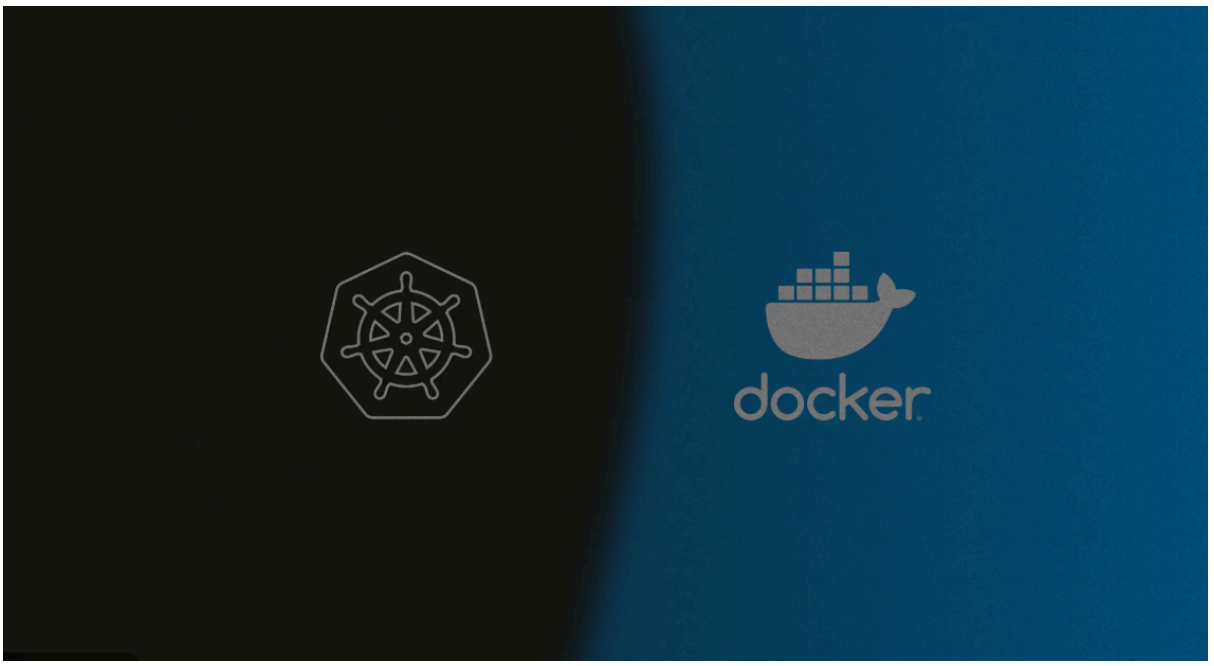
🐳 ☸ GUÍA DE DESPLIEGUE

Plataforma Cloud Native con Docker y Kubernetes



Arquitectura moderna basada en microservicios, orquestación y observabilidad

Incluye:

- ✔ **Docker & Docker Compose**
- ✔ **Kubernetes (k3d)**
- ✔ **NGINX Ingress Controller**
- ✔ **Load Balancing con MetalLB**
- ✔ **Monitorización con Prometheus & Grafana**
- ✔ **TLS automático con Cert-Manager**
- ✔ **Arquitectura escalable y segura**

Autor: Luis Rodríguez
Especialización: DevOps & Cloud Infrastructure
Proyecto: Wellness Platform Deployment
Fecha: 2026

📦 1. Preparación del entorno Docker

Verificar instalación de Docker

Confirma que Docker esté instalado:

```
tester@vbox:~$ docker --version
Docker version 29.1.3, build f52814d
tester@vbox:~$ kubectl version --client
Client Version: v1.35.0
Kustomize Version: v5.7.1
tester@vbox:~$ helm version
version.BuildInfo{Version:"v3.20.0", GitCommit:"b2e4314fa0f229a1de7b4c981273f61d69ee5a59", GitTreeState:"clean", GoVersion:"go1.25.6"}
```

Se recomienda usar una versión reciente y estable.

Clonar el repositorio

git clone https://github.com/luisrodvilladaorg/wellnes-ops.git

cd wellnes-ops/
ls -l

```
tester@vbox:~$ git clone https://github.com/luisrodvilladaorg/wellnes-ops.git
Cloning into 'wellnes-ops'...
remote: Enumerating objects: 9205, done.
remote: Counting objects: 100% (85/85), done.
remote: Compressing objects: 100% (58/58), done.
remote: Total 9205 (delta 43), reused 54 (delta 27), pack-reused 9120 (from 3)
Receiving objects: 100% (9205/9205), 24.21 MiB | 5.96 MiB/s, done.
Resolving deltas: 100% (2395/2395), done.
tester@vbox:~$
```

```
tester@vbox:~$ cd wellnes-ops/
tester@vbox:~/wellnes-ops$ ls -l
total 1404
drwxr-xr-x  4 tester tester    4096 Feb 13 10:13 backend
drwxr-xr-x  2 tester tester    4096 Feb 13 10:13 db
-rwxr-xr-x  1 tester tester    2973 Feb 13 10:13 docker-compose.dev.yml
-rwxr-xr-x  1 tester tester    1434 Feb 13 10:13 docker-compose.prod.yml
-rwxr-xr-x  1 tester tester    1948 Feb 13 10:13 docker-compose.yml
-rwxr-xr-x  1 tester tester     241 Feb 13 10:13 Dockerfile.dev
drwxr-xr-x  3 tester tester    4096 Feb 13 10:13 docs
drwxr-xr-x  3 tester tester    4096 Feb 13 10:13 frontend
-rw-r--r--  1 tester tester    8470 Feb 13 10:13 HTTPS.md
-rw-r--r--  1 tester tester     357 Feb 13 10:13 k3d-cluster.yml
drwxr-xr-x 10 tester tester    4096 Feb 13 10:13 k8s
-rw-r--r--  1 tester tester   10174 Feb 13 10:13 LICENSE
drwxr-xr-x  3 tester tester    4096 Feb 13 10:13 monitoring
drwxr-xr-x  3 tester tester    4096 Feb 13 10:13 nginx
-rw-r--r--  1 tester tester    8093 Feb 13 10:13 README.md
-rwxr-xr-x  1 tester tester    1119 Feb 13 10:13 setup-local.sh
-rwxr-xr-x  1 tester tester 1347871 Feb 13 10:13 Wellnes-ops-guide.pdf
```

🔧 Configuración de Variables de Entorno (PostgreSQL y Backend)

Antes de iniciar los contenedores, es necesario definir las variables de entorno que utilizarán:

- la base de datos PostgreSQL
- el servicio Backend
- la conexión entre ambos servicios

Estas variables permiten configurar credenciales, puertos y parámetros sin modificar el código fuente.

