RequestResponse:** Registro de metadatos y cuerpos de la petición y la respuesta. (Tampoco aplica a peticiones que no sean de recurso)

Configuraciones por defecto

Cuando llevamos a cabo proyectos sobre K8s, instalamos muchas utilidades, servicios y componentes que se instalan sobre puertos por defecto y por tanto conocidos. No sólo ocurre con K8s, *mysql* típicamente se lanza sobre el puerto 3306, al igual que muchas otras herramientas necesarias para el desarrollo de nuestro proyecto.

En términos de seguridad, toda la información que expongamos al exterior tiene un riesgo. Y las configuraciones por defecto no están excluidas, ya que si una amenaza analiza los puertos que tiene abiertos o utiliza

internamente nuestra aplicación, puede identificar a qué herramientas se está enfrentando y dirigir un ataque hacia las que considere más vulnerables.

Hemos hablado de puertos predeterminados, pero no son la única configuración insegura que se presenta por defecto. Estas configuraciones peligrosas empiezan en los propios contenedores, con aspectos como los siguientes:

- **Imágenes comprometidas:** La configuración por defecto no verifica las imágenes descargadas, lo que puede dar lugar a lanzar nuestra aplicación sobre imágenes comprometidas.
- **Herramientas y dependencias innecesarias:** Al crear contenedores se instalan herramientas y dependencias por defecto, que si no utilizamos nunca, deben ser eliminadas. Estas herramientas hoy no suponen un problema, pero si la versión que tenemos acaba siendo vulnerable, tendremos un hueco en nuestra seguridad del que no somos conscientes.
- **Superusuario por defecto:** Siempre debemos crear un usuario ajeno al superusuario, con los privilegios mínimos para poder llevar a cabo la actividad asignada al contenedor. De lo contrario, si un atacante accede al contenedor, tendría permisos de superusuario y podría intentar acceder a nuestro host.
- **Límites de recursos:** Los límites de memoria y CPU no vienen definidos por defecto. Para asegurar la disponibilidad de recursos para todos los componentes del clúster, y protegernos de un ataque de inanición de recursos o DOS (Denial Of Service), es recomendable establecer límites que, una vez alcanzados, impidan que el contenedor solicite más recursos.

Además de securizar el contenedor debemos confirmar que la configuración de K8s es segura. Para ello tendremos en cuenta las siguientes recomendaciones:

- Configurar **controles de acceso basado en roles (RBAC):** Aunque este sistema de autorización está habilitado por defecto, se recomienda configurar *Role*s y *RoleBinding*s para que sean los propios espacios de nombres quienes brinden acceso a una cuenta.
- **Políticas de red de K8s:** Para hacer uso de estas políticas debemos instalar nosotros el plugin de red que nos permite controlar el tráfico de entrada y salida de la aplicación. Es muy importante que

configuremos las políticas de red para garantizar que sólo se establecerán las conexiones necesarias con los contenedores, y sólo en los puertos designados.

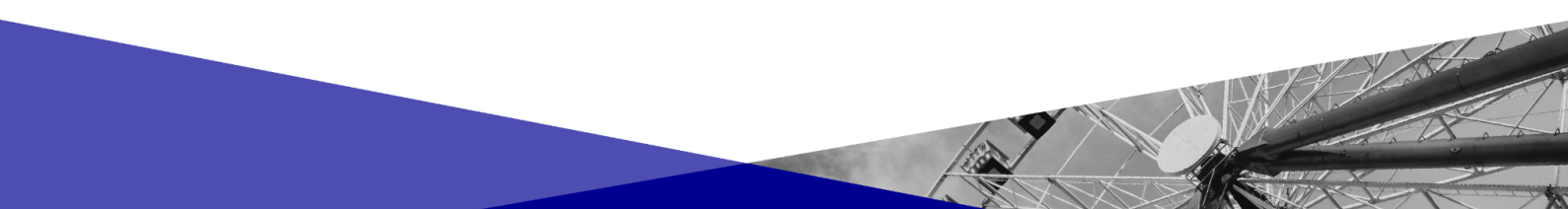
- **Espacios de nombres:** K8s define un único espacio de nombres por defecto. Mezclar cargas de trabajo supone un riesgo para los recursos que comparten, la garantía de aislamiento de pods y para la seguridad, pues todos los contenedores tienen visibilidad entre sí. Se recomienda separar los recursos en espacios de nombres para aislar las cargas de trabajo y los pods.
- **Registro de auditorías:** Es muy importante configurar esta función que no se activa por defecto. Las auditorías nos permitirán llevar un registro de las peticiones sospechosas o los fallos de autorización para poder detectar un error en el desarrollo o un intento de ataque.

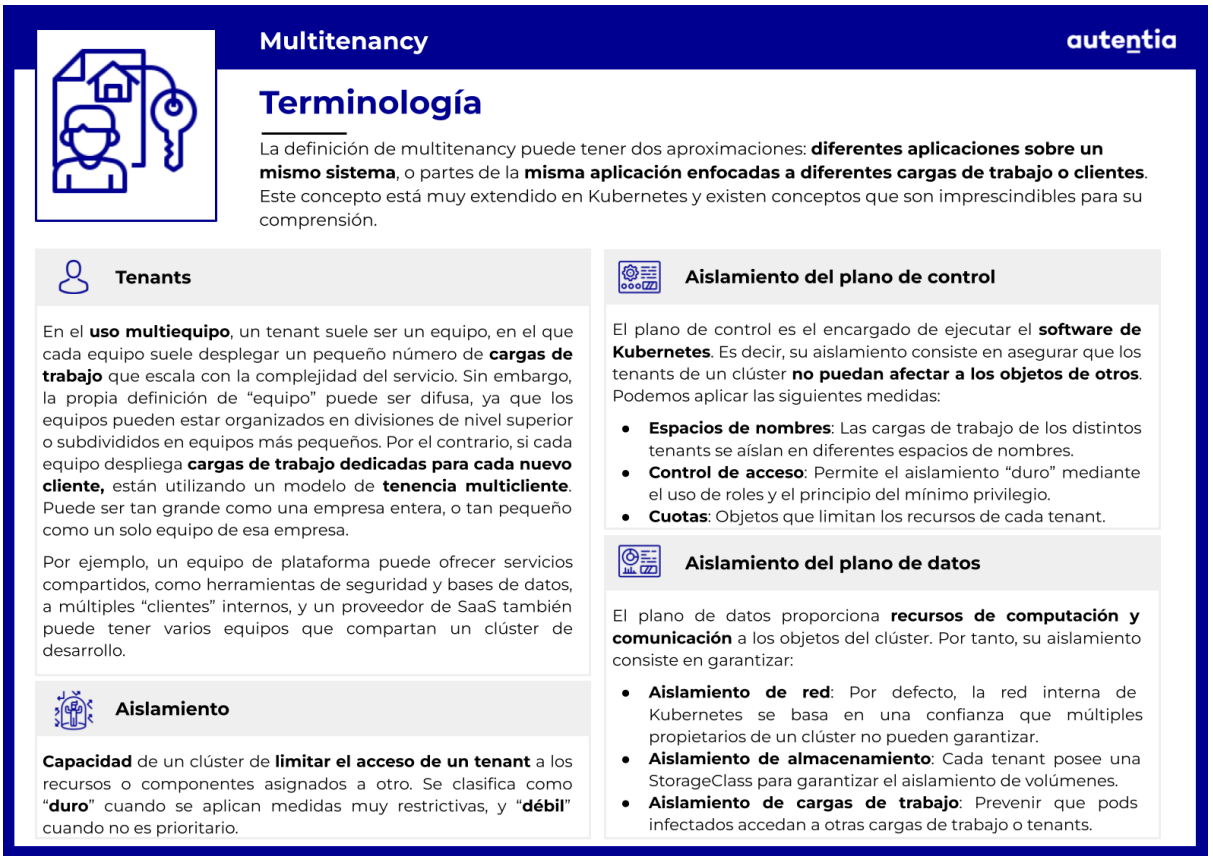
Llevar a cabo estas recomendaciones no nos exime de riesgo, pero aporta mucho valor a la seguridad de nuestra aplicación y reduce su riesgo de ser vulnerada. Si te interesa seguir descubriendo cómo mejorar tu seguridad mediante el uso de buenas prácticas y técnicas de detección de puntos débiles te recomendamos el siguiente artículo.

Multitenancy

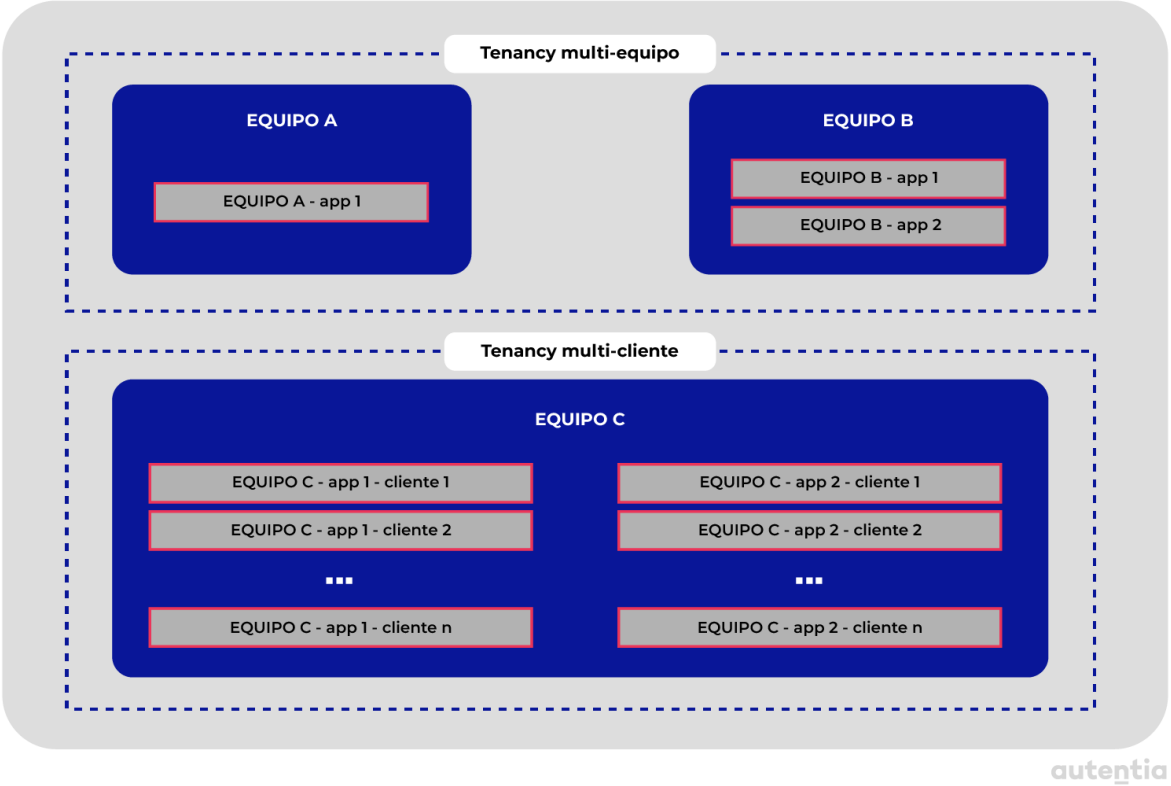
Un sistema *multitenancy* se basa en compartir clústers sobre una misma instancia, también conocido como **arquitectura multipropietario**. Compartir clústers ayuda a ahorrar costes y simplificar la gestión de la aplicación.

