Lo que pasa es que desarrollar en Kubernetes

Guía completa de desarrollo de aplicaciones en Kubernetes en lenguaje sencillo - Muestra gratuita (Capítulo 1)



Luis Felipe Ariza Vesga

Lo que pasa es que desarrollar en Kubernetes

Lo que pasa es que desarrollar en Kubernetes

Guía completa de desarrollo de aplicaciones en Kubernetes en lenguaje sencillo - Muestra gratuita (Capítulo 1)

Luis Felipe Ariza Vesga

Lo que pasa es que desarrollar en Kubernetes
Guía completa de desarrollo de aplicaciones en Kubernetes en lenguaje sencillo - Muestra gratuita (*Capítulo 1*)

Copyright © Luis Felipe Ariza Vesga.
Todos los derechos reservados.

La información de este libro se distribuye "tal cual", sin garantía. Aunque se han tomado todas las precauciones en su preparación, ni el autor ni la editorial asumen responsabilidad alguna por pérdidas o daños derivados directa o indirectamente de su contenido.

ISBN: 9798161946701

Para el zarapito esquimal (*Numenius borealis*)*, que vivió en la tundra del oeste ártico de Canadá y Alaska con poblaciones de millones, y que hoy la UICN clasifica como en peligro crítico y posiblemente extinta. Y para quien despliega una aplicación en un clúster que no administra: tener muchas réplicas no protege por sí solo; protege que el resto del sistema pueda absorber la pérdida de una.*

Índice general

Convenciones de este libro **1**
Tipografía y referencias 1
Las cuatro cajas de aviso 2
Créditos y licencias de terceros 3

Prefacio **5**
De qué se trata este libro 7
De qué no se trata este libro 7
Cómo está organizado 8
Sobre los comandos y ejemplos de código 8
Cómo contactarnos 10
Agradecimientos 10

I Diseño y construcción de aplicaciones (20%) **13**

1 La receta que se lleva a cualquier cocina **15**
1.1 La receta que nunca quedó escrita 15
1.2 La receta escrita: qué es una imagen de contenedor 17
1.3 Cocinar la receta una sola vez: capas, y por qué no se repite el trabajo 19
1.4 El nombre en la fachada: tags, digests y el registro donde queda guardada 23
1.5 Practique en el laboratorio 32
1.6 Resumen: la receta que llega intacta a cualquier cocina 32
Bibliografía 33

Convenciones de este libro

El pequeño sistema visual de este libro, demostrado en vivo, no solo descrito.

Antes de arrancar, media página sobre cómo está escrito este libro. Se lee en dos minutos y le ahorra confusiones después.

Este libro le habla de **usted**. No es formalidad ni distancia: es la manera respetuosa de dirigirse a alguien con quien se trabaja hombro a hombro.

Cada capítulo abre con una escena cotidiana. Esa escena no es adorno ni relleno: **es** el concepto técnico contado de otra manera, y vuelve al final del capítulo, en el resumen, ya explicada con las palabras correctas. Si en algún momento no entiende una explicación técnica, devuélvase a la escena del comienzo: casi siempre la respuesta está ahí en versión sencilla.

Este libro traduce el lenguaje técnico a lenguaje cotidiano, con situaciones de la rutina diaria que todos reconocemos, usadas como analogía para entender rápido un concepto de Kubernetes que suele explicarse en lenguaje técnico puro. Eso incluye datos como que una readiness probe fallida saca al Pod del tráfico del Service sin reiniciar el contenedor. La meta es que usted entienda. Por eso la analogía nunca reemplaza el rigor: después de ella viene siempre la definición oficial, citada tal cual, con su enlace a la fuente.

Tipografía y referencias

Tipografía monoespaciada se usa para todo lo que usted tecleería o leería en una pantalla: nombres de objetos como `Deployment` y `CronJob`, campos de YAML como `restartPolicy` y `revisionHistoryLimit`, comandos como `kubectl rollout history`, y rutas como `/var/log/pods/`.

Negrita marca la primera aparición de un término importante, y también las palabras que el examen suele usar textualmente. Cuando usted vea una palabra en negrita en medio de una definición, es porque esa palabra exacta importa.

Los **paréntesis angulares** marcan lo que usted debe reemplazar: en `kubectl rollout status deployment/<nombre>`, usted escribe el nombre real del Deployment y no escribe los paréntesis.

Las **citas de la documentación oficial** aparecen integradas en el texto, no como bloques aparte, y siempre llevan un enlace a la fuente la primera vez que se usan en el capítulo. Eso es intencional: usted debe poder verificar cualquier afirmación de este libro sin buscarla, haciendo clic.

Cada capítulo cierra con una **bibliografía** de al menos tres fuentes, todas oficiales y todas con enlace.

Las **figuras** se referencian en el texto y llevan pie explicativo. Si una figura aparece cerca de una explicación, es porque esa explicación se entiende mejor mirándola.

Las cuatro cajas de aviso

A lo largo del libro va a encontrar cuatro tipos de caja. Cada una significa algo distinto: un comando para verificar, un atajo para la terminal del examen, un comportamiento observado en un clúster real o un riesgo de producción. Ninguna le advierte sobre preguntas capciosas, porque el CKAD no las tiene: es un examen práctico que le asigna tareas y califica el estado final del clúster. Conviene reconocerlas de un vistazo, porque no todas tienen la misma urgencia.

> **Nota**
> Información complementaria que amplía lo que acaba de leer, casi siempre con la forma de comprobarlo usted mismo. Se puede saltar sin perder el hilo. Ejemplo: `kubectl rollout history deployment/<nombre>` le muestra las revisiones que guarda un Deployment, y con `--revision=<número>` le muestra la plantilla de Pod de una revisión concreta.

Consejo

Un atajo para la terminal del examen, una regla mnemotécnica o una forma más rápida de hacer algo. Ejemplo: `kubectl create deployment cocina --image=nginx --replicas=2 --dry-run=client -o yaml` le escribe el esqueleto del manifiesto en un segundo, sin crear nada en el clúster; usted lo redirige a un archivo y edita solo lo que la tarea pide.



Advertencia

Un error frecuente o un comportamiento que sorprende en un clúster real, de esos que dejan una tarea a medio resolver. Si usted solo va a leer un tipo de caja antes de presentar el examen, que sean estas. Ejemplo: un Job no acepta `restartPolicy: Always` en la plantilla de su Pod; Kubernetes solo permite `Never` u `OnFailure`. Si necesita algo que corra para siempre, eso es un Deployment.



Precaución

Algo que puede causar daño real en un clúster de producción: pérdida de datos, caída de servicio o un hueco de seguridad. Son pocas y van en serio. Ejemplo: una política de egress restrictiva también gobierna la consulta DNS de su aplicación. Si solo permite el tráfico hacia el Pod de la base de datos y olvida la salida hacia el DNS del clúster, la aplicación ni siquiera puede resolver el nombre del Service con el que quiere hablar.

