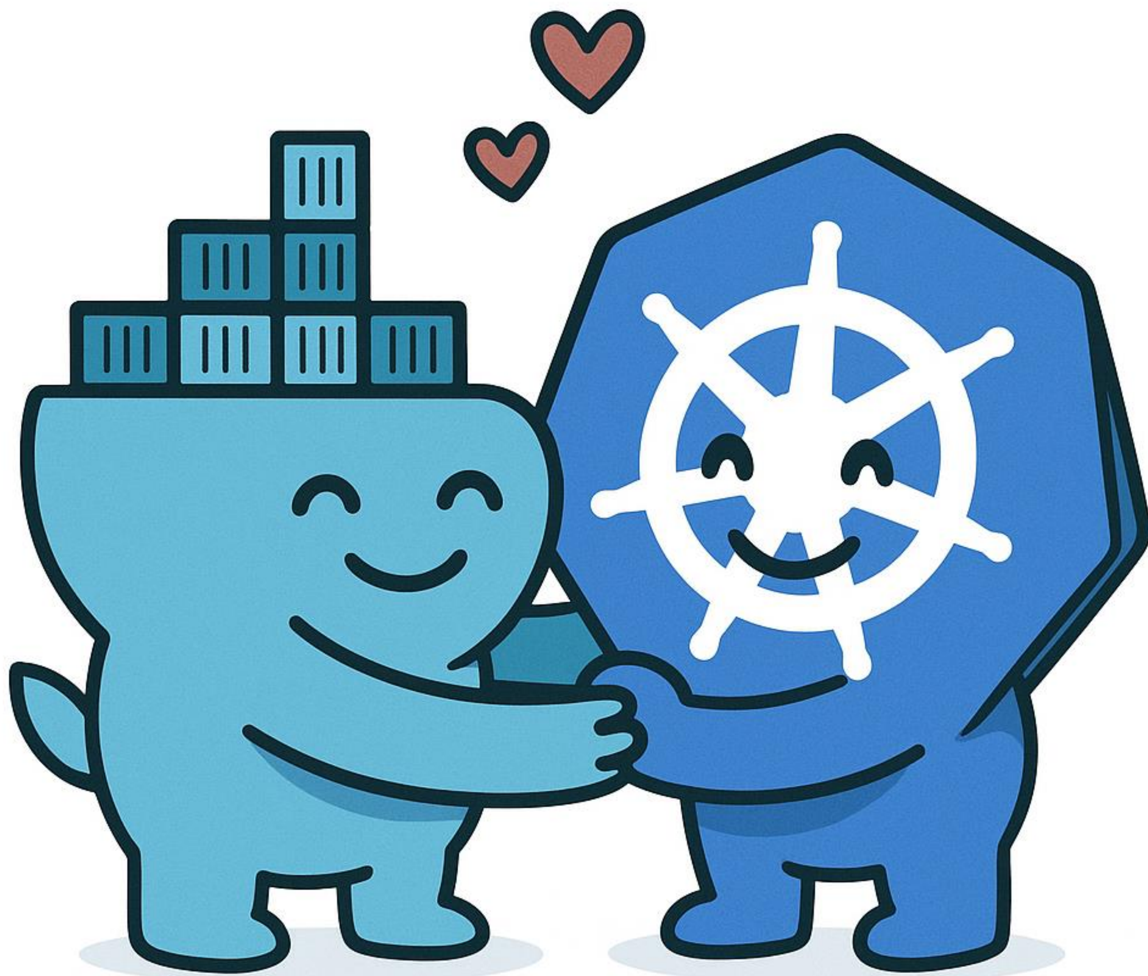
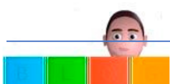
# De cero a cien con Docker y Kubernetes





Vivimos en una era en la que la **velocidad**, la **eficiencia** y la **escala** marcan la diferencia entre el éxito y el estancamiento de un proyecto tecnológico. En este contexto, **Docker** y **Kubernetes** han surgido como herramientas esenciales que están redefiniendo cómo concebimos, desarrollamos, desplegamos y mantenemos nuestras aplicaciones.

Este libro nace con un propósito claro: **llevarte de 0 a 100** en el mundo de la contenerización y la orquestación de contenedores. No importa si apenas has oído hablar de Docker o Kubernetes, o si ya has dado tus primeros pasos; aquí encontrarás una guía práctica, estructurada por partes, para acompañarte en tu aprendizaje de forma progresiva y sólida.

Comenzaremos explorando los **conceptos básicos**: qué es un contenedor, en qué se diferencia de una máquina virtual y por qué ha revolucionado el desarrollo de software. Luego, nos sumergiremos en **Docker**, la herramienta que facilita la creación, empaquetado y ejecución de contenedores, descubriendo tanto su uso básico como técnicas más avanzadas de optimización y seguridad.

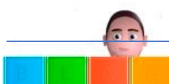
Una vez dominado Docker, daremos el salto a **Kubernetes**, la plataforma líder en orquestación de contenedores, aprendiendo desde su arquitectura fundamental hasta cómo desplegar aplicaciones reales, gestionar configuraciones, escalar servicios y asegurar entornos productivos.

Además, dedicaremos secciones especiales a abordar **temas avanzados**, **casos prácticos reales** y **buenas prácticas**, para que no solo adquieras conocimiento técnico, sino también la perspectiva necesaria para tomar buenas decisiones en proyectos reales.

Este libro está diseñado para ser práctico y aplicable. Cada capítulo te invita a poner manos a la obra, ya sea montando tu primer contenedor o desplegando una arquitectura de microservicios en la nube.

**Docker y Kubernetes no son solo tecnologías: son llaves que abren la puerta al futuro del desarrollo y operación de software.**
Hoy, tú tienes esa llave en tus manos.

¡Bienvenido al viaje! 🚀



# Prólogo

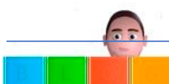
Recuerdo la primera vez que escuché hablar de contenedores.
En aquel momento, el concepto me sonó como algo innecesariamente complejo, una moda pasajera para quienes disfrutaban complicándose la vida.
Hoy, años después, puedo decirte con toda certeza: **Docker y Kubernetes no son solo herramientas, son una revolución**.

Este libro es el que me habría gustado tener cuando comencé.
Una guía clara, práctica y progresiva, que no solo explique los comandos y configuraciones, sino que te ayude a entender **el "por qué"** y **el "cómo"** detrás de cada paso.
Porque la tecnología no se trata solo de saber qué botón apretar; se trata de comprender **el impacto** que nuestras decisiones tienen en el funcionamiento de sistemas reales.

Mi objetivo aquí no es solo enseñarte a usar Docker o desplegar clusters de Kubernetes.
Mi objetivo es que termines este libro **pensando como un verdadero ingeniero de sistemas modernos**, capaz de construir soluciones escalables, resilientes y preparadas para el futuro.

No importa si estás dando tus primeros pasos o si ya tienes experiencia en desarrollo o infraestructura: si tienes la curiosidad y la determinación de aprender, **este libro es para ti**.

**Prepárate. Lo que vas a aprender cambiará tu manera de construir software para siempre.**



# Sobre el autor

**Óscar de la Cuesta**
@oscardelacuesta — palentino.es

---

Apasionado de la tecnología, el cloud computing y el desarrollo de soluciones innovadoras, a través de mi trabajo, mi blog y la presencia en redes, comparto conocimiento práctico y accesible, buscando siempre acercar la tecnología a más personas y fomentar una comunidad abierta y colaborativa.

Este libro nace del deseo de ofrecer un camino completo —desde cero hasta nivel avanzado— en Docker y Kubernetes, para que cualquier profesional pueda dominarlo y construir soluciones sólidas, escalables y preparadas para el futuro.

Este libro está licenciado bajo la **Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional (CC BY-SA 4.0)**. Esto significa que eres libre de:

✓ **Compartir** — Copiar y redistribuir el material en cualquier medio o formato.
✓ **Adaptar** — Remezclar, transformar y construir sobre el material para cualquier propósito, incluso comercialmente.

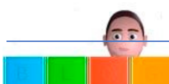
Bajo las siguientes condiciones:

✓ **Atribución** — Debes dar el crédito adecuado, proporcionar un enlace a la licencia e indicar si se han realizado cambios. Puedes hacerlo de cualquier manera razonable, pero no de una forma que sugiera que el licenciante te respalda a ti o a tu uso.
✓ **CompartirIgual** — Si remezclas, transformas o creas a partir del material, debes distribuir tus contribuciones bajo la misma licencia que el original.

Para más información, visita: https://creativecommons.org/licenses/by-sa/4.0/

Autor: Oscar de la Cuesta Campillo
@oscardelacuesta

**Palentino.es**



# Índice


## Parte I: Introducción a la Contenerización

1. **¿Qué es la contenerización?**
2. **Diferencias entre máquinas virtuales y contenedores**
3. **Ventajas de usar contenedores**
4. **Panorama general: Docker y Kubernetes**

## Parte II: Dominando Docker

1. **Instalación de Docker en diferentes sistemas operativos**
2. **Primeros pasos: tu primer contenedor**
3. **Entendiendo imágenes y contenedores**
4. **Dockerfiles: Cómo construir tus propias imágenes**
5. **Gestión de imágenes y contenedores**
6. **Docker Compose: Orquestación básica**
7. **Persistencia de datos: Volúmenes y Bind Mounts**
8. **Redes en Docker**
9. **Buenas prácticas de seguridad en Docker**
10. **Optimización de imágenes Docker**

## Parte III: Introducción a Kubernetes

1. **¿Qué es Kubernetes y por qué usarlo?**
2. **Arquitectura de Kubernetes: Componentes principales**
3. **Instalación local: Minikube, Kind y alternativas**
4. **Tu primer despliegue en Kubernetes**
5. **Pods, ReplicaSets y Deployments**
6. **Servicios en Kubernetes: ClusterIP, NodePort y LoadBalancer**
7. **ConfigMaps y Secrets**
8. **Volúmenes y almacenamiento persistente (PVCs)**
9. **Namespaces y control de recursos**

## Parte IV: Desplegando Aplicaciones en Kubernetes

1. **Despliegue de aplicaciones multi-contenedor**
2. **Helm: gestión de paquetes en Kubernetes**
3. **Actualizaciones y Rollbacks**
4. **Estrategias de escalado: Horizontal y Vertical**
5. **Health Checks: Liveness y Readiness Probes**
6. **Logs, métricas y monitorización básica**

## Parte V: Kubernetes Avanzado

1. **Ingress Controllers y gestión de tráfico**
2. **RBAC: Control de acceso basado en roles**
3. **Políticas de red**
4. **Operadores en Kubernetes**
5. **Kubernetes en la nube: GKE, EKS y AKS**
6. **CI/CD con Docker y Kubernetes**

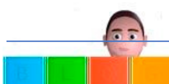
## Parte VI: Buenas Prácticas y Casos Reales

1. **Diseño de aplicaciones nativas de Kubernetes**
2. **Errores comunes y cómo evitarlos**
3. **Caso práctico: Desplegando una aplicación web completa**
4. **Optimización de clústeres y costes**
5. **Tendencias futuras: Serverless y Kubernetes**

## Apéndices

- **Glosario de términos**
- **Recursos recomendados (libros, cursos, documentación oficial)**
- **Comandos útiles de referencia rápida**

---



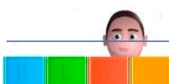
# Introducción

En un mundo cada vez más digital y acelerado, el desarrollo y despliegue de aplicaciones debe ser ágil, seguro y eficiente. La contenerización ha emergido como una respuesta a estos desafíos, permitiendo a desarrolladores y administradores de sistemas construir, empaquetar y desplegar software de manera consistente, en cualquier entorno.

Este libro nace con el objetivo de guiarte desde **cero absoluto** hasta un nivel **avanzado** en **Docker** y **Kubernetes**, las dos tecnologías líderes en contenerización y orquestación de contenedores. No necesitas conocimientos previos: cada capítulo construye una base sólida, ampliando gradualmente tu dominio en la creación, administración y escalamiento de aplicaciones modernas.

Prepárate para aprender no solo la teoría, sino también aplicarla en ejemplos reales, buenas prácticas y casos prácticos diseñados para simular entornos de producción.
Ya sea que busques optimizar el flujo de trabajo en tu empresa, prepararte para nuevas oportunidades laborales, o simplemente dominar dos herramientas esenciales del mundo tech, este libro será tu guía de confianza.

**¡Comencemos!**



# Capítulo 1: ¿Qué es la contenerización?

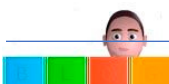
Durante años, los desarrolladores enfrentaron un problema clásico: "Funciona en mi máquina, pero no en producción."
La contenerización surgió para eliminar esa brecha, ofreciendo una forma de empaquetar una aplicación junto con todas sus dependencias, configuraciones y librerías necesarias para funcionar, en cualquier entorno.

Un **contenedor** es una unidad de software ligera, independiente y ejecutable, que incluye todo lo necesario para ejecutar una aplicación: código, librerías, dependencias y variables de entorno.
A diferencia de las máquinas virtuales, los contenedores no necesitan un sistema operativo completo; comparten el núcleo del sistema anfitrión, haciéndolos mucho más ligeros y rápidos.

### Beneficios principales de la contenerización:

- **Portabilidad:** ejecuta tu aplicación en cualquier lugar sin modificaciones.
- **Eficiencia:** menos consumo de recursos que las máquinas virtuales tradicionales.
- **Velocidad:** despliegues rápidos y arranque casi instantáneo.
- **Escalabilidad:** integración perfecta con plataformas de orquestación como Kubernetes.

En los próximos capítulos, descubrirás cómo Docker y Kubernetes han hecho de la contenerización una práctica estándar en la industria, cambiando la forma en que construimos, enviamos y ejecutamos software.



# 1. ¿ Contenerización?

La contenerización es una tecnología que permite **empaquetar** una aplicación y **todo su entorno de ejecución** —como librerías, configuraciones y dependencias— en un **contenedor** único y portátil.

Un contenedor es como una "caja" que guarda todo lo que tu aplicación necesita para funcionar. Gracias a ello, puedes ejecutar esa misma caja en tu ordenador, en un servidor, en la nube o en un centro de datos, **sin importar** las diferencias del sistema operativo o de la infraestructura.

En otras palabras, contenerizar una aplicación significa:

- Aislarla del sistema operativo anfitrión.
- Asegurar que se ejecutará de la **misma manera** en cualquier entorno.
- Evitar conflictos como los típicos "en mi máquina funciona, pero en producción no".

Esta técnica ha revolucionado el desarrollo de software porque ofrece una solución elegante al problema de la compatibilidad de entornos.
Imagina que tu aplicación necesita versiones específicas de Python, Node.js, o de cualquier librería: con los contenedores, **todo queda dentro** del propio contenedor, sin alterar ni depender del sistema operativo donde se ejecuta.

# 2. Diferencias entre máquinas virtuales y contenedores

Aunque los contenedores y las máquinas virtuales (**VMs**) buscan resolver problemas similares —como la portabilidad y el aislamiento de aplicaciones—, su enfoque y arquitectura son muy diferentes.

Aquí están las principales diferencias:

## 2.1. Arquitectura

- **Máquinas Virtuales:**
  Simulan un hardware completo. Cada VM incluye su propio sistema operativo (SO), además de la aplicación y sus dependencias. Esto implica un consumo considerable de recursos.
- **Contenedores:**
  Comparten el mismo núcleo (kernel) del sistema operativo anfitrión. Solo empaquetan la aplicación y sus dependencias, sin necesidad de un SO completo dentro de cada contenedor.
  Resultado: son mucho más ligeros y rápidos.

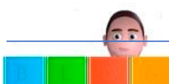


**Gráfico conceptual:**

```
less
CopiarEditar
Máquina Virtual:
[Servidor físico] → [Hipervisor] → [VM1: SO + App] [VM2: SO + App]

Contenedor:
[Servidor físico] → [SO] → [Docker Engine] → [Contenedor 1: App] [Contenedor
2: App]
```

## 2.2. Peso y velocidad

- **Máquinas Virtuales:**
  Tardan más en arrancar (minutos) y requieren gigabytes de espacio.
- **Contenedores:**
  Se inician en segundos (incluso menos) y ocupan solo megabytes.

## 2.3. Aislamiento

- **Máquinas Virtuales:**
  Proporcionan aislamiento completo a nivel de hardware.
- **Contenedores:**
  Ofrecen aislamiento a nivel de procesos y sistema operativo.
  Aunque muy seguro, es más liviano y rápido que el aislamiento por hardware.

## 2.4. Casos de uso

- **VMs:**
  Ideal cuando necesitas simular entornos completamente distintos (por ejemplo, correr Windows sobre Linux).
- **Contenedores:**
  Perfectos para desplegar, escalar y administrar aplicaciones modernas, especialmente en arquitecturas de microservicios.

---

**Resumen clave:**

Los contenedores no reemplazan a las máquinas virtuales en todos los casos, pero sí representan una evolución más ágil para desarrollar, desplegar y escalar aplicaciones de forma rápida, eficiente y portátil.

# 3. Ventajas de usar contenedores

Los contenedores no solo son una moda tecnológica; son una respuesta práctica y poderosa a muchos de los desafíos modernos en el desarrollo y operación de software.

A continuación, te explico las principales ventajas:

## 3.1. Portabilidad total

Un contenedor encapsula todo lo necesario para ejecutar una aplicación.
Esto significa que puedes moverlo entre:

- Tu portátil
- Un servidor físico
- Un centro de datos
- Servicios en la nube (AWS, Azure, Google Cloud)
  **…sin tener que hacer ningún cambio.**

## 3.2. Eficiencia de recursos

Al no incluir un sistema operativo completo, los contenedores:

- Consumen menos memoria.
- Ocupan menos espacio en disco.
- Usan menos recursos de CPU.

Esto permite correr **decenas o cientos** de contenedores donde antes solo era posible tener unas pocas máquinas virtuales.

## 3.3. Rapidez de despliegue

Los contenedores se crean, actualizan y destruyen en **cuestión de segundos**.
Esto facilita enormemente:

- La entrega continua (**CI/CD**).
- El escalamiento dinámico de aplicaciones.
- La recuperación ante fallos.

## 3.4. Aislamiento

Cada contenedor corre su propio proceso de forma aislada:

- Evitando conflictos de dependencias.
- Mejorando la seguridad.
- Facilitando la gestión de errores: si un contenedor falla, no afecta a los demás.

### 3.5. Facilidad para escalar aplicaciones

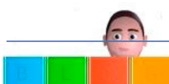
Con herramientas como Kubernetes, puedes escalar tu aplicación automáticamente:

- Aumentando el número de contenedores si sube la demanda.
- Reduciéndolos cuando baja el tráfico, optimizando costes.

---

**En resumen:**

Los contenedores permiten construir aplicaciones más rápidas, portables, eficientes y escalables, abriendo las puertas a nuevas formas de trabajar como los microservicios, DevOps y la computación en la nube.



# 4. Panorama general: Docker y Kubernetes

Ahora que sabes qué son los contenedores y sus ventajas, es momento de conocer a los dos grandes protagonistas de esta revolución: **Docker** y **Kubernetes**.

## 4.1. ¿Qué es Docker?

**Docker** es una plataforma que facilita la creación, ejecución y gestión de contenedores.
Con Docker puedes:

- Crear imágenes que contienen tu aplicación y sus dependencias.
- Ejecutar esas imágenes como contenedores en cualquier máquina.
- Administrar contenedores de forma sencilla con comandos intuitivos.

**Docker cambió para siempre** la manera de desarrollar y distribuir software.
Antes, desplegar una aplicación podía tardar días o semanas; con Docker, puedes hacerlo en minutos.

## 4.2. ¿Qué es Kubernetes?

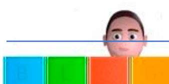
**Kubernetes** es un sistema de **orquestación de contenedores**.
¿Qué significa esto?
Cuando tienes pocos contenedores, puedes gestionarlos manualmente.
Pero ¿qué pasa cuando tienes:

- Cientos o miles de contenedores?
- Diferentes versiones de tu aplicación?
- Necesidad de escalarlos automáticamente según la carga?