📄 Archivo de ejemplo incluido en el repositorio:

.env.example

Este archivo contiene una plantilla con las variables necesarias para el funcionamiento del sistema.

Se utiliza como base para crear el archivo `.env` que será usado en el entorno local.

📁 Crear el directorio de variables de entorno

```
sudo mkdir -p env/dev
```

📄 Copiar la plantilla de variables

```
sudo cp .env.example env/dev/.env
```

Revisar y editar las variables

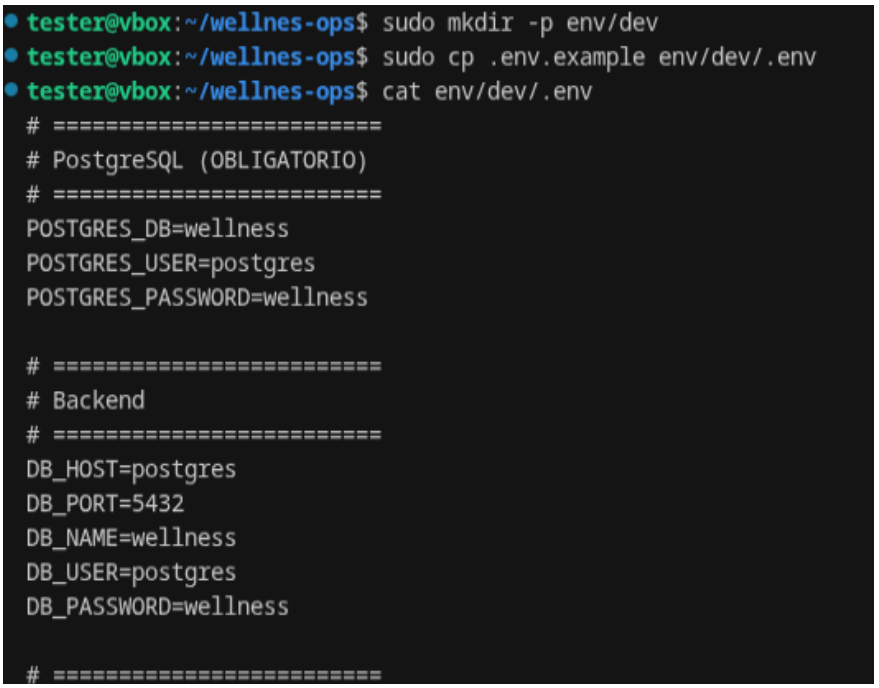
Puedes visualizar el contenido:

```
cat env/dev/.env
```

Dentro encontrarás variables como:

```
POSTGRES_DB=
POSTGRES_USER=
POSTGRES_PASSWORD=
DATABASE_URL=
BACKEND_PORT=
```

👉 Ajusta valores si necesitas personalizar tu entorno.



¿Por qué usar variables de entorno?

✔ Separan configuración del código

✔ Permiten múltiples entornos (dev / prod)

✔ Mejoran la seguridad

✔ Facilitan despliegues automatizados

✔ Evitan exponer credenciales en el repositorio

🚀 Iniciar el stack de contenedores

Una vez configuradas las variables, podemos iniciar la plataforma.

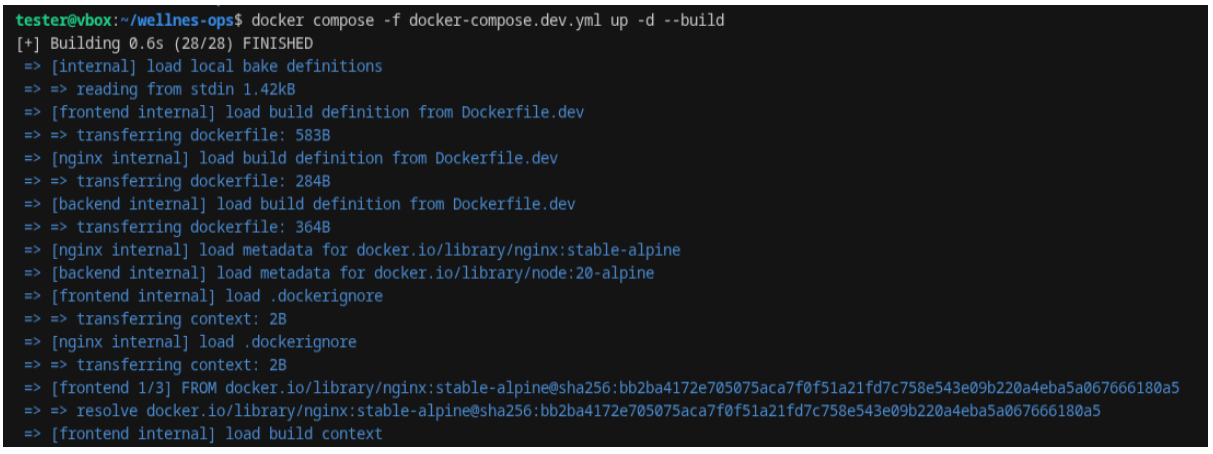
Este proyecto utiliza **Docker Compose** para levantar todos los servicios.

▶️ Levantar entorno de desarrollo

docker compose -f docker-compose.dev.yml up -d --build

🔎 Qué hace este comando:

✔ construye las imágenes Docker
✔ crea la red interna una para Postgres y Backend y otra red separada para Frontend y Nginx
✔ inicia PostgreSQL
✔ inicia Backend
✔ inicia Frontend
✔ inicia NGINX
✔ aplica variables del archivo `.env`
✔ ejecuta contenedores en segundo plano



🌍 Entornos disponibles

En este ejemplo se utiliza el entorno:

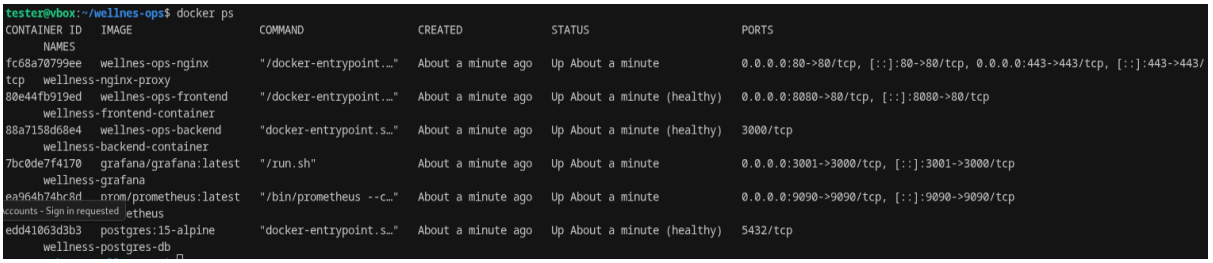
dev (desarrollo)

El entorno prod puede iniciarse de la misma forma cambiando el archivo compose correspondiente, dependiendo del escenario de despliegue

⏳ Verificar que los contenedores están corriendo

Deberías ver los servicios activos.

docker ps



🔍 Verificación del Backend y Conectividad Interna

Por motivos de seguridad, el servicio **backend** no está expuesto directamente al exterior.