Créditos y licencias de terceros

Este libro reproduce y traduce fragmentos de la documentación oficial de Kubernetes (https://kubernetes.io/docs/), publicada por The Kubernetes Authors bajo licencia Creative Commons Attribution 4.0 International (CC BY 4.0). El texto original está disponible en https://creativecommons.org/licenses/by/4.0/. También cita la documentación de Helm (https://helm.sh/docs/), que el sitio de Helm declara distribuida bajo CC-BY-4.0, y enlaza la documentación de Docker y de Kustomize, siempre con enlace a cada página.

Las definiciones citadas textualmente aparecen entre comillas y enlazadas a la página de la que provienen. Las traducciones al español son propias de este libro y no han sido revisadas ni avaladas por The Kubernetes Authors ni por los demás proyectos

citados.

Kubernetes y Helm figuran entre las marcas registradas de The Linux Foundation, y el nombre de la certificación Certified Kubernetes Application Developer también, según su lista oficial de marcas. Este libro es material de estudio independiente y no está afiliado a The Linux Foundation, a la Cloud Native Computing Foundation ni a ninguno de los proyectos o empresas mencionados, ni cuenta con su respaldo o patrocinio.

Prefacio

Cómo está organizado este libro, con qué voz explica el examen CKAD, y por qué usted arrienda el local en lugar de administrar la finca.

Este libro nace de la misma pregunta que le dio origen a toda la serie "Lo que pasa es que": ¿y si un examen de turno se explicara con la misma voz de "lo que pasa es que" con la que se explica la vida en cualquier negocio de la esquina? El Certified Kubernetes Application Developer lo pintan como un examen de sintaxis: manifiestos YAML, probes, ConfigMaps, Secrets, NetworkPolicies, palabras que suenan a curso de programación y a nada más. Pero lo que pasa es que, debajo de ese vocabulario, desarrollar aplicaciones para un clúster de Kubernetes se parece mucho a montar un negocio de comida en un local arrendado dentro de un centro comercial que administra otra persona: usted no levanta el edificio ni contrata al celador de la entrada, pero si tiene que escoger la receta que se puede preparar en cualquier cocina, decidir el tipo de negocio que va a abrir, cambiar de turno sin cerrar las puertas, y contestar el domicilio que si sabe llegar. Ese es el hilo que atraviesa los quince capítulos de este libro: usted desarrolla y despliega aplicaciones, no administra la finca completa, y ese matiz cambia todo lo que necesita saber para presentar el examen.

Este es el tercer libro de la serie. Si usted está estudiando para el examen y quiere un orden lógico, empiece por el libro 1, "Lo que pasa es que Kubernetes", donde se explican las bases de Kubernetes. Siga con el libro 2, "Lo que pasa es que administrar Kubernetes", donde se aprende lo más importante de instalar un clúster desde cero y los retos que tiene un ingeniero de infraestructura para facilitarle al desarrollador el ciclo de vida de sus aplicaciones. Este libro 3 agradece todo ese conocimiento y se dedica a lo que le importa al desarrollador: el código y el control del ciclo de vida de la aplicación en Kubernetes. Aquí usted piensa como un desarrollador con mirada de SRE y DevOps. No le preocupa que el clúster funcione, porque de eso se encargó el

libro 2. Le preocupa que la aplicación se pueda desplegar, escalar, observar y entregar de forma continua. Cuando un tema ya se explicó en los libros anteriores, como RBAC o las NetworkPolicies, aquí se recuerda en pocas líneas y se mira desde el lado de la aplicación.

El temario oficial del CKAD, publicado por la CNCF, describe qué sabe hacer ese desarrollador en el clúster:

- Construir y modificar imágenes de contenedor.
- Escoger el recurso de carga de trabajo correcto: Deployment, DaemonSet, CronJob y otros.
- Diseñar Pods con varios contenedores, con patrones como sidecar e init.
- Usar volúmenes persistentes y efímeros.
- Implementar estrategias de despliegue como blue/green o canary con las primitivas de Kubernetes.
- Hacer rolling updates con Deployments.
- Desplegar paquetes existentes con Helm y adaptar manifiestos con Kustomize.
- Atender las obsolescencias de la API.
- Implementar probes y chequeos de salud.
- Monitorear aplicaciones con las herramientas de la línea de comandos, leer logs y depurar.
- Extender Kubernetes con CRD y operadores.
- Entender la autenticación, la autorización y el control de admisión.
- Declarar requests, limits y cuotas.
- Usar ConfigMaps, crear y consumir Secrets, y entender los ServiceAccounts.
- Aplicar seguridad a la aplicación con SecurityContexts y capabilities.
- Entender las NetworkPolicies, dar acceso con Services y exponer aplicaciones con Ingress.

El temario no incluye herramientas de CI/CD como tal. Sí incluye lo que esos flujos usan para desplegar: manifiestos declarativos, Helm, Kustomize y rollouts controlados. Este libro los trata como las piezas de esa entrega continua.

De qué se trata este libro

Este es un libro de preparación para el examen de certificación CKAD, Certified Kubernetes Application Developer, de la Cloud Native Computing Foundation (CNCF) y la Linux Foundation. Sigue el temario oficial v1.37, organizado en cinco partes que siguen los cinco dominios vigentes del examen: Diseño y construcción de aplicaciones; Despliegue de aplicaciones; Observabilidad y mantenimiento de aplicaciones; Entorno, configuración y seguridad de aplicaciones, el dominio más pesado; y Servicios y redes. Son quince capítulos, cada uno abierto con una escena cotidiana del negocio en un local arrendado (la receta que se lleva a cualquier cocina, el cambio de turno sin cerrar el negocio, el menú del día y la caja fuerte, el domicilio que sí sabe llegar) que se convierte en la explicación técnica precisa de un mecanismo real de Kubernetes, y cerrado con un resumen que recoge esa misma escena. Cada una de las cinco partes tiene su propio laboratorio sobre un clúster kind real con kubectl.

Un aviso sobre la versión. Al escribir esta edición, el temario publicado por la CNCF es el v1.37, con el mismo contenido que el v1.35, y la página oficial de la certificación indica que el examen se basa en Kubernetes v1.37. La misma página explica que el entorno del examen se alinea con la versión menor más reciente de Kubernetes en un plazo aproximado de cuatro a ocho semanas. Antes de agendar su examen, revise esa página y el repositorio github.com/cncf/curriculum: si el temario cambió, esa página manda sobre este libro.

De qué no se trata este libro

Este libro no es un manual de referencia exhaustivo de cada campo, cada bandera ni cada objeto de la API de Kubernetes; para eso está la documentación oficial en kubernetes.io/docs, que este libro cita en cada capítulo. Tampoco es un curso de contenedores desde cero: se asume que usted ya sabe qué es una imagen de contenedor, ya construyó al menos un Dockerfile, y ya sabe moverse en una terminal y leer un archivo YAML sin que se lo tengan que explicar aquí. Tampoco es un manual de una distribución específica de Kubernetes ni de un proveedor de nube en particular: el libro se queda en lo que el examen CKAD evalúa sobre un clúster genérico, no en cómo usar el registro de contenedores o el servicio administrado de tal o cual proveedor. Y definitivamente no reproduce preguntas de exámenes reales ni de simuladores

comerciales como killer.sh: eso violaría tanto la ética como el acuerdo de confidencialidad del examen, y además sería mal negocio, porque memorizar una pregunta no le sirve el día que el examen le pregunta lo mismo con otro nombre de namespace.