Ahí es donde entra Kubernetes:

- **Automatiza** el despliegue, escalado y gestión de contenedores.
- **Recupera** contenedores caídos.
- **Distribuye** la carga entre diferentes nodos del clúster.
- **Expone** aplicaciones al exterior de manera segura.

**Docker** se centra en empaquetar y ejecutar aplicaciones en contenedores.
**Kubernetes** se encarga de coordinar, escalar y administrar esos contenedores en entornos de producción.



### 4.3. Relación entre Docker y Kubernetes

Piensa en Docker como la **herramienta que crea y ejecuta** contenedores, y en Kubernetes como el **cerebro que organiza miles de contenedores** distribuidos entre muchos servidores.

**Juntos forman una combinación poderosa** para construir aplicaciones modernas: rápidas, escalables y resistentes.

---

**Conclusión del capítulo:**

Entendiendo la contenerización, sus ventajas y el rol de Docker y Kubernetes, estás listo para comenzar a trabajar con estas tecnologías y dar tus primeros pasos en el mundo de la infraestructura moderna.

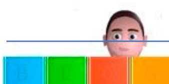
# Miniresumen del Capítulo 1

En este primer capítulo, hemos aprendido los conceptos fundamentales que sustentan todo el ecosistema moderno de aplicaciones:

- La **contenerización** permite empaquetar aplicaciones junto con todas sus dependencias, garantizando que funcionen de forma consistente en cualquier entorno.
- **Contenedores** y **máquinas virtuales** resuelven problemas similares, pero los contenedores son más ligeros, rápidos y eficientes, ideales para el desarrollo y despliegue ágil.
- Entre las **ventajas** de usar contenedores destacan su portabilidad, eficiencia en el uso de recursos, rapidez de despliegue, aislamiento de procesos y facilidad para escalar.
- **Docker** facilita la creación y ejecución de contenedores, mientras que **Kubernetes** orquesta su despliegue, escalado y gestión a gran escala.

Con esta base clara, estamos listos para **entrar de lleno en Docker**, construir nuestros primeros contenedores y preparar el terreno para trabajar con Kubernetes.

**¡Vamos allá!**



# Capítulo 2: Dominando Docker

Ahora que entiendes qué es la contenerización y su impacto en el desarrollo moderno, ha llegado el momento de ensuciarnos las manos.

En este capítulo aprenderás a usar **Docker**, la plataforma líder para crear, administrar y ejecutar contenedores.
Partiremos desde lo más básico: instalar Docker en tu máquina, ejecutar tu primer contenedor, y entender qué son las **imágenes** y los **contenedores** realmente.

No te preocupes si nunca has usado Docker antes. Iremos paso a paso, asegurándonos de que cada comando y concepto quede claro, para que puedas construir tus propios contenedores con total confianza.

Prepárate para convertir tu ordenador en una poderosa fábrica de aplicaciones contenerizadas.

**¡Empecemos nuestro viaje práctico con Docker!**

## 1. Instalación de Docker en diferentes sistemas operativos

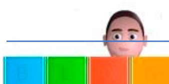
Antes de crear nuestros primeros contenedores, necesitamos instalar Docker en nuestro equipo. La instalación varía ligeramente dependiendo del sistema operativo que uses. A continuación, te guío para cada uno:

### 1.1. Instalación en Windows

1. **Descargar Docker Desktop:**
   - Visita la página oficial: https://www.docker.com/products/docker-desktop
   - Descarga la versión para Windows.
2. **Requisitos previos:**
   - Windows 10/11 Pro, Enterprise o Education (también es posible en Home usando WSL 2).
   - Virtualización habilitada en la BIOS.
3. **Instalación:**
   - Ejecuta el instalador y sigue las instrucciones.
   - Habilita WSL 2 (Subsistema de Windows para Linux) si se solicita.
4. **Verificación:**
   - Abre una terminal (CMD o PowerShell) y ejecuta:

     ```
     bash
     CopiarEditar
     docker --version
     ```

   - Deberías ver algo como `Docker version 24.0.0, build abc123`.



## 1.2. Instalación en macOS

1. **Descargar Docker Desktop:**
   - Desde la misma página: https://www.docker.com/products/docker-desktop.
2. **Instalación:**
   - Arrastra el icono de Docker a tu carpeta de Aplicaciones.
   - Abre Docker Desktop y permite los permisos solicitados.
3. **Verificación:**
   - Abre Terminal y ejecuta:

     ```
     bash
     CopiarEditar
     docker --version
     ```

## 1.3. Instalación en Linux

La instalación depende de la distribución.
Aquí te muestro el ejemplo para **Ubuntu**:

```
bash
CopiarEditar
# Actualizar paquetes
sudo apt update

# Instalar paquetes necesarios
sudo apt install apt-transport-https ca-certificates curl software-
properties-common

# Añadir la clave GPG oficial de Docker
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor
-o /etc/apt/trusted.gpg.d/docker.gpg

# Añadir repositorio de Docker
echo "deb [arch=$(dpkg --print-architecture)]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo
tee /etc/apt/sources.list.d/docker.list

# Actualizar repositorios e instalar Docker
sudo apt update
sudo apt install docker-ce docker-ce-cli containerd.io

# Verificar instalación
docker --version
```

**Tip:** En Linux, suele ser necesario usar `sudo` para ejecutar Docker o agregar tu usuario al grupo `docker` para evitarlo.

### 1.4. Comprobación final

Independientemente de tu sistema, para comprobar que Docker funciona correctamente, ejecuta:

```
bash
CopiarEditar
docker run hello-world
```

Este comando descargará una pequeña imagen de prueba y ejecutará un contenedor que imprime un mensaje de éxito.
Si ves el mensaje `Hello from Docker!`, ¡Docker está listo para usar!

# 2. Primeros pasos: tu primer contenedor

Ahora que tienes Docker instalado, es momento de lanzar tu primer contenedor y entender cómo funciona.

Vamos a usar una imagen oficial muy sencilla: **`alpine`**, una pequeña distribución de Linux ideal para pruebas.

### 2.1. Ejecutar un contenedor básico

Abre una terminal o consola y escribe:

```
bash
CopiarEditar
docker run alpine echo "¡Hola desde un contenedor!"
```

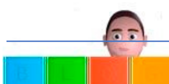
¿Qué pasa aquí?

- `docker run`: le dices a Docker que quieres **crear** y **ejecutar** un contenedor.
- `alpine`: es el **nombre de la imagen** que Docker buscará. Si no la tienes localmente, la descargará automáticamente.
- `echo "¡Hola desde un contenedor!"`: es el **comando** que se ejecutará dentro del contenedor.

**Resultado esperado:**

```
bash
CopiarEditar
¡Hola desde un contenedor!
```

¡Acabas de lanzar tu primer contenedor Docker! 🚀



## 2.2. Entendiendo lo que sucedió

- Docker descargó la imagen de **Alpine** (si no la tenías).
- Creó un nuevo contenedor a partir de esa imagen.
- Ejecutó el comando `echo`.
- Terminó la ejecución y **apagó** el contenedor.

**Importante:**
Por defecto, los contenedores de Docker **viven solo mientras su proceso principal esté activo**. Cuando el proceso (`echo`) terminó, el contenedor se cerró.

---

## 2.3. Listar contenedores

Para ver los contenedores que están corriendo:

```
bash
CopiarEditar
docker ps
```

Para ver todos los contenedores (aunque ya estén parados):

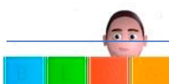
```
bash
CopiarEditar
docker ps -a
```

Verás una lista donde estará tu contenedor de Alpine, probablemente con estado "Exited".

---

## 2.4. Resumen rápido

- Cada vez que ejecutas `docker run`, creas un nuevo contenedor.
- Si el proceso principal termina, el contenedor se detiene.
- Puedes ver y administrar contenedores usando `docker ps` y otros comandos que veremos más adelante.



# 3. Entendiendo imágenes y contenedores

Uno de los conceptos más importantes en Docker es entender bien la diferencia entre **imágenes** y **contenedores**.
Aunque suelen mencionarse juntos, **no son lo mismo**.

---

## 3.1. ¿Qué es una imagen?

Una **imagen** es una **plantilla de solo lectura**.
Contiene:

- El sistema de archivos.
- El código de la aplicación.
- Todas las dependencias necesarias.

Piensa en la imagen como el **molde** de una aplicación.

**Ejemplo:**
La imagen `alpine` que usamos antes es una plantilla de una pequeña distribución de Linux.

---

## 3.2. ¿Qué es un contenedor?

Un **contenedor** es una **instancia en ejecución** de una imagen.
Cuando usas `docker run`, Docker:

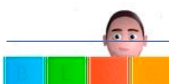
- **Toma** una imagen.
- **Crea** una copia aislada de ella.
- **Ejecuta** los procesos necesarios dentro de ese contenedor.

El contenedor puede:

- Modificar su sistema de archivos temporalmente.
- Ejecutar procesos en aislamiento.
- Tener su propio entorno de red y variables.

**Importante:** cuando un contenedor se detiene o se elimina, sus cambios desaparecen (a menos que uses volúmenes, que veremos más adelante).

---



## 3.3. Analogía sencilla

Para entenderlo aún mejor:

| Concepto | Comparación práctica |
|---|---|
| Imagen | Receta de cocina (instrucciones y materiales) |
| Contenedor | Plato ya cocinado usando esa receta |

- La **receta** (imagen) no cambia.
- Cada **plato** (contenedor) puede ser preparado, modificado, consumido o desechado.

---

## 3.4. Comandos básicos

- **Listar imágenes descargadas:**

```
bash
CopiarEditar
docker images
```

- **Eliminar una imagen:**

```
bash
CopiarEditar
docker rmi nombre_de_la_imagen
```

- **Eliminar un contenedor:**

```
bash
CopiarEditar
docker rm id_del_contenedor
```

- **Ver contenedores detenidos:**

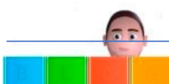
```
bash
CopiarEditar
docker ps -a
```

---

**Resumen:**

Las imágenes son las plantillas; los contenedores son instancias vivas basadas en esas plantillas.

Dominar este concepto te permitirá construir y gestionar tus aplicaciones de forma ordenada y eficiente.



# 4. Dockerfiles: Cómo construir tus propias imágenes

Hasta ahora hemos usado imágenes creadas por otros (como `alpine`), pero una de las mayores potencias de Docker es que **puedes crear tus propias imágenes**.
Para eso existe el **Dockerfile**.

---

## 4.1. ¿Qué es un Dockerfile?

Un **Dockerfile** es un archivo de texto que contiene una lista de **instrucciones** que Docker sigue para construir una imagen personalizada.

Cada instrucción agrega una capa a la imagen final.
Por ejemplo:

- Definir el sistema operativo base.
- Copiar archivos.
- Instalar dependencias.
- Configurar comandos de arranque.

---

## 4.2. Estructura básica de un Dockerfile

Aquí tienes un ejemplo muy sencillo:

```
dockerfile
CopiarEditar
# Usar una imagen base
FROM alpine

# Añadir un mensaje al contenedor
CMD ["echo", "¡Hola desde mi propia imagen!"]
```

**Explicación:**

- `FROM`: indica la imagen base desde la cual construir.
- `CMD`: define el comando que se ejecutará cuando el contenedor arranque.

---

## 4.3. Crear una imagen a partir de un Dockerfile

Supongamos que guardas el contenido anterior en un archivo llamado `Dockerfile`.

1. **Construir la imagen:**

```
bash
CopiarEditar
docker build -t mi-primer-imagen .
```

- `-t mi-primer-imagen`: etiqueta (nombre) que quieres darle a la imagen.
- `.` (punto): indica que el Dockerfile está en el directorio actual.

2. **Verificar la imagen:**

```
bash
CopiarEditar
docker images
```

3. **Ejecutar un contenedor basado en tu imagen:**

```
bash
CopiarEditar
docker run mi-primer-imagen
```

**Resultado esperado:**

```
bash
CopiarEditar
¡Hola desde mi propia imagen!
```

¡Acabas de crear y ejecutar tu primera imagen personalizada! 🚀

## 4.4. Instrucciones comunes en Dockerfile

- `FROM`: Imagen base.
- `COPY`: Copia archivos al contenedor.
- `RUN`: Ejecuta comandos durante la construcción.
- `CMD`: Define el comando predeterminado al iniciar.
- `EXPOSE`: Expone un puerto.
- `ENV`: Define variables de entorno.
- `WORKDIR`: Establece el directorio de trabajo dentro del contenedor.

Con estas instrucciones, puedes construir imágenes **muy potentes y adaptadas** a tus aplicaciones.

**Resumen:**

Un Dockerfile es como una receta que define paso a paso cómo construir una imagen de Docker, permitiéndote automatizar la creación de entornos exactamente como los necesitas.

# 5. Gestión de imágenes y contenedores

A medida que creas y usas contenedores, tu sistema empezará a acumular imágenes, contenedores detenidos, capas intermedias, volúmenes y más.
Para mantener todo organizado y bajo control, es esencial saber cómo **gestionar eficientemente imágenes y contenedores**.

---

## 5.1. Ver contenedores activos y detenidos

- **Contenedores en ejecución:**

```
bash
CopiarEditar
docker ps
```

- **Todos los contenedores (incluidos los detenidos):**

```
bash
CopiarEditar
docker ps -a
```

---

## 5.2. Eliminar contenedores

- **Eliminar un contenedor específico (debe estar detenido):**

```
bash
CopiarEditar
docker rm <ID o nombre>
```

- **Eliminar todos los contenedores detenidos:**

```
bash
CopiarEditar
docker container prune
```

⚠️ Este comando eliminará todos los contenedores que no estén activos. Confirmará antes de hacerlo.

---

## 5.3. Ver y eliminar imágenes

- **Ver todas las imágenes descargadas:**

```
bash
CopiarEditar
docker images
```

- **Eliminar una imagen específica:**

```
bash
CopiarEditar
docker rmi <nombre o ID>
```

- **Eliminar todas las imágenes sin usar (dangling):**

```
bash
CopiarEditar
docker image prune
```

- **Eliminar todas las imágenes no utilizadas:**

```
bash
CopiarEditar
docker image prune -a
```

---

## 5.4. Limpiar todo lo no usado (contenedores, imágenes, redes, volúmenes)

Si quieres liberar espacio en tu sistema de forma más agresiva:

```
bash
CopiarEditar
docker system prune
```

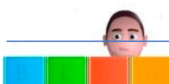
Usa `docker system prune -a` si también deseas eliminar **todas** las imágenes no utilizadas.

---

## 5.5. Detener y reiniciar contenedores

- **Detener un contenedor en ejecución:**

```
bash
CopiarEditar
docker stop <ID o nombre>
```



- **Reiniciar un contenedor detenido:**

  ```
  bash
  CopiarEditar
  docker start <ID o nombre>
  ```

- **Eliminar un contenedor en ejecución (forzado):**

  ```
  bash
  CopiarEditar
  docker rm -f <ID o nombre>
  ```

---

## 5.6. Renombrar contenedores

Puedes asignar un nombre a un contenedor desde el principio:

```
bash
CopiarEditar
docker run --name mi_contenedor alpine
```

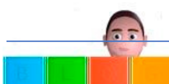
O renombrarlo después:

```
bash
CopiarEditar
docker rename viejo_nombre nuevo_nombre
```

---

**Resumen:**

Docker te ofrece una variedad de comandos para listar, detener, eliminar y limpiar imágenes y contenedores. Una buena gestión te ahorrará espacio y mantendrá tu entorno limpio y eficiente.



# 6. Docker Compose: Orquestación básica

Hasta ahora hemos trabajado con un solo contenedor a la vez.
¿Pero qué pasa si tu aplicación necesita varios servicios a la vez? (por ejemplo, una app web + base de datos + caché).

**Docker Compose** es la herramienta de Docker que permite **definir y administrar múltiples contenedores** fácilmente usando un simple archivo de configuración.

---

## 6.1. ¿Qué es Docker Compose?

Docker Compose usa un archivo llamado `docker-compose.yml` donde describes:

- Las imágenes a usar.
- Las redes necesarias.
- Los volúmenes para persistencia de datos.
- Cómo deben comunicarse los contenedores entre sí.

Con un solo comando, puedes levantar toda una aplicación completa compuesta por varios servicios.

---

## 6.2. Instalación de Docker Compose

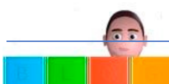
Actualmente, **Docker Desktop** (Windows y macOS) ya incluye Docker Compose integrado.
En Linux, suele instalarse con el paquete de Docker o manualmente.

Verifica si lo tienes:

```
bash
CopiarEditar
docker compose version
```

(Sí, en versiones recientes es `docker compose` con espacio, no `docker-compose`).

---



## 6.3. Ejemplo básico de `docker-compose.yml`

Supongamos que queremos correr una aplicación que necesita:

- Un contenedor web basado en **Nginx**.
- Un contenedor de base de datos **MySQL**.

Archivo `docker-compose.yml`:

```
yaml
CopiarEditar
version: '3'
services:
  web:
    image: nginx
    ports:
      - "80:80"

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: ejemplo123
```

**Explicación rápida:**

- `services`: define los diferentes contenedores.
- `web`: usa la imagen de nginx y expone el puerto 80.
- `db`: usa la imagen de MySQL 5.7 con una contraseña de root.

---

## 6.4. Comandos básicos de Docker Compose

- **Levantar los servicios:**

```
bash
CopiarEditar
docker compose up
```

- **Levantar en segundo plano (modo "detached"):**

```
bash
CopiarEditar
docker compose up -d
```

- **Detener y eliminar todos los servicios:**

```
bash
CopiarEditar
docker compose down
```

- **Ver el estado de los servicios:**

```
bash
CopiarEditar
docker compose ps
```

---

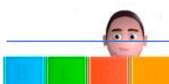
## 6.5. Ventajas de usar Docker Compose

- Gestionar varios contenedores con un solo archivo.
- Configurar redes, volúmenes y variables de entorno de manera organizada.
- Facilitar el despliegue de entornos completos de desarrollo o producción.

---

**Resumen:**

Docker Compose te permite orquestar aplicaciones multi-contenedor de forma sencilla, usando un solo archivo de texto. Ideal para ambientes de desarrollo, pruebas y despliegues iniciales.



# 7. Persistencia de datos: Volúmenes y Bind Mounts

Por defecto, **todo lo que ocurre dentro de un contenedor desaparece** cuando el contenedor se elimina.
Esto es un problema si tu aplicación necesita **guardar datos** como archivos, bases de datos o configuraciones.

Para solucionar esto, Docker ofrece dos mecanismos de **persistencia**:

- **Volúmenes**
- **Bind Mounts**

---

## 7.1. ¿Qué es un volumen?

Un **volumen** es un almacenamiento **gestionado por Docker**.
Se guarda fuera del ciclo de vida del contenedor y puede ser compartido entre varios contenedores.

**Crear un volumen:**

```
bash
CopiarEditar
docker volume create mi_volumen
```

**Usar un volumen en un contenedor:**

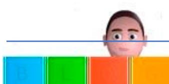
```
bash
CopiarEditar
docker run -d --name contenedor-con-volumen -v mi_volumen:/datos alpine
```

Aquí `mi_volumen` es la carpeta en tu sistema, y `/datos` es donde el contenedor verá esos archivos.

**Ventajas de los volúmenes:**

- Docker se encarga de su ubicación y gestión.
- Fáciles de respaldar o migrar.
- Son independientes del contenedor.

---



## 7.2. ¿Qué es un Bind Mount?

Un **bind mount** enlaza directamente una carpeta de tu máquina host a una carpeta dentro del contenedor.

**Ejemplo de uso:**

```
bash
CopiarEditar
docker run -d --name contenedor-con-mount -v /ruta/local:/datos alpine
```

Aquí `/ruta/local` es una carpeta en tu máquina y `/datos` es su equivalente dentro del contenedor.

**Ventajas de los Bind Mounts:**

- Útiles para desarrollo: cambios en tu carpeta local se reflejan **inmediatamente** en el contenedor.
- Total control sobre los archivos.

**Desventajas:**

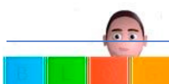
- Dependen de la estructura de tu máquina, no son tan portables.