Sin embargo, el problema principal de esta aproximación es que esas aplicaciones comparten recursos, seguridad y tráfico, por lo que debemos asegurar un correcto aislamiento y una administración justa para todos los propietarios.





La definición de *multitenancy* puede tener dos aproximaciones: diferentes aplicaciones sobre un mismo sistema, o partes de la misma aplicación enfocadas a diferentes cargas de trabajo. En un mismo cluster es posible encontrar distintos tipos de tenancy, por ejemplo:



En la imagen anterior podemos ver cómo un único clúster está dividido en varios equipos, cada uno trabajando en distintas aplicaciones.

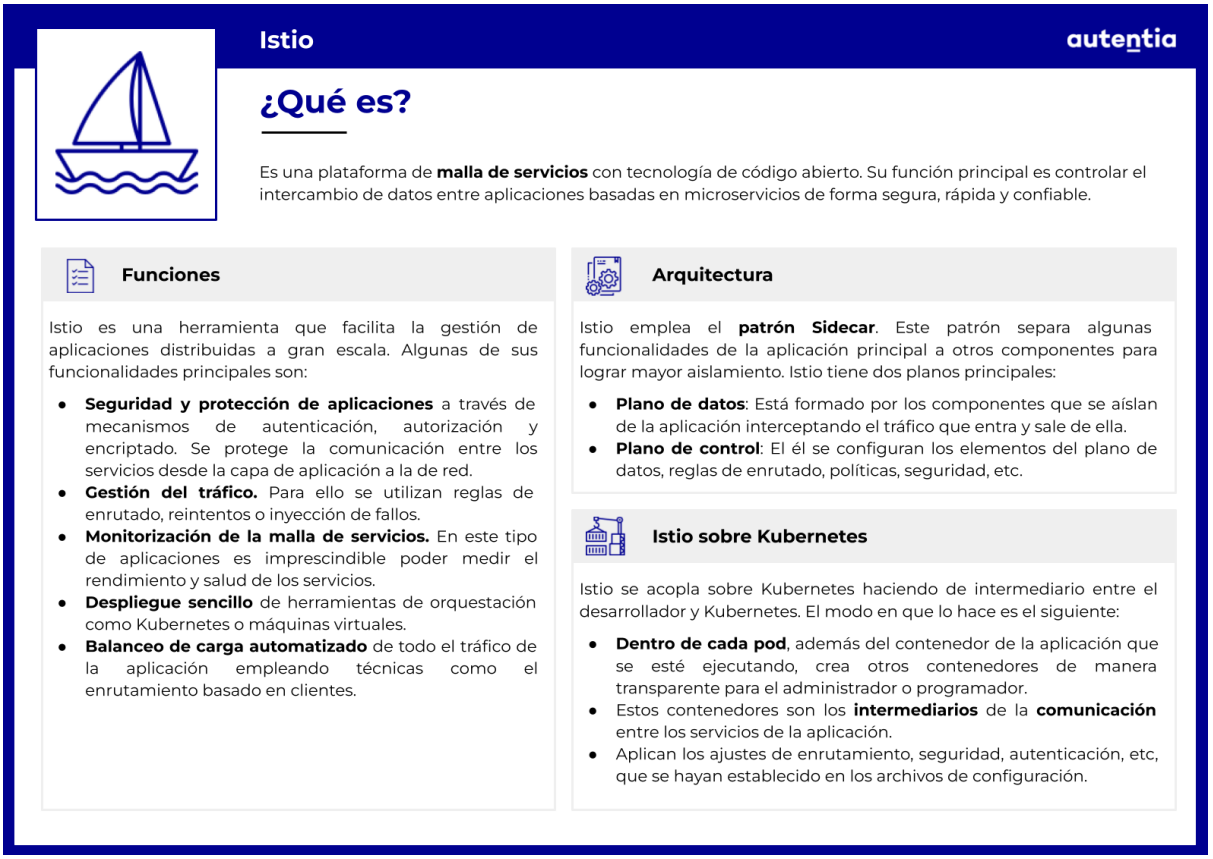
Si observamos los equipos A y B, en la parte superior, se nos presenta un ejemplo claro de tenencia multi-equipo. En esta, ambos equipos despliegan sus aplicaciones sobre el mismo clúster, donde, normalmente, las cargas de trabajo necesitan comunicarse entre sí o con otras de diferentes clústers. Este tipo de tenencia necesita un cierto grado de confianza entre los equipos, los límites de los recursos que pueden utilizar bien definidos, y una capa de seguridad compuesta por acceso basado en roles (RBAC) y políticas de red.

En la parte inferior vemos un ejemplo de tenencia multi-cliente, en la que un único equipo da servicio a múltiples clientes de una o varias aplicaciones. Este tipo de tenencia normalmente divide los recursos del clúster entre los clientes, creando cargas de trabajo independientes para cada uno. En este caso, los clientes no tienen acceso directo al clúster, por lo que las medidas de seguridad se centran en el aislamiento y la principal prioridad es la optimización de recursos.

Istio

Las aplicaciones que forman sistemas grandes son complicadas de gestionar de forma manual. K8s automatiza y orquesta tareas de despliegue y escalado de aplicaciones mediante contenedores. Istio se puede utilizar junto a K8s para simplificar las operaciones de red entre servicios, las de auditoría y las de observabilidad.

A modo de ejemplo, cuando una aplicación tiene una arquitectura de microservicios desplegada sobre K8s, éstos se ejecutan en los contenedores alojados a lo largo de los diferentes pods que se distribuyen sobre los diferentes nodos del clúster. K8s gestiona los recursos de los nodos añadiendo pods según la demanda. La función de **Istio** en este punto es **inyectar contenedores** adicionales en los pods para aumentar la seguridad, gestión y supervisión.



¿Qué es y para qué sirve?

Istio es una plataforma de **malla de servicios** con tecnología de código abierto que controla el intercambio de datos entre aplicaciones distribuidas basadas en microservicios de forma segura, rápida y confiable.

Gracias a esta herramienta, se pueden gestionar correctamente aplicaciones distribuidas a gran escala. Algunas de las funcionalidades de Istio son:

- **Proteger aplicaciones** nativas de la nube mediante autenticación, autorización y encriptado. Aplicado a K8s, se puede proteger la comunicación entre pods y servicios desde la capa de aplicación a la de red.
- **Gestionar el tráfico** mediante reglas de enrutado, reintentos o inyección de fallos. Esto permite controlar el flujo del tráfico y las llamadas a la API.
- **Monitorizar la malla de servicios** de forma que se pueda medir el rendimiento de los servicios.
- **Despliegues sencillos** de K8s y máquinas virtuales.
- **Balanceo de carga automatizado** en todo el tráfico con funciones como el enrutamiento basado en clientes.

El uso de la malla de servicios de Istio simplifica las implementaciones y además reduce la carga de trabajo de los desarrolladores.

Arquitectura de Istio

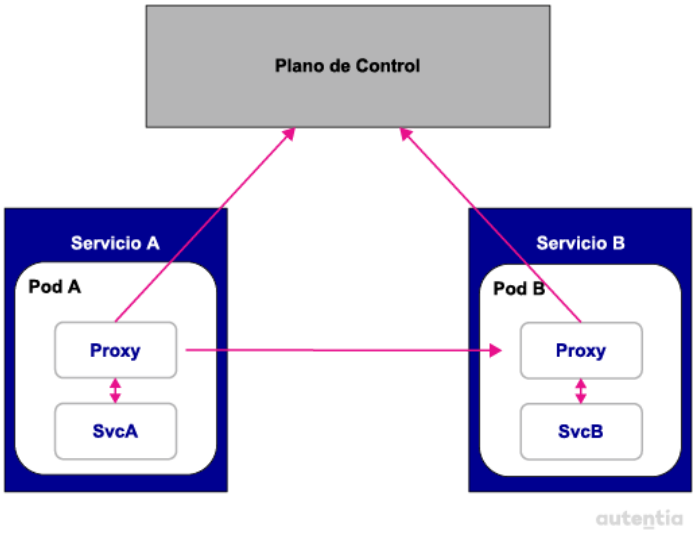
Istio emplea el **patrón Sidecar**. Este patrón consiste en trasladar algunas funcionalidades de las aplicaciones a otros componentes para desacoplarlas del resto de la aplicación. Así se logra un mayor aislamiento.

A modo de ejemplo, si tenemos un **sidecar-proxy**, la funcionalidad desacoplada que acompaña a la aplicación es un proxy que intercepta todas las comunicaciones de red. Aplicado a K8s, dentro de un pod tenemos el propio contenedor de la aplicación y además, otro contenedor que será el proxy.

Istio tiene dos **piezas principales** que se conocen como planos:

- **Plano de datos**: Está formado por los sidecar de tipo proxy que se ejecutan con cada pieza de la aplicación e interceptan el tráfico que entra y sale de ella.
- **Plano de control**: Se encarga de gestionar y configurar los proxies, es decir, las reglas de enrutado, políticas, seguridad, etc.