Cómo está organizado

El libro sigue el orden de los cinco dominios oficiales del examen, agrupados en cinco partes con su peso declarado. La primera parte cubre el diseño y la construcción de aplicaciones: la receta que se lleva a cualquier cocina con imágenes de contenedor portables, cómo escoger el tipo de negocio entre Deployment, StatefulSet, DaemonSet y Job, y el ayudante de cocina con contenedores de inicio y volúmenes compartidos. La segunda parte cubre el despliegue: el cambio de turno sin cerrar el negocio con rollouts controlados, probar el plato nuevo con la mitad de los clientes con estrategias de despliegue progresivo, y la franquicia con instrucciones de armado con Helm y Kustomize. La tercera parte cubre la observabilidad y el mantenimiento: el letrero de abierto y el pulso del negocio con probes de vida y disponibilidad, y el libro de quejas con logs y el catálogo que cambia de página con la política de obsolescencia de la API. La cuarta parte, la más pesada del examen, cubre el entorno, la configuración y la seguridad: cuánto espacio pide cada puesto en la plaza con requests y limits, el menú del día y la caja fuerte con ConfigMaps y Secrets, pedir un servicio que el barrio no tenía con recursos personalizados, y el carnet del empleado con ServiceAccounts y contextos de seguridad. La quinta y última parte cubre servicios y redes: el domicilio que sí sabe llegar con Services, la entrada principal del centro comercial con Ingress, y con quién sí se puede hablar desde el local con NetworkPolicies. Dentro de cada parte, los capítulos avanzan de lo más fundamental a lo más específico. El apéndice de glosario reúne los términos clave de todo el libro para repasar antes del examen, y el laboratorio de cada parte asume que usted ya leyó los capítulos de esa parte, no los de partes futuras.

Sobre los comandos y ejemplos de código

Todos los ejemplos de YAML y de comandos kubectl de este libro son reales y verificados contra la documentación oficial de kubernetes.io/docs, con los campos que la API de Kubernetes espera y con los nombres de recurso que define el propio esquema OpenAPI del clúster, no pseudocódigo inventado para que se vea bonito. Cuando

un ejemplo usa kubectl, el libro le indica el verbo preciso (create, apply, exec, logs, rollout) y el recurso sobre el que actúa, porque confundir un despliegue imperativo con `kubectl create` con uno declarativo con `kubectl apply` deja el clúster en un estado distinto al que usted esperaba. Cuando un comportamiento depende de la versión instalada, el libro le enseña a consultarla con `kubectl version` y en las notas de lanzamiento de kubernetes.io, en lugar de fijar un número que quede desactualizado el día que usted lea esto. Solo los laboratorios fijan una versión concreta del clúster, por digest de imagen de kind, porque un laboratorio sí necesita ser reproducible de principio a fin.

Todo el código de los laboratorios está publicado en https://github.com/lfarizav/lo-que-pasa-es-que-desarrollar-en-kubernetes. El repositorio tiene dos carpetas: `es/` para la edición en español y `en/` para la edición en inglés. Dentro de cada una hay un laboratorio por parte, de `lab-part-01` a `lab-part-05`, con sus comandos, sus comprobaciones y las versiones con las que se probó, listo para clonar y ejecutar en una terminal de Linux:

```
git clone https://github.com/lfarizav/lo-que-pasa-\
es-que-desarrollar-en-kubernetes.git
cd lo-que-pasa-es-que-desarrollar-en-kubernetes/es
```

Cada laboratorio levanta su propio clúster kind aislado y escribe su kubeconfig en un archivo privado, sin tocar su kubeconfig de siempre ni otros clústeres que usted tenga, así que nunca lo corra contra un clúster de producción. Cada paso le dice el objetivo y el entregable, y la secuencia de comandos queda guardada aparte para que usted la revele después de intentarlo por su cuenta. La calificación consulta el clúster en vivo y no lee su historial de comandos. Se parecen a lo que le van a pedir, porque el CKAD no es de opción múltiple: según la página oficial de la certificación, es una prueba práctica de dos horas en la línea de comandos que asigna tareas y califica el estado que usted deja en el clúster. Aun así no son un simulador ni reproducen preguntas del examen.

Cada capítulo cierra, justo antes de su resumen, con una sección **Practique en el laboratorio** que le dice qué laboratorio le corresponde y qué temas va a trabajar allí; los pasos, los comandos y las comprobaciones están solo en el repositorio.

Cómo contactarnos

Este libro es parte del catálogo de Luis Felipe Ariza Vesga. Si encuentra un ejemplo que no aplica como el libro dice, un campo que cambió de nombre en una versión reciente de Kubernetes, un comportamiento de una probe o de una estrategia de despliegue que se actualizó después de escribir este capítulo, o una afirmación que no coincide con la documentación oficial vigente, esa retroalimentación es bienvenida y se toma en serio: un libro técnico que no se corrige deja de ser confiable. Kubernetes es un proyecto que evoluciona rápido, con nuevas versiones cada pocos meses y objetos de la API que se gradúan de beta a estable sin previo aviso, así que un libro sobre el tema tiene que aceptar que ninguna edición es la última palabra, y que la corrección hecha a tiempo vale más que la promesa de que nunca hará falta. Este libro prefiere quedar corregido a quedar bonito, igual que un desarrollador prefiere un readinessProbe honesto que avise a tiempo que el negocio todavía no está listo para atender, y no un letrero de abierto que miente mientras el local está vacío por dentro.

Agradecimientos

Este libro no inventó nada desde cero, y eso es justamente lo que quiere agradecer. Kubernetes existe porque miles de personas decidieron no reinventar la rueda cada una por su lado, sino construir una sola rueda entre todas, a la vista de todos.

A la Cloud Native Computing Foundation (CNCF) y a la Linux Foundation, por darle una casa neutral a Kubernetes y a los proyectos que lo rodean, por publicar abierto el temario del examen CKAD y por sostener la certificación que este libro prepara.

A las comunidades y a las empresas que donaron proyectos completos a la CNCF. Entregar un proyecto entero a una fundación, con su código, su nombre y su gobierno, en lugar de guardarlo como ventaja propia, es lo que permite que un desarrollador en cualquier parte del mundo use las mismas herramientas sin pedirle permiso a nadie.

A los mantenedores de Kubernetes, kubectl, Helm, Kustomize, la Gateway API y los runtimes de contenedores que aparecen en estas páginas. Revisar código ajeno, responder issues a medianoche y decir que no a una función mal pensada es un trabajo que casi nadie ve y del que todo el mundo depende.

A todos los contribuidores: a quienes escriben código, pero también a quienes escriben y traducen la documentación de kubernetes.io, reportan errores, prueban versiones beta y responden preguntas en los foros. La documentación oficial que este libro cita en cada capítulo existe gracias a ellos, abierta y gratis para cualquiera.