---

## 7.3. Comparativa rápida

| Característica | Volúmenes | Bind Mounts |
|---|---|---|
| Gestión | Docker | Sistema de archivos |
| Portabilidad | Alta | Baja |
| Casos de uso | Producción | Desarrollo local |
| Control | Limitado (pero seguro) | Total sobre los archivos |

---



## 7.4. Comandos útiles de volúmenes

- **Listar volúmenes existentes:**

```
bash
CopiarEditar
docker volume ls
```

- **Inspeccionar un volumen:**

```
bash
CopiarEditar
docker volume inspect mi_volumen
```

- **Eliminar un volumen:**

```
bash
CopiarEditar
docker volume rm mi_volumen
```

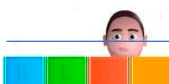
- **Eliminar volúmenes no usados:**

```
bash
CopiarEditar
docker volume prune
```

---

**Resumen:**

Para guardar datos más allá de la vida de un contenedor, puedes usar volúmenes gestionados por Docker o bind mounts directos a tu máquina. Cada uno tiene su lugar dependiendo del objetivo: producción o desarrollo.



# 8. Redes en Docker

Cuando trabajas con varios contenedores, es esencial que puedan **comunicarse entre ellos**. Docker facilita esto creando y gestionando **redes virtuales**.

Entender las redes de Docker te permitirá conectar servicios de manera segura, flexible y controlada.

---

## 8.1. ¿Qué es una red en Docker?

Una **red** en Docker es un espacio virtual donde los contenedores pueden:

- **Detectarse automáticamente** por nombre.
- **Comunicarse** usando direcciones IP internas.
- **Aislarse** del mundo exterior si es necesario.

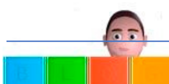
---

## 8.2. Tipos de redes en Docker

Docker ofrece varios tipos de redes. Las principales son:

- **bridge** (puente):
  - Red privada por defecto cuando no se especifica otra.
  - Ideal para conectar contenedores en una misma máquina.
- **host**:
  - El contenedor usa directamente la red del host (sin aislamiento).
  - Puede ser útil para necesidades de máximo rendimiento.
- **none**:
  - Sin conexión de red. El contenedor está completamente aislado.
- **overlay** (solo en Swarm o Kubernetes):
  - Conecta contenedores que corren en distintas máquinas.

---



## 8.3. Crear y usar una red personalizada

### Crear una red tipo bridge:

```
bash
CopiarEditar
docker network create mi_red_personalizada
```

### Ejecutar un contenedor en esa red:

```
bash
CopiarEditar
docker run -d --name contenedor1 --network mi_red_personalizada alpine sleep
3600
```

### Ejecutar otro contenedor en la misma red:

```
bash
CopiarEditar
docker run -d --name contenedor2 --network mi_red_personalizada alpine sleep
3600
```

Ahora, `contenedor1` y `contenedor2` pueden comunicarse **por nombre**:

```
bash
CopiarEditar
docker exec -it contenedor1 ping contenedor2
```

---

## 8.4. Listar y administrar redes

- **Ver redes disponibles:**

  ```
  bash
  CopiarEditar
  docker network ls
  ```

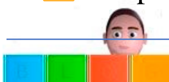
- **Inspeccionar una red:**

  ```
  bash
  CopiarEditar
  docker network inspect mi_red_personalizada
  ```

- **Eliminar una red:**

  ```
  bash
  CopiarEditar
  docker network rm mi_red_personalizada
  ```

⚠️ No puedes eliminar una red que aún tenga contenedores conectados.



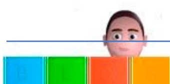
## 8.5. Redes en Docker Compose

Cuando usas Docker Compose, **se crea automáticamente** una red para que todos los servicios definidos puedan comunicarse fácilmente entre sí, sin necesidad de configurarlo manualmente.

---

**Resumen:**

Las redes en Docker permiten que los contenedores se comuniquen de forma segura y controlada. Crear redes personalizadas mejora la organización y el aislamiento de tus aplicaciones.



# 9. Buenas prácticas de seguridad en Docker

Trabajar con contenedores trae grandes ventajas, pero también nuevos desafíos de **seguridad**. Un contenedor mal configurado puede abrir puertas a vulnerabilidades críticas.

Aquí te comparto las principales **buenas prácticas** para mantener seguros tus contenedores y tu infraestructura.

---

## 9.1. Usa imágenes oficiales y verificadas

Siempre que sea posible:

- Utiliza **imágenes oficiales** del Docker Hub o fuentes de confianza.
- Verifica la integridad de las imágenes usando firmas digitales o herramientas como **Docker Content Trust**.

⚠️ No confíes en imágenes de fuentes desconocidas.

---

## 9.2. Minimiza el tamaño de tus imágenes

- Usa **imágenes base ligeras** (por ejemplo, `alpine` en lugar de `ubuntu`).
- Instala solo lo que sea **estrictamente necesario**.
- Esto reduce la **superficie de ataque** y acelera los despliegues.

---

## 9.3. No ejecutes contenedores como root

Por defecto, los contenedores corren como usuario `root`. Esto es un riesgo si alguien logra escapar del contenedor.

**Solución:**

- Define un usuario no privilegiado en tu Dockerfile:

  ```
  dockerfile
  CopiarEditar
  USER 1000
  ```

- O crea un usuario específico si tu aplicación lo necesita.

---

## 9.4. Restringe el acceso a la API de Docker

La API de Docker permite control total sobre contenedores y el sistema.
**Nunca expongas** la API sin protección.

Si necesitas exponerla:

- Usa certificados TLS.
- Restringe el acceso a IPs de confianza.

---

## 9.5. Usa herramientas de escaneo de imágenes

Antes de usar o subir una imagen:

- Analízala en busca de vulnerabilidades con herramientas como:
  - **Docker Scan** (`docker scan imagen`)
  - **Trivy**
  - **Clair**

Esto te ayudará a identificar librerías o configuraciones inseguras.

---

## 9.6. Limita capacidades del contenedor

Usa flags como `--cap-drop` o `--security-opt` para reducir los permisos de los contenedores.

Ejemplo:

```
bash
CopiarEditar
docker run --cap-drop ALL alpine
```

Esto ejecuta un contenedor **sin capacidades privilegiadas**.

---

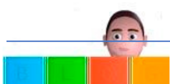
## 9.7. Actualiza imágenes y contenedores regularmente

- Mantén las imágenes actualizadas con las últimas correcciones de seguridad.
- Automatiza el proceso si es posible mediante pipelines de CI/CD.

---

**Resumen:**

La seguridad en Docker comienza desde la elección de imágenes, sigue en la configuración de contenedores, y culmina en una gestión continua de actualizaciones y escaneos. Mejor prevenir que lamentar.



# 10. Optimización de imágenes Docker

Crear imágenes funcionales es solo el primer paso.
Para entornos reales, es fundamental optimizarlas para que sean:

- **Más pequeñas** (menos consumo de espacio y red).
- **Más rápidas** (despliegues instantáneos).
- **Más seguras** (menos superficie de ataque).

Veamos cómo conseguirlo.

---

## 10.1. Usa imágenes base ligeras

Siempre que puedas:

- Usa imágenes como `alpine` (5 MB) en lugar de `ubuntu` (más de 70 MB).

Ejemplo:

```
dockerfile
CopiarEditar
FROM alpine
```

Cuanto más ligera la imagen base, más pequeña y rápida será la imagen final.

## 10.2. Minimiza el número de capas

Cada instrucción en el Dockerfile (`RUN`, `COPY`, etc.) crea una **nueva capa**.
Agrupar operaciones en una sola instrucción ayuda a reducir el tamaño.

**Mejor:**

```
dockerfile
CopiarEditar
RUN apt update && apt install -y python3 \
    && apt clean \
    && rm -rf /var/lib/apt/lists/*
```

**No tan óptimo:**

```
dockerfile
CopiarEditar
RUN apt update
RUN apt install -y python3
RUN apt clean
```

---

## 10.3. Elimina archivos temporales y cachés

Después de instalar paquetes, limpia todos los archivos innecesarios:

```
dockerfile
CopiarEditar
RUN apt-get update && apt-get install -y paquete \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*
```

Esto evita que tu imagen cargue cachés obsoletos.

---

## 10.4. Usa `.dockerignore`

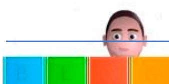
Crea un archivo `.dockerignore` (similar a `.gitignore`) para **excluir archivos** que no quieres copiar a la imagen.

Ejemplo de `.dockerignore`:

```
nginx
CopiarEditar
node_modules
.git
*.log
Dockerfile
```

Esto evita que se copien carpetas o archivos innecesarios al contenedor, reduciendo el peso.

---



## 10.5. Usa multistage builds (compilación por etapas)

Si tu aplicación necesita compilarse (por ejemplo, apps en Go, Node.js o Java), puedes **compilar en una etapa** y **copiar solo lo necesario** al contenedor final.

**Ejemplo:**

```
dockerfile
CopiarEditar
# Etapa de construcción
FROM golang:1.20 AS builder
WORKDIR /app
COPY . .
RUN go build -o app

# Etapa final
FROM alpine
WORKDIR /app
COPY --from=builder /app/app .
CMD ["./app"]
```

Resultado: una imagen **pequeñísima** solo con el binario.

---

## 10.6. Analiza y mide el tamaño de tus imágenes

Usa herramientas como:

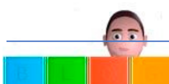
- **Docker Scout** (`docker scout quickview`)
- **Dive** (análisis visual de capas)

Estas te ayudarán a ver qué ocupa espacio y cómo puedes optimizarlo.

---

**Resumen:**

Una imagen Docker optimizada arranca más rápido, consume menos ancho de banda y es más segura. ¡Dedicar tiempo a optimizarlas te ahorrará problemas en producción!



# Capítulo 3: Introducción a Kubernetes

Hasta ahora hemos aprendido a crear, optimizar y gestionar contenedores con Docker.
Pero en aplicaciones reales, donde debes desplegar **docenas o cientos de contenedores**, surgen nuevos retos:

- ¿Qué pasa si uno de los contenedores falla?
- ¿Cómo escalamos automáticamente cuando aumenta la demanda?
- ¿Cómo actualizamos aplicaciones sin interrumpir el servicio?
- ¿Cómo organizamos todo entre múltiples servidores?

Aquí es donde **Kubernetes** entra en escena.

Kubernetes, también conocido como **K8s**, es una plataforma de **orquestación de contenedores** que automatiza:

- El despliegue.
- El escalado.
- El balanceo de carga.
- La actualización.
- Y la recuperación de contenedores.

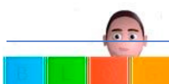
Diseñado por Google y hoy administrado por la **Cloud Native Computing Foundation (CNCF)**, Kubernetes es el estándar de facto para gestionar infraestructura basada en contenedores, tanto en entornos locales como en la nube.

**En este capítulo:**

- Descubrirás qué es Kubernetes y por qué es tan importante.
- Aprenderás su arquitectura básica y sus componentes esenciales.
- Darás tus primeros pasos desplegando aplicaciones en un clúster de Kubernetes.

**Prepárate** para llevar tus habilidades con contenedores al **siguiente nivel**: construir infraestructuras resilientes, escalables y listas para producción.

¡Comencemos nuestro viaje con Kubernetes! 🚀



# 1. ¿Qué es Kubernetes y por qué usarlo?

Kubernetes es una **plataforma de orquestación de contenedores** de código abierto, diseñada para automatizar el despliegue, la gestión, el escalado y la recuperación de aplicaciones contenerizadas.

Fue creado originalmente por Google basándose en su experiencia interna gestionando millones de contenedores, y hoy en día es mantenido por la **Cloud Native Computing Foundation (CNCF)**.

**En pocas palabras:**

Kubernetes es el "sistema operativo" de los clústeres de contenedores.

## 1.1. ¿Qué resuelve Kubernetes?

Trabajar con unos pocos contenedores es fácil.
Pero a medida que crecen tus aplicaciones, gestionar contenedores manualmente se vuelve **inviable**:

- ¿Qué pasa si un contenedor se cae?
- ¿Cómo aseguramos alta disponibilidad?
- ¿Cómo balanceamos tráfico entre múltiples instancias?
- ¿Cómo escalamos automáticamente ante picos de usuarios?
- ¿Cómo actualizamos aplicaciones sin downtime?

Kubernetes resuelve todos estos problemas de forma **automatizada**.

---

## 1.2. Funciones principales de Kubernetes

- **Orquestación automática:**
  Gestiona dónde y cuándo se ejecutan los contenedores en un clúster de máquinas.
- **Escalado automático:**
  Aumenta o disminuye el número de contenedores según la carga de trabajo.
- **Recuperación automática:**
  Si un contenedor falla, Kubernetes lo reinicia o reemplaza automáticamente.
- **Actualizaciones sin interrupciones:**
  Permite desplegar nuevas versiones de aplicaciones sin afectar a los usuarios (rolling updates).
- **Balanceo de carga y descubrimiento de servicios:**
  Distribuye el tráfico de red de manera eficiente entre los contenedores.
- **Gestión de almacenamiento:**
  Proporciona acceso a volúmenes persistentes para los contenedores.

## 1.3. ¿Por qué Kubernetes y no solo Docker?

Docker **por sí solo** ejecuta y gestiona contenedores **en una sola máquina**.
Cuando necesitas múltiples máquinas trabajando juntas, o alta disponibilidad, o escalado dinámico, **Docker solo no es suficiente**.

**Kubernetes**:

- Orquesta contenedores **en múltiples servidores** (clúster).
- Automatiza el escalado, la recuperación y el despliegue.
- Proporciona una capa de abstracción sobre la infraestructura.

---

## 1.4. ¿Dónde se usa Kubernetes?

Hoy en día, Kubernetes se usa en:

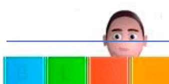
- Grandes compañías tecnológicas (Google, Netflix, Spotify, Airbnb).
- Startups que buscan escalar rápido.
- Proyectos de código abierto.
- Entornos de nube híbrida y multicloud.

Es el **estándar industrial** para la gestión de aplicaciones modernas.

---

**Resumen:**

Kubernetes es la plataforma que te permite pasar de ejecutar contenedores manualmente a gestionar aplicaciones altamente disponibles, escalables y resilientes de forma automatizada.



# 2. Arquitectura de Kubernetes: Componentes principales

Para entender cómo funciona Kubernetes, primero necesitas conocer su **arquitectura interna**. Aunque parece complejo al principio, todo se basa en **pocos componentes fundamentales** que trabajan juntos para mantener el clúster funcionando de forma automática.

Vamos a desglosarlo.

---

## 2.1. Nodo maestro y nodos de trabajo

Un **clúster de Kubernetes** está formado por:

- **Nodo maestro (Control Plane):**
  Coordina el clúster: decide dónde y cómo deben ejecutarse los contenedores.
- **Nodos de trabajo (Workers):**
  Son las máquinas (físicas o virtuales) donde realmente corren los contenedores.

**Simplificando:**

El maestro piensa y organiza; los workers ejecutan.

---

## 2.2. Componentes del nodo maestro (Control Plane)

**a) kube-apiserver:**

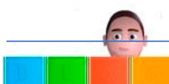
- Es el "cerebro" de Kubernetes.
- Expone una API RESTful que permite comunicarse con el clúster (usuarios, componentes internos y herramientas externas).

**b) etcd:**

- Base de datos distribuida y ligera que guarda todo el **estado** del clúster (configuraciones, información de servicios, secretos...).

**c) kube-scheduler:**

- Decide en qué nodo debe ejecutarse cada contenedor basándose en recursos disponibles, restricciones y políticas.



**d) kube-controller-manager:**

- Supervisa el estado del clúster y ejecuta "controladores" para mantener ese estado deseado (por ejemplo, si un contenedor cae, lo vuelve a levantar).

**e) cloud-controller-manager:** *(opcional)*

- Se encarga de integrar Kubernetes con servicios de nubes públicas (como AWS, GCP o Azure).

## 2.3. Componentes de los nodos de trabajo

**a) kubelet:**

- Agente que corre en cada worker.
- Recibe instrucciones del kube-apiserver y se asegura de que los contenedores estén funcionando como deben.

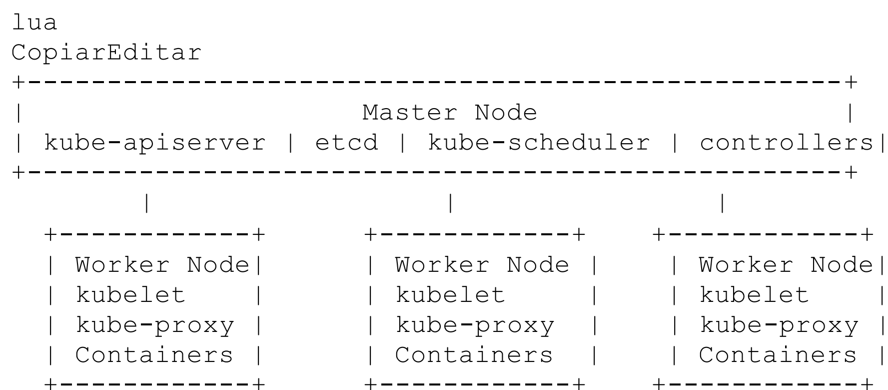
**b) kube-proxy:**

- Gestiona la red en el nodo.
- Permite el acceso de red a los contenedores, incluyendo el balanceo de carga interno.

**c) Container Runtime (Motor de contenedores):**

- Es el software que corre los contenedores en el nodo.
- Docker, containerd o CRI-O son ejemplos de runtimes compatibles.

## 2.4. Diagrama simple de la arquitectura



```
lua
CopiarEditar
+----------------------------------------------------+
|                      Master Node                   |
| kube-apiserver | etcd | kube-scheduler | controllers|
+----------------------------------------------------+
        |                  |                |
  +-------------+     +-------------+    +-------------+
  | Worker Node|     | Worker Node |    | Worker Node|
  | kubelet    |     | kubelet     |    | kubelet    |
  | kube-proxy |     | kube-proxy  |    | kube-proxy |
  | Containers |     | Containers  |    | Containers |
  +-------------+     +-------------+    +-------------+
```

**Resumen:**

Kubernetes separa la inteligencia (Control Plane) de la ejecución (Workers). Esta arquitectura modular le permite ser escalable, resiliente y altamente disponible.

# 3. Instalación local: Minikube, Kind y alternativas

Antes de desplegar Kubernetes en producción o en la nube, es ideal **aprender y practicar** en un clúster local.
Existen herramientas ligeras diseñadas para crear entornos Kubernetes de forma sencilla en tu ordenador.

Veamos las principales opciones:

---

## 3.1. Minikube

**Minikube** es una herramienta que despliega un clúster de Kubernetes de un solo nodo en tu máquina local.

**Ventajas:**

- Fácil de instalar y usar.
- Soporta la mayoría de características estándar de Kubernetes.
- Perfecto para aprender y probar configuraciones.