En el siguiente diagrama vemos un ejemplo de arquitectura de Istio sobre K8s.



Algunos **componentes** de Istio son:

- **Proxy Envoy**: La función de un proxy es interceptar el tráfico de entrada y salida al contenedor de la aplicación. Es el intermediario de comunicación entre servicios.
- **Mixer**: Controla el acceso y recibe las métricas que generan las llamadas al proxy. Separa de los servicios funciones como el control de accesos, gestión de cuotas, logging, etc., y las configura aparte para evitar acoplamiento. Mixer utiliza distintos plugins para realizar estas tareas.
- **Pilot**: Se encarga de gestionar el registro de servicios, enrutado, timeouts, reintentos, etc.
- **Istio-Auth**: Sirve para securizar la comunicación entre servicios y usuarios finales mediante TLS. Utiliza políticas de acceso en base a identidad mediante claves y certificados.

Istio sobre Kubernetes

Istio se acopla sobre K8s en **capas** siguiendo el **patrón Sidecar**. Se encarga de añadir contenedores de manera transparente para el programador y administrador del sistema. Dirigen el **tráfico** y se encargan de supervisar las **interacciones** entre componentes. Si nos fijamos en el diagrama anterior, vemos cómo los servicios no se comunican directamente entre ellos, sino a través del proxy de Istio.

La configuración de Istio y K8s se realiza del mismo modo que todos los

elementos de K8s, a través de archivos YAML. Istio permite supervisar el estado de las aplicaciones en ejecución en K8s y proporciona información sobre el clúster y los nodos. La interfaz de Istio es similar a la de K8s. La principal diferencia es que se pueden aplicar políticas sobre el clúster de K8s para que su gestión sea menos tediosa y no necesite generar código manualmente.

Instalación del panel de control de Istio en Kubernetes

En la documentación oficial de Istio podemos encontrar distintas guías de instalación en función de las necesidades de nuestro sistema.

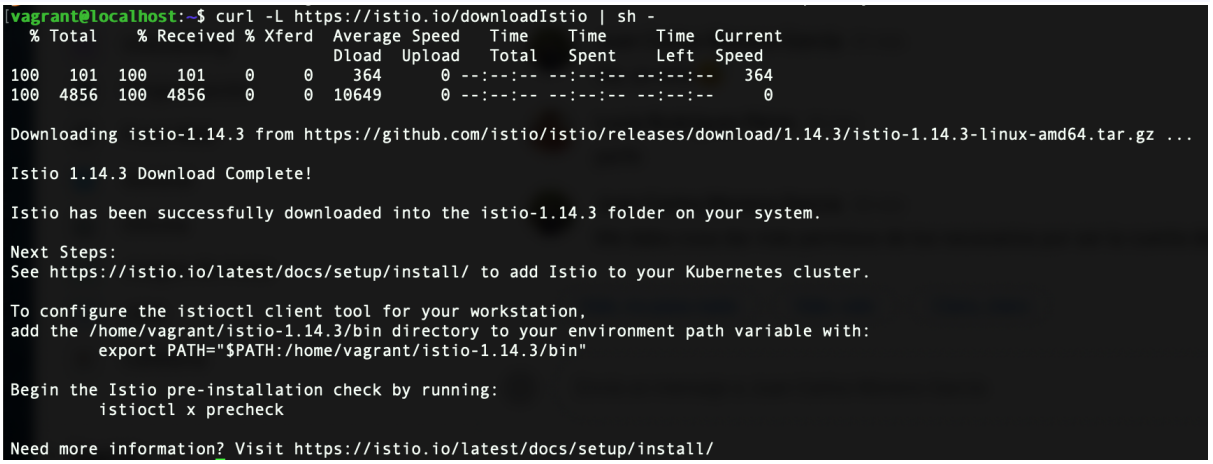
En esta sección, vamos a explicar cómo hacer la instalación básica de Istio sobre un clúster de Minikube.

En primer lugar, arrancamos Minikube:

```
minikube start
```

Una vez que hemos arrancado el clúster, vamos a instalar la última versión de Istio con el siguiente comando:

```
curl -L https://istio.io/downloadIstio | sh -
```

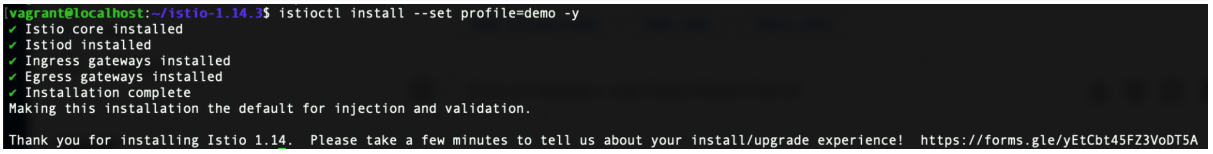


Ahora tenemos que configurar la variable de entorno para utilizar el cliente **istioctl**. En primer lugar, la carpeta que utilizaremos tendrá el nombre de la versión que hayamos instalado, en este caso será istio-1.14.3. Puedes ver el nombre de la carpeta en la imagen anterior: “Istio has been succesfully downloaded into de istio-{version} folder on your system.” Vamos a ejecutar los siguientes comandos:

```
cd istio-1.14.3
export PATH=$PATH:$PWD/bin
```

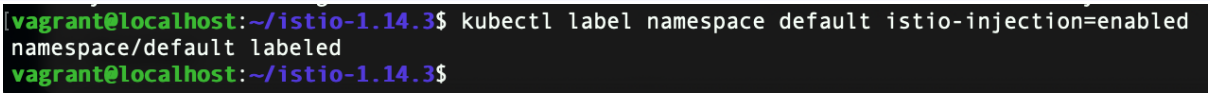
Una vez que tenemos configurada la variable de entorno, vamos a instalar con **istioctl** el perfil demo. Este perfil es bueno para pruebas, aunque también hay perfiles para producción o pruebas de rendimiento.

```
istioctl install --set profile=demo -y
```



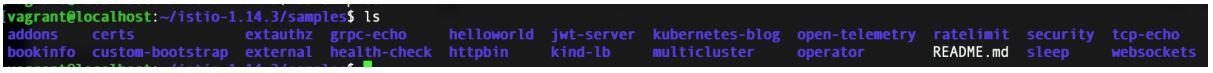
Por último, vamos a configurar una etiqueta con **kubectl** para que Istio pueda crear automáticamente proxies Envoy en la aplicación que vamos a configurar más tarde.

```
kubectl label namespace default istio-injection=enabled
```



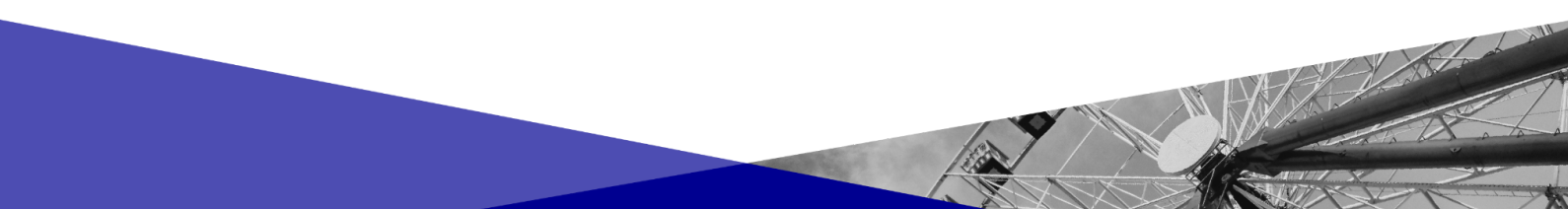
Desarrollo de una aplicación

Siguiendo el punto anterior, vamos a utilizar la instalación básica para desarrollar una aplicación sencilla a modo de ejemplo. Vamos a utilizar una de las aplicaciones de la carpeta de ejemplos/samples que se ha descargado al instalar Istio.



Para empezar, cogemos el archivo de configuración YAML y lo ejecutamos con kubectl como hemos hecho a lo largo de toda esta guía.

```
kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml
```



```
vagrant@localhost:~/istio-1.14.3$ kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml
service/details created
serviceaccount/bookinfo-details created
deployment.apps/details-v1 created
service/ratings created
serviceaccount/bookinfo-ratings created
deployment.apps/ratings-v1 created
service/reviews created
serviceaccount/bookinfo-reviews created
deployment.apps/reviews-v1 created
deployment.apps/reviews-v2 created
deployment.apps/reviews-v3 created
service/productpage created
serviceaccount/bookinfo-productpage created
deployment.apps/productpage-v1 created
vagrant@localhost:~/istio-1.14.3$
```

Para comprobar que la aplicación ha arrancado, comprobamos por un lado los pods y por otro los servicios. Para los pods, ten en cuenta que tardarán un poco en estar todos listos, espera hasta que todos estén en estado “running” y “READY 2/2”.

```
kubectl get pods
```

```
vagrant@localhost:~/istio-1.14.3/samples/bookinfo/src/reviews$ kubectl get pods
NAME                              READY   STATUS    RESTARTS      AGE
details-v1-b48c969c5-z84rt        2/2     Running   1 (15m ago)   22m
hello-node-6d5f754cc9-7h5kw       2/2     Running   0             10m
productpage-v1-74fdfbd7c7-nl29n   2/2     Running   0             10m
ratings-v1-b74b895c5-zncfv        2/2     Running   1 (15m ago)   22m
reviews-v1-68b4dcbdb9-s9ml6       2/2     Running   2 (15m ago)   22m
reviews-v2-565bcd7987-vgp79       2/2     Running   2 (15m ago)   22m
reviews-v3-d88774f9c-7tzg4        2/2     Running   2 (15m ago)   22m
vagrant@localhost:~/istio-1.14.3/samples/bookinfo/src/reviews$
```

```
kubectl get services
```

```
vagrant@localhost:~/istio-1.14.3$ kubectl get services
NAME          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
details       ClusterIP   10.104.197.183   <none>        9080/TCP   115s
kubernetes    ClusterIP   10.96.0.1        <none>        443/TCP    18d
productpage   ClusterIP   10.111.98.182    <none>        9080/TCP   115s
ratings       ClusterIP   10.107.220.66    <none>        9080/TCP   115s
reviews       ClusterIP   10.102.228.155   <none>        9080/TCP   115s
vagrant@localhost:~/istio-1.14.3$
```