Para comprobar que está funcionando correctamente, debemos acceder al contenedor y probar el endpoint interno.

Acceder al contenedor del backend

👉 Este comando abre una terminal dentro del contenedor backend.

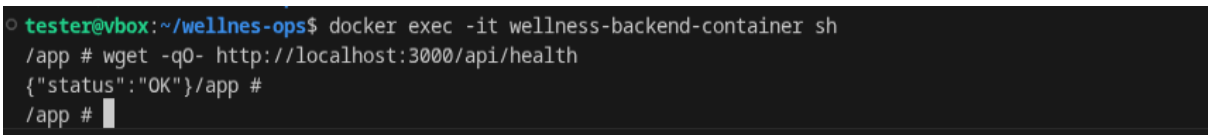
docker exec -it wellness-backend-container sh

Probar el endpoint de salud interno

wget -qO- http://localhost:3000/api/health

✔ Resultado esperado:

Debe devolver una respuesta indicando que el servicio está activo, por ejemplo: **OK**



🌐 Acceso seguro a la aplicación

Una vez verificado el backend, puedes acceder a la aplicación desde tu navegador.

Abrir

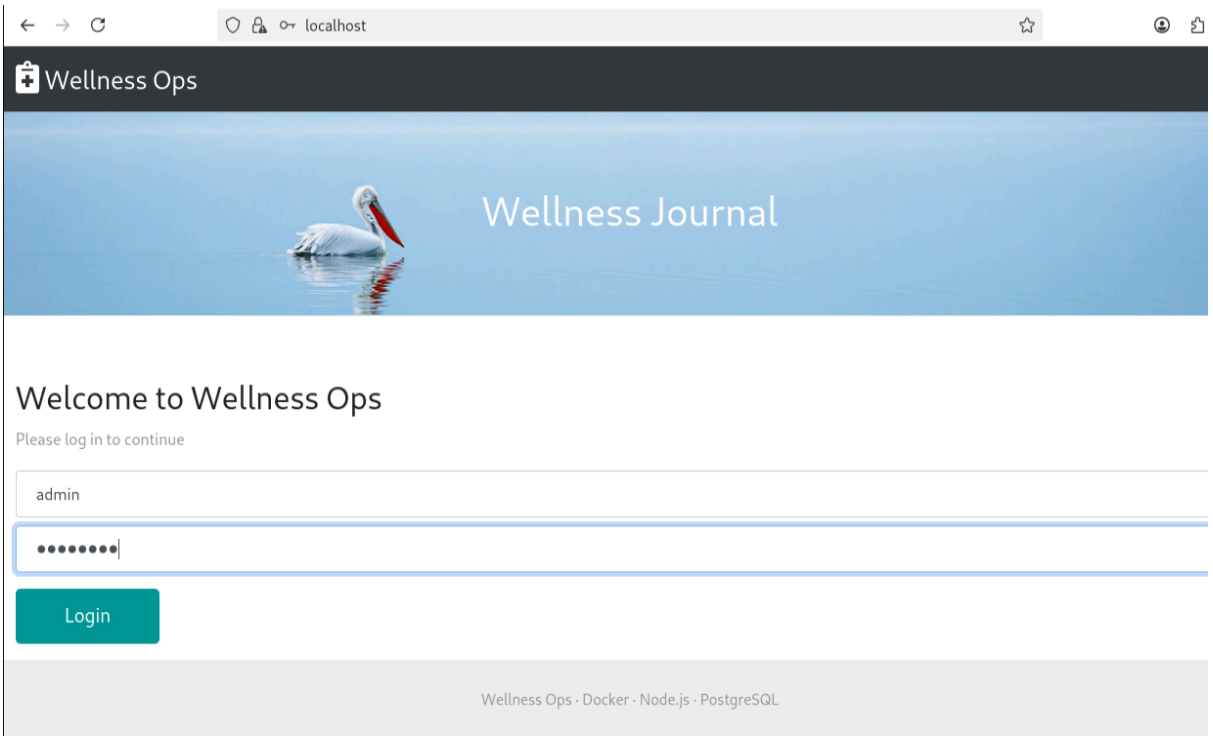
https://localhost

👉 La conexión se realiza mediante HTTPS, garantizando comunicación cifrada.

🔐 Credenciales por defecto (entorno desarrollo)

Usuario: admin
Contraseña: admin123

⚠️ Estas credenciales son solo para desarrollo.



💾 Verificar persistencia de datos

Para confirmar que la base de datos funciona correctamente:

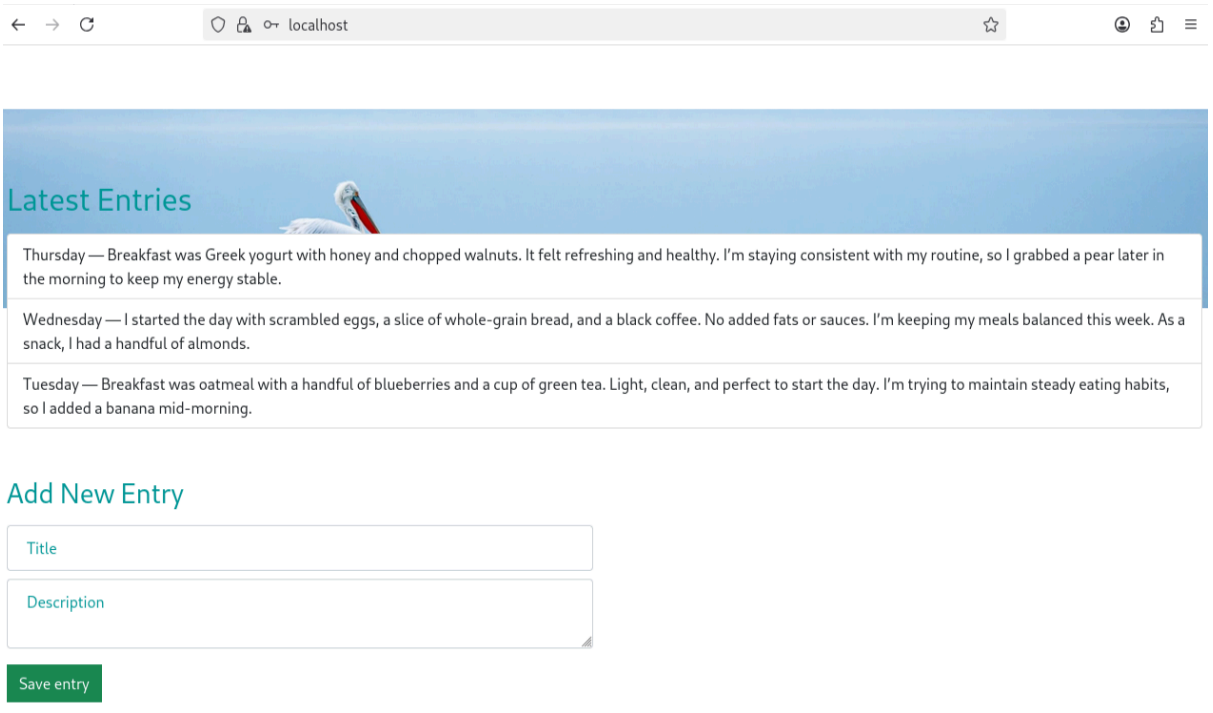
1️⃣ Crear nuevos registros desde la interfaz.
2️⃣ Guardar los cambios.
3️⃣ Refrescar la página o cerrar sesión.
4️⃣ Volver a entrar haciendo logout y login

✔ Resultado esperado:

Los datos deben permanecer almacenados.

👉 Esto confirma que:

✔ PostgreSQL está funcionando
✔ el backend guarda correctamente los datos
✔ los volúmenes Docker persisten la información



🧪 Qué estás validando realmente

Este proceso verifica:

✅ conectividad frontend → backend
✅ backend → base de datos
✅ persistencia de datos
✅ funcionamiento del sistema completo

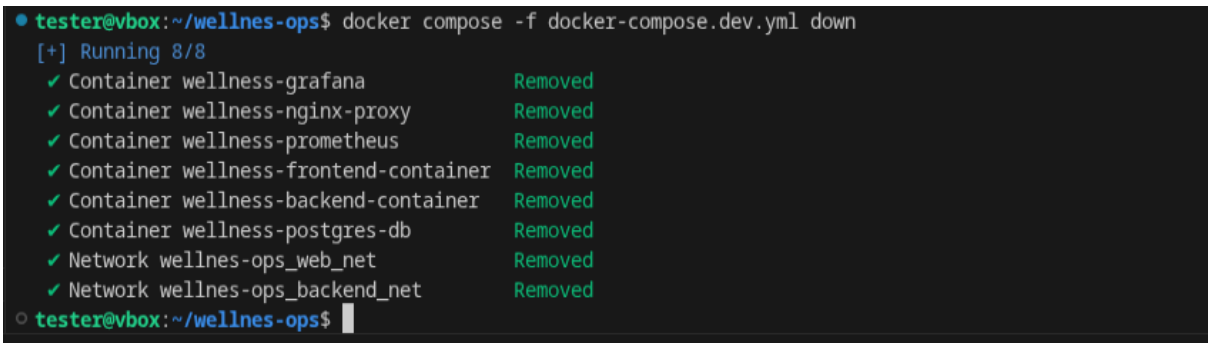
🛑 Detener el entorno antes de continuar

Una vez validado que todo funciona correctamente:

👉 **Este comando:**

✔ detiene los contenedores
✔ elimina la red creada
✔ mantiene los volúmenes de datos
✔ libera recursos del sistema

docker compose -f docker-compose.dev.yml down



✅ Resultado esperado

Ahora tienes:

✔ stack validado correctamente
✔ backend funcionando
✔ persistencia confirmada
✔ entorno listo para migrar a Kubernetes

☸️ Despliegue en Kubernetes

Después de validar la aplicación con Docker Compose, el siguiente paso es desplegar la plataforma en Kubernetes.

Esto permite:

✔ alta disponibilidad

✔ escalabilidad

✔ balanceo de carga

✔ automatización del despliegue

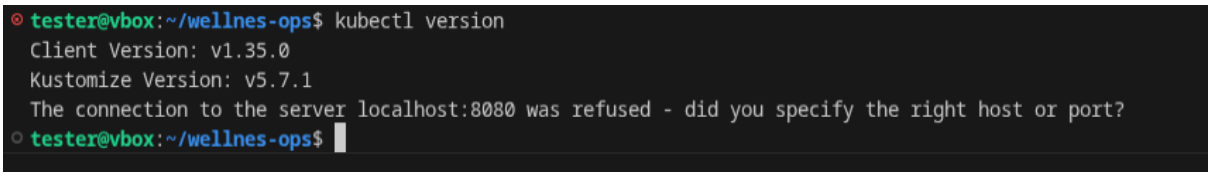
✔ arquitectura cloud-native

🔧 Verificar instalación de kubectl

Antes de interactuar con Kubernetes, debemos confirmar que la herramienta CLI está disponible.

kubectl version

👉 Este comando verifica que kubectl está instalado correctamente.



☸️ Crear el clúster Kubernetes local (k3d)

Para entornos locales utilizamos **k3d**, que permite ejecutar Kubernetes dentro de Docker.

Esto facilita pruebas rápidas sin necesidad de infraestructura cloud.

Crear el clúster

```
k3d cluster create cluster-wellness-local \
--api-port 6443 \
--servers 1 \
--agents 0 \
--port 80:80@loadbalancer \
--port 443:443@loadbalancer \
--k3s-arg "--disable=traefik@server:0"
```