**Instalación rápida en Windows, macOS o Linux:**

1. Descargar Minikube:
    - https://minikube.sigs.k8s.io/docs/start/
2. Instalar usando un gestor de paquetes (ejemplo en Linux):

```
bash
CopiarEditar
curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

3. Iniciar el clúster:

```
bash
CopiarEditar
minikube start
```

4. Verificar:

```
bash
CopiarEditar
kubectl get nodes
```

Si ves un nodo activo, ¡ya tienes Kubernetes funcionando localmente!

## 3.2. Kind (Kubernetes IN Docker)

**Kind** crea clústeres de Kubernetes usando **contenedores Docker** como nodos.
Ideal para ambientes de testing y CI/CD.

**Ventajas:**

- No necesita máquinas virtuales.
- Rápido de levantar y destruir clústeres.
- Configurable para clústeres multinodo.

**Instalación básica:**

1. Instalar Kind:

```
bash
CopiarEditar
go install sigs.k8s.io/kind@latest
```

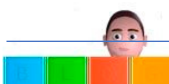
2. Crear un clúster:

```
bash
CopiarEditar
kind create cluster
```

3. Comprobar:

```
bash
CopiarEditar
kubectl get nodes
```

---



## 3.3. Otras alternativas

- **k3s:** Kubernetes ligero y optimizado para IoT y edge computing.
- **microk8s:** Versión empaquetada de Kubernetes para Linux, fácil de instalar y gestionar.

Ambos son opciones excelentes si quieres clústeres rápidos y muy livianos, especialmente en Linux.

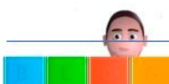
## 3.4. ¿Qué opción elegir?

| Herramienta | Ideal para | Notas |
|---|---|---|
| Minikube | Principiantes y entornos locales | Muy estable y documentado. |
| Kind | Testing, pipelines CI/CD | Requiere Docker instalado. |
| k3s | IoT, edge, servidores ligeros | Súper liviano. |
| microk8s | Ubuntu/Linux rápidos | Integración sencilla. |

---

**Resumen:**

Para empezar con Kubernetes localmente, Minikube es la opción más amigable. Si prefieres algo basado en contenedores, Kind es excelente. Ambas te permitirán practicar y aprender de manera efectiva.



# 4. Tu primer despliegue en Kubernetes

Ahora que tienes tu clúster local funcionando (con Minikube, Kind o cualquier otra herramienta), es momento de realizar **tu primer despliegue** en Kubernetes.
Vamos a desplegar una aplicación muy sencilla para entender el flujo básico.

---

## 4.1. ¿Qué es un despliegue (Deployment)?

En Kubernetes, un **Deployment** es un recurso que:

- Define **qué aplicación** quieres correr.
- Especifica **cuántas réplicas** quieres tener.
- Se encarga de **crear** y **mantener** esos contenedores activos.

En otras palabras:

El Deployment es el "contrato" que le dice a Kubernetes: “quiero 3 copias de esta aplicación corriendo siempre”.

---

## 4.2. Crear tu primer Deployment

Primero, crea un archivo llamado `deployment.yaml` con el siguiente contenido:

```
yaml
CopiarEditar
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hola-mundo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hola-mundo
  template:
    metadata:
      labels:
        app: hola-mundo
    spec:
      containers:
      - name: hola-mundo
        image: nginx
        ports:
        - containerPort: 80
```

**Explicación rápida:**

- `replicas: 2`: queremos 2 copias (pods) corriendo.
- `image: nginx`: usaremos la imagen oficial de Nginx como contenedor.
- `containerPort: 80`: expone el puerto 80 dentro del contenedor.

---

## 4.3. Aplicar el Deployment

Con tu clúster corriendo, ejecuta:

```
bash
CopiarEditar
kubectl apply -f deployment.yaml
```

Esto creará el Deployment en Kubernetes.

---

## 4.4. Verificar el estado

- Para ver los Deployments:

  ```
  bash
  CopiarEditar
  kubectl get deployments
  ```

- Para ver los Pods creados:

  ```
  bash
  CopiarEditar
  kubectl get pods
  ```

Deberías ver **dos pods** activos, basados en Nginx.

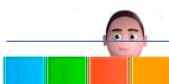
---

## 4.5. Exponer la aplicación

Para acceder a tu aplicación desde fuera del clúster, puedes crear un **Service**.

Comando rápido:

```
bash
CopiarEditar
kubectl expose deployment hola-mundo --type=NodePort --port=80
```



Esto crea un Service que expone el Deployment en un puerto aleatorio del nodo.

Para saber qué puerto usar:

```
bash
CopiarEditar
kubectl get services
```

Minikube te facilita abrirlo directamente en el navegador:

```
bash
CopiarEditar
minikube service hola-mundo
```

---

## 4.6. ¿Qué has logrado?

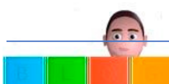
- Kubernetes ha creado 2 copias (pods) de Nginx.
- Las mantiene activas (si una falla, la recrea).
- Ha expuesto tu servicio para acceso externo.

¡Tu primer despliegue en Kubernetes es un éxito! 🎉

---

**Resumen:**

En Kubernetes, los Deployments gestionan aplicaciones asegurando que siempre haya el número correcto de copias corriendo, mientras los Services permiten que esas aplicaciones sean accesibles.



# 5. Pods, ReplicaSets y Deployments

Cuando trabajas en Kubernetes, escucharás continuamente tres términos clave: **Pod**, **ReplicaSet** y **Deployment**.
Entender cómo se relacionan es fundamental para dominar la orquestación de contenedores.

Vamos a explicarlo de manera clara y sencilla.

---

## 5.1. ¿Qué es un Pod?

Un **Pod** es la **unidad más pequeña** que puedes desplegar en Kubernetes.

- Un pod encapsula:
  - Uno o varios contenedores (normalmente uno solo).
  - Almacenamiento compartido (si es necesario).
  - Red compartida.

**Importante:**

Un Pod es efímero. Si un Pod falla, **no se repara solo** (Kubernetes lo reemplaza creando uno nuevo).

---

## 5.2. ¿Qué es un ReplicaSet?

Un **ReplicaSet** asegura que un número determinado de réplicas de un Pod estén corriendo **en todo momento**.

**Ejemplo:**

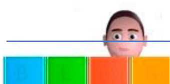
- Si dices que quieres 3 réplicas y uno de los pods falla, el ReplicaSet detecta el fallo y crea un nuevo pod para reemplazarlo.

**ReplicaSet = Policía que patrulla para que siempre haya el número correcto de Pods vivos.**

---

## 5.3. ¿Qué es un Deployment?

Un **Deployment** es una **capa superior** que administra ReplicaSets y gestiona la evolución de tu aplicación.



Con un Deployment puedes:

- Crear ReplicaSets automáticamente.
- Actualizar aplicaciones de forma controlada (rolling updates).
- Hacer rollback a una versión anterior si algo falla.

**Deployment = Gerente de recursos: crea y actualiza ReplicaSets según tus instrucciones.**

---

## 5.4. Relación entre ellos

La estructura sería:

```
nginx
CopiarEditar
Deployment ➔ ReplicaSet ➔ Pods
```

- **Deployment** define el estado deseado.
- **ReplicaSet** asegura que ese estado se cumpla.
- **Pods** son las instancias reales que ejecutan tu aplicación.

---

## 5.5. Visualización práctica

Supongamos que despliegas una aplicación:

```
yaml
CopiarEditar
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mi-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mi-app
  template:
    metadata:
      labels:
        app: mi-app
    spec:
      containers:
      - name: mi-app
        image: nginx
```

**¿Qué pasa?**

- Kubernetes crea un **Deployment** llamado `mi-app`.
- El Deployment crea un **ReplicaSet** asociado.
- El ReplicaSet lanza **3 Pods** corriendo Nginx.

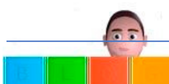
Si quieres actualizar Nginx a otra versión, el Deployment:

- Crea un nuevo ReplicaSet.
- Va migrando los Pods gradualmente.
- Elimina el antiguo ReplicaSet al terminar.

---

**Resumen:**

El Deployment es quien da las órdenes, el ReplicaSet es quien asegura que las órdenes se cumplan, y los Pods son los trabajadores que ejecutan la aplicación.



# 6. Servicios en Kubernetes: ClusterIP, NodePort y LoadBalancer

Cuando despliegas Pods en Kubernetes, te enfrentas a un problema:
los Pods tienen IPs efímeras y privadas, y **pueden cambiar** cada vez que un Pod se reinicia o migra a otro nodo.

**¿Cómo hacemos para exponer de forma estable nuestras aplicaciones?**
La respuesta son los **Servicios** (**Services**).

Un **Service** es una **abstracción de red** que:

- Da un **nombre DNS estable** a los Pods.
- Permite el **balanceo de carga** entre réplicas.
- Expone aplicaciones **dentro o fuera** del clúster.

---

## 6.1. Tipos principales de Servicios

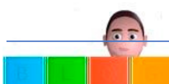
**Kubernetes ofrece tres tipos básicos de Servicios:**

| Tipo | ¿Qué hace? | ¿Uso principal? |
|---|---|---|
| ClusterIP | Acceso interno dentro del clúster | Comunicación entre Pods |
| NodePort | Expone el servicio en un puerto del nodo | Acceso externo sencillo |
| LoadBalancer | Usa un balanceador de carga externo | Entornos en la nube pública |

---

## 6.2. ClusterIP (por defecto)

- Es el tipo **por defecto** si no especificas otro.
- Solo es accesible **desde dentro** del clúster.
- Se utiliza para comunicación interna entre servicios (backend, bases de datos, APIs internas).



**Ejemplo de Service tipo ClusterIP:**

```
yaml
CopiarEditar
apiVersion: v1
kind: Service
metadata:
  name: mi-servicio
spec:
  selector:
    app: mi-app
  ports:
    - port: 80
      targetPort: 8080
  type: ClusterIP
```

**Nota:** `port` es el puerto del Service; `targetPort` es el puerto donde escucha el Pod.

---

## 6.3. NodePort

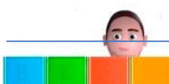
- Permite acceder a un Servicio **desde fuera del clúster** usando la IP del nodo + un puerto elevado (30000-32767).
- Sencillo, pero poco flexible en producción.

**Ejemplo de Service tipo NodePort:**

```
yaml
CopiarEditar
apiVersion: v1
kind: Service
metadata:
  name: mi-servicio-nodeport
spec:
  type: NodePort
  selector:
    app: mi-app
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30080
```

- Acceso: `http://IP-del-nodo:30080`

---



## 6.4. LoadBalancer

- Funciona **en entornos de nube** (AWS, GCP, Azure).
- Solicita un **balanceador de carga** al proveedor.
- Expone el servicio automáticamente al exterior.

**Ejemplo de Service tipo LoadBalancer:**

```
yaml
CopiarEditar
apiVersion: v1
kind: Service
metadata:
  name: mi-servicio-loadbalancer
spec:
  type: LoadBalancer
  selector:
    app: mi-app
  ports:
    - port: 80
      targetPort: 8080
```

- El proveedor creará una IP pública para tu aplicación.

**Importante:** En Minikube puedes simular LoadBalancer usando:

```
bash
CopiarEditar
minikube tunnel
```

---

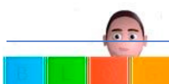
## 6.5. ¿Cómo elegir el tipo de Service?

| Escenario | Tipo recomendado |
| --- | --- |
| Comunicación interna entre microservicios | ClusterIP |
| Exponer un servicio para testing rápido | NodePort |
| Producción en la nube | LoadBalancer |

---

**Resumen:**

Los Servicios en Kubernetes crean redes estables para acceder a tus Pods. Dependiendo del caso, puedes usar ClusterIP, NodePort o LoadBalancer para hacer que tu aplicación sea accesible donde y cuando la necesites.



# 7. ConfigMaps y Secrets

En una aplicación real, necesitas manejar **configuraciones** como:

- Variables de entorno.
- Rutas de conexión a bases de datos.
- Claves API o contraseñas.

En Kubernetes, para **separar configuración de código**, usamos dos recursos especiales:

- **ConfigMaps** para configuraciones normales.
- **Secrets** para información sensible.

Vamos a ver cómo funcionan.

---

## 7.1. ¿Qué es un ConfigMap?

Un **ConfigMap** almacena datos de configuración no sensibles en forma de **pares clave-valor**.

**¿Para qué sirve?**

- Definir variables de entorno para Pods.
- Configurar aplicaciones sin necesidad de modificar la imagen del contenedor.
- Compartir archivos de configuración completos.

---

**Ejemplo de ConfigMap:**

Archivo `configmap.yaml`:

```
yaml
CopiarEditar
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-app
data:
  APP_ENV: "produccion"
  API_URL: "https://api.ejemplo.com"
```

**Crear el ConfigMap:**

```
bash
CopiarEditar
kubectl apply -f configmap.yaml
```

**Usarlo en un Pod:**

```yaml
yaml
CopiarEditar
spec:
  containers:
  - name: app
    image: miapp:latest
    envFrom:
    - configMapRef:
        name: config-app
```

Esto inyecta las variables `APP_ENV` y `API_URL` en el contenedor.

---

## 7.2. ¿Qué es un Secret?

Un **Secret** almacena **datos sensibles** (contraseñas, tokens, certificados).

**¿Por qué usar Secrets?**

- Porque Kubernetes trata esos datos de manera más segura:
    - Los cifra (opcionalmente) al almacenarlos.
    - Limita su exposición en los Pods.

---

**Ejemplo de Secret:**

Archivo `secret.yaml`:

```yaml
yaml
CopiarEditar
apiVersion: v1
kind: Secret
metadata:
  name: secret-db
type: Opaque
data:
  username: dXN1YXJpbyA=    # base64 de "usuario"
  password: cGFzc3dvcmQ=    # base64 de "password"
```

⚠️ **Nota:** Los valores deben estar codificados en **Base64**.

**Crear el Secret:**

```bash
bash
CopiarEditar
kubectl apply -f secret.yaml
```

**Usarlo en un Pod:**

```
yaml
CopiarEditar
spec:
  containers:
  - name: app
    image: miapp:latest
    env:
    - name: DB_USER
      valueFrom:
        secretKeyRef:
          name: secret-db
          key: username
    - name: DB_PASS
      valueFrom:
        secretKeyRef:
          name: secret-db
          key: password
```

Así el contenedor tendrá las variables `DB_USER` y `DB_PASS` cargadas de forma segura.

---

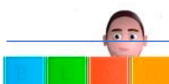
## 7.3. Diferencias entre ConfigMap y Secret

| Característica | ConfigMap | Secret |
|---|---|---|
| Tipo de datos | Configuración normal | Datos sensibles |
| Codificación Base64 | No requerida | Obligatoria |
| Seguridad especial | No | Sí |
| Casos de uso | Variables de entorno, configuración | Contraseñas, tokens, claves |

---

**Resumen:**

Usa ConfigMaps para almacenar configuraciones generales y Secrets para guardar información sensible. Así mantienes tu aplicación limpia, segura y flexible en Kubernetes.



# 8. Volúmenes y almacenamiento persistente (PVCs)

Por defecto, los datos generados dentro de un Pod **se pierden** si el Pod se elimina o reinicia.
Pero muchas aplicaciones necesitan **guardar datos** de forma duradera (bases de datos, archivos, configuraciones).
Aquí es donde entran los **volúmenes persistentes** en Kubernetes.

Vamos a entender cómo funcionan.

---

## 8.1. ¿Qué es un Volumen en Kubernetes?

Un **Volumen** en Kubernetes es un recurso de almacenamiento que puede ser:

- Interno (disco local de un nodo).
- Externo (NFS, discos en la nube, Ceph, etc.).

**Importante:**

A diferencia del almacenamiento interno del contenedor, el contenido de un Volumen **sobrevive** si el contenedor falla o se reinicia.

---

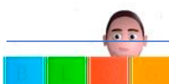
## 8.2. ¿Qué es un PersistentVolume (PV)?

Un **PersistentVolume (PV)** es una **pieza de almacenamiento real** en el clúster que fue previamente provisionada:

- Manualmente (por un administrador).
- Dinámicamente (por Kubernetes en la nube).

Es un recurso del clúster que existe **independientemente de los Pods**.

---



## 8.3. ¿Qué es un PersistentVolumeClaim (PVC)?

Un **PersistentVolumeClaim (PVC)** es una **petición** de almacenamiento hecha por un Pod.

En el PVC defines:

- Cuánto espacio necesitas.
- Qué tipo de almacenamiento quieres.

**Kubernetes busca un PV** disponible que cumpla con esas condiciones y **lo conecta** al Pod.

---

## 8.4. Flujo básico de almacenamiento en Kubernetes

```
java
CopiarEditar
Administrador crea PersistentVolume (PV)  ➔
Aplicación crea PersistentVolumeClaim (PVC) ➔
Kubernetes vincula PVC a un PV disponible ➔
Pod usa el PVC como volumen.
```

---

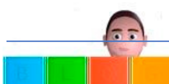
## 8.5. Ejemplo práctico: usar un PVC

**Definir un PVC:** (archivo `pvc.yaml`)

```
yaml
CopiarEditar
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mi-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

- `accessModes`: modo de acceso (por ejemplo, lectura/escritura por un solo nodo).
- `storage`: cantidad de almacenamiento solicitado.

---



**Usar el PVC en un Pod:**

```
yaml
CopiarEditar
apiVersion: v1
kind: Pod
metadata:
  name: pod-con-pvc
spec:
  containers:
  - name: app
    image: nginx
    volumeMounts:
    - mountPath: "/usr/share/nginx/html"
      name: almacenamiento
  volumes:
  - name: almacenamiento
    persistentVolumeClaim:
      claimName: mi-pvc
```

**¿Qué pasa aquí?**

- El contenedor Nginx tendrá un volumen montado en `/usr/share/nginx/html`.
- Ese volumen es el almacenamiento proporcionado por el PVC `mi-pvc`.

---

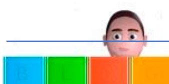
## 8.6. StorageClass: provisión dinámica

En entornos de nube o clusters bien configurados, Kubernetes puede **crear automáticamente** los PV necesarios usando **StorageClasses**.
Así no necesitas crear volúmenes manualmente.

Ejemplo rápido:

```
yaml
CopiarEditar
spec:
  storageClassName: standard
```

---



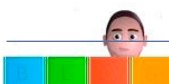
## 8.7. Modos de acceso más comunes

- **ReadWriteOnce (RWO):** Un único nodo puede montar el volumen en lectura y escritura.
- **ReadOnlyMany (ROX):** Múltiples nodos pueden montar el volumen en solo lectura.
- **ReadWriteMany (RWX):** Múltiples nodos pueden montar el volumen en lectura y escritura (menos común).

---

**Resumen:**

Kubernetes separa el almacenamiento físico (PV) de las peticiones de almacenamiento (PVC), permitiendo un manejo dinámico, seguro y escalable de los datos de tus aplicaciones.



# 9. Namespaces y control de recursos

A medida que un clúster Kubernetes crece, se vuelve necesario **organizar**, **aislar** y **gestionar** mejor sus recursos.
Aquí es donde entran dos conceptos fundamentales:

- **Namespaces**: para organizar los recursos.
- **Control de recursos**: para limitar el uso de CPU y memoria.

Vamos a entender cómo funcionan.

---

## 9.1. ¿Qué es un Namespace?

Un **Namespace** en Kubernetes es como una **carpeta virtual** donde agrupas objetos (Pods, Services, Deployments, etc.).

**¿Para qué sirve?**

- **Separar ambientes** (por ejemplo: desarrollo, pruebas, producción).
- **Gestionar permisos** de usuarios distintos.
- **Aplicar cuotas y límites** específicos a cada grupo de recursos.

**Importante:** Dentro de un mismo clúster, los Namespaces permiten crear "mini clústeres" aislados.

---

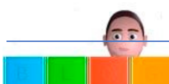
**Namespaces comunes en Kubernetes:**

- `default`: donde caen los recursos si no especificas Namespace.
- `kube-system`: componentes internos de Kubernetes.
- `kube-public`: datos públicos accesibles para todos.
- `kube-node-lease`: gestiona información de salud de los nodos.

---

**Crear un Namespace:**

```
bash
CopiarEditar
kubectl create namespace desarrollo
```



**Usar un Namespace al aplicar un recurso:**

```
bash
CopiarEditar
kubectl apply -f deployment.yaml --namespace=desarrollo
```

**O definirlo dentro del manifiesto YAML:**

```
yaml
CopiarEditar
metadata:
  name: mi-app
  namespace: desarrollo
```

---

## 9.2. ¿Qué es el control de recursos?

Kubernetes permite **definir límites** para controlar el uso de **CPU** y **memoria RAM** por contenedor.

Esto evita:

- Que una aplicación acapare todos los recursos.
- Que otras aplicaciones sufran por falta de CPU o RAM.