Con el siguiente comando, verificamos que la aplicación está corriendo dentro del clúster y sirviendo páginas HTML.

```
kubectl exec "$(kubectl get pod -l app=ratings -o \
jsonpath='{.items[0].metadata.name}')" -c ratings -- curl -sS \
productpage:9080/productpage | grep -o "<title>.*</title>"
```

```
<title>Simple Bookstore App</title>
```

Si has llegado correctamente al último paso, la aplicación está corriendo. Ahora vamos a abrir la aplicación al exterior. Lo primero es asociar la

aplicación con el *gateway* de Istio y comprobar que no ha habido ningún problema con la configuración:

```
kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml
```

```
[vagrant@localhost:~/istio-1.14.3$ kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml
gateway.networking.istio.io/bookinfo-gateway created
virtualservice.networking.istio.io/bookinfo created
```

Con el siguiente comando, Istio comprueba que la configuración que hemos realizado es correcta. En caso de que hayamos cometido algún error o haya algún componente como pods que no esté supervisado por Istio, se mostrará por pantalla un mensaje indicando cuál es el problema. Por ejemplo, si hemos creado algún pod antes de instalar Istio, seguramente aparezca un *warning* indicando que ese pod no tiene el Proxy Sidecar de Istio.

```
istioctl analyze
```

```
[vagrant@localhost:~/istio-1.14.3$ istioctl analyze

✔ No validation issues found when analyzing namespace: default.
```

Para ver cómo los pods que se han creado tienen el Proxy Sidecar, puedes utilizar este comando y abrir el archivo de configuración de cualquier pod de la aplicación:

```
kubectl describe pod {nombre_del_pod}
```

```
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  16h                default-scheduler  Successfully assigned default/details-v1-b48c969c5-2zq9k to minikube
  Normal   Pulled     16h                kubelet            Container image "docker.io/istio/proxyv2:1.15.0" already present on machine
  Normal   Created    16h                kubelet            Created container istio-init
  Normal   Started    16h                kubelet            Started container istio-init
  Normal   Pulled     16h                kubelet            Container image "docker.io/istio/examples-bookinfo-details-v1:1.16.4" already present on machine
  Normal   Created    16h                kubelet            Created container details
  Normal   Started    16h                kubelet            Started container details
  Normal   Pulled     16h                kubelet            Container image "docker.io/istio/proxyv2:1.15.0" already present on machine
  Normal   Created    16h                kubelet            Created container istio-proxy
  Normal   Started    16h                kubelet            Started container istio-proxy
```

Si observas la salida, en el proceso de creación del pod se han creado dos contenedores, uno con la imagen de docker de la aplicación “details” y otro con la imagen del Proxy Sidecar de Istio. También puedes verlo en el propio archivo de configuración del pod con el comando:

```
kubectl edit pod {nombre_del_pod}
```

Por último, vamos a configurar las variables **INGRESS_HOST** e **INGRESS_PORT** para poder acceder al gateway Ingress de Istio.

En una nueva terminal ejecutamos:

```
minikube tunnel
```

Este comando sirve para crear una ruta de red al clúster utilizando su IP como puerta de enlace y de esta forma envíar tráfico al Gateway de Istio desde Minikube.

En la terminal que teníamos antes, vamos a configurar el *host*, los puertos y la URL del *gateway*:

```
export INGRESS_HOST=$(kubectl -n istio-system get service \
istio-ingressgateway -o jsonpath='{.status.loadBalancer.ingress[0].ip}')

export INGRESS_PORT=$(kubectl -n istio-system get service \
istio-ingressgateway -o
jsonpath='{.spec.ports[?(@.name=="http2")].port}')

export SECURE_INGRESS_PORT=$(kubectl -n istio-system get service \
istio-ingressgateway -o
jsonpath='{.spec.ports[?(@.name=="https")].port}')

export GATEWAY_URL=$INGRESS_HOST:$INGRESS_PORT
```

Ahora, vamos a ejecutar el siguiente comando para obtener la dirección web de la aplicación y copiamos la salida por pantalla en el navegador:

```
echo “http://$GATEWAY_URL/productpage”
```

```
vagrant@localhost:~$ echo "http://$GATEWAY_URL/productpage"
http://10.98.118.19:80/productpage
```

Para ver la estructura de la malla de servicio y su topología, vamos a utilizar el panel de Kiali, aunque Istio se integra con otras aplicaciones de telemetría como Prometheus, Grafana o Jaeger.

Con los siguientes comandos, instalaremos Kiali y otros complementos necesarios para usar su panel de control:

```
kubectl apply -f samples/addons
kubectl rollout status deployment/kiali -n istio-system
istioctl dashboard kiali
```

Este último comando, abrirá en una nueva ventana del navegador el Dashboard de Kiali. Kiali es una consola de administración para Istio, requiere haber configurado Istio previamente (lo que hemos hecho hasta ahora). Una vez que se abre, podemos acceder en el menú de la izquierda

al apartado “graphs” y establecer el filtro de “Namespace” en “default”. Esto nos mostrará los diferentes servicios y pods de la aplicación.

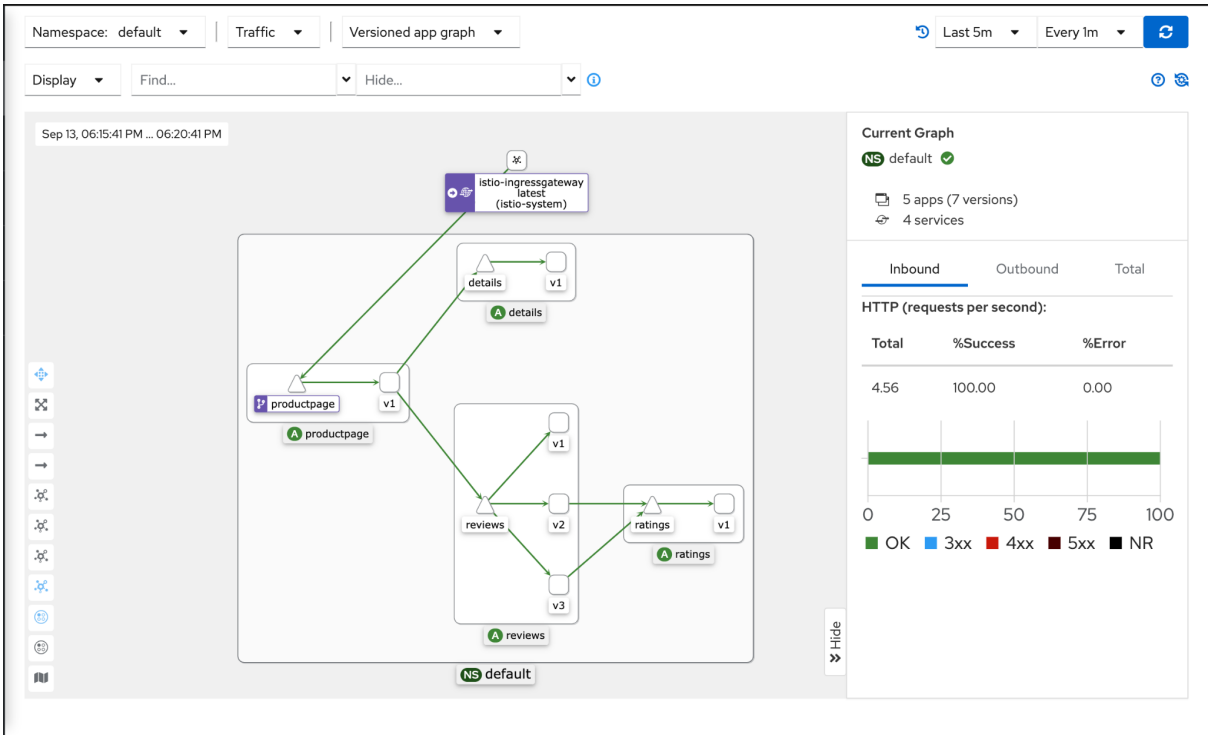
Los servicios son los iconos triangulares y los pods son los cuadrados. Cada servicio va unido a su pod.

Para ver el tráfico de la malla que hemos creado, puedes lanzar, en una nueva terminal, una serie de solicitudes con este comando:

```
for i in $(seq 1 100); do curl -s -o /dev/null \
"http://$GATEWAY_URL/productpage"; done
```

Nota: Recuerda que si quieres utilizar la variable GATEWAY_URL en una nueva terminal, **debes definirla de nuevo** de la misma forma que has hecho anteriormente. En su defecto, puedes utilizar su valor directamente.

El grafo mostrará la dirección de las conexiones necesarias para responder a la solicitud.



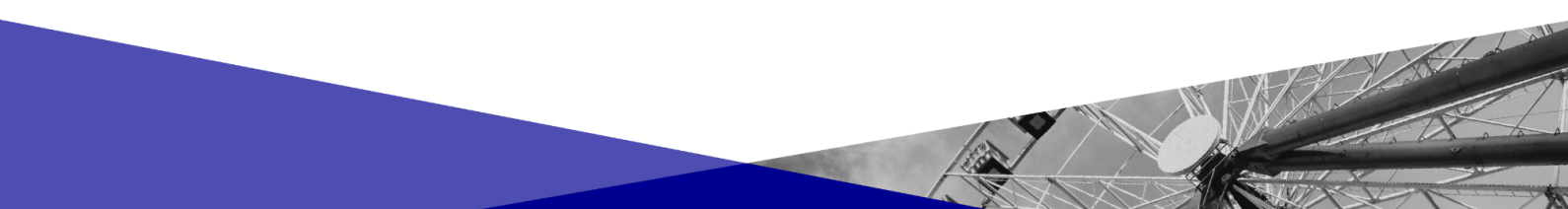
Despliegue

Hacer el despliegue de una aplicación en K8s es laborioso al tener que partir de cero y definir en distintos ficheros todos los componentes necesarios para llevarlo a cabo. Esta tarea es aún más compleja si no se tiene conocimiento sobre cómo funciona internamente K8s y qué clase de recursos hay que definir para conseguir el funcionamiento esperado de la aplicación.