A las comunidades locales de código abierto: a los miembros de Cloud Native Colombia, de KCD Colombia (Kubernetes Community Days) y del Colombia OpenInfra User Group. Organizan meetups, talleres y eventos para fomentar el uso del código abierto sin cobrar por ello, y muchas veces ponen de su propio bolsillo el refrigerio, el transporte o el alquiler del lugar para que cada encuentro se haga realidad. Buena parte de lo que aquí se cuenta se aprendió en esas reuniones.

Y gracias, por último, al zarapito esquimal (Numenius borealis), cuya historia le presta a la portada su lección más dura. Fue una de las aves playeras más numerosas de Norteamérica, y la caza intensiva de finales del siglo XIX la llevó a un declive del que no volvió a recuperarse; la UICN la clasifica hoy como en peligro crítico y posiblemente extinta. La lección para quien despliega aplicaciones es que tener muchas réplicas no protege por sí solo: una aplicación sobrevive a la caída de un Pod cuando el resto del sistema puede absorber esa pérdida.

PARTE I

Diseño y construcción de aplicaciones (20%)

1

La receta que se lleva a cualquier cocina

Por qué una imagen de contenedor es una receta completa que cualquier cocina puede seguir sin preguntar nada, y qué diferencia realmente un tag móvil de un digest inmutable.

1.1 La receta que nunca quedó escrita

Imagínese que usted por fin decide montar su propio negocio de comida en un local que consigue arrendado en la plaza de mercado. El primer día quiere replicar la misma receta del restaurante de su mamá, la que lleva veinte años haciendo el mismo mute santandereano sin que nadie se queje. El problema es que su mamá nunca escribió esa receta en ningún lado: la tiene en la cabeza, y cuando usted le pregunta cuánto cilantro le echa, ella le contesta "lo que vea que hace falta". Y su mamá, ese día, mientras revuelve la olla, le dice de pasada, cortico y en tono serio, que en la cocina el orden lo es todo: primero lo que más se demora en ablandar, después la carne, y las papas al final. Usted asiente sin ponerle mucha atención, porque lo suyo, ese día, es el cilantro. Eso funciona en su cocina, con su estufa, con sus ollas de toda la vida, pero el día que usted intenta hacer el mismo mute santandereano en el local nuevo, con otra estufa y otras ollas, el resultado sale distinto, y nadie sabe bien por qué.

Lo que pasa es que ese problema, la receta que solo funciona en una cocina específi-

ca porque nadie la escribió completa, es justo el problema que resuelve una imagen de contenedor. La documentación oficial de Kubernetes la define así: una imagen de contenedor "representa datos binarios que encapsulan una aplicación y todas sus dependencias de software", y son "paquetes de software ejecutables que pueden correr de forma autónoma y que hacen suposiciones muy bien definidas sobre su entorno de ejecución" (documentación oficial de Kubernetes sobre imágenes de contenedor). Es la receta escrita hasta el último gramo: no dice "el cilantro que haga falta", dice cuánto, de qué marca, y en qué momento se echa. Cualquier cocina que reciba esa receta completa produce el mismo mute santandereano, sin necesidad de llamar a preguntar nada.

Este capítulo sigue esa receta desde que se escribe hasta que llega al mostrador del cliente: cómo se cocina una sola vez y se reutiliza sin repetir el trabajo (las capas de la imagen), y cómo se le pone nombre en la fachada del local para que el cliente sepa qué está pidiendo (tags, digests, y el registro donde vive esa receta).

Y en una receta el orden importa tanto como los ingredientes. Si usted echa el arroz antes que el agua, o las papas al mismo tiempo que lo que tarda horas en ablandar, la comida igual sale, pero se pega en el fondo o queda con unos pedazos duros y otros deshechos. Eso era lo que su mamá le quería decir, y la Figura 1.1 pone las dos ollas una al lado de la otra. Con una imagen de contenedor pasa lo mismo: el Dockerfile se construye instrucción por instrucción, de arriba hacia abajo, y el orden en que usted las escribe decide qué tan rápido se reconstruye la imagen. La documentación oficial de Docker sobre la caché de construcción lo resume así: "si una capa cambia, todas las capas que vienen después también se ven afectadas". A escribir las instrucciones en el orden correcto se le llama seguir las buenas prácticas, y la siguiente sección le muestra cómo.

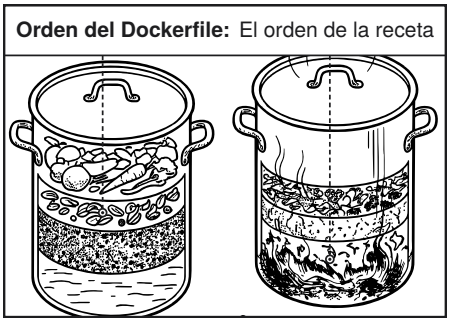


Figura 1.1. Dos ollas con los mismos ingredientes: en la de la izquierda el agua va primero y el arroz después; en la de la derecha el orden se invirtió y el fondo se quemó. En un Dockerfile, el orden de las instrucciones decide qué capas se pueden reutilizar.

1.2 La receta escrita: qué es una imagen de contenedor

Sigamos con la receta escrita. Una imagen de contenedor no es un programa a medio hacer que necesita que alguien complete instrucciones en el momento de correrlo: es el paquete completo, listo para ejecutarse solo, sin depender de que la cocina de turno tenga instalado tal condimento o tal utensilio por su cuenta. La misma documentación oficial insiste en ese punto: la imagen "hace suposiciones muy bien definidas sobre su entorno de ejecución" (Kubernetes, Container Images), lo cual quiere decir que la receta no asume nada que no traiga ella misma empacado: si necesita determinada versión de una librería, esa librería va dentro de la receta, no en la estantería de la cocina donde a lo mejor está, a lo mejor no.

En la práctica, esa receta se escribe con un Dockerfile (en la Figura 1.2, el cuaderno donde el cocinero joven anota cada paso): una lista de instrucciones, en orden, que describen paso a paso cómo se construye esa imagen desde una imagen base. Cada instrucción (una línea `FROM`, una `RUN`, una `COPY`) agrega algo concreto a la receta: la base sobre la que se cocina, un paso de preparación, un ingrediente que se incorpora. La comparación con el restaurante es directa: en vez de decirle al cocinero nuevo "usted ya sabe cómo se hace un sancocho", la receta escrita dice, en orden, corte tanto, sofría tanto tiempo, agregue esto después de aquello. Nada queda a criterio del cocinero de turno, y por eso la misma receta produce el mismo plato sin importar en qué sucursal de la cadena se cocine.



Figura 1.2. La mamá cocina de memoria mientras el cocinero joven anota cada paso en el cuaderno. Esa receta escrita completa es la imagen de contenedor, y el cuaderno donde se anota en orden es el Dockerfile.