---

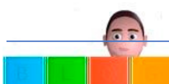
**Definir límites de recursos en un Pod:**

```
yaml
CopiarEditar
spec:
  containers:
  - name: app
    image: miapp:latest
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

**¿Qué significa esto?**

- **Requests**: lo que el contenedor **necesita** para arrancar (reservado).
- **Limits**: el **máximo** que puede usar.

`250m` CPU = 250 milicores = 0.25 de un core.



---

## 9.3. Cuotas de recursos en Namespaces

Puedes aplicar **cuotas** a un Namespace entero, limitando la cantidad de recursos que los objetos dentro pueden consumir.

**Ejemplo de ResourceQuota:**

```
yaml
CopiarEditar
apiVersion: v1
kind: ResourceQuota
metadata:
  name: cuota-desarrollo
  namespace: desarrollo
spec:
  hard:
    pods: "10"
    requests.cpu: "2"
    requests.memory: "4Gi"
    limits.cpu: "4"
    limits.memory: "8Gi"
```

- Limita el Namespace `desarrollo` a:
  - 10 Pods máximo.
  - 2 cores reservados.
  - 4 GB de RAM solicitada.
  - 4 cores y 8 GB de RAM como máximo absoluto.

---

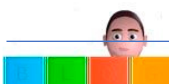
## 9.4. Beneficios de usar Namespaces y control de recursos

- **Organización:** Separas proyectos, ambientes o equipos.
- **Seguridad:** Permites o restringes acceso a distintos espacios.
- **Eficiencia:** Controlas cuánto recurso consume cada área o aplicación.
- **Escalabilidad:** Mantienes el clúster ordenado al crecer.

---

**Resumen:**

Namespaces agrupan y aíslan recursos; Requests, Limits y Quotas controlan el uso de CPU y memoria, garantizando un clúster eficiente y estable.



# 10. Despliegue de aplicaciones multi-contenedor

Hasta ahora hemos visto ejemplos donde **un Pod ejecuta un solo contenedor**.
Pero Kubernetes también permite tener **varios contenedores dentro de un mismo Pod**, trabajando juntos y compartiendo recursos.

Este enfoque se usa en situaciones donde **los contenedores son dependientes entre sí** y deben:

- Compartir red.
- Compartir almacenamiento.
- Coordinar su ciclo de vida.

---

## 10.1. ¿Por qué usar múltiples contenedores en un Pod?

Algunos casos comunes:

- **Sidecar:** Un contenedor complementa al principal (por ejemplo, para manejar logs, métricas o configuraciones).
- **Adapter:** Un contenedor convierte datos o tráfico antes de llegar al contenedor principal.
- **Ambassador:** Un contenedor actúa como proxy o traductor de red para el contenedor principal.

**Importante:** Los contenedores de un mismo Pod:

- Comparten la misma **IP** y **puertos de red**.
- Pueden compartir **volúmenes**.

---

## 10.2. Ejemplo de Pod multi-contenedor

Supongamos una aplicación donde:

- Un contenedor sirve la app web (Nginx).
- Otro contenedor recolecta los logs de acceso.

### Archivo `multi-pod.yaml`:

```
yaml
CopiarEditar
apiVersion: v1
kind: Pod
metadata:
  name: web-con-logs
spec:
```

```
  containers:
  - name: nginx-web
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - mountPath: /var/log/nginx
      name: logs
  - name: log-agent
    image: busybox
    command: ["sh", "-c", "tail -f /var/log/nginx/access.log"]
    volumeMounts:
    - mountPath: /var/log/nginx
      name: logs
  volumes:
  - name: logs
    emptyDir: {}
```

---

## 10.3. ¿Qué pasa aquí?

- Ambos contenedores comparten un volumen temporal (`emptyDir`).
- `nginx-web` escribe sus logs en `/var/log/nginx`.
- `log-agent` los lee continuamente usando `tail -f`.

Todo sucede **dentro del mismo Pod**, con recursos compartidos y sincronización natural.

---

## 10.4. Consideraciones importantes

- **Ciclo de vida conjunto:** Si el Pod se elimina, **todos** los contenedores desaparecen.
- **Recursos compartidos:** Uso compartido de red y volúmenes facilita la colaboración entre contenedores.
- **Responsabilidad separada:** Cada contenedor puede especializarse en una única tarea (principio de "Single Responsibility").

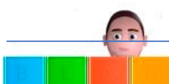
---

**Resumen:**

Kubernetes permite agrupar varios contenedores complementarios dentro de un mismo Pod, para que colaboren de manera eficiente y compartan red o almacenamiento.

---

Con esto **cerramos el Capítulo 3**: ¡ya sabes contenerizar, orquestar, exponer, almacenar y escalar aplicaciones en Kubernetes! 🚀



# Capítulo 4: Desplegando aplicaciones en Kubernetes

Hasta ahora has aprendido a crear Pods, Deployments, Servicios, ConfigMaps, Secrets y almacenamiento persistente en Kubernetes.
Conoces los componentes básicos para levantar aplicaciones simples.

**Pero en el mundo real**, las aplicaciones no se despliegan solas:

- Necesitan múltiples contenedores trabajando juntos (bases de datos, APIs, frontends).
- Requieren configuraciones dinámicas, seguridad, persistencia y escalabilidad.
- Deben actualizarse sin interrupciones y responder a variaciones de tráfico.

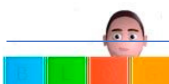
**En este capítulo**, vamos a dar el salto de ejemplos básicos a **desplegar aplicaciones más completas y estructuradas** en Kubernetes, paso a paso.

Aprenderás a:

- Crear despliegues de aplicaciones multi-contenedor.
- Administrar configuraciones de forma más profesional.
- Implementar escalado automático.
- Realizar actualizaciones controladas (rolling updates).
- Monitorizar el estado de tus aplicaciones.

El objetivo es que al finalizar este capítulo **puedas construir una aplicación real** capaz de correr de forma robusta en un clúster Kubernetes.

**¡Vamos a poner todo lo aprendido en acción!**



# 1. Despliegue de aplicaciones multi-contenedor completo

Ahora que sabes que un Pod puede contener varios contenedores trabajando juntos, vamos a desplegar una **aplicación real** que combine varios servicios esenciales en Kubernetes.

Imagina un escenario sencillo pero muy común:

- Un **frontend** (web) que sirve una aplicación web.
- Un **backend** (API) que maneja la lógica de negocio.
- Una **base de datos** para almacenar datos persistentes.

**Todo desplegado, coordinado y comunicado dentro de Kubernetes.**

---

## 1.1. Componentes que vamos a desplegar

| Componente | Tecnología | Descripción |
|---|---|---|
| Frontend | Nginx | Servirá archivos HTML estáticos. |
| Backend | Node.js API | Proveerá una pequeña API REST. |
| Base de datos | MySQL | Almacenará información persistente. |

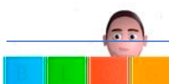
---

## 1.2. Estructura general

Tendremos:

- **3 Deployments**: uno para el frontend, otro para el backend, y otro para MySQL.
- **3 Services**: para permitir la comunicación entre los Deployments.
- **Un ConfigMap**: para inyectar configuraciones a la API.
- **Un Secret**: para gestionar la contraseña de la base de datos.
- **Un PVC**: para la persistencia de datos en MySQL.

---



## 1.3. Ejemplo de estructura de archivos

```
CopiarEditar
k8s/
├── frontend-deployment.yaml
├── backend-deployment.yaml
├── mysql-deployment.yaml
├── frontend-service.yaml
├── backend-service.yaml
├── mysql-service.yaml
├── configmap.yaml
├── secret.yaml
├── pvc.yaml
```

Cada archivo YAML definirá un componente del ecosistema de la aplicación.

---

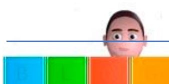
## 1.4. Principios que aplicaremos

- Cada componente será **independiente** y **escalable**.
- La comunicación entre servicios se hará a través de **nombres de servicio DNS** (`backend`, `mysql`).
- Guardaremos información sensible (contraseñas) en **Secrets**.
- Usaremos **volúmenes persistentes** para evitar pérdida de datos en MySQL.

---

**Resumen del objetivo:**

Al final de este punto, tendrás una aplicación funcional completa, desplegada en Kubernetes, siguiendo buenas prácticas de arquitectura de microservicios.



# 1.5. Desplegando el Backend (API REST)

Nuestro **backend** será una pequeña API en Node.js que simula un servicio real.

**Suposiciones:**

- Tenemos una imagen lista en Docker Hub (por ejemplo: `miusuario/backend-api:v1`).
- La API necesita:
  - Conectarse a MySQL.
  - Leer variables como `DB_HOST`, `DB_USER`, `DB_PASS` desde un ConfigMap y un Secret.

---

### Archivo `backend-deployment.yaml`

```
yaml
CopiarEditar
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: miusuario/backend-api:v1
        ports:
        - containerPort: 3000
        env:
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: DB_HOST
        - name: DB_USER
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: username
        - name: DB_PASS
          valueFrom:
            secretKeyRef:
              name: db-secret
```

```
              key: password
```

---

## Archivo `backend-service.yaml`

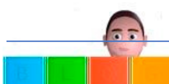
Este Service permitirá que otros componentes (como el frontend) contacten al backend.

```
yaml
CopiarEditar
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: backend
  ports:
  - protocol: TCP
    port: 3000
    targetPort: 3000
  type: ClusterIP
```

---

## Resumen de lo que hemos hecho:

- Creamos un Deployment de 2 réplicas del backend API.
- El backend obtendrá sus variables de configuración desde un ConfigMap y un Secret.
- Creamos un Service llamado `backend` accesible internamente dentro del clúster.



# 1.6. Desplegando la Base de Datos MySQL

---

### Archivo `pvc.yaml` (PersistentVolumeClaim)

Primero creamos el almacenamiento persistente para MySQL:

```
yaml
CopiarEditar
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

---

### Archivo `secret.yaml` (Credenciales de MySQL)

Creamos un Secret para guardar las credenciales codificadas en base64:

```
yaml
CopiarEditar
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: bXlzcWx1c3Vhcmlv  # "mysqlusuario" en base64
  password: bXlzcWxjbGF2ZQ==  # "mysqlclave" en base64
```

*(Puedes codificar tus valores usando `echo -n "valor" | base64`)*

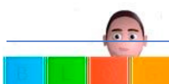
---

### Archivo `mysql-deployment.yaml`

Ahora desplegamos MySQL usando el PVC y el Secret:

```
yaml
CopiarEditar
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.7
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: password
        - name: MYSQL_USER
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: username
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: password
        - name: MYSQL_DATABASE
          value: appdb
        ports:
        - containerPort: 3306
        volumeMounts:
        - name: mysql-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-storage
        persistentVolumeClaim:
          claimName: mysql-pvc
```



### Archivo `mysql-service.yaml`

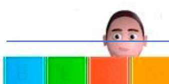
Creamos un Service para que otros servicios (como el backend) puedan acceder a MySQL:

```
yaml
CopiarEditar
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
```

---

## ¿Qué logramos aquí?

- La base de datos tendrá almacenamiento persistente (aunque el Pod caiga, los datos seguirán).
- Las credenciales estarán seguras en un Secret.
- El backend podrá comunicarse con MySQL usando el nombre DNS `mysql` y el puerto `3306`.



# 1.7. Desplegando el Frontend (servidor web)

El **Frontend** será una aplicación web sencilla servida usando **Nginx**.

---

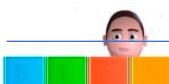
### Archivo `frontend-deployment.yaml`

```
yaml
CopiarEditar
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: nginx:stable
        ports:
        - containerPort: 80
```

- Usamos la imagen oficial de Nginx.
- Creamos **2 réplicas** para alta disponibilidad.

*(En un proyecto real, reemplazarías esta imagen por una personalizada que sirva tu aplicación HTML/JS compilada.)*

---



### Archivo `frontend-service.yaml`

Ahora creamos el Service que expondrá el frontend:

```
yaml
CopiarEditar
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  selector:
    app: frontend
  ports:
  - port: 80
    targetPort: 80
  type: NodePort
```

- El Service será de tipo **NodePort**, para que puedas acceder a la web desde tu navegador utilizando la IP del nodo y un puerto aleatorio entre 30000-32767.
- Kubernetes asignará automáticamente un puerto si no lo defines explícitamente.

---

## Acceso al Frontend

Para ver tu aplicación:

1. Listar los servicios y ver el puerto asignado:

```
bash
CopiarEditar
kubectl get services
```

Busca la línea del servicio `frontend`, por ejemplo:

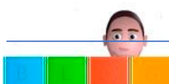
```
nginx
CopiarEditar
frontend    NodePort    10.96.0.10    <none>    80:31600/TCP    5m
```

2. Si estás usando Minikube, puedes hacer:

```
bash
CopiarEditar
minikube service frontend
```

¡Y se abrirá automáticamente en tu navegador! 🚀

---



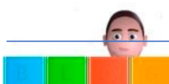
## ¿Qué hemos logrado hasta aquí?

- 2 réplicas de Nginx sirviendo el frontend.
- Un Service expuesto al exterior para acceder a la web.
- El frontend podrá comunicarse internamente con el backend si lo configuramos en la app (usando el nombre DNS `backend`).

---

**En resumen:**

Ahora tenemos un sistema completo: Frontend + Backend + Base de datos, desplegados, coordinados y expuestos correctamente en Kubernetes.



# 2. Helm: gestión de paquetes en Kubernetes

A medida que tu aplicación crece, mantener **muchos archivos YAML** individuales puede volverse **caótico y difícil de manejar**.
¿Te imaginas desplegar decenas de servicios, bases de datos y configuraciones a mano?

Aquí es donde entra **Helm**, el **gestor de paquetes de Kubernetes**.

---

## 2.1. ¿Qué es Helm?

Helm es una herramienta que te permite:

- **Empaquetar** todos tus recursos Kubernetes (YAMLs) en un "paquete" llamado **Chart**.
- **Instalar**, **actualizar** y **eliminar** aplicaciones enteras con **un solo comando**.
- **Configurar** tu aplicación fácilmente usando valores modificables.

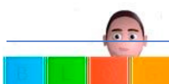
**Idea clave:**
Helm para Kubernetes es como `apt` para Linux o `npm` para Node.js.

---

## 2.2. Conceptos principales de Helm

| Concepto | Definición breve |
|---|---|
| **Chart** | Paquete de Kubernetes que incluye todos los manifiestos necesarios. |
| **Release** | Una instancia desplegada de un Chart en el clúster. |
| **Repository** | Lugar donde se almacenan Charts públicos o privados. |

---



## 2.3. Instalación de Helm

### En Linux/macOS:

```
bash
CopiarEditar
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 |
bash
```

### En Windows:

- Puedes instalar Helm usando `choco`:

```
bash
CopiarEditar
choco install kubernetes-helm
```

---

## 2.4. Crear tu propio Chart

1. Crear la estructura de un Chart:

```
bash
CopiarEditar
helm create mi-aplicacion
```

Esto genera una estructura como:

```
pgsql
CopiarEditar
mi-aplicacion/
├── charts/
├── templates/
│   ├── deployment.yaml
│   ├── service.yaml
│   └── _helpers.tpl
├── values.yaml
├── Chart.yaml
```

- `templates/`: donde colocas los manifiestos YAML que has estado usando.
- `values.yaml`: contiene las configuraciones que puedes personalizar (puertos, replicas, nombres de imagen, etc.).
- `Chart.yaml`: define el nombre, versión y descripción del Chart.

---

## 2.5. Instalar un Chart en tu clúster

Una vez tengas el Chart listo:

```
bash
CopiarEditar
helm install nombre-de-release ./mi-aplicacion
```

Ejemplo:

```
bash
CopiarEditar
helm install miapp ./mi-aplicacion
```

Helm:

- Lee todos los templates.
- Sustituye los valores.
- Despliega todos los recursos automáticamente.

---

## 2.6. Actualizar una Release

Si cambias algo en los archivos o `values.yaml`:

```
bash
CopiarEditar
helm upgrade nombre-de-release ./mi-aplicacion
```

Helm actualizará automáticamente solo los componentes necesarios.

---

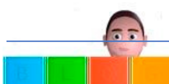
## 2.7. Eliminar una Release

Para borrar todo lo desplegado con Helm:

```
bash
CopiarEditar
helm uninstall nombre-de-release
```

Helm limpia automáticamente los Deployments, Services, PVCs, ConfigMaps, etc.

---



## 2.8. Helm Charts oficiales

Helm tiene un **repositorio público** lleno de Charts listos para usar:

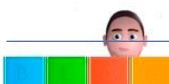
- Bases de datos (MySQL, PostgreSQL, MongoDB).
- Plataformas (WordPress, Jenkins, GitLab).
- Herramientas de monitorización (Prometheus, Grafana).

Puedes verlos aquí: https://artifacthub.io/

---

**Resumen:**

Helm simplifica enormemente la gestión de aplicaciones en Kubernetes, permitiéndote empaquetar, desplegar y actualizar todo tu sistema de manera ordenada y profesional.



# 3. Actualizaciones y Rollbacks

Una de las grandes ventajas de Kubernetes es su capacidad para realizar **actualizaciones controladas** y, si algo sale mal, **revertir fácilmente** a una versión anterior.

Esto es crucial para mantener tus aplicaciones disponibles sin tiempos de inactividad (downtime).

Vamos a ver cómo funciona.

---

## 3.1. Rolling Updates (actualizaciones progresivas)

Kubernetes realiza actualizaciones usando el patrón de **rolling update**:

- Actualiza **gradualmente** los Pods de un Deployment.
- No elimina todos los Pods antiguos de golpe.
- Va creando nuevos Pods con la nueva versión mientras elimina los viejos uno por uno.

**Ventaja:**
Tu aplicación siempre está disponible durante la actualización.

---

**Ejemplo de actualización:**

Supongamos que tienes un Deployment con 3 réplicas de tu backend versión `v1`.

Para actualizar la imagen:

```
bash
CopiarEditar
kubectl set image deployment/backend backend=miusuario/backend-api:v2
```

- Cambiamos el contenedor `backend` del Deployment `backend` a la nueva imagen `v2`.

**Kubernetes hará automáticamente:**

1. Crear un nuevo Pod con la imagen `v2`.
2. Verificar que esté saludable.
3. Eliminar un Pod viejo.
4. Repetir hasta actualizar todos los Pods.

---

### 3.2. Configurar la estrategia de Rolling Update

En tu `deployment.yaml`, puedes definir cómo de rápido o lento quieres que sea el update:

```
yaml
CopiarEditar
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
```

- `maxSurge: 1`: Puede haber **1 Pod adicional** mientras se actualiza.
- `maxUnavailable: 1`: Solo **1 Pod** puede estar fuera de servicio durante la actualización.

Así controlas el equilibrio entre **disponibilidad** y **velocidad**.

---

### 3.3. Ver el historial de revisiones

Cada vez que actualizas un Deployment, Kubernetes guarda una **revisión**.

Ver las revisiones:

```
bash
CopiarEditar
kubectl rollout history deployment/backend
```

Te mostrará la lista de cambios realizados.

---

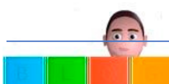
### 3.4. Hacer un Rollback (revertir a una versión anterior)

Si algo falla después de una actualización, puedes **volver a la versión anterior** fácilmente:

```
bash
CopiarEditar
kubectl rollout undo deployment/backend
```

Kubernetes revertirá el Deployment a la última versión estable automáticamente.

---



## 3.5. Verificar el estado del rollout

Para asegurarte de que la actualización (o rollback) va bien:

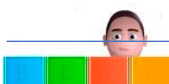
```
bash
CopiarEditar
kubectl rollout status deployment/backend
```

Esto te dirá si Kubernetes ya terminó de actualizar o si sigue en proceso.

---

**Resumen:**

Kubernetes permite hacer actualizaciones sin interrumpir el servicio (rolling updates) y ofrece un mecanismo fácil para revertir cambios problemáticos (rollback). Esto te da flexibilidad y seguridad en la gestión de aplicaciones.



# 4. Estrategias de escalado: Horizontal y Vertical

Uno de los superpoderes de Kubernetes es su capacidad de **escalar automáticamente** las aplicaciones:

- **Escalado Horizontal:** más copias (Pods) cuando sube la carga.
- **Escalado Vertical:** más CPU o memoria a los contenedores existentes.

Con el escalado, tu aplicación **responde dinámicamente** al tráfico, sin intervención manual.

---

## 4.1. Escalado Horizontal (Horizontal Pod Autoscaler - HPA)

**Escalar horizontalmente** significa **aumentar o disminuir** el número de Pods en función de la carga (normalmente CPU o uso de memoria).