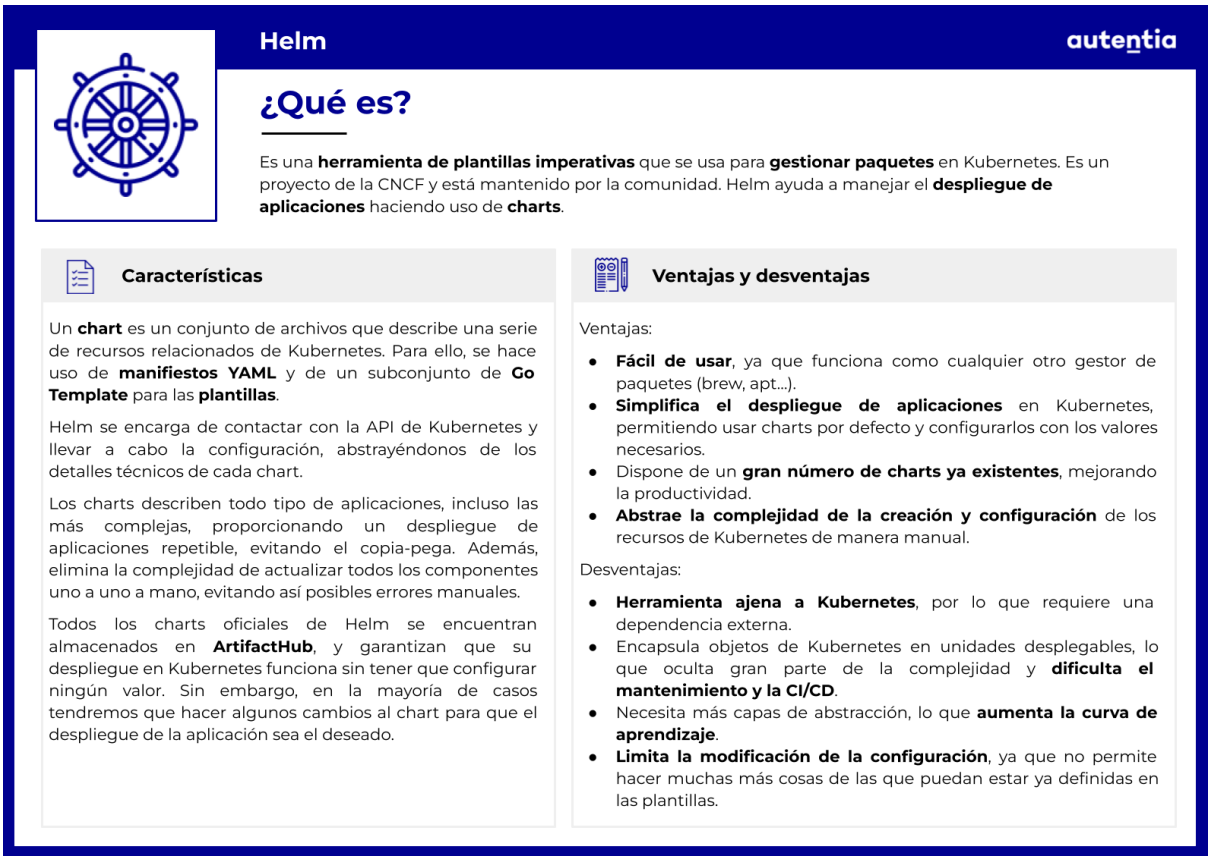
Si hacemos el despliegue de manera manual, tendremos que ejecutar cada fichero con la definición del recurso de K8s uno a uno con el comando *kubectl create/apply* y esperar que las variables definidas en el fichero consigan desplegar correctamente la aplicación.

De esta manera, cuantas más aplicaciones tengamos desplegadas en K8s, más complicado será mantenerlas y gestionarlas, además de tratarse de un proceso bastante repetitivo y propenso a errores. Cualquier cambio necesario implicaría tener que volver a desplegar todos los ficheros afectados y, para eliminar un despliegue, habría que ejecutar de nuevo múltiples comandos de kubectl.

Es por ello que existen herramientas, como son Helm y Kustomize, que simplifican esta tarea, haciendo más rápidos y mantenibles los despliegues.



Helm



Helm es el administrador de paquetes para K8s. Se trata de una herramienta de plantillas imperativas que se usa para gestionar paquetes en K8s. Es un proyecto de la CNCF y está mantenido por la comunidad.

Helm ayuda a manejar el despliegue de aplicaciones en K8s. Para ello, existe el concepto de chart. Un chart es un conjunto de archivos que describe una serie de recursos relacionados de K8s. Los charts hacen uso de los manifiestos YAML de K8s, además de un subconjunto de Go Template para las plantillas.

Los charts ayudan a definir un marco común de uso básico de la aplicación y sus posibles configuraciones. Helm se encarga de contactar con la API de K8s y llevar a cabo la configuración, lo cual nos abstrae de entrar en detalles técnicos de cada chart. Se puede usar un solo chart para implementar algo simple, como un pod *memcached*, o algo complejo, como aplicaciones web completas con servidores HTTP, bases de datos, cachés, etc.

Todos los charts oficiales de Helm, almacenados en ArtifactHub, garantizan que su despliegue en K8s funciona sin tener que configurar ningún valor. Sin embargo, esto no quiere decir que con desplegar un chart de Helm, este cubra todas las necesidades de nuestra aplicación. En la mayoría de casos tendremos que hacer algunos cambios al chart para que el despliegue de la aplicación sea el deseado.

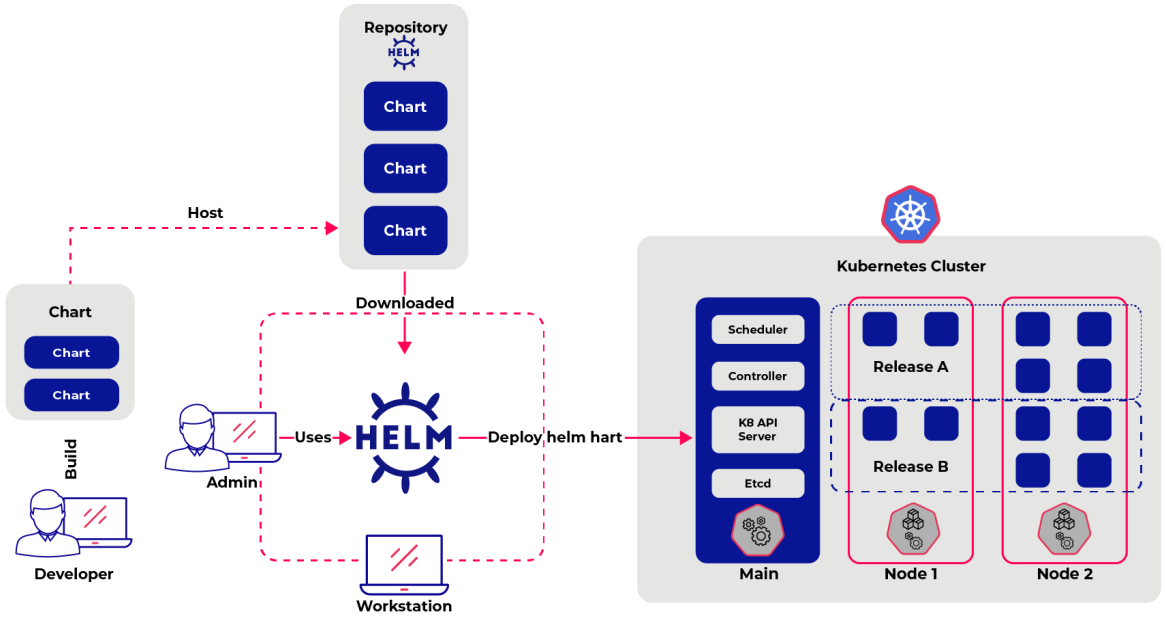
Entrando más en detalle, un chart no es más que un directorio con una cierta estructura que se empaqueta en un único archivo, el cual se puede subir a un repositorio para su posterior uso y despliegue en K8s. En la siguiente imagen podemos ver un ejemplo de un chart, que recibe el nombre de "my-app":

```
my-app/
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── ingress.yaml
│   ├── NOTES.txt
│   └── service.yaml
└── values.yaml
```

La estructura básica de un chart debe contener al menos los siguientes archivos:

- **Archivo "Chart.yaml"**: Describe el nombre del chart y su descripción, y establece las versiones.
- **Carpeta "templates"**: Contiene todos los archivos de recursos de K8s que se quieren desplegar en el clúster. Además, estos contienen una serie de variables que pueden tomar distintos valores.
- **Archivo "values.yaml"**: Define todas las variables que se usan en los manifiestos YAML. Podemos configurar cualquier chart modificando los valores de las variables aquí definidas.

De forma aclaratoria, con este diagrama podemos entender mejor cuál es el flujo de trabajo con Helm:

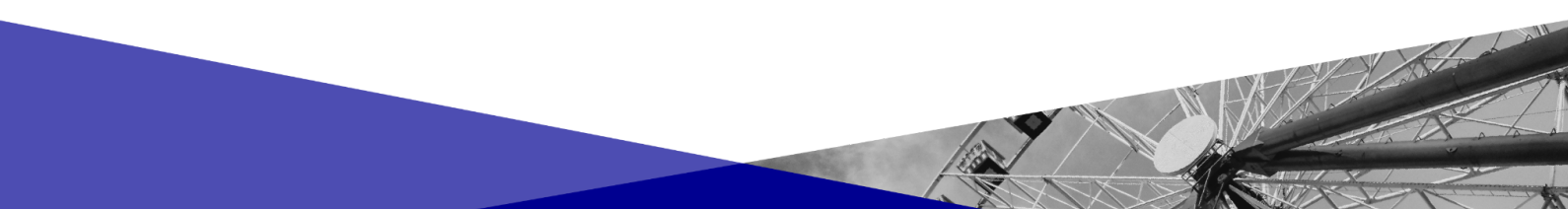


Helm Workflow

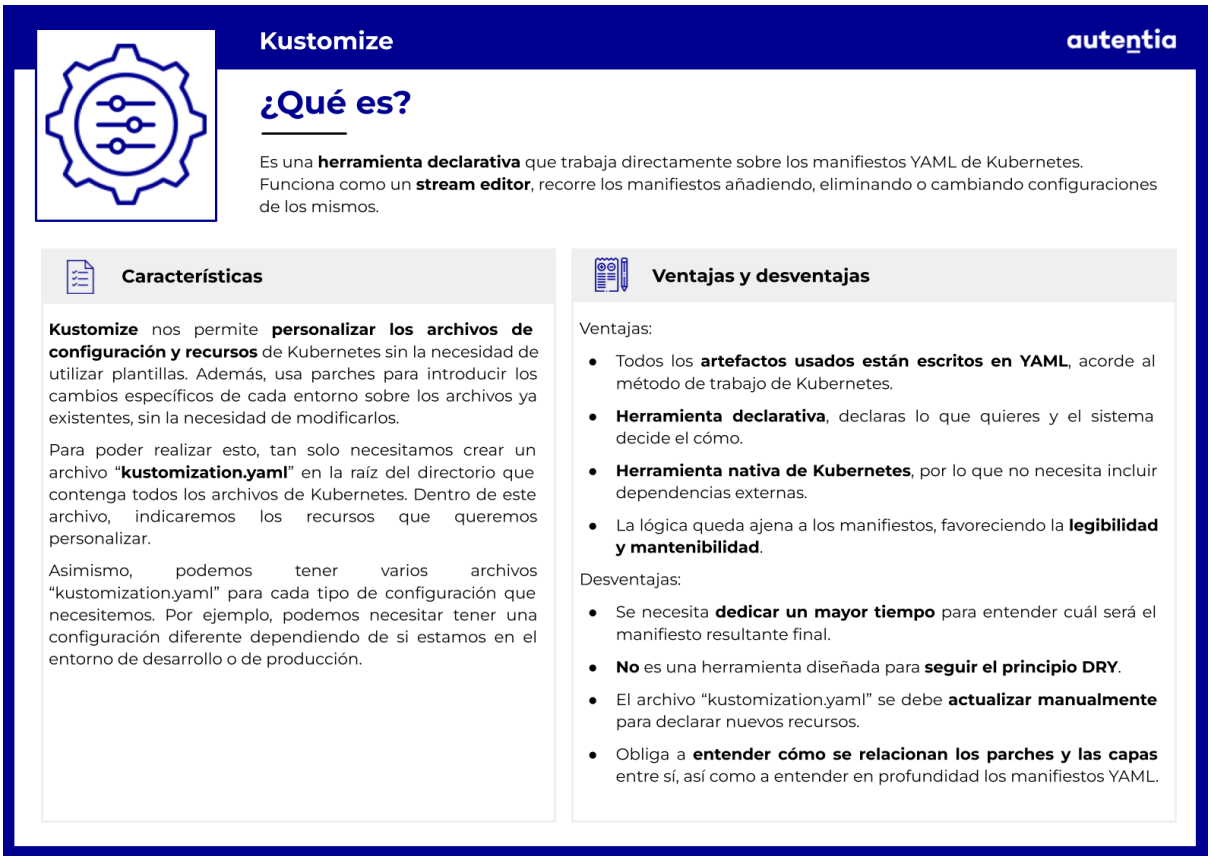
En él vemos cómo un desarrollador programa y construye el chart, con sus correspondientes plantillas y valores configurables. Acto seguido, lo sube a un repositorio de Helm junto con el resto de charts. Finalmente, un usuario que requiera de su uso utilizará Helm para descargarlo, configurarlo y desplegarlo sobre el clúster de K8s

Entre las características más importantes del uso de Helm, destacan las siguientes:

- **Administra la complejidad**: Los charts describen todo tipo de aplicaciones, incluso las más complejas, proporcionando un despliegue de aplicaciones repetible, evitando el copia-pega, y que sirve como punto único de autoridad.
- **Actualizaciones fáciles**: Elimina la complejidad de actualizar todos los componentes uno a uno a mano.
- **Uso compartido simple**: Los charts son fáciles de versionar, compartir y publicar en servidores públicos o privados.
- **Retrocesos**: Podemos hacer uso del comando "*helm rollback*" para volver con facilidad a una versión anterior.



Kustomize



Kustomize es una herramienta declarativa que trabaja directamente sobre los manifiestos YAML de K8s. Funciona como un stream editor, recorre los manifiestos añadiendo, eliminando o cambiando configuraciones de los mismos, pero dejando el archivo original sin tocar. Es una herramienta interna de K8s y sigue el mismo paradigma declarativo, es decir, declaras lo que quieres y el sistema decide el cómo.

Kustomize nos permite personalizar los archivos de configuración y recursos de K8s sin la necesidad de utilizar plantillas. Nos provee de una serie de métodos, como los generadores, que nos ayudan a realizar esa personalización de manera más sencilla. Además, usa parches para introducir los cambios específicos de cada entorno sobre los archivos ya existentes, sin la necesidad de modificarlos.

Para poder realizar esto, tan solo necesitamos crear un archivo "**kustomization.yaml**" en la raíz del directorio que contenga todos los archivos de configuración de los distintos recursos de K8s. Dentro de este archivo, indicaremos los recursos a los que queremos aplicar los diferentes cambios para personalizarlos.

Asimismo, podemos tener varios archivos “kustomization.yaml” para cada tipo de configuración que necesitemos. Por ejemplo, podemos necesitar tener una configuración diferente dependiendo de si estamos en el entorno de desarrollo o de producción.

Para conseguir esto, dentro de nuestro directorio crearemos una carpeta “base” en la que meteremos todos los archivos de configuración comunes, incluido el “kustomization.yaml” general, y una carpeta “overlays” donde incluiremos las configuraciones concretas de cada entorno, con sus respectivos archivos de customización. En la siguiente imagen se muestra un ejemplo de este tipo:

```
~/someApp
├── base
│   ├── deployment.yaml
│   ├── kustomization.yaml
│   └── service.yaml
└── overlays
    ├── development
    │   ├── cpu_count.yaml
    │   ├── kustomization.yaml
    │   └── replica_count.yaml
    └── production
        ├── cpu_count.yaml
        ├── kustomization.yaml
        └── replica_count.yaml
```

Para usar Kustomize, no necesitamos nada más que tener instalado kubectl y ejecutar los comandos necesarios. Sin embargo, también podemos instalarlo como una herramienta aparte, siguiendo los pasos que se explican en la guía oficial.

Kustomize vs Helm

Por una parte, Helm tiene la ventaja de ser fácil de usar, ya que funciona como cualquier gestor de paquetes (brew o apt). Además, dispone de un gran número de charts ya existentes que pueden ser útiles para mejorar la productividad si nuestro problema ya tiene una solución implementada.

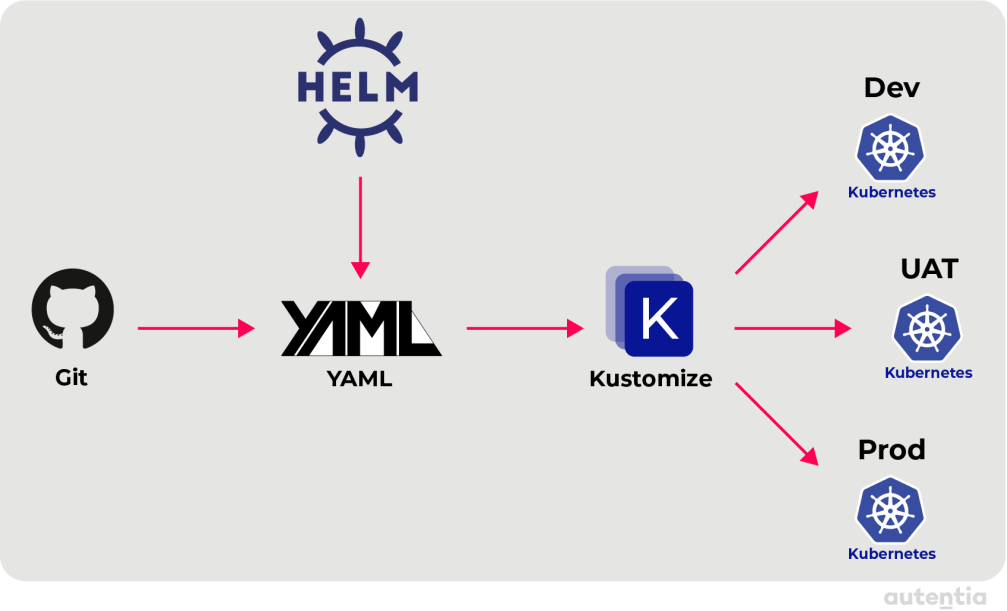
Sin embargo, es una herramienta ajena a K8s, por lo que requiere una dependencia externa. Helm encapsula objetos de K8s en unidades desplegables, lo que oculta gran parte de la complejidad y puede acabar dificultando el mantenimiento y la CI/CD. También necesita más capas de

abstracción, lo que aumenta la curva de aprendizaje. Por último, Helm presenta algunas limitaciones a la hora de poder modificar la configuración, ya que no permite hacer muchas más cosas de las que estén ya definidas en las plantillas.

Por otro lado, Kustomize tiene la principal ventaja de que todos los artefactos usados están escritos en YAML, acorde al método de trabajo de K8s. Es una herramienta nativa de K8s, por lo que no necesita incluir dependencias externas. Kustomize solo puede combinar archivos YAML sin plantilla, por lo que la lógica queda ajena a los manifiestos, favoreciendo la legibilidad y mantenibilidad. Sin embargo, esto también implica dedicar un mayor tiempo a entender cuál será el manifiesto resultante final.

Las desventajas de Kustomize frente a Helm son la productividad en aquellos casos en los que los charts de Helm pueden ofrecernos la base de nuestros objetos. No es una herramienta diseñada para seguir el principio DRY y el archivo “kustomization.yaml” se debe actualizar manualmente para declarar nuevos recursos, lo que supone una gran cantidad de tiempo dedicado a los archivos en comparación con Helm.