Así queda escrita la receta de la sopa del día de La Buena Sazón, con la imagen base fijada por digest y el archivo `carta.html` al lado:

```
FROM busybox:1.37.0@sha256:9db7b59979c38555a39def84a31fb9
 ↪  8b5296952f9e3afd4f6f11f05b07adfab0
RUN mkdir -p /carta && echo "La Buena Sazon, sopa del dia"
 ↪  > /carta/fachada.txt
COPY carta.html /carta/index.html
CMD ["httpd", "-f", "-p", "8080", "-h", "/carta"]
```

Se cocina con un solo comando, que termina con la línea `naming to docker.io/library/sopa-del-dia:1.0 done`:

```
docker build -t sopa-del-dia:1.0 .
```

Y `docker history` muestra los pasos que quedaron guardados, del último al primero:

```
docker history sopa-del-dia:1.0
```

```
IMAGE          CREATED        CREATED BY
 ↪  SIZE      COMMENT
48fa57f12af3   1 second ago   CMD ["httpd" "-f" "-p"
 ↪  "8080" "-h" "/carta"]    0B
 ↪  buildkit.dockerfile.v0
<missing>      1 second ago   COPY carta.html
 ↪  /carta/index.html # buildkit    12.3kB
 ↪  buildkit.dockerfile.v0
<missing>      1 second ago   RUN /bin/sh -c mkdir -p
 ↪  /carta && echo "La B…   20.5kB
 ↪  buildkit.dockerfile.v0
<missing>      2 years ago    BusyBox 1.37.0 (glibc),
 ↪  Debian 13               4.55MB
```

La base de busybox llegó hecha (4.55MB) y encima quedaron dos capas pequeñas, una por `RUN` y otra por `COPY`, más un `CMD` que no ocupa espacio: la receta anota los pasos, no carga la cocina entera.

Esa receta escrita completa es, además, portable: la misma imagen que usted arma en su computador portátil corre igual en el clúster de producción, en el computador de un colega, o en el nodo que el scheduler eligió esta mañana, porque ninguno de esos destinos tiene que adivinar nada que la receta no traiga ya resuelto. Esa portabilidad es justamente el motivo por el que Kubernetes construye toda su unidad de despliegue, el Pod, alrededor de contenedores que corren a partir de imágenes, y no alrededor de binarios sueltos instalados a mano en cada máquina.

1.3 Cocinar la receta una sola vez: capas, y por qué no se repite el trabajo

Una receta completa no sirve de mucho si hay que cocinarla desde cero cada vez que alguien la pide. Lo que pasa es que una imagen de contenedor tampoco se cocina de una sola pieza: se construye en capas, y cada capa queda guardada por separado, lista para reutilizarse sin repetir el trabajo. La documentación oficial de Docker sobre buenas prácticas de construcción explica que, para cada instrucción del Dockerfile, "Docker verifica si puede reutilizar la instrucción desde la caché de construcción". Si usted ya cocinó el sofrito ayer y nadie tocó esa parte de la receta hoy, no hace falta volver a picar cebolla: se reutiliza el sofrito de ayer tal cual quedó guardado, como la

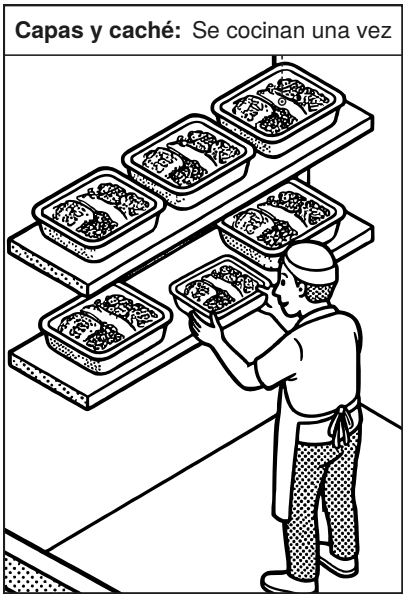


Figura 1.3. Un cocinero toma de la repisa una bandeja que ya estaba preparada en vez de cocinarla otra vez. Así funciona la caché de construcción: una capa que no cambió se reutiliza tal cual quedó guardada.



Figura 1.4. Detrás de la pared queda la cocina de preparación, con tablas, cáscaras y ollas; por la ventanilla solo sale el plato servido. Eso hace una construcción en varias etapas: las herramientas de construcción se quedan en una etapa anterior y la imagen final lleva solo lo necesario.

bandeja que el cocinero toma de la repisa en la Figura 1.3.

Para ver la caché trabajando basta con cambiar solo la carta y volver a construir; el `grep` deja a la vista los tres pasos de la receta y lo que pasó con cada uno:

```
echo 'sopa del dia de La Buena Sazon, v2 con mas cilantro'
 ↪  > carta.html
docker build -t sopa-del-dia:1.1 . 2>&1 | grep -A1 -E
 ↪  '^#[0-9]+ \[[0-9]/[0-9]\]'
```

```
#5 [1/3] FROM
 ↪  docker.io/library/busybox:1.37.0@sha256:9db7b59979c38
 ↪  555a39def84a31fb98b5296952f9e3afd4f6f11f05b07adfab0
#5 resolve
 ↪  docker.io/library/busybox:1.37.0@sha256:9db7b59979c38
 ↪  555a39def84a31fb98b5296952f9e3afd4f6f11f05b07adfab0
 ↪  0.0s done
--
#6 [2/3] RUN mkdir -p /carta && echo "La Buena Sazon, sopa
 ↪  del dia" > /carta/fachada.txt
#6 CACHED
--
#7 [3/3] COPY carta.html /carta/index.html
#7 DONE 0.0s
```

El `RUN` salió `CACHED` porque nadie lo tocó, y solo el `COPY` se volvió a ejecutar, porque la carta cambió. Es el sofrito de ayer tomado de la repisa: lo único que se picó hoy fue lo nuevo.

Ese ahorro depende por completo del orden en que se escriben las instrucciones. La misma documentación de Docker señala el ejemplo clásico del error: usar `apt-get update` solo, en una instrucción `RUN` separada de `apt-get install`, "causa problemas de caché y hace que las instrucciones posteriores de `apt-get install` fallen", porque si la caché reutiliza la línea del `update` sin volver a ejecutarla, el `install` de después puede terminar instalando paquetes de una lista de fuentes desactualizada. La recomendación oficial recomienda: "siempre combine `RUN apt-get update` con `apt-get install` en la misma instrucción `RUN`". En el restaurante: no tiene sentido guardar "revisar qué hay en la despensa" como un paso aparte de "comprar lo que falta", separados por un día de por medio; si los separa, puede terminar comprando con una lista de despensa que ya no es la de hoy.

Más allá de una sola imagen, cuando la receta necesita herramientas de construcción pesadas que el plato final no necesita (compiladores, dependencias de desarrollo, código fuente completo), la práctica recomendada es la construcción en varias etapas (multi-stage build). La documentación de Docker lo resume así: las construcciones en varias etapas "permiten reducir el tamaño de la imagen final, creando una separación más limpia entre la construcción de la imagen y la salida final", y además "permiten

construir de forma más eficiente ejecutando pasos de construcción en paralelo". Es la diferencia entre entregarle al cliente el plato ya servido y entregarle, además del plato, toda la batería de ollas sucias, las tablas de picar y las bolsas de mercado vacías que se usaron para prepararlo: la cocina de preparación se queda atrás, y a la mesa solo sale lo que el cliente pidió, que es lo que muestra la Figura 1.4.