**Crear un HPA básico:**

```
bash
CopiarEditar
kubectl autoscale deployment backend --cpu-percent=50 --min=2 --max=10
```

¿Qué hace esto?

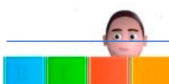
- Mantendrá el **uso de CPU** de cada Pod cerca del **50%**.
- **Mínimo 2 Pods**, **máximo 10 Pods**.

---

**¿Cómo funciona?**

- Kubernetes mide constantemente la carga.
- Si la CPU promedio de los Pods supera el 50%, **crea más Pods**.
- Si baja mucho, **reduce el número de Pods**.

---



### 4.2. Requisitos para HPA

- Tu clúster necesita tener habilitado el **metrics-server** (que recolecta métricas de CPU y RAM).

**Instalar metrics-server en Minikube:**

```
bash
CopiarEditar
minikube addons enable metrics-server
```

---

### 4.3. Escalado Manual

También puedes escalar manualmente tu Deployment:

```
bash
CopiarEditar
kubectl scale deployment backend --replicas=5
```

Esto forzará tener exactamente 5 Pods en ejecución.

---

### 4.4. Escalado Vertical (Vertical Pod Autoscaler - VPA)

**Escalar verticalmente** significa **ajustar los recursos** (CPU/RAM) de un Pod individual para adaptarse mejor a su carga.

⚠️ A diferencia del HPA, el escalado vertical normalmente **requiere reiniciar** el Pod para aplicar los nuevos recursos.

**Crear un VPA básico:**

```
yaml
CopiarEditar
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: backend-vpa
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: backend
  updatePolicy:
    updateMode: "Auto"
```

- `Auto`: Kubernetes ajustará automáticamente los recursos del Pod basado en su uso real.

---

## 4.5. ¿Qué estrategia elegir?

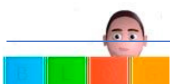
| Estrategia | Cuándo usar |
| --- | --- |
| **Escalado Horizontal (HPA)** | Cuando quieres manejar picos de tráfico aumentando el número de Pods. |
| **Escalado Vertical (VPA)** | Cuando un contenedor requiere más recursos, pero no más instancias. |

**Combinados**, te permiten construir aplicaciones altamente **resilientes** y **adaptativas**.

---

**Resumen:**

Kubernetes permite escalar horizontalmente (más Pods) y verticalmente (más recursos por Pod) de forma automática, asegurando que tus aplicaciones siempre respondan eficientemente a la demanda.



# 5. Health Checks: Liveness y Readiness Probes

Para que Kubernetes pueda **gestionar bien la vida de tus aplicaciones**, necesita saber:

- **¿Están funcionando correctamente?**
- **¿Están listas para recibir tráfico?**

Para esto se usan las **Probes**:

- **Liveness Probe:** ¿El contenedor sigue vivo?
- **Readiness Probe:** ¿El contenedor está listo para recibir tráfico?

**Idea clave:**
Kubernetes usa estas comprobaciones para **matar**, **reiniciar** o **aislar** Pods automáticamente si algo falla.

---

## 5.1. Liveness Probe (¿sigue vivo?)

La **Liveness Probe** detecta si un contenedor **está atascado** o **no responde**.

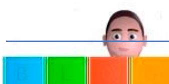
- Si la Liveness falla, **Kubernetes reinicia** el contenedor automáticamente.

**Ejemplo básico:**

```
yaml
CopiarEditar
livenessProbe:
  httpGet:
    path: /health
    port: 3000
  initialDelaySeconds: 5
  periodSeconds: 10
```

- Hace un GET a `/health` en el puerto 3000.
- Comienza 5 segundos después de arrancar.
- Repite cada 10 segundos.

---



## 5.2. Readiness Probe (¿está listo?)

La **Readiness Probe** indica si el contenedor **está listo** para recibir tráfico.

- Mientras falla la Readiness, **el Pod no recibe tráfico**.
- No se mata el contenedor, solo se retira del balanceador de carga.

**Ejemplo básico:**

```
yaml
CopiarEditar
readinessProbe:
  httpGet:
    path: /ready
    port: 3000
  initialDelaySeconds: 5
  periodSeconds: 5
```

---

## 5.3. Tipos de Probes

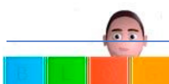
Puedes hacer las Probes de tres maneras:

- **HTTP GET:** Consulta un endpoint (`/health`, `/ready`, etc.).
- **TCP Socket:** Verifica si un puerto está abierto.
- **Exec Command:** Ejecuta un comando en el contenedor y espera código de salida `0`.

**Ejemplo de Exec Probe:**

```
yaml
CopiarEditar
livenessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

---



## 5.4. Beneficios de usar Probes

- **Mayor resiliencia:** Si una app se queda colgada, Kubernetes la reinicia.
- **Actualizaciones seguras:** Los Pods solo reciben tráfico cuando realmente están listos.
- **Despliegues más confiables:** Detectas problemas automáticamente antes de afectar a los usuarios.

---

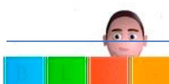
## 5.5. Buenas prácticas

- No pongas los Health Checks demasiado estrictos: tolera pequeños retrasos para evitar reinicios innecesarios.
- Usa diferentes endpoints para `readiness` y `liveness` si tiene sentido.
- Empieza con `initialDelaySeconds` generosos, sobre todo en aplicaciones que tardan en arrancar.

---

**Resumen:**

Las Liveness y Readiness Probes permiten a Kubernetes supervisar, recuperar y equilibrar el tráfico hacia los Pods, mejorando la disponibilidad y robustez de tus aplicaciones.



# 6. Logs, métricas y monitorización básica

Una vez que tus aplicaciones están corriendo en Kubernetes, necesitas **ver qué está pasando**:

- ¿Funcionan correctamente?
- ¿Hay errores?
- ¿Cómo está su consumo de recursos?

Sin visibilidad, estarías **"volando a ciegas"**.

Vamos a ver cómo obtener **logs**, **métricas** y **monitorear** tu clúster de forma básica.

---

## 6.1. Ver logs de los Pods

Cada contenedor escribe sus logs en la salida estándar (**stdout**) y error (**stderr**).
Kubernetes captura esos logs automáticamente.

**Comando básico para ver logs:**

```
bash
CopiarEditar
kubectl logs nombre-del-pod
```

**Ejemplo:**

```
bash
CopiarEditar
kubectl logs backend-5d6f7d9f9b-abcde
```

Si el Pod tiene varios contenedores, debes especificar el contenedor:

```
bash
CopiarEditar
kubectl logs nombre-del-pod -c nombre-del-contenedor
```

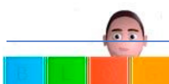
---

## 6.2. Seguir logs en tiempo real

Para "seguir" los logs en tiempo real (como `tail -f`):

```
bash
CopiarEditar
kubectl logs -f nombre-del-pod
```

---



## 6.3. Ver métricas básicas del clúster

Con el **metrics-server** instalado, puedes ver el uso de CPU y memoria:

```
bash
CopiarEditar
kubectl top nodes
```

Muestra estadísticas por nodo:

```
scss
CopiarEditar
NAME        CPU(cores)    MEMORY(bytes)
minikube    300m          1200Mi
```

---

Ver estadísticas por Pod:

```
bash
CopiarEditar
kubectl top pods
```

Así ves qué Pods están usando más recursos.

---

## 6.4. Monitorización ligera

Para un primer nivel de monitorización puedes usar:

- **kubectl describe pod nombre-del-pod**
  Verás eventos recientes, detalles de estado, errores, etc.
- **kubectl get events**
  Ver lista de eventos recientes del clúster (crashes, rescheduling, fallos de readiness, etc.).

---

## 6.5. Herramientas más avanzadas (más adelante)

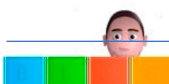
Cuando tu clúster crezca, puedes instalar soluciones de monitorización más robustas:

| Herramienta | ¿Para qué sirve? |
|---|---|
| Prometheus | Recopilar métricas del clúster y las aplicaciones. |
| Grafana | Visualizar gráficas y dashboards personalizados. |
| Loki | Centralizar y consultar logs. |
| ELK Stack | ElasticSearch + Logstash + Kibana para logs avanzados. |

🚀 **Prometheus + Grafana** es una de las combinaciones más populares para Kubernetes.

**Resumen:**

Kubernetes facilita la recolección de logs y métricas básicas desde el primer momento. Con prácticas sencillas puedes detectar errores, analizar el comportamiento y mejorar la disponibilidad de tus aplicaciones.



# Parte V: Kubernetes Avanzado

Hasta este punto ya manejas Kubernetes en su nivel esencial:

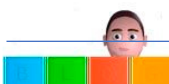
- Desplegar aplicaciones.
- Crear servicios.
- Gestionar volúmenes, configuraciones y seguridad básica.

**Ahora es momento de dar el siguiente salto: dominar Kubernetes en entornos reales y de alta demanda.**

En esta parte aprenderás:

- Cómo **gestionar el tráfico de entrada** de forma avanzada usando **Ingress Controllers**.
- Cómo **controlar accesos** de usuarios y aplicaciones a recursos del clúster con **RBAC**.
- Cómo aplicar **Políticas de red** para proteger la comunicación entre Pods.
- Qué son los **Operadores** de Kubernetes y cómo automatizan tareas complejas.
- Cómo desplegar Kubernetes en la **nube pública** usando **GKE (Google Cloud)**, **EKS (AWS)** y **AKS (Azure)**.
- Cómo integrar **Docker, Kubernetes y CI/CD** para automatizar todo el flujo de despliegue.

**Esta sección está diseñada para convertirte en un operador de Kubernetes capaz de diseñar, proteger, automatizar y escalar infraestructuras modernas a nivel profesional.**



# 1. Ingress Controllers y gestión de tráfico (Avanzado)

Cuando trabajas en entornos de producción, gestionar el tráfico de entrada a los servicios de tu clúster se convierte en un tema **crítico**:

- **Balanceo de carga**.
- **Seguridad TLS/SSL**.
- **Reglas de enrutamiento complejas**.
- **Políticas de red** para controlar qué puede entrar y qué no.

Los **Ingress Controllers** son esenciales para lograr esto de manera escalable, segura y flexible.

---

## 1.1. ¿Qué es un Ingress Controller?

Un **Ingress Controller** es un **proxy inverso especializado** que:

- Escucha las reglas de tráfico definidas en los recursos **Ingress** de Kubernetes.
- Se encarga de dirigir el tráfico externo al servicio interno adecuado.
- Puede gestionar certificados SSL/TLS, reescrituras de URL, redirecciones, autenticación, políticas de acceso, etc.

**Importante:** Kubernetes **no incluye** un Ingress Controller por defecto. Debes instalarlo o usar el que ofrezca tu proveedor de nube.

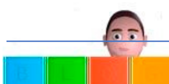
---

## 1.2. Principales Ingress Controllers

| Ingress Controller | Características principales |
|---|---|
| **NGINX Ingress** | Popular, muy flexible, soporta TLS, redirecciones, autenticación. |
| **Traefik** | Ligero, dinámico, ideal para microservicios y auto-discovery. |
| **HAProxy Ingress** | Alto rendimiento, muy configurable a bajo nivel. |
| **Istio Gateway** | Basado en malla de servicios, tráfico ultra avanzado. |

En la mayoría de los casos de producción inicial, **NGINX** o **Traefik** son las mejores opciones.

---



## 1.3. Arquitectura de tráfico con Ingress

```
plaintext
CopiarEditar
[Usuario externo]
    ↓ (petición HTTP/HTTPS)
[Ingress Controller]
    ↓ (aplicando reglas)
[Service de Kubernetes]
    ↓
[Pod correspondiente]
```

---

## 1.4. Funcionalidades avanzadas que maneja un Ingress Controller

- **TLS/SSL termination**: Gestiona certificados para tráfico seguro HTTPS.
- **URL Path routing**: Ruta peticiones según la ruta de la URL (`/api`, `/web`, `/admin`, etc.).
- **Name-based virtual hosting**: Soporta múltiples dominios (`app1.miempresa.com`, `app2.miempresa.com`).
- **Rate Limiting**: Limita el número de peticiones por segundo.
- **Autenticación externa**: Integración con OAuth2, OpenID Connect, LDAP.
- **Balanceo de carga**: Distribuye tráfico entre múltiples Pods de un mismo servicio.
- **Políticas de reintento y timeout**: Mejor manejo de fallos en la red.

---

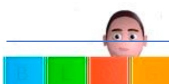
## 1.5. Instalación típica de un Ingress Controller (ejemplo con NGINX)

```
bash
CopiarEditar
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-
nginx/main/deploy/static/provider/cloud/deploy.yaml
```

Esto despliega:

- Un Deployment del Ingress Controller.
- Un Service de tipo LoadBalancer (si estás en la nube) o NodePort.

---

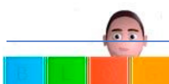


# 1.6. Ejemplo de recurso Ingress avanzado

```
yaml
CopiarEditar
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
  tls:
  - hosts:
    - app.miempresa.com
    secretName: app-tls-secret
  rules:
  - host: app.miempresa.com
    http:
      paths:
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: backend-service
            port:
              number: 3000
      - path: /web
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
```

- Gestiona tráfico HTTPS con certificado TLS.
- Divide rutas `/api` y `/web` hacia servicios diferentes.

---



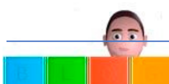
## 1.7. Buenas prácticas de Ingress en producción

- Siempre usa **TLS/SSL**.
- Usa **autenticación** para rutas sensibles.
- Configura **timeout** y **rate limits** para proteger de abusos.
- Implementa **redirects automáticos** de HTTP a HTTPS.
- Monitorea el estado del Ingress Controller (CPU, memoria, latencia).

---

**Resumen:**

En entornos avanzados de Kubernetes, el Ingress Controller se convierte en la pieza clave para gestionar todo el tráfico externo de forma segura, flexible y eficiente.



# 2. RBAC: Control de acceso basado en roles (Avanzado)

En clústeres de producción, no basta con tener acceso limitado:
hay que construir **políticas de seguridad de acceso bien diseñadas**, basadas en **roles mínimos**, **separación de responsabilidades** y **auditoría de permisos**.

Aquí veremos cómo aplicar **RBAC real** en Kubernetes de forma **efectiva y segura**.

---

## 2.1. Recordatorio: ¿Qué es RBAC?

- RBAC (Role-Based Access Control) permite **autorizar acciones** sobre los recursos del clúster.
- Define **quién** puede realizar **qué acción** sobre **qué recurso** en **qué espacio (Namespace)**.

---

## 2.2. Diseño avanzado de roles en Kubernetes

**1. Principio de mínimo privilegio:**

- Un usuario o servicio debe tener **solo** los permisos estrictamente necesarios.

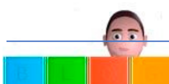
**2. Separación por tipo de usuario:**

- **Usuarios humanos**: operaciones de administración, lectura o despliegue.
- **Service Accounts**: automatización y tareas de aplicaciones.

**3. Roles específicos por Namespace:**

- No otorgues ClusterRoles si no es absolutamente necesario.
- Un proyecto = un Namespace = un conjunto de Roles dedicados.

---



## 2.3. Crear un Role avanzado (lectura + actualización de Pods)

**Ejemplo `role-pods.yaml`:**

```
yaml
CopiarEditar
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: gestor-pods
  namespace: produccion
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch", "patch", "update"]
```

Permite:

- Ver Pods.
- Listarlos.
- Observar eventos.
- Actualizar o modificar Pods.

---

## 2.4. Crear un RoleBinding

**Ejemplo `rolebinding-pods.yaml`:**

```
yaml
CopiarEditar
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-gestor-pods
  namespace: produccion
subjects:
- kind: User
  name: administrador-apps
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: gestor-pods
  apiGroup: rbac.authorization.k8s.io
```

Asocia el **Role** `gestor-pods` al **usuario** `administrador-apps` en el Namespace `produccion`.

---

## 2.5. ClusterRole y ClusterRoleBinding: acceso global

Si un usuario necesita actuar **sobre todo el clúster** (por ejemplo, ver todos los Nodes o StorageClasses), usa **ClusterRole**.

**Ejemplo de ClusterRole:**

```
yaml
CopiarEditar
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: lector-global
rules:
- apiGroups: [""]
  resources: ["nodes", "namespaces"]
  verbs: ["get", "list", "watch"]
```

Luego enlazas con un ClusterRoleBinding.

---

## 2.6. Escenario real: Automatización segura con Service Accounts

Supón que un sistema CI/CD necesita desplegar aplicaciones automáticamente.

Pasos:

- Creas una **ServiceAccount** dedicada.
- Le asocias un **RoleBinding** o **ClusterRoleBinding** mínimo.
- Limita su acceso solo a `deployments`, `pods`, `services`, nada más.

**Ejemplo ServiceAccount:**

```
yaml
CopiarEditar
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-cicd
  namespace: staging
```

**Y su RoleBinding específico.**

---

## 2.7. Comprobar permisos

Ver qué puede hacer un usuario o ServiceAccount:

```
bash
CopiarEditar
kubectl auth can-i create pods --as=usuario --namespace=produccion
```

Consulta muy útil para validar permisos **sin probar a ciegas**.

---

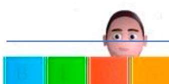
## 2.8. Buenas prácticas de RBAC avanzado

- **Automatiza la creación de Roles** para nuevos proyectos.
- **Revoca permisos** que ya no sean necesarios.
- **Audita** los permisos existentes periódicamente.
- **Documenta** los Roles creados y su propósito.
- **Divide roles entre lectura (readonly)** y **escritura (admin)**.

---

**Resumen:**

En Kubernetes avanzado, RBAC es la primera línea de defensa. Bien aplicado, garantiza que cada usuario y servicio opere dentro de sus límites, fortaleciendo la seguridad y organización del clúster.



# 3. Políticas de red

En un clúster Kubernetes sin configuración extra, **cualquier Pod puede comunicarse con cualquier otro Pod**, en cualquier Namespace.
Esto **no es seguro** en entornos de producción.

Con **Network Policies** podemos **controlar qué Pods pueden comunicarse entre sí**, y **restringir** el tráfico entrante o saliente.

**Idea clave:**
Las Network Policies son las **firewalls internas** de Kubernetes.

---

## 3.1. ¿Qué es una Network Policy?

Una **NetworkPolicy** define **reglas de tráfico**:

- Desde qué Pods se puede **recibir** tráfico (**Ingress**).
- Hacia qué Pods o direcciones se puede **enviar** tráfico (**Egress**).

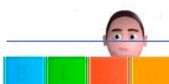
Estas reglas se aplican a nivel de **Pods** y **Namespaces**, no de nodos.

---

## 3.2. Requisitos

- Tu clúster necesita un **plugin de red** que soporte Network Policies.
  Ejemplos: **Calico**, **Cilium**, **Weave Net**.
- Si no hay un plugin compatible, las políticas **no tendrán efecto**.

---



## 3.3. Ejemplo de NetworkPolicy básica

**Política para permitir solo tráfico interno desde Pods con una etiqueta específica.**

```
yaml
CopiarEditar
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: permitir-desde-backend
  namespace: produccion
spec:
  podSelector:
    matchLabels:
      app: mysql
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: backend
```

**¿Qué hace esta política?**

- Solo los Pods etiquetados con `app=backend` pueden acceder a los Pods `app=mysql`.
- Todo el resto del tráfico **es bloqueado**.

## 3.4. Ejemplo de política de Egress

**Restringir que un Pod solo pueda comunicarse hacia Internet a través del puerto 443 (HTTPS):**