🔍 Qué hace este comando

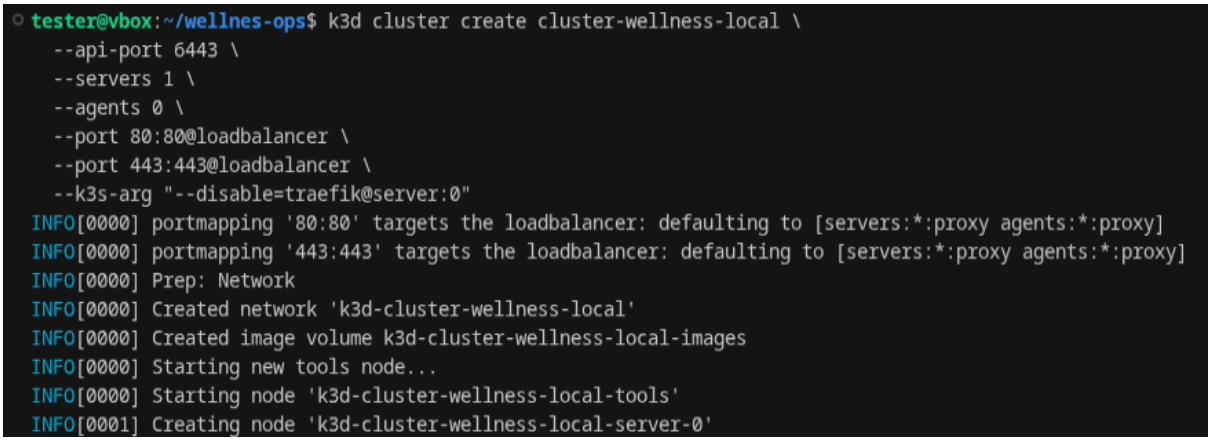
✔ crea un clúster Kubernetes local

✔ expone los puertos 80 y 443

✔ habilita el balanceador de carga interno

✔ desactiva Traefik (usaremos NGINX Ingress)

✔ permite acceso a servicios HTTP y HTTPS



🔐 Configurar acceso al clúster

Crear directorio de configuración

mkdir -p ~/.kube

Obtener configuración del cluster

👉 **Esto permite que kubectl se conecte al clúster correcto.**

k3d kubeconfig get cluster-wellness-local > ~/.kube/config

Ajustar permisos de seguridad

👉 **Solo el usuario actual podrá leer el archivo.**

chmod 600 ~/.kube/config

Verificar contexto activo

kubectl config get-contexts

🔍 Verificar estado del clúster

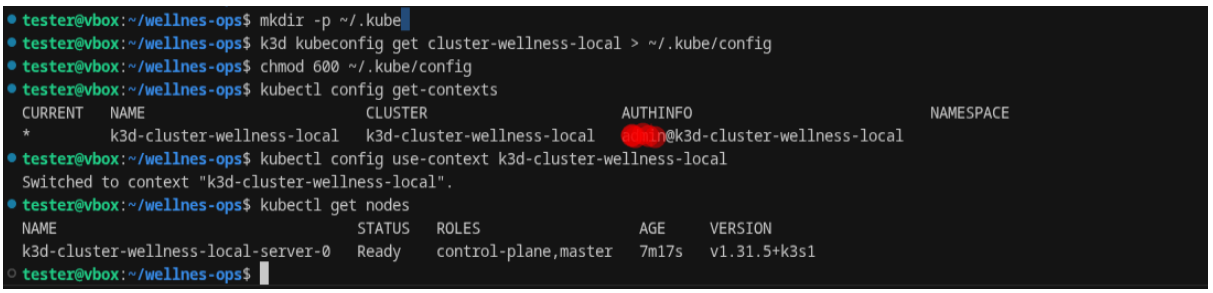
Confirmar que el cluster está activo

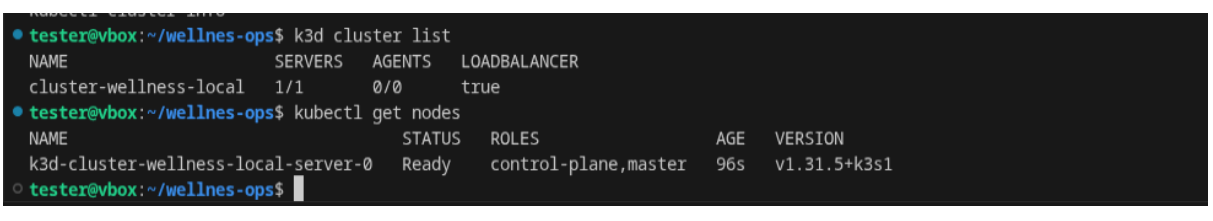
k3d cluster list

Verificar nodos

kubectl get nodes

👉 **Esto confirma que el plano de control está operativo.**





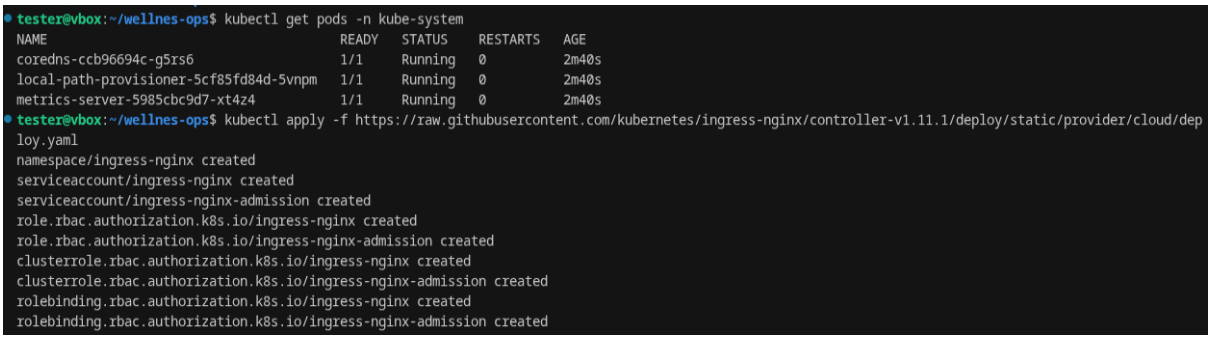
Verificar componentes del sistema

kubectl get pods -n kube-system

👉 Este comando muestra los componentes internos de Kubernetes:

- DNS
- CNI (red)
- controladores
- servicios del sistema

Todos deben estar en estado **Running**.



Qué hemos logrado hasta ahora

✔ clúster Kubernetes funcionando

✔ acceso configurado

✔ nodos operativos

✔ servicios internos activos

⚙️ Instalación y uso de Helm en Kubernetes

🧠 ¿Qué es Helm?

Helm es el gestor de paquetes de Kubernetes.

Funciona de forma similar a:

- apt en Linux
- npm en Node.js
- pip en Python

Permite instalar aplicaciones complejas usando plantillas reutilizables llamadas Charts.

🎯 ¿Por qué usamos Helm?