Una construcción en varias etapas lleva dos `FROM` en el mismo archivo. La etapa `preparacion` ensucia 20 MB de ollas para producir el plato, y la etapa final solo se lleva el plato con `COPY --from`:

```
FROM busybox:1.37.0@sha256:9db7b59979c38555a39def84a31fb9
 ↪  8b5296952f9e3afd4f6f11f05b07adfab0 AS preparacion
RUN dd if=/dev/zero of=/ollas-sucias bs=1M count=20
RUN echo "sopa del dia de La Buena Sazon, v3" > /plato.html

FROM busybox:1.37.0@sha256:9db7b59979c38555a39def84a31fb9
 ↪  8b5296952f9e3afd4f6f11f05b07adfab0
COPY --from=preparacion /plato.html /carta/index.html
CMD ["httpd", "-f", "-p", "8080", "-h", "/carta"]
```

Se construyen la etapa de preparación sola, con `--target`, y la imagen final, y se comparan los tamaños:

```
docker build -q -f Dockerfile.etapas --target preparacion
 ↪  -t sopa-del-dia:preparacion .
docker build -q -f Dockerfile.etapas -t sopa-del-dia:2.0 .
docker image ls sopa-del-dia
```

```
sha256:2b16298afbb1c4478a9f2650a32d2d36b89089d03baee1fdaf
 ↪  82cf1cc3139be0
sha256:06cc70a322b29f678756fd95175c5bd25db7ab40dc4b24448d
 ↪  7d579f7fd29ccc
IMAGE                        ID             DISK USAGE
 ↪  CONTENT SIZE   EXTRA
sopa-del-dia:1.0             48fa57f12af3        6.8MB
 ↪  2.22MB
```

```
sopa-del-dia:1.1           0882b9770e3a        6.8MB
 ↪  2.22MB
sopa-del-dia:2.0           06cc70a322b2       6.78MB
 ↪  2.22MB
sopa-del-dia:preparacion   2b16298afbb1       27.8MB
 ↪  2.24MB
```

Y se busca la olla sucia en cada una:

```
docker run --rm sopa-del-dia:2.0 ls /ollas-sucias
docker run --rm sopa-del-dia:preparacion ls -lh
 ↪  /ollas-sucias
```

```
ls: /ollas-sucias: No such file or directory
-rw-r--r--    1 root     root       20.0M Oct  9 17:01
 ↪  /ollas-sucias
```

La imagen final ocupa 6.78MB contra 27.8MB de la etapa de preparación, y las ollas sucias no viajaron con ella: a la mesa solo salió el plato.

1.4 El nombre en la fachada: tags, digests y el registro donde queda guardada

Con la receta ya cocinada y guardada en capas, falta ponerle nombre en la fachada del local para que el cliente sepa qué está pidiendo. Una imagen normalmente se identifica con un nombre como `pause`, `example/micontenedor`, o `kube-apiserver`, y puede incluir además el nombre de un registro, por ejemplo `fictional.registry.example/imagename`, según la página sobre imágenes de contenedor en kubernetes.io. Y si usted no específica ningún registro, "Kubernetes asume que se refiere al registro público de Docker", así que un nombre corto siempre termina resolviéndose contra algún registro, aunque sea implícito: el archivo de recetas de la Figura 1.6.

Ahora bien, el nombre en la fachada no es lo único que importa: importa también qué tan firme es esa promesa. La misma documentación distingue con toda claridad entre dos cosas que a primera vista se parecen: "los tags le permiten identificar diferentes

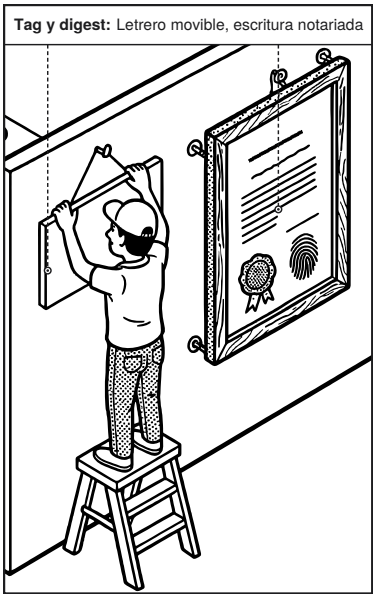


Figura 1.5. Un trabajador descuelga el letrero de sus ganchos para poner otro, y al lado la escritura notariada, con sello y huella, queda atornillada a la pared. El tag se puede mover a otra imagen; el digest no cambia nunca.



Figura 1.6. En el archivo de recetas, la encargada le entrega una caja sellada a un domiciliario, frente a estantes de cajas idénticas. El registro guarda las imágenes y las entrega a cualquier nodo que las pida.

versiones de la misma serie de imágenes. Los digests son un identificador único para una versión específica de una imagen. Los digests son hashes del contenido de la imagen, y son inmutables. Los tags se pueden mover para apuntar a imágenes distintas, pero los digests son fijos". Un tag, entonces, es como el letrero "Restaurante La Buena Sazón" en la fachada: sigue siendo el mismo letrero aunque el dueño del local cambie de proveedor de carne la semana entrante y el mute santandereano ya no sepa igual. Un digest es distinto: es el número de escritura notariada de ese local exacto, en ese estado exacto, y no hay forma de que apunte a otra cosa después; en la Figura 1.5 aparecen los dos, el letrero que se descuelga y la escritura atornillada. Esa misma garantía es la que documenta Docker cuando explica cómo descargar una imagen apuntando directamente a su digest, el identificador inmutable: al fijar el digest en el pull no queda ninguna ambigüedad sobre qué bytes exactos se están descargando, sin importar qué le haya pasado al tag mientras tanto.

La diferencia cabe en cuatro comandos: se cuelga el letrero `estable` sobre la versión 1.0, se mira, se cuelga el mismo letrero sobre la 1.1 y se vuelve a mirar:

```
docker tag sopa-del-dia:1.0 sopa-del-dia:estable
docker image ls sopa-del-dia:estable
docker tag sopa-del-dia:1.1 sopa-del-dia:estable
docker image ls sopa-del-dia:estable
```

```
IMAGE                  ID             DISK USAGE   CONTENT
 ↪  SIZE   EXTRA
sopa-del-dia:estable   48fa57f12af3        6.8MB
 ↪  2.22MB
IMAGE                  ID             DISK USAGE   CONTENT
 ↪  SIZE   EXTRA
sopa-del-dia:estable   0882b9770e3a        6.8MB
 ↪  2.22MB
```

Mismo letrero, otro ID, y nadie le avisó al cliente que la receta detrás de `estable` cambió. El digest, en cambio, sale del contenido mismo, y es lo que este libro usa para fijar busybox:

```
docker image ls --digests busybox:1.37.0
```

```
REPOSITORY   TAG       DIGEST
 ↪  IMAGE ID       CREATED       SIZE
busybox      1.37.0    sha256:9db7b59979c38555a39def84a31f
 ↪  b98b5296952f9e3afd4f6f11f05b07adfab0   9db7b59979c3
 ↪  2 years ago   6.77MB
```

Ese `sha256:9db7b599...` es la escritura notariada: quien descarga por digest recibe esos bytes, aunque mañana alguien vuelva a publicar otra cosa con el tag `1.37.0`.