En conclusión, Helm es una herramienta que ofrece una productividad y simplicidad mayor que Kustomize, a costa de legibilidad y mantenibilidad en el futuro. Mientras que Kustomize nos permite hacer cambios más personalizados y siguiendo la filosofía de K8s, nos obliga a entender cómo se relacionan los parches y las capas entre sí, y entender en profundidad los manifiestos YAML. Por tanto, la mejor opción parece ser utilizar ambas herramientas, ya que no son incompatibles. Podemos aprovechar los charts de Helm para crear una base inicial, evitando así tener que definir todos los YAML necesarios, y posteriormente personalizarlos y mantenerlos con Kustomize, a la vez que nos permite cambiar de entorno fácilmente. Este proceso se puede ver reflejado en la siguiente imagen:



GitOps

A partir de Git, el sistema de control de versiones de código abierto que casi todos conocemos y utilizamos, surgen variantes como GitOps. En este caso, se trata de mantener el flujo de trabajo de la gestión de configuración de aplicaciones e infraestructuras a través de Git.

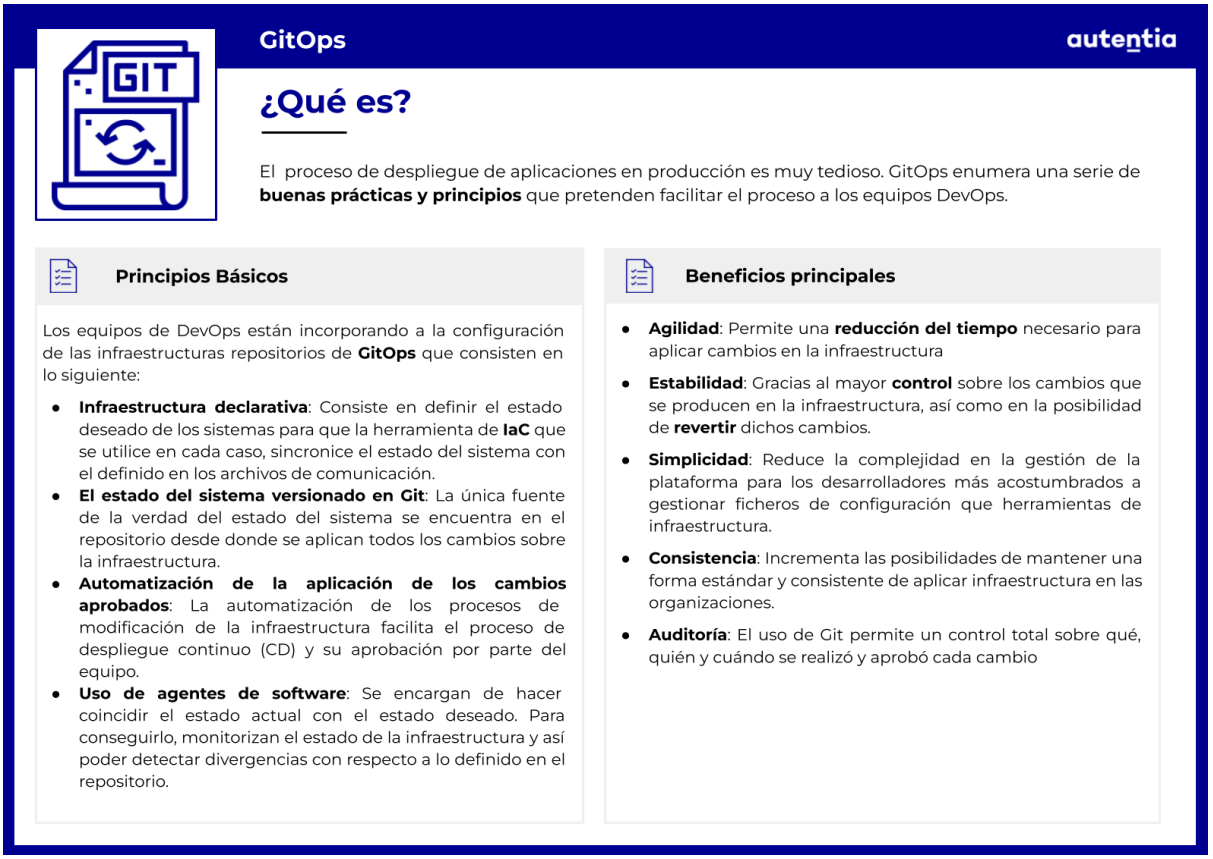
Para poder mantener el control del flujo es necesario que las configuraciones se realicen de forma declarativa, como ocurre con K8s. Por eso, consideramos importante dedicar un punto en esta guía a mencionar en qué consisten las buenas prácticas definidas por GitOps.

Con este concepto, lo que almacenamos en Git no es un repositorio de código fuente de nuestras aplicaciones, sino el estado completo del sistema. Esto permite ver y administrar las modificaciones que se realizan en la configuración.

Ahora que conocemos el término GitOps, es el momento de analizar cómo funciona. En este caso, dado el tema de la guía, vamos a centrarnos en aplicar GitOps sobre K8s, pero como mencionamos anteriormente, se puede hacer con cualquier infraestructura que permita configuración declarativa.

Normalmente, cuando trabajamos con CI/CD, los canales se activan por eventos, como la inserción de cambios en el código de un repositorio. En este caso, cuando queremos implementar cambios en la configuración utilizando un flujo de trabajo de GitOps, lo hacemos mediante una solicitud

de incorporación de cambios en Git. El operador, que en nuestro caso será el operador de K8s, se encargará de extraer el nuevo estado deseado del repositorio Git y aplicarlo a la infraestructura.



Flux

Uno de los operadores de K8s más conocidos es Flux. Es una herramienta de código abierto que automatiza las implementaciones de configuración de GitOps en el clúster. Se puede aplicar sobre repositorios Git, pero también sobre plantillas de Helm, por ejemplo.

Flux

autentia

¿Qué es?

Flux es una herramienta diseñada para mantener los clústeres de Kubernetes sincronizados con fuentes de configuración como Git de forma automática. Flux funciona bien con herramientas como Helm, Istio, Grafana o Prometeus.

Características

Algunos de los aspectos más destacados de esta herramienta son:

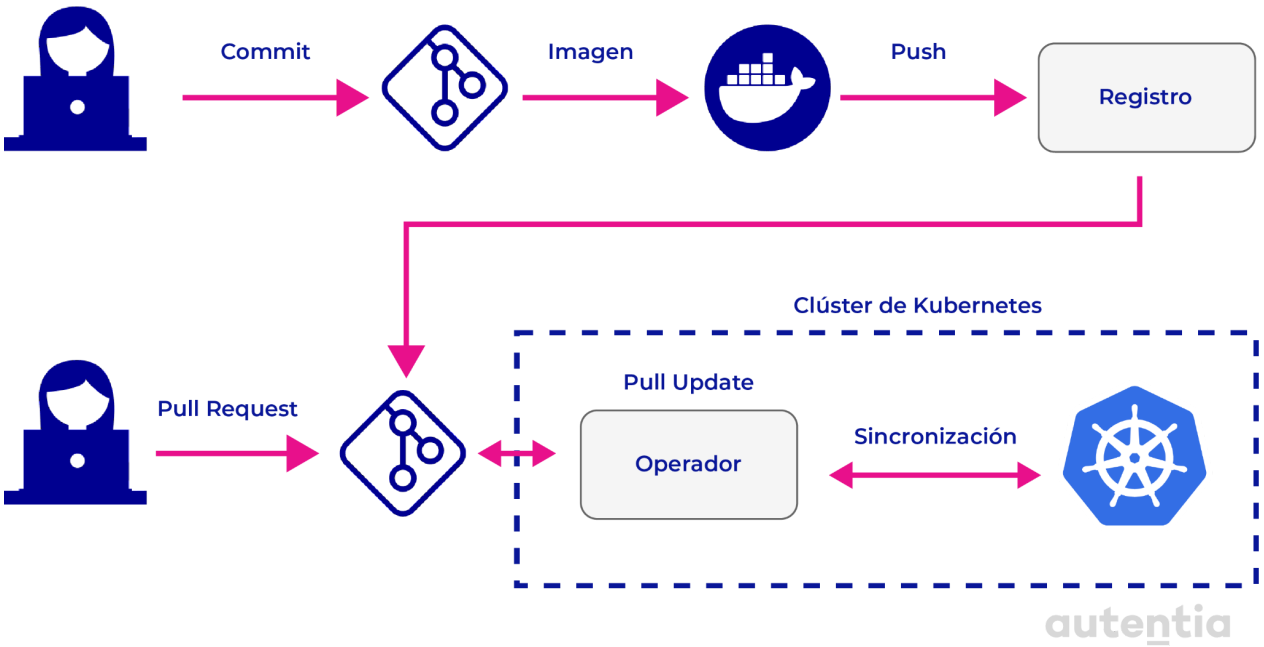
- Flux se puede utilizar para **GitOps** sobre aplicaciones e infraestructuras. Además, puede gestionar cualquier recurso de Kubernetes, así como las cargas de trabajo.
- Habilita el despliegue continuo de aplicaciones (CD) y la entrega progresiva (PD) a través de la **reconciliación automática**. También puede actualizar automáticamente imágenes de contenedores en git (Push).
- Se adapta a casi todos los **proveedores de Git**.
- Es compatible con todas las **herramientas de Kubernetes** como Kustomize, Helm, Istio...
- Soporta **más de un repositorio a la vez** y también más de un **clúster**. Flux utiliza un clúster de Kubernetes para gestionar las aplicaciones que hay dentro de él pero también dentro de otros clústeres.
- Evalúa la **salud del sistema** y genera alarmas en función de las vulnerabilidades detectadas.

Conceptos de Flux

Para poder utilizar esta herramienta, es importante conocer los **conceptos** con los que trabaja:

- **GitOps**: Es un modo de gestionar la infraestructura de las aplicaciones de forma declarativa y aplicando el control de versiones de Git. El fin es que el estado del entorno del sistema coincida con el deseado en el repositorio.
- **Fuentes**: Son el origen de los repositorios. Almacenan el estado deseado del sistema.
- **Reconciliación**: Consiste en asegurar que el estado deseado coincide con el estado del sistema.
- **Customización**: Representa un conjunto de recursos que Flux debe reconciliar en el clúster.
- **Bootstrap**: Es el nombre que se da al proceso de instalación de los componentes de Flux.

Si quieres empezar a utilizar Flux, puedes seguir este tutorial que solo te llevará unos minutos. El tutorial consiste en instalar el operador y crear un repositorio en GitHub para guardar la configuración de una aplicación de prueba.