Helm permite:

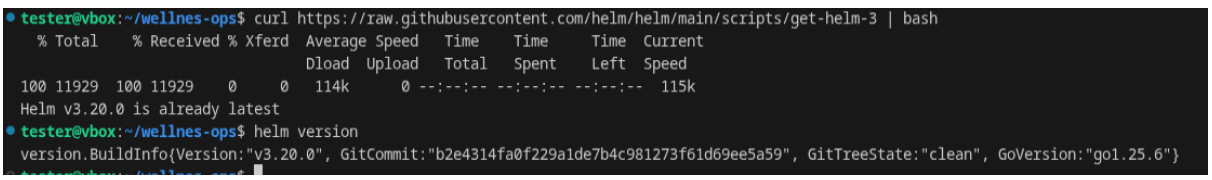
✔ desplegar aplicaciones complejas rápidamente
✔ gestionar configuraciones fácilmente
✔ actualizar servicios sin interrupciones
✔ reutilizar plantillas
✔ simplificar despliegues en producción

🛠️ Instalar Helm

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

Este script oficial:

✔ descarga Helm
✔ lo instala en el sistema
✔ configura el binario automáticamente



🌐 Añadir repositorio oficial del Ingress NGINX

Antes de instalar componentes, debemos añadir su repositorio Helm.

helm repo update

helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

```
tester@vbox:~/wellnes-ops$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "ingress-nginx" chart repository
Update Complete. ⎈Happy Helming!⎈
tester@vbox:~/wellnes-ops$
```

🧠 ¿Por qué necesitamos un Ingress Controller?

Por defecto, Kubernetes no expone servicios HTTP externamente.

El Ingress Controller:

✔ gestiona tráfico HTTP/HTTPS
✔ enruta solicitudes a los servicios correctos
✔ permite múltiples aplicaciones bajo una misma IP
✔ habilita TLS y certificados

👉 Es equivalente a un reverse proxy inteligente.

🚀 Instalar NGINX Ingress Controller

helm install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx \
  --create-namespace \
  --set controller.watchIngressWithoutClass=true

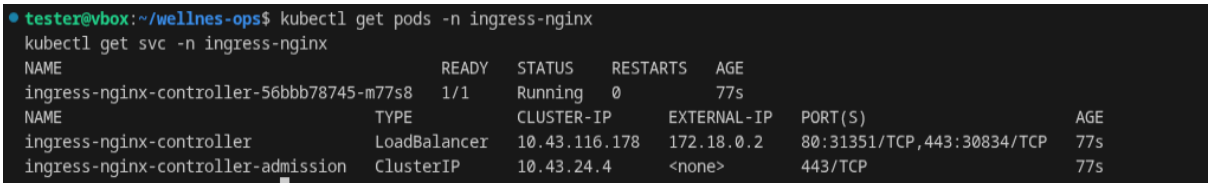
```
tester@vbox:~/wellnes-ops$ helm install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx \
  --create-namespace \
  --set controller.watchIngressWithoutClass=true
NAME: ingress-nginx
LAST DEPLOYED: Mon Feb  9 12:48:43 2026
NAMESPACE: ingress-nginx
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The ingress-nginx controller has been installed.
It may take a few minutes for the load balancer IP to be available.
You can watch the status by running 'kubectl get service --namespace ingress-nginx ingress-nginx-controller --output wide --watch'

An example Ingress that makes use of the controller:
  apiVersion: networking.k8s.io/v1
  kind: Ingress
  metadata:
    name: example
    namespace: foo
  spec:
    ingressClassName: nginx
```

⏳ Verificar instalación

kubectl get pods -n ingress-nginx

✔ Resultado esperado:

Pods en estado: Running



🌐 MetalLB: Load Balancer para Kubernetes Local

🧠 ¿Qué es MetalLB?

MetalLB proporciona funcionalidad de **LoadBalancer externo** en clústeres Kubernetes que no están en la nube.

En proveedores cloud (AWS, Azure, GCP), el LoadBalancer se crea automáticamente.

En local → necesitamos MetalLB.

🎯 ¿Por qué necesitamos MetalLB?

Permite:

✔ asignar IP externas a servicios

✔ acceder a aplicaciones desde el navegador

✔ simular un entorno cloud real

✔ balancear tráfico hacia los pods

✔ trabajar con Ingress correctamente

👉 Sin MetalLB no tendrías IP externa.

🚀 Instalar MetalLB

🔍 Qué hace este comando

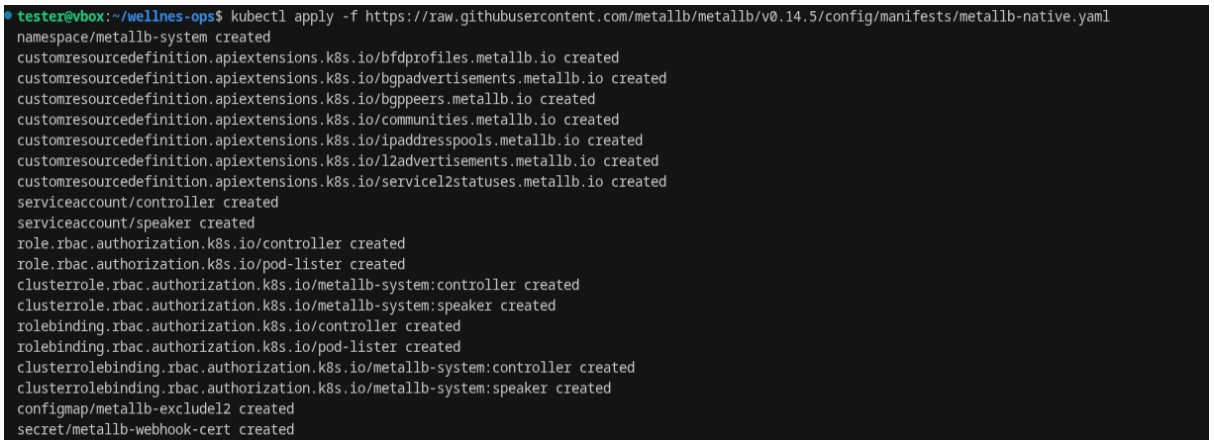
✔ instala el controlador MetalLB

✔ crea el namespace metallb-system

✔ despliega los controladores necesarios

✔ habilita el balanceo de carga

kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.14.5/config/manifests/metallb-native.yaml



⏳ Verificar instalación

Con el siguiente comando comprobamos que se haya instalado correctamente.

Resultado esperado, pods = Running.

kubectl get pods -n metallb-system

```
tester@vbox:~/wellnes-ops$ kubectl get pods -n metallb-system
NAME                          READY   STATUS    RESTARTS   AGE
controller-58859b4d4f-zn4qc   1/1     Running   0          53s
speaker-wzqtw                 1/1     Running   0          53s
tester@vbox:~/wellnes-ops$
```

⚙️ Configurar el pool de direcciones IP

MetalLB necesita un rango de IPs para asignarlas a los servicios.