Cuidado: fijar un tag específico en su manifiesto, por ejemplo `mi-app:v2`, no garantiza que reciba los mismos bytes cada vez que se descargue esa imagen. Nada impide que alguien vuelva a publicar contenido distinto bajo ese mismo tag `v2`, porque los

tags son apuntadores móviles, no huellas fijas. La única garantía real de "esto es lo mismo, byte por byte" la da el digest, no el tag; de hecho, cuando `imagePullPolicy` está ausente y usted sí específica el digest, Kubernetes fija automáticamente la política en `IfNotPresent`, precisamente porque un digest ya es inmutable y no hace falta revisarlo de nuevo en cada arranque.

Y hablando de `imagePullPolicy`, esa es la última pieza: decide cuándo la cocina vuelve a pedir la receta al registro en vez de usar la copia que ya tiene guardada. Con `IfNotPresent`, "la imagen se descarga solo si no está ya presente localmente"; con `Always`, cada vez que el kubelet arranca un contenedor "solicita al container runtime que descargue la imagen", y el runtime "resuelve el tag o el nombre de la imagen a un digest, y descarga cualquier capa que no esté ya en caché localmente"; con `Never`, "el kubelet no intenta descargar la imagen", y si no está ya presente localmente el arranque simplemente falla. Y cuando usted omite el campo por completo, Kubernetes decide por usted según cómo nombró la imagen: si usó `:latest` o no puso ningún tag, la política se fija en `Always`; si puso cualquier otro tag, o un digest, se fija en `IfNotPresent`. Esa regla explica por qué dos manifiestos que a simple vista se ven parecidos pueden comportarse muy distinto la próxima vez que se reinicie el Pod.

En el desarrollo local con kind, la receta no pasa por ningún registro: `kind load docker-image` la copia del docker de su máquina al containerd de cada nodo.

```
kind load docker-image sopa-del-dia:1.0 --name
 ↪  ckad-evidence
docker exec ckad-evidence-worker crictl images | grep
 ↪  sopa-del-dia
```

```
Image: "sopa-del-dia:1.0" with ID "sha256:48fa57f12af380a
 ↪  61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node
 ↪  "ckad-evidence-control-plane", loading...
```

```
Image: "sopa-del-dia:1.0" with ID "sha256:48fa57f12af380a
 ↪  61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node "ckad-evidence-worker",
 ↪  loading...
Image: "sopa-del-dia:1.0" with ID "sha256:48fa57f12af380a
 ↪  61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node "ckad-evidence-worker2",
 ↪  loading...
docker.io/library/sopa-del-dia                  1.0
 ↪  0a4a6a5aee0cf       2.22MB
```

El nodo la lista como `docker.io/library/sopa-del-dia` porque containerd normaliza el nombre corto, pero la imagen nunca estuvo en Docker Hub. El Deployment que la pone a trabajar en el namespace `la-cocina` declara `imagePullPolicy: IfNotPresent`, para que el kubelet use la copia que ya está en el nodo:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: el-restaurante
  namespace: la-cocina
spec:
  replicas: 2
  selector:
    matchLabels:
      app: el-restaurante
  template:
    metadata:
      labels:
        app: el-restaurante
    spec:
      containers:
      - name: cocinero
        image: sopa-del-dia:1.0
        imagePullPolicy: IfNotPresent
        ports:
```

```
        - containerPort: 8080
```

```
kubectl create namespace la-cocina
kubectl apply -f el-restaurante.yaml
kubectl -n la-cocina rollout status
 ↪  deployment/el-restaurante
kubectl -n la-cocina get pods -o wide
```

```
namespace/la-cocina created
deployment.apps/el-restaurante created
Waiting for deployment "el-restaurante" rollout to
 ↪  finish: 0 of 2 updated replicas are available...
Waiting for deployment "el-restaurante" rollout to
 ↪  finish: 1 of 2 updated replicas are available...
deployment "el-restaurante" successfully rolled out
NAME                              READY   STATUS
 ↪  RESTARTS   AGE   IP            NODE
 ↪  NOMINATED NODE   READINESS GATES
el-restaurante-6f7c867554-sphpw   1/1     Running   0
 ↪  1s    10.244.2.23   ckad-evidence-worker2   <none>
 ↪  <none>
el-restaurante-6f7c867554-xvnhm   1/1     Running   0
 ↪  1s    10.244.1.14   ckad-evidence-worker    <none>
 ↪  <none>
```

Para confirmar de dónde salió la imagen, se le pregunta a cada Pod su `imageID` y se le pide la carta al cocinero:

```
kubectl -n la-cocina get pods -o jsonpath='{range
 ↪  .items[*]}{.metadata.name}{"
 ↪  "}{.status.containerStatuses[0].imageID}{"\n"}{end}'
kubectl -n la-cocina exec deploy/el-restaurante -- wget
 ↪  -qO- http://localhost:8080
```

```
el-restaurante-6f7c867554-sphpw  docker.io/library/import
 ↪  -2026-10-09@sha256:83a21087dff924682c6a3669729792b65e
 ↪  f26ec6dd797dad0afc797bf9994c9f
el-restaurante-6f7c867554-xvnhm  docker.io/library/import
 ↪  -2026-10-09@sha256:83a21087dff924682c6a3669729792b65e
 ↪  f26ec6dd797dad0afc797bf9994c9f
sopa del dia de La Buena Sazon, v1
```

El `imageID` dice `import-2026-10-09`, que es la huella de `kind load` (una imagen bajada de un registro no se llamaría así), y el cocinero sirve la carta v1 que se empacó en la receta.

Ahora el contraejemplo: la misma receta, con el tag `latest`, cargada en los nodos de la misma forma, pero sin escribir la política.