En este diagrama se muestra el proceso de control de versiones de Git sobre archivos de configuración. Es un proceso similar a cuando añadimos código de una aplicación al repositorio.

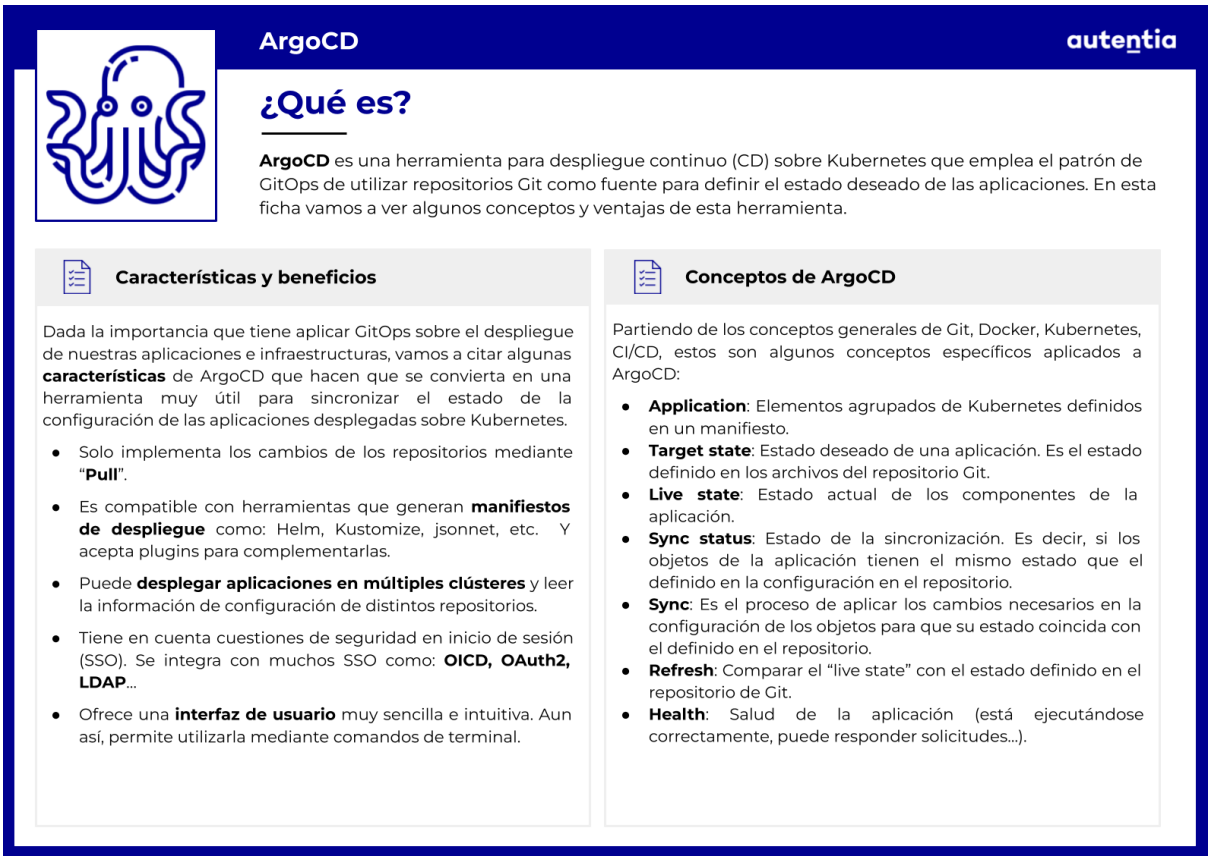
En primer lugar se hace un *commit* que genera una nueva imagen, en este caso de Docker, y esta nueva versión de la imagen se registra. Después, mediante un *pull Request* se sube el commit al repositorio y un operador de K8s (si es la herramienta de orquestación que estamos utilizando) se encarga de hacer un *pull Update* para obtener los cambios realizados en los archivos de configuración.

La segunda misión del operador es comprobar que el estado del clúster es el mismo que el indicado por la última actualización de los archivos de configuración y, si no lo es, realizar los cambios necesarios en el clúster para que lo sea.

ArgoCD

ArgoCD es otra herramienta de despliegue continuo de aplicaciones mediante GitOps para K8s. Las condiciones de uso de la herramienta son las mismas que hemos visto previamente: la configuración de las aplicaciones debe ser declarativa y se debe emplear una herramienta de control de versiones como Git.

Aquí tienes un ejemplo básico de uso de ArgoCD para que compruebes cómo se aplican los cambios realizados en la configuración una vez que los añades al repositorio.



ArgoCD vs Flux

En primer lugar, debemos tener en cuenta que ambas son herramientas muy potentes para mantener el estado de nuestras aplicaciones igual al estado deseado que el desarrollador establece en el repositorio.

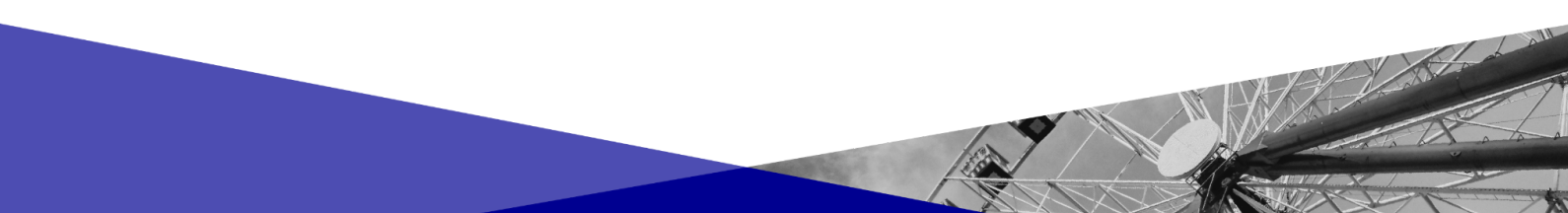
La instalación de ambos es bastante sencilla. Además, como hemos visto en los puntos anteriores, la documentación de los dos incluye guías de inicio rápido.

En cuanto a la interfaz de usuario, en este caso ArgoCD cuenta con una interfaz bastante intuitiva y limpia. Sin embargo, Flux carece de ella.

Respecto al funcionamiento interno de ambas herramientas, ArgoCD agrupa todos los recursos relacionados y los despliega como una única aplicación. Cada aplicación pertenece a un proyecto y cada *tenant* puede restringirse a un clúster o espacio de nombres concreto. Cuando los recursos se han desplegado correctamente y el estado de la aplicación es igual al deseado en el repositorio, se dice que los recursos están en estado *live*. Para poder aplicar los cambios del repositorio, el desarrollador debe realizar una sincronización. Por último, ArgoCD monitorea la salud de la aplicación y permite retroceder a versiones anteriores de la configuración en caso de fallos.

Por otro lado, Flux es una herramienta con mucha menos intervención humana. El proceso es instalar los componentes de Flux en el clúster en lo que se conoce como Bootstraping. Esto realiza una monitorización del directorio en el repositorio de Git para que cualquier cambio se aplique en el clúster mediante el proceso de reconciliación.

En definitiva, ambas herramientas tienen el mismo objetivo. La principal diferencia reside en que, con Flux el proceso de sincronización (nombre de ArgoCD) o reconciliación (nombre de Flux) es automático y en ArgoCD se produce cuando el desarrollador da la orden. Esta es su configuración por defecto, sin embargo, se puede configurar de forma que ocurra automáticamente, igual que Flux.



Bibliografía

Estas son las fuentes que hemos consultado y en las que nos hemos basado para la redacción de este material:

- https://docs.docker.com/reference/
- https://docs.ansible.com/ansible/latest/user_guide/index.html
- https://kubernetes.io/es/docs/home/

Lecciones aprendidas en esta guía

La gran presencia de **Linux** en servidores y otros sistemas hace que prácticamente todo desarrollador tenga que saber desenvolverse con naturalidad en estos entornos.

Gracias a la **virtualización,** es muy sencillo crear un sistema Linux desde nuestro propio ordenador sin necesidad de tener a nuestra disposición múltiples equipos. Estos sistemas virtualizados son el entorno ideal para familiarizarnos con Linux, ya que, al ser tan sencillos y rápidos de configurar, podemos hacer todas las pruebas que queramos sin miedo ya que, aunque cometamos errores, no tendrán prácticamente ninguna repercusión.

Hemos aprendido la gestión de **usuarios y permisos** que tiene Linux, y con nuestro usuario, hemos visto que la terminal es una herramienta muy potente con la que podemos controlar nuestro ordenador sin necesidad de herramientas gráficas. Conocer y saber usar estos comandos, nos permite realizar tareas tan diversas como **ver y editar ficheros** o **controlar los procesos** que se ejecutan en un ordenador.

Quizá el aspecto más importante de la terminal es que está presente en **todos los sistemas del mundo** y por lo tanto, aprender a usar esta herramienta implica que podremos manejarnos en cualquier sistema sin haber tenido un contacto previo con este.

Uniendo los conceptos de **virtualización y manejo con la terminal** vistos en puntos anteriores, hemos aprendido a cómo automatizar todo el proceso de aprovisionamiento y despliegue de los servicios y cómo éstos se integran entre sí gracias a **herramientas de gestión de la infraestructura como código**, en particular Ansible. Eliminar el proceso manual ayuda a reducir el tiempo invertido, el coste y los errores humanos. Además, durante la creación de la configuración se realizan procesos de verificación que reducen casi a cero el riesgo de cometer errores.

En esta línea, también hemos aprendido el concepto de contenedor y su gestión en particular con Docker, así como las ventajas que nos ofrece la contenerización de aplicaciones y servicios, como son la portabilidad, eficiencia y facilidad de despliegue. Sin embargo, a medida que aumenta el número de contenedores que necesitamos desplegar, su gestión de forma manual se complica, tendiendo a cometer más errores. Es por ello que surgen las herramientas de orquestación de contenedores, como Kubernetes en este caso, con la que hemos podido comprobar las grandes ventajas y facilidades que nos ofrece a la hora de desplegar nuestras aplicaciones en la nube.

¡Ya estamos preparados para poder iniciar con garantías los siguientes pasos en este inmenso y apasionante mundo del DevOps!

¡Conoce más!