Aplicamos el manifiesto que incluye la configuración.

kubectl apply -f /k8s/metallb/

🔍 Verificar que el pool fue creado

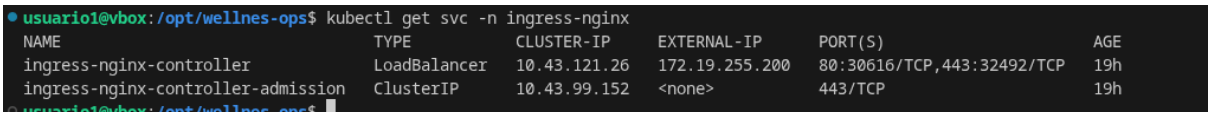
kubectl get ipaddresspools -n metallb-system

```
tester@vbox:~/wellnes-ops$ kubectl apply -f k8s/metallb/
l2advertisement.metallb.io/local-l2 created
ipaddresspool.metallb.io/local-pool created
tester@vbox:~/wellnes-ops$ kubectl get ipaddresspools -n metallb-system
NAME         AUTO ASSIGN   AVOID BUGGY IPS   ADDRESSES
local-pool   true          false             ["172.19.255.200-172.19.255.250"]
```

🌍 Verificar asignación de IP externa

Ahora revisa el servicio del Ingress:

👉 Esa IP permitirá acceder a tu aplicación desde el navegador.



🧠 Qué acabamos de habilitar

✔ LoadBalancer funcional en entorno local

✔ IP externa para acceder al clúster

✔ balanceo de tráfico hacia los pods

✔ simulación real de entorno cloud

🎯 Cómo funciona el flujo ahora

Cliente → IP MetalLB → Ingress NGINX → Servicio → Pods

📊 Monitorización con Prometheus y Grafana

Una vez que el clúster está operativo y el tráfico externo configurado, el siguiente paso es habilitar la monitorización.

La monitorización permite:

✔ supervisar el rendimiento del clúster
✔ detectar problemas de recursos
✔ analizar consumo de CPU y memoria
✔ visualizar métricas en tiempo real
✔ mejorar la disponibilidad del sistema

En este entorno utilizaremos:

- **Prometheus → recopilación de métricas**
- **Grafana → visualización y dashboards**

➕ Añadir el repositorio Helm de Prometheus

Primero añadimos el repositorio oficial que contiene los charts de monitorización:

helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

📁 Crear un namespace dedicado para monitorización

Separar los componentes de monitorización mejora la organización y seguridad.

kubectl create namespace monitoring

🚀 Instalar Prometheus y Grafana

Instalamos el stack completo:

helm install monitoring prometheus-community/kube-prometheus-stack -n monitoring

🔍 Qué instala este chart

✔ Prometheus Server
✔ Alertmanager
✔ Grafana
✔ Node Exporter
✔ kube-state-metrics
✔ ServiceMonitors
✔ dashboards preconfigurados

👉 Es una solución completa de observabilidad.

```
tester@vbox:~/wellnes-ops$ helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
"prometheus-community" has been added to your repositories
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "ingress-nginx" chart repository
...Successfully got an update from the "prometheus-community" chart repository
Update Complete. ⎈Happy Helming!⎈
tester@vbox:~/wellnes-ops$ kubectl create namespace monitoring
namespace/monitoring created
tester@vbox:~/wellnes-ops$ helm install monitoring prometheus-community/kube-prometheus-stack -n monitoring
NAME: monitoring
LAST DEPLOYED: Fri Feb 13 16:37:07 2026
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
```

🔎 Verificar que Prometheus está operativo

kubectl get pods -n monitoring

También puedes comprobar los servicios:

kubectl get svc -n monitoring

```
tester@vbox:~/wellnes-ops$ kubectl get pods -n monitoring
NAME                                                READY   STATUS    RESTARTS   AGE
prometheus-alertmanager-0                           1/1     Running   0          97s
prometheus-kube-state-metrics-74bb7cc548-f722f      1/1     Running   0          97s
prometheus-prometheus-node-exporter-slln7           1/1     Running   0          97s
prometheus-prometheus-pushgateway-665f7ffd4c-5qgls  1/1     Running   0          97s
prometheus-server-6496dc96-wvttx                    2/2     Running   0          97s
tester@vbox:~/wellnes-ops$
```

📊 Verificación de CRDs y configuración de monitorización

El stack `kube-prometheus-stack` instala recursos personalizados (CRDs) que permiten a Prometheus descubrir automáticamente servicios a monitorizar.

Uno de los más importantes es:

👉 **ServiceMonitor**

Este recurso define qué servicios deben ser monitoreados.

🔎 Verificar que los CRDs están instalados

```
kubectl get crds | grep servicemonitor
```

Debes ver algo similar a: servicemonitors.monitoring.coreos.com

🧠 ¿Qué es un ServiceMonitor?

Un ServiceMonitor permite:

✔ definir endpoints a monitorizar
✔ indicar puertos y rutas de métricas
✔ automatizar el descubrimiento de servicios
✔ integrar aplicaciones con Prometheus

👉 Es el método estándar en Kubernetes moderno.

📥 Aplicar los manifiestos de monitorización del proyecto

Tu repositorio incluye configuraciones específicas para monitorizar los servicios de la aplicación.

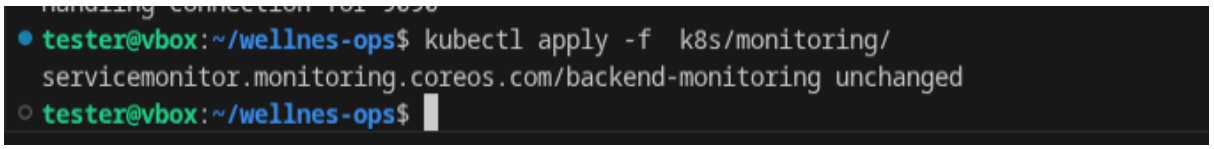
Aplicarlas permite que Prometheus recoja métricas del backend, frontend u otros componentes.