```
docker tag sopa-del-dia:1.0 sopa-del-dia:latest
kind load docker-image sopa-del-dia:latest --name
 ↪  ckad-evidence
kubectl -n la-cocina run plato-del-dia
 ↪  --image=sopa-del-dia:latest
```

```
Image: "sopa-del-dia:latest" with ID "sha256:48fa57f12af3
 ↪  80a61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node
 ↪  "ckad-evidence-control-plane", loading...
Image: "sopa-del-dia:latest" with ID "sha256:48fa57f12af3
 ↪  80a61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node "ckad-evidence-worker",
 ↪  loading...
Image: "sopa-del-dia:latest" with ID "sha256:48fa57f12af3
 ↪  80a61d9fb86cabe0a068ea7f85a6b9de9f32144c77ada5885435"
 ↪  not yet present on node "ckad-evidence-worker2",
 ↪  loading...
pod/plato-del-dia created
```

Quince segundos después:

```
kubectl -n la-cocina get pod plato-del-dia -o jsonpath='{
 ↪  .spec.containers[0].imagePullPolicy}{"\n"}'
kubectl -n la-cocina get pod plato-del-dia
kubectl -n la-cocina get events --field-selector
 ↪  involvedObject.name=plato-del-dia -o
 ↪  custom-columns=REASON:.reason,MESSAGE:.message
```

```
Always
NAME            READY   STATUS             RESTARTS   AGE
plato-del-dia   0/1     ImagePullBackOff   0          15s
REASON      MESSAGE
Scheduled   Successfully assigned la-cocina/plato-del-dia
 ↪  to ckad-evidence-worker
Pulling     Pulling image "sopa-del-dia:latest"
Failed      Failed to pull image "sopa-del-dia:latest":
 ↪  failed to pull and unpack image
 ↪  "docker.io/library/sopa-del-dia:latest": failed to
 ↪  resolve reference
 ↪  "docker.io/library/sopa-del-dia:latest": pull access
 ↪  denied, repository does not exist or may require
 ↪  authorization: server message: insufficient_scope:
 ↪  authorization failed
Failed      Error: ErrImagePull
BackOff     Back-off pulling image "sopa-del-dia:latest"
Failed      Error: ImagePullBackOff
```

La política salió `Always` sin que nadie la escribiera, porque el tag es `latest`; el kubelet fue a buscar la imagen a docker.io aunque la tenía en el nodo, los eventos muestran el `ErrImagePull` (`pull access denied`), y el Pod quedó esperando en `ImagePullBackOff`. Mismo plato, otro letrero, y la cocina no abre.

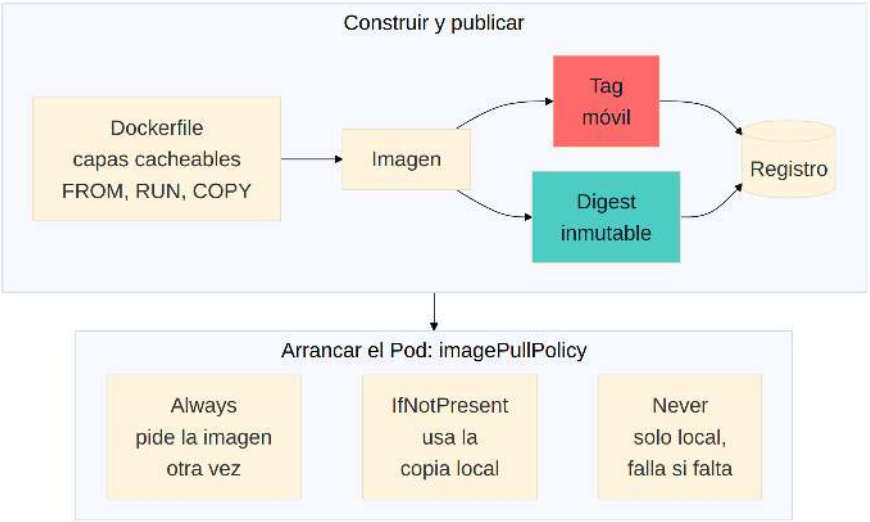


Figura 1.7. El camino de una imagen de contenedor: capas cacheables, tag móvil contra digest inmutable, y las tres políticas de imagePullPolicy al momento de arrancar el Pod.

> **Advertencia**
> Un tag como `mi-app:v2` no garantiza que siempre reciba los mismos bytes: es un apuntador que alguien puede volver a publicar con otro contenido. Solo el digest, un hash del contenido, da esa garantía. Para saber qué imagen corrió en realidad, `kubectl get pod <nombre> -o jsonpath='{.status.containerStatuses[0].imageID}'` le devuelve la referencia con su digest, del tipo `docker.io/library/nginx@sha256:...`.

> **Nota**
> El valor por defecto de `imagePullPolicy` depende del tag: es `Always` si el tag es `:latest`, e `IfNotPresent` en cualquier otro caso. Además, el campo no se puede cambiar en un Pod que ya existe; para cambiarlo hay que recrear el Pod.

Consejo
Cuando no recuerde un campo, `kubectl explain` es la documentación que tiene a mano en la terminal del examen. `kubectl explain pod.spec.containers.imagePullPolicy` le muestra los valores permitidos y el valor por defecto sin salir de la terminal.

1.5 Practique en el laboratorio

Temas del laboratorio `lab-part-01` (Parte I) para este capítulo:

- Empacar la receta del restaurante: construir una imagen con `docker` a partir de un `Dockerfile` que ya está en el disco.
- Llevar esa imagen al clúster `kind` con `kind load docker-image`, sin pedirle nada prestado a un registro público.
- Poner a trabajar la imagen como un Deployment de dos réplicas, sin forzar `imagePullPolicy: Always`.
- Comprobar que el contenedor en ejecución sale de esa imagen local y mirar el nombre con el que el nodo la guarda.
- Pedirle a la cocina qué carta está sirviendo y dejar el resultado anotado.

1.6 Resumen: la receta que llega intacta a cualquier cocina

Volvamos al local de la plaza de mercado. La receta que su mamá nunca escribió se volvió, en este capítulo, una imagen de contenedor: datos binarios que empaquetan la aplicación y todas sus dependencias, hechos para correr solos sin que ninguna cocina tenga que adivinar nada (documentación de Kubernetes, Container Images). Esa receta no se cocina de una sola pieza cada vez: se construye en capas que Docker reutiliza desde la caché siempre que el orden de las instrucciones lo permita, y cuando la preparación pesada no le sirve de nada al plato final, una construcción en varias etapas deja esa cocina de preparación atrás y solo entrega lo que el cliente pidió.

Y por último, esa receta necesita un nombre en la fachada: un tag, que es un letrero móvil que puede terminar apuntando a otra receta distinta con el tiempo, y un digest, que es la escritura notariada de esa versión exacta y nada más. La política `imagePullPolicy` decide cuándo la cocina vuelve a tocar la puerta del registro para pedir la

receta de nuevo, con reglas por defecto que dependen de si usted nombró la imagen con `:latest`, sin tag, con un tag fijo, o directamente con un digest.

Con la receta ya clara, el próximo capítulo deja de hablar de qué se cocina y empieza a hablar de qué tipo de negocio se monta con esa receta: uno de un solo día, uno que nunca cierra, uno con mesas reservadas, o uno replicado en cada esquina del barrio.

La Figura 1.7 resume ese recorrido completo, desde el Dockerfile hasta la decisión de imagePullPolicy en cada arranque.

Bibliografía

- Kubernetes documentation, Container Images https://kubernetes.io/docs/concepts/containers/images/
- Docker documentation, Building best practices https://docs.docker.com/build/building/best-practices/
- Docker documentation, docker pull (Pull an image by digest) https://docs.docker.com/engine/reference/commandline/pull/#pull-an-image-by-digest-immutable-identifier