```
yaml
CopiarEditar
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: limitar-egress
  namespace: produccion
spec:
  podSelector:
    matchLabels:
      app: api
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 0.0.0.0/0
    ports:
    - protocol: TCP
      port: 443
```

- Los Pods `app=api` **solo pueden hacer conexiones HTTPS** hacia el exterior.

---

## 3.5. ¿Cómo funcionan las políticas?

- **Sin política:** Todos los Pods pueden comunicarse libremente.
- **Con al menos una política Ingress/Egress:**
  - Solo el tráfico explícitamente permitido está permitido.
  - Todo lo que no está definido queda **bloqueado**.

---

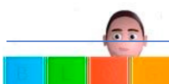
## 3.6. Buenas prácticas de Network Policies

- Aplica políticas **desde el inicio** del proyecto, no como corrección después.
- **Define comunicaciones explícitas**: solo lo necesario.
- Usa **Namespaces separados** para ambientes (dev, prod) con reglas específicas.
- **Combina Ingress y Egress** para máximo control.
- Documenta y revisa las políticas periódicamente.

---

**Resumen:**

Las Network Policies son la herramienta clave para controlar la comunicación entre Pods en Kubernetes, mejorando la seguridad interna y reduciendo el riesgo de movimientos laterales en el clúster.



# 4. Operadores en Kubernetes

Cuando manejas aplicaciones o servicios complejos (bases de datos, sistemas distribuidos, aplicaciones de negocio), no basta con crear Pods y Deployments:

- Hay que gestionar actualizaciones.
- Recuperar fallos automáticamente.
- Escalar con lógica personalizada.
- Realizar backups, migraciones, configuraciones especiales.

Para automatizar todo eso **de manera nativa en Kubernetes**, existen los **Operadores**.

**Idea clave:**
Un **Operador** es como un **administrador humano** programado en Kubernetes.

---

## 4.1. ¿Qué es un Operador?

Un **Operador** es una **aplicación especializada** que:

- Extiende Kubernetes añadiendo **nuevos recursos** (CRDs: Custom Resource Definitions).
- Contiene **lógica de gestión** específica para una aplicación o sistema.
- Automatiza tareas complejas que normalmente haría un administrador manualmente.

---

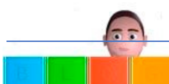
## 4.2. Componentes básicos de un Operador

| Componente | Función |
|---|---|
| **CRD (Custom Resource Definition)** | Define un nuevo tipo de recurso en Kubernetes. |
| **Controller** | Observa cambios en los CRDs y ejecuta acciones para mantener el estado deseado. |

Ejemplo:

- CRD: `MySQLCluster`
- Operador: Crea, escala, realiza backups de clústeres MySQL automáticamente.

---



## 4.3. ¿Qué tareas puede hacer un Operador?

- Desplegar aplicaciones complejas (PostgreSQL, Redis, Cassandra, Kafka, etc.).
- Gestionar upgrades automáticos.
- Monitorear el estado de los servicios.
- Aplicar parches de seguridad.
- Gestionar backups, restauraciones, failover.

Todo eso **dentro** del flujo normal de Kubernetes, como si fueran recursos nativos.

---

## 4.4. Ejemplos de Operadores reales

| Operador | ¿Qué gestiona? |
|---|---|
| **Prometheus Operator** | Despliega y gestiona clústeres de Prometheus. |
| **MongoDB Community Operator** | Administra bases de datos MongoDB en Kubernetes. |
| **ElasticSearch Operator (ECK)** | Automatiza ElasticSearch en Kubernetes. |
| **Postgres Operator (Crunchy Data)** | Gestión completa de PostgreSQL clusters. |

---

## 4.5. Beneficios de usar Operadores

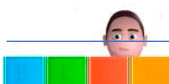
- **Automatización total:** Menos errores humanos.
- **Consistencia:** Despliegues y configuraciones estandarizadas.
- **Escalabilidad:** Respuesta automática a la demanda o fallos.
- **Extendibilidad:** Añades lógica personalizada adaptada a tu entorno.

---

## 4.6. ¿Cómo se crean Operadores?

- Lenguajes más comunes: **Go**, **Python**, **Java**.
- Herramientas como **Operator SDK** y **Kubebuilder** facilitan la creación de Operadores.
- Puedes programar desde pequeños controladores personalizados hasta Operadores empresariales completos.

---



## Flujo básico de un Operador:

```
plaintext
CopiarEditar
Usuario crea un objeto personalizado (CRD) ➔
El Controller lo detecta ➔
Aplica la lógica necesaria ➔
Actualiza el estado del clúster según el deseo del usuario
```

---

### Resumen:

Los Operadores en Kubernetes permiten automatizar la gestión completa de aplicaciones complejas, extendiendo las capacidades del clúster de manera nativa, fiable y escalable.

# 5. Kubernetes en la nube: GKE, EKS y AKS

Aunque puedes montar tu propio clúster Kubernetes manualmente, en producción suele ser **mejor y más eficiente** usar una **solución administrada** por un proveedor de nube.

Tres grandes plataformas ofrecen Kubernetes como servicio:

- **GKE** (Google Kubernetes Engine)
- **EKS** (Elastic Kubernetes Service, Amazon AWS)
- **AKS** (Azure Kubernetes Service)

Estas soluciones **se encargan del mantenimiento** del clúster:

- Provisionan automáticamente nodos.
- Actualizan versiones.
- Gestionan escalado y alta disponibilidad.
- Proveen integraciones de red, almacenamiento, monitoreo y seguridad.

---

## 5.1. GKE (Google Kubernetes Engine)

**Ventajas:**

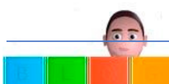
- Kubernetes fue creado en Google, por lo que GKE ofrece la **implementación más nativa y optimizada**.
- Integración profunda con servicios de Google Cloud (IAM, Stackdriver, VPCs).
- Actualizaciones automáticas opcionales.

**Comando típico de creación:**

```
bash
CopiarEditar
gcloud container clusters create mi-cluster --zone us-central1-a
```

- Puedes integrar Identity-Aware Proxy para autenticaciones más seguras.
- Fácil de gestionar con **gcloud CLI** y la **Google Cloud Console**.

---



## 5.2. EKS (Elastic Kubernetes Service)

**Ventajas:**

- Kubernetes administrado sobre la infraestructura de AWS.
- Muy buena integración con servicios como IAM, ELB, EBS, CloudWatch.
- Compatible con **Fargate** (serverless para pods individuales).

**Comando típico usando eksctl:**

```
bash
CopiarEditar
eksctl create cluster --name mi-cluster --region us-west-2
```

- Con soporte completo para escalado automático de nodos y balanceadores de carga de AWS.

---

## 5.3. AKS (Azure Kubernetes Service)

**Ventajas:**

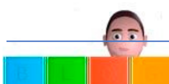
- Kubernetes integrado en Azure.
- Fuerte conexión con Active Directory y servicios de seguridad de Azure.
- Facilita el despliegue de soluciones Microsoft (.NET, SQL Server) en contenedores.

**Comando típico usando Azure CLI:**

```
bash
CopiarEditar
az aks create --resource-group mi-grupo --name mi-cluster --node-count 3 --
enable-addons monitoring --generate-ssh-keys
```

- Buen soporte de CI/CD automático con GitHub Actions y Azure DevOps.

---



## 5.4. Comparativa rápida

| Característica | GKE | EKS | AKS |
|---|---|---|---|
| Actualizaciones automáticas | Sí (muy sencillo) | Parcialmente manual | Sí |
| Integración con servicios | Fuerte con GCP | Fuerte con AWS | Fuerte con Azure |
| Facilidad de inicio | Muy alta | Requiere más configuración inicial | Alta |
| Precio | Ajustado, pagarás más por comodidad | Puede ser caro según nodos y tráfico | Bastante competitivo |

---

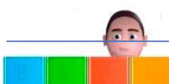
## 5.5. ¿Por qué elegir un Kubernetes administrado?

- **Ahorras tiempo**: no debes mantener nodos maestros ni actualizar manualmente el clúster.
- **Reduces errores humanos**: los proveedores garantizan alta disponibilidad.
- **Seguridad y compliance**: integración sencilla con sistemas de seguridad nativos.
- **Escalabilidad instantánea**: pasa de 3 a 300 nodos en minutos.

---

**Resumen:**

Usar Kubernetes en la nube (GKE, EKS o AKS) te libera del mantenimiento del clúster, te da escalabilidad automática y permite concentrarte en desplegar y evolucionar tus aplicaciones.



# 6. CI/CD con Docker y Kubernetes

Hoy en día, desplegar una aplicación **a mano** cada vez que hay cambios **no es viable**.
Por eso, la mayoría de las empresas implementan **CI/CD** (**Integración Continua / Entrega Continua**) para automatizar:

- Construcción de imágenes Docker.
- Testeo automático del código.
- Despliegue automático a Kubernetes.

**Idea clave:**
CI/CD conecta el desarrollo con la producción de forma fluida, segura y rápida.

---

## 6.1. ¿Qué es CI/CD?

- **CI (Continuous Integration):**
  - Cada vez que subes código (push/pull request), se ejecutan tests automáticos.
  - Se genera una nueva imagen Docker automáticamente.
- **CD (Continuous Delivery/Deployment):**
  - Se despliega automáticamente la nueva imagen en el clúster Kubernetes.
  - Puede ser automático (Deployment) o manual (Delivery).

---

## 6.2. Flujo típico de CI/CD con Kubernetes

```
plaintext
CopiarEditar
Desarrollador ➔ Push a GitHub ➔ Pipeline CI/CD ➔
Construir imagen Docker ➔ Push a Docker Registry ➔
Actualizar Deployment en Kubernetes ➔ Nueva versión online
```

---

## 6.3. Herramientas comunes para CI/CD

| Herramienta | Rol principal |
|---|---|
| GitHub Actions | Pipelines de CI/CD integrados en GitHub. |
| GitLab CI | Pipelines integrados en GitLab. |
| Jenkins | Plataforma extensible para CI/CD. |
| ArgoCD | Automatiza despliegues GitOps en Kubernetes. |

---

# 6.4. Ejemplo básico: Pipeline con GitHub Actions

**Archivo `.github/workflows/deploy.yaml`:**

```
yaml
CopiarEditar
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Login to DockerHub
      run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u ${{
secrets.DOCKER_USERNAME }} --password-stdin

    - name: Build and push Docker image
      run: |
        docker build -t miusuario/miapp:${{ github.sha }} .
        docker push miusuario/miapp:${{ github.sha }}

    - name: Set up kubectl
      uses: azure/setup-kubectl@v3

    - name: Update Kubernetes deployment
      run: |
        kubectl set image deployment/mi-deployment miapp=miusuario/miapp:${{
github.sha }}
```

---

### ¿Qué hace este flujo?

- Cada push a `main`:
  - Compila una nueva imagen Docker.
  - La sube a DockerHub.
  - Actualiza automáticamente el Deployment de Kubernetes con la nueva imagen.

---

## 6.5. Buenas prácticas de CI/CD con Kubernetes

- **Versiona imágenes** con tags únicos (`latest` no es seguro).
- **Automatiza pruebas** unitarias antes de construir imágenes.
- **Usa Secrets seguros** para contraseñas, tokens de acceso, etc.
- **Haz rollout controlados**: despliega nuevas versiones gradualmente.
- **Implementa notificaciones** (Slack, Teams, correo) en caso de error.

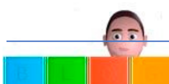
---

**Resumen:**

Integrar Docker y Kubernetes en un flujo CI/CD profesional acelera el desarrollo, mejora la calidad del software y permite entregas más frecuentes, confiables y seguras.

---

🚀 ¡Con esto terminamos toda la **Parte V: Kubernetes Avanzado**!



# Parte VI: Buenas Prácticas y Casos Reales

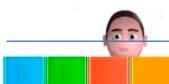
Tras recorrer todo el viaje desde los conceptos básicos hasta la operación avanzada de Kubernetes,
llega el momento de **consolidar** el conocimiento con **buenas prácticas**, **casos prácticos reales** y una mirada hacia **el futuro**.

En esta última parte del libro aprenderás a:

- **Diseñar aplicaciones nativas de Kubernetes** desde el principio, sacando el máximo partido de la plataforma.
- **Evitar errores comunes** que pueden afectar rendimiento, seguridad y estabilidad de los clústeres.
- **Desplegar una aplicación web completa** en Kubernetes siguiendo un flujo profesional, paso a paso.
- **Optimizar clústeres** para mejorar eficiencia y **reducir costes** en la nube o en entornos propios.
- Entender hacia dónde se mueve la industria con conceptos como **Serverless**, **Kubernetes para inteligencia artificial**, y **nuevas tendencias cloud-native**.

**Objetivo final:**
Que no solo sepas usar Kubernetes, sino que seas capaz de **diseñar soluciones completas**, **operarlas eficientemente** y **adaptarte a su evolución** en el futuro.

**¡Vamos a por el cierre perfecto de este viaje de aprendizaje!** 🚀 🌟



# 1. Diseño de aplicaciones nativas de Kubernetes

Kubernetes no es solo un lugar donde "empaquetas lo que ya tienes".
Para aprovechar todo su potencial, debes **diseñar aplicaciones específicamente pensadas para ejecutarse en Kubernetes**.

**Idea clave:**
Una aplicación "cloud-native" está diseñada para ser **resiliente, escalable y modular** desde su nacimiento.

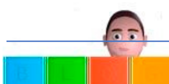
Vamos a ver cómo hacerlo.

---

## 1.1. Características de una aplicación nativa de Kubernetes

- **Contenerización:**
  Cada componente de la aplicación corre dentro de su propio contenedor.
- **Microservicios:**
  La aplicación se divide en pequeños servicios independientes que se comunican por APIs.
- **Configuraciones externas:**
  Toda la configuración (variables, credenciales) vive fuera del código, usando **ConfigMaps** y **Secrets**.
- **Escalabilidad horizontal:**
  Los servicios están preparados para escalar mediante **réplicas** de Pods.
- **Stateless o separación de estado:**
  Los servicios deben ser preferiblemente **stateless** (sin almacenar información interna), o guardar el estado fuera (bases de datos, volúmenes).
- **Auto-recuperación:**
  Las aplicaciones deben ser tolerantes a fallos, apoyándose en **liveness** y **readiness probes**.

---



## 1.2. Principios de diseño para Kubernetes

| Principio | ¿Qué significa? |
|---|---|
| 12 Factor App | Sigue las mejores prácticas de aplicaciones cloud-native. |
| Desacoplamiento | Cada servicio debe ser lo más independiente posible. |
| Tolerancia a fallos | No asumir que un Pod, nodo o red siempre estará disponible. |
| Escalabilidad dinámica | Tu app debe poder escalar sin cambios manuales. |
| Despliegues continuos | Ser capaz de actualizar versiones sin downtime. |

---

## 1.3. Patrones de diseño comunes en Kubernetes

- **Sidecar:**
  Un contenedor auxiliar que complementa la funcionalidad de otro contenedor principal (por ejemplo, para logs o métricas).
- **Adapter:**
  Convierte la entrada o salida de un servicio para hacerlo compatible con otros.
- **Ambassador:**
  Gestiona la comunicación entre el servicio y el mundo exterior (proxy o gateway).
- **Leader Election:**
  Para servicios que requieren un único líder activo entre múltiples réplicas.

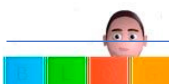
---

## 1.4. Ejemplo práctico: arquitectura básica para una app web

```
plaintext
CopiarEditar
Frontend (Nginx, React) ➔ Backend API (Node.js, Express) ➔ Base de datos
(PostgreSQL)
```

- Cada componente en su propio Deployment.
- Configuración sensible almacenada en Secrets.
- Comunicación interna mediante Services.
- Entrada única mediante Ingress.
- Almacenamiento persistente para la base de datos mediante PVCs.
- Autoscaling habilitado en backend y frontend.

---



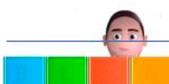
## 1.5. Beneficios de diseñar bien para Kubernetes

- **Mayor resiliencia**: la aplicación puede recuperarse sola de errores.
- **Mayor escalabilidad**: adapta los recursos a la demanda real.
- **Facilidad de actualizaciones**: rolling updates y rollbacks sencillos.
- **Mejor experiencia de usuario**: menos downtime, mejor rendimiento.
- **Facilidad de operación**: monitorización, backups y recuperación simplificados.

---

**Resumen:**

Diseñar aplicaciones pensando en Kubernetes desde el principio no solo mejora su rendimiento, sino que también maximiza su disponibilidad, escalabilidad y mantenimiento futuro.



# 2. Errores comunes y cómo evitarlos

Aprender de los errores propios es valioso, pero aprender de los errores **de otros** es aún mejor. En Kubernetes, muchos fallos **se repiten** porque no se aplican buenas prácticas desde el inicio.

Vamos a ver los **errores más comunes** y, lo más importante, **cómo evitarlos**.

---

## 2.1. No establecer límites de recursos

**Error:**

- No definir `requests` y `limits` de CPU y memoria en los contenedores.

**Consecuencia:**

- Un Pod puede consumir todos los recursos del nodo y afectar a otros servicios.

**Solución:**

- Siempre define `resources` en tus Pods.

```
yaml
CopiarEditar
resources:
  requests:
    cpu: "100m"
    memory: "128Mi"
  limits:
    cpu: "500m"
    memory: "512Mi"
```

---

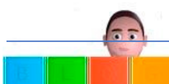
## 2.2. No usar Liveness y Readiness Probes

**Error:**

- No configurar health checks en los contenedores.

**Consecuencia:**

- Kubernetes no detecta cuándo un contenedor está colgado o no está listo, afectando disponibilidad.



**Solución:**

- Implementa siempre `livenessProbe` y `readinessProbe`.

---

## 2.3. No externalizar configuraciones sensibles

**Error:**

- Hardcodear contraseñas, APIs o configuraciones dentro del contenedor o el código.

**Consecuencia:**

- Pérdida de seguridad, dificultad para modificar configuraciones dinámicamente.

**Solución:**

- Usa **ConfigMaps** para configuraciones normales y **Secrets** para datos sensibles.

---

## 2.4. Escalar solo verticalmente

**Error:**

- Pensar que aumentando CPU o RAM de un Pod es suficiente.

**Consecuencia:**

- No aprovechas la elasticidad real de Kubernetes.

**Solución:**

- Diseña tus aplicaciones para escalar **horizontalmente** (más réplicas).

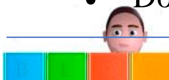
## 2.5. No controlar la política de actualización

**Error:**

- Hacer despliegues sin controlar `rollingUpdate`, causando caídas o inconsistencias.

**Consecuencia:**

- Downtime inesperado.



**Solución:**

- Define estrategias claras en los Deployments.

```
yaml
CopiarEditar
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 1
```

## 2.6. Exponer todos los servicios innecesariamente

**Error:**

- Usar NodePort o LoadBalancer para todo.

**Consecuencia:**

- Incrementa los riesgos de seguridad y el coste.

**Solución:**

- Usa **ClusterIP** para servicios internos.
- **Ingress** para exponer varios servicios detrás de una única entrada.

## 2.7. Ignorar la actualización del clúster

**Error:**

- Dejar pasar años sin actualizar versiones de Kubernetes o de los componentes.

**Consecuencia:**

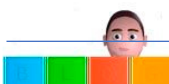
- Vulnerabilidades de seguridad, falta de soporte, incompatibilidades.

**Solución:**

- Planifica un ciclo regular de **actualización** de clúster y aplicaciones.

---

**Resumen:**



Evitar estos errores comunes mejora la disponibilidad, seguridad y eficiencia de tus aplicaciones Kubernetes, y te posiciona como un operador de plataformas de primer nivel.

# 3. Caso práctico: Desplegando una aplicación web completa

Ahora vamos a poner en práctica **todo lo aprendido** a lo largo del libro, desplegando una **aplicación web completa** en Kubernetes, siguiendo buenas prácticas reales.

---

## 3.1. Arquitectura de la aplicación

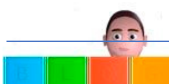
- **Frontend:**
  Aplicación web React servida con Nginx.
- **Backend:**
  API REST desarrollada en Node.js + Express.
- **Base de datos:**
  PostgreSQL persistente.
- **Otros componentes:**
  - Configuraciones externas (ConfigMaps y Secrets).
  - Ingress Controller para gestionar tráfico.
  - Escalado automático (HPA).
  - Monitorización básica de recursos.

---

## 3.2. Estructura de recursos Kubernetes

```
plaintext
CopiarEditar
k8s/
├── frontend-deployment.yaml
├── backend-deployment.yaml
├── postgres-deployment.yaml
├── frontend-service.yaml
├── backend-service.yaml
├── postgres-service.yaml
├── configmap.yaml
├── secret.yaml
├── pvc.yaml
├── ingress.yaml
├── hpa.yaml
```

---



## 3.3. Paso 1: Configurar el almacenamiento de la base de datos

### Archivo `pvc.yaml`:

```
yaml
CopiarEditar
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

---

## 3.4. Paso 2: Crear Secrets y ConfigMaps

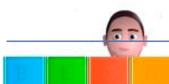
### Archivo `secret.yaml`:

```
yaml
CopiarEditar
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  username: cG9zdGdyZXM=   # "postgres" en base64
  password: c2VjdXJv     # "securo" en base64
```

### Archivo `configmap.yaml`:

```
yaml
CopiarEditar
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  DATABASE_URL: "postgres://postgres:securo@postgres-service:5432/appdb"
```

---



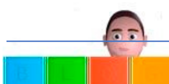
## 3.5. Paso 3: Desplegar PostgreSQL

### Archivo `postgres-deployment.yaml`:

yaml
CopiarEditar

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:13
        envFrom:
        - secretRef:
            name: db-credentials
        ports:
        - containerPort: 5432
        volumeMounts:
        - mountPath: /var/lib/postgresql/data
          name: postgres-storage
      volumes:
      - name: postgres-storage
        persistentVolumeClaim:
          claimName: postgres-pvc
```

---



## 3.6. Paso 4: Desplegar Backend y Frontend

**Backend `backend-deployment.yaml`:**

```
yaml
CopiarEditar
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: miusuario/backend:latest
        envFrom:
        - configMapRef:
            name: app-config
        ports:
        - containerPort: 3000
```

**Frontend `frontend-deployment.yaml`:**

```
yaml
CopiarEditar
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: miusuario/frontend:latest
        ports:
        - containerPort: 80
```

## 3.7. Paso 5: Crear Servicios y el Ingress

### Archivo `frontend-service.yaml`:

```
yaml
CopiarEditar
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  ports:
  - port: 80
    targetPort: 80
  type: ClusterIP
```

### Archivo `backend-service.yaml`:

```
yaml
CopiarEditar
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
  - port: 3000
    targetPort: 3000
  type: ClusterIP
```

### Archivo `postgres-service.yaml`:

```
yaml
CopiarEditar
apiVersion: v1
kind: Service
metadata:
  name: postgres-service
spec:
  selector:
    app: postgres
  ports:
  - port: 5432
    targetPort: 5432
  type: ClusterIP
```

### Archivo `ingress.yaml`:

```
yaml
```

```
CopiarEditar
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: app.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
```

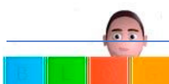
---

# 3.8. Paso 6: Configurar el HPA

### Archivo `hpa.yaml`:

```
yaml
CopiarEditar
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: backend-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: backend
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

---



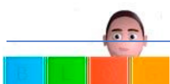
**Resultado final:**

- Una aplicación web completa, desplegada con buenas prácticas.
- Configuración externa.
- Escalabilidad automática.
- Tráfico gestionado con Ingress.

---

**Resumen:**

Este caso práctico demuestra cómo construir y desplegar una arquitectura moderna en Kubernetes de manera profesional, combinando todos los elementos aprendidos.



# 4. Optimización de clústeres y costes

Kubernetes te da potencia y flexibilidad, pero si no gestionas bien tu infraestructura, **los costes pueden dispararse**.
Una parte clave de ser un operador eficiente es saber **optimizar el clúster** para que sea:

- Más barato.
- Más eficiente.
- Más rápido.

Vamos a ver cómo hacerlo.

---

## 4.1. Dimensionar correctamente los recursos

**Error común:**

- Sobredimensionar requests y limits de CPU/memoria "por si acaso".

**Optimización:**

- Analiza el consumo real de los Pods (`kubectl top pods`).
- Ajusta `requests` y `limits` basados en observaciones, no en suposiciones.
- Usa **Vertical Pod Autoscaler (VPA)** para recomendaciones automáticas.

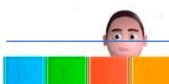
---

## 4.2. Escalado automático inteligente

- Configura **Horizontal Pod Autoscalers (HPA)** para adaptar los Pods a la carga real.
- Usa **Cluster Autoscaler** para añadir o quitar nodos dinámicamente según el uso.

**Así pagas solo por lo que realmente necesitas.**

---



## 4.3. Uso de nodos adecuados

**Optimización de nodos:**

- No uses siempre máquinas grandes; analiza si necesitas más nodos pequeños o menos nodos grandes.
- Usa **grupos de nodos (Node Pools)** diferentes:
    - Un grupo para cargas ligeras (aplicaciones web).
    - Otro para cargas pesadas (bases de datos, procesamiento intensivo).

---

## 4.4. Borrar recursos no utilizados

- **Pod olvidado** = CPU/RAM desperdiciados.
- Haz limpiezas regulares:
    - Pods en estado CrashLoopBackOff.
    - Servicios sin Pods conectados.
    - Deployments obsoletos.

Automatiza tareas de limpieza usando scripts o herramientas como **kubectl prune**.

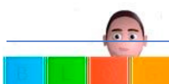
---

## 4.5. Uso de instancias Spot o Preemptibles

En la nube (AWS, GCP, Azure):

- Considera usar **instancias spot/preemptibles** para cargas no críticas.
- Son mucho más baratas (~70% de descuento), ideales para procesamientos batch, cargas redundantes o entornos de desarrollo.

---



## 4.6. Monitorear el gasto

Utiliza herramientas de monitoreo de costes:

| Herramienta | Uso principal |
|---|---|
| Kubernetes Metrics Server | Métricas básicas de consumo. |
| Prometheus + Grafana | Análisis avanzado de consumo y eficiencia. |
| Kubecost | Estimación de costes Kubernetes en tiempo real. |
| Cloud-specific billing | Dashboards de AWS, GCP, Azure. |

**Visualizar el gasto** te permitirá optimizar mejor.

---

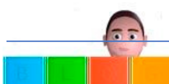
## 4.7. Buenas prácticas de optimización

- **Haz "right-sizing"** de Pods y nodos.
- **Aprovecha autoscalado** en todos los niveles (Pods y nodos).
- **Divide workloads** críticos y no críticos.
- **Apaga ambientes de desarrollo** fuera de horario laboral.
- **Define políticas de lifecycle**: elimina recursos automáticamente cuando no se usen.

---

**Resumen:**

Optimizar clústeres y costes en Kubernetes es clave para una operación sostenible y eficiente: debes ajustar recursos, automatizar escalado, limpiar recursos olvidados y monitorear gastos constantemente.



# 5. Tendencias futuras: Serverless y Kubernetes

Kubernetes ha revolucionado el mundo de la infraestructura, pero el ecosistema **no se detiene**: surgen nuevas tendencias que transforman cómo diseñamos y desplegamos aplicaciones en la nube.

Vamos a ver hacia dónde se dirige el futuro de Kubernetes.

---

## 5.1. Kubernetes + Serverless

**Serverless** no significa "sin servidores", sino que **el usuario no gestiona los servidores**.
La infraestructura se escala automáticamente y los recursos se asignan dinámicamente según el uso real.

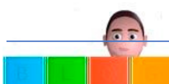
**¿Cómo encaja Serverless en Kubernetes?**

- Kubernetes puede ser extendido con **frameworks serverless** para ejecutar funciones a demanda.
- Gestiona solo el código y la lógica; Kubernetes se encarga del resto.

**Ejemplos populares:**

- **Knative:**
  Plataforma serverless sobre Kubernetes. Permite desplegar funciones y servicios autoscalables hasta cero.
- **OpenFaaS:**
  Framework para desplegar funciones como servicios en Kubernetes de forma sencilla.
- **Kubeless:**
  Ligero y simple, pensado para funciones pequeñas en Kubernetes.

---



## 5.2. Tendencias fuertes en Kubernetes

| Tendencia | Descripción |
| --- | --- |
| **GitOps** | Automatizar despliegues declarativos directamente desde Git. |
| **Service Mesh** | Controlar la comunicación entre servicios de manera segura (ej: Istio, Linkerd). |
| **Multi-cluster y multicloud** | Gestionar múltiples clústeres Kubernetes desde una sola capa de control. |
| **AI y Machine Learning en Kubernetes** | Usar Kubernetes para entrenar y desplegar modelos de ML. |
| **Zero Trust Security** | Aumentar la seguridad aplicando principios de desconfianza total en redes internas. |

---

## 5.3. Kubernetes no desaparecerá, evolucionará

Kubernetes ha dejado de ser "solo para expertos".
Ahora es:

- Base de plataformas modernas (Cloud Native).
- Impulsor de innovación serverless.
- Centro de automatización y autosanación de infraestructura.

**Se integrará más y más** con tecnologías de IA, big data, IoT y operaciones autónomas en los próximos años.

---

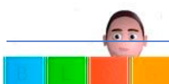
## 5.4. ¿Qué debes aprender para el futuro?

- **Profundizar en GitOps** (ArgoCD, Flux).
- **Dominar Service Mesh** (Istio, Linkerd).
- **Optimizar clústeres multi-región**.
- **Incorporar serverless nativo en tus arquitecturas** (Knative, OpenFaaS).
- **Aplicar seguridad avanzada** en redes y cargas de trabajo.

---

**Resumen:**

Kubernetes seguirá siendo el corazón de las plataformas modernas, pero complementándose cada vez más con serverless, GitOps, Service Mesh y arquitecturas autónomas. ¡El futuro es híbrido, automatizado y cloud-native!



# Conclusiones y siguientes pasos

Al llegar al final de este viaje, has recorrido todo el universo de Kubernetes:
desde los conceptos más básicos hasta las prácticas más avanzadas para operar, optimizar y proyectar el futuro de tus clústeres y aplicaciones cloud-native.

Aprendiste a:

- Contenerizar aplicaciones de forma profesional.
- Desplegarlas, escalarlas y exponerlas de manera segura en Kubernetes.
- Gestionar configuraciones, secretos, volúmenes persistentes y tráfico de red.
- Optimizar recursos y costes, asegurando la eficiencia de los clústeres.
- Incorporar automatización, serverless y buenas prácticas de CI/CD.
- Entender hacia dónde se mueve Kubernetes y prepararte para las tendencias que vienen.

**Pero esto no es un final: es un comienzo.**

Kubernetes no es solo una tecnología:
es una **cultura de diseño** basada en la resiliencia, la automatización y la evolución continua.

---

## Siguientes pasos recomendados

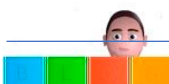
- **Profundizar en GitOps:** Automatizar todo el ciclo de vida usando ArgoCD o Flux.
- **Explorar Service Mesh:** Implementar Istio o Linkerd para gestionar comunicaciones seguras y observables entre servicios.
- **Prepararte para multiclúster y multicloud:** Aprender herramientas como Rancher, Anthos o Amazon EKS Anywhere.
- **Contribuir a la comunidad:** Participar en proyectos open source, asistir a meetups, leer la Kubernetes Enhancement Proposals (KEPs).
- **Seguir actualizándote:** Kubernetes evoluciona constantemente, cada versión trae mejoras de seguridad, rendimiento y operatividad.

---

## Un mensaje final

**Dominar Kubernetes te convierte en un arquitecto del futuro de la infraestructura.**
No te detengas aquí. Sigue construyendo, aprendiendo, optimizando y, sobre todo, soñando con lo que puedes crear.

**¡Bienvenido al mundo del cloud-native!** 🌎 🚀



# Apéndice A: Glosario de términos

## Cluster

Conjunto de nodos (servidores físicos o virtuales) que forman una plataforma de ejecución para aplicaciones contenerizadas, gestionada por Kubernetes.

## Node

Un solo servidor (físico o virtual) dentro del clúster Kubernetes que ejecuta Pods y proporciona recursos de CPU, memoria, almacenamiento y red.

## Pod

La unidad más pequeña desplegable en Kubernetes. Puede contener uno o varios contenedores que comparten red, almacenamiento y configuraciones.

## Container

Instancia ligera, aislada y ejecutable de una aplicación con todo su entorno de ejecución empaquetado (Docker es el ejemplo más común).

## Deployment

Objeto que administra un conjunto de Pods idénticos, proporcionando despliegue controlado, actualizaciones y escalado automático.

## ReplicaSet

Objeto que asegura que siempre exista un número especificado de réplicas de un Pod en ejecución.

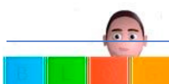
## Service

Abstracción que expone una aplicación ejecutándose en un conjunto de Pods como un único punto de acceso de red (IP o nombre DNS).

## Ingress

Objeto que gestiona el acceso externo al clúster Kubernetes, dirigiendo peticiones HTTP/HTTPS a los servicios internos.

## Namespace



Mecanismo para dividir recursos de Kubernetes en espacios aislados, útiles para organizar equipos, entornos o aplicaciones.

### ConfigMap

Recurso que permite almacenar configuraciones no sensibles (por ejemplo, variables de entorno) en pares clave-valor.

### Secret

Objeto diseñado para almacenar datos sensibles (contraseñas, tokens, certificados) de forma segura.

### PersistentVolume (PV)

Porción de almacenamiento en el clúster provisionada para ser utilizada por los Pods de manera persistente.

### PersistentVolumeClaim (PVC)

Petición de almacenamiento hecha por un usuario para utilizar un PersistentVolume en Kubernetes.

### Horizontal Pod Autoscaler (HPA)

Controlador que ajusta automáticamente el número de réplicas de Pods basándose en el consumo de recursos como CPU o memoria.

### Vertical Pod Autoscaler (VPA)

Mecanismo que ajusta automáticamente los recursos asignados a un Pod (CPU y memoria) para optimizar su rendimiento.

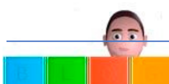
### ResourceQuota

Restricción aplicada en un Namespace para limitar el consumo total de recursos (CPU, memoria, almacenamiento).

### Liveness Probe

Chequeo de salud configurado para verificar si un contenedor está funcionando correctamente. Si falla, Kubernetes reinicia el contenedor.

### Readiness Probe



Chequeo de disponibilidad que determina si un contenedor está listo para recibir tráfico.

### Volume

Disco o sistema de archivos que puede ser montado en un Pod para almacenar datos compartidos o persistentes.

### ServiceAccount

Identidad específica que un Pod puede usar para comunicarse de manera segura con el API Server de Kubernetes.

### Role / ClusterRole

Conjunto de reglas de acceso que definen qué acciones puede realizar un usuario o un servicio dentro de Kubernetes.

### RoleBinding / ClusterRoleBinding

Vincula un Role o ClusterRole a un usuario, grupo o ServiceAccount, aplicando sus permisos.

### Ingress Controller

Componente que implementa las reglas de Ingress y gestiona el tráfico entrante al clúster.

### Operator

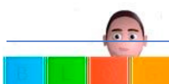
Extensión que automatiza tareas complejas en Kubernetes mediante la combinación de CRDs (Custom Resources) y Controladores personalizados.

### Custom Resource Definition (CRD)

Extensión de la API de Kubernetes que permite definir y gestionar recursos personalizados.

### GitOps

Modelo de gestión de infraestructura donde Git actúa como la fuente única de verdad para las configuraciones y despliegues.



# Apéndice B: Recursos recomendados

## Libros recomendados

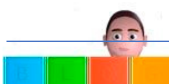
- **Kubernetes Up & Running**
  *Autores:* Kelsey Hightower, Brendan Burns y Joe Beda
  *Descripción:* Una introducción práctica y profunda a Kubernetes escrita por tres de sus creadores.
- **The Kubernetes Book**
  *Autor:* Nigel Poulton
  *Descripción:* Un enfoque claro y directo para entender y desplegar Kubernetes rápidamente.
- **Cloud Native DevOps with Kubernetes**
  *Autores:* John Arundel y Justin Domingus
  *Descripción:* Aprende a construir sistemas escalables y resilientes usando Kubernetes y principios de DevOps.
- **Kubernetes Patterns**
  *Autores:* Bilgin Ibryam y Roland Huß
  *Descripción:* Estudia patrones de diseño para arquitecturas cloud-native sobre Kubernetes.

---

## Cursos recomendados

- **Certified Kubernetes Administrator (CKA) – Linux Foundation Training**
  Curso oficial para prepararte como administrador certificado de Kubernetes.
- **Kubernetes for the Absolute Beginners – Udemy (Mumshad Mannambeth)**
  Curso perfecto para quienes empiezan desde cero.
- **Learn Kubernetes – Katacoda**
  Entornos de práctica interactivos que no requieren instalaciones locales.
- **Pluralsight – Kubernetes Deep Dive**
  Serie de cursos más detallados para usuarios intermedios y avanzados.

---



# Documentación oficial y tutoriales

- Documentación oficial de Kubernetes
  La referencia más completa y actualizada sobre todos los componentes de Kubernetes.
- Guía de Conceptos Fundamentales
  Ideal para construir una base sólida de conocimientos.
- Tutoriales Kubernetes (Hands-on)
  Prácticas oficiales paso a paso para aprender haciendo.
- Artifact Hub
  Repositorio de Helm Charts, Operators y otros recursos para Kubernetes.
- Awesome Kubernetes (GitHub)
  Colección de enlaces, recursos, libros, cursos y herramientas de la comunidad.

# Apéndice C: Comandos útiles de referencia rápida

## Gestión de Pods

| Acción | Comando |
|---|---|
| Listar todos los Pods en el Namespace actual | `kubectl get pods` |
| Listar Pods en todos los Namespaces | `kubectl get pods --all-namespaces` |
| Ver detalles de un Pod | `kubectl describe pod <nombre-del-pod>` |
| Ver logs de un Pod | `kubectl logs <nombre-del-pod>` |
| Ver logs de un contenedor específico en un Pod | `kubectl logs <nombre-del-pod> -c <nombre-contenedor>` |
| Ejecutar un comando en un Pod | `kubectl exec -it <nombre-del-pod> -- bash` |

## Gestión de Deployments

| Acción | Comando |
|---|---|
| Crear un Deployment rápido | `kubectl create deployment <nombre> --image=<imagen>` |
| Escalar un Deployment | `kubectl scale deployment <nombre> --replicas=<número>` |
| Actualizar la imagen de un Deployment | `kubectl set image deployment/<nombre> <contenedor>=<imagen:tag>` |
| Hacer rollback de un Deployment | `kubectl rollout undo deployment/<nombre>` |
| Ver historial de rollouts | `kubectl rollout history deployment/<nombre>` |
| Pausar un Deployment | `kubectl rollout pause deployment/<nombre>` |
| Reanudar un Deployment pausado | `kubectl rollout resume deployment/<nombre>` |

# Gestión de Servicios

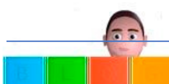
| Acción | Comando |
|---|---|
| Listar todos los Servicios | `kubectl get services` |
| Describir un Servicio | `kubectl describe service <nombre>` |
| Exponer un Deployment como Servicio | `kubectl expose deployment <nombre> --port=<puerto>` |

# Gestión de Clúster

| Acción | Comando |
|---|---|
| Listar los nodos del clúster | `kubectl get nodes` |
| Ver detalles de un nodo | `kubectl describe node <nombre-del-nodo>` |
| Ver consumo de CPU y RAM de nodos | `kubectl top nodes` |
| Ver consumo de CPU y RAM de Pods | `kubectl top pods` |

# Gestión de ConfigMaps y Secrets

| Acción | Comando |
|---|---|
| Crear un ConfigMap desde archivo | `kubectl create configmap <nombre> --from-file=<archivo>` |
| Crear un Secret desde archivo | `kubectl create secret generic <nombre> --from-file=<archivo>` |
| Ver ConfigMaps existentes | `kubectl get configmaps` |
| Ver Secrets existentes | `kubectl get secrets` |



# Aplicar y eliminar archivos YAML

| Acción | Comando |
| --- | --- |
| Aplicar un archivo de configuración | `kubectl apply -f <archivo.yaml>` |
| Eliminar un recurso definido en YAML | `kubectl delete -f <archivo.yaml>` |
| Ver todos los objetos en un Namespace | `kubectl get all` |

**Consejo final:**

Cuando tengas dudas sobre un comando, puedes usar `kubectl explain <recurso>` para obtener una breve explicación directamente en la terminal.