Ejecuta:

```
kubectl apply -f k8s/monitoring/
```

🔍 Qué hace este paso

✔ crea ServiceMonitors personalizados
✔ registra endpoints de métricas
✔ integra la aplicación con Prometheus
✔ habilita observabilidad específica



⏳ Verificar recursos creados

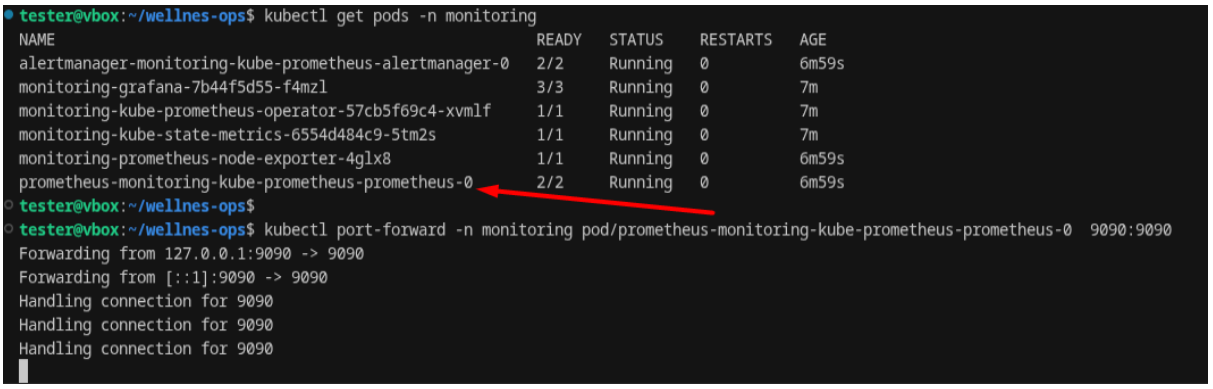
Puedes comprobar: kubectl get servicemonitors -n monitoring

🌐 Exponer la interfaz de Prometheus

Para acceder a la interfaz web de Prometheus desde tu equipo local utilizamos **port-forward**.

Ejecuta:

kubectl port-forward -n monitoring pod/prometheus-stac-prometheus-0 9090:9090



Acceder a Prometheus

Abrir en el navegador:

http://localhost:9090

📈 Verificar que Prometheus está recolectando métricas

Dentro de la interfaz:

👉 **Status → Targets**

Debes ver servicios con estado: UP

👉 **Esto confirma que Prometheus está recopilando métricas correctamente.**

🧠 Qué hemos logrado

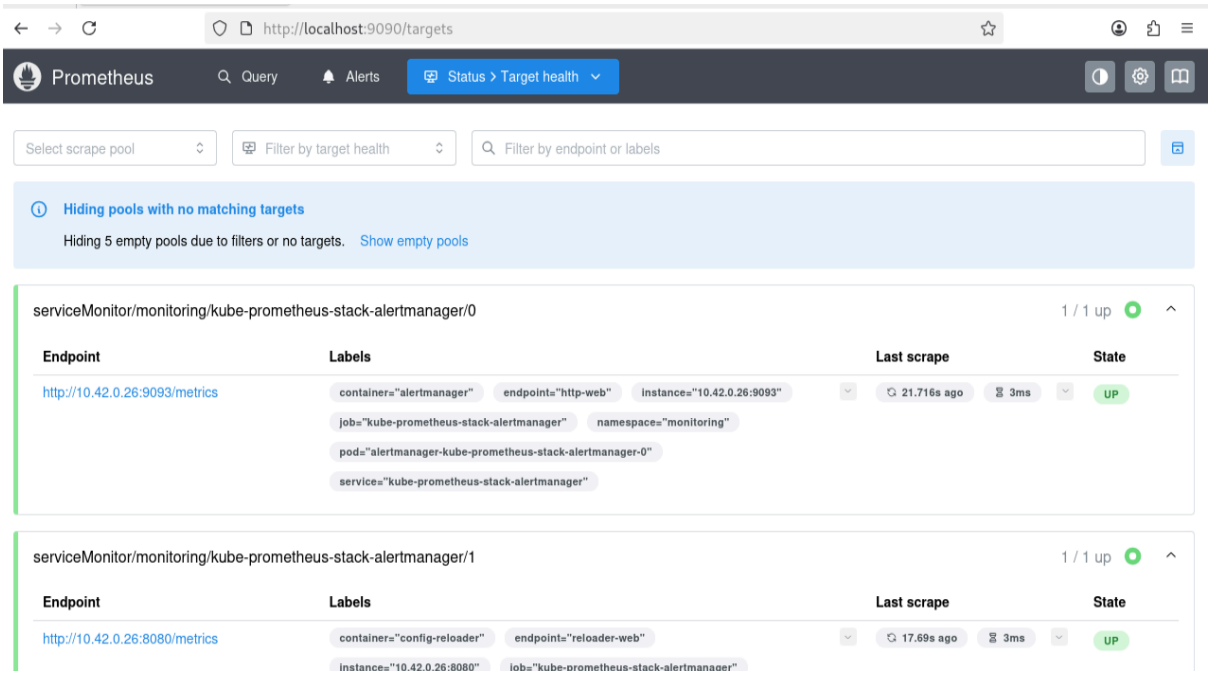
✔ CRDs instalados correctamente

✔ ServiceMonitor habilitado

✔ monitorización personalizada activa

✔ Prometheus recolectando métricas

✔ observabilidad lista



📊 Acceso y verificación de Grafana

Grafana se instala automáticamente junto con el stack de Prometheus y proporciona dashboards para visualizar las métricas del clúster y las aplicaciones.

Permite:

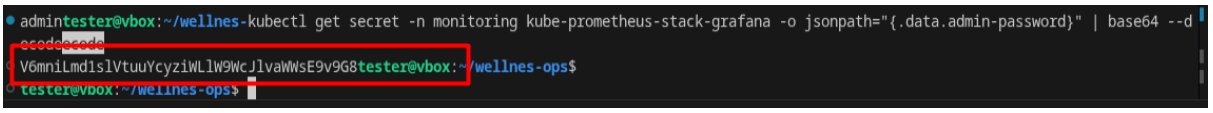
✔ visualizar métricas en tiempo real
✔ analizar rendimiento del sistema
✔ detectar cuellos de botella
✔ monitorizar servicios y nodos
✔ crear paneles personalizados

🔐 Obtener la contraseña de administrador

Grafana genera automáticamente una contraseña almacenada en un secreto de Kubernetes.

Ejecuta:

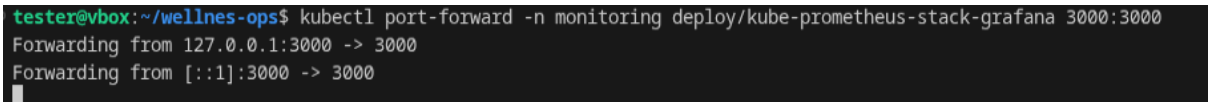
kubectl get secret -n monitoring kube-prometheus-stack-grafana -o jsonpath="{.data.admin-password}" | base64 --decode



🌐 Exponer Grafana localmente

Para acceder a Grafana desde el navegador utilizamos port-forward:

kubectl port-forward -n monitoring deploy/kube-prometheus-stack-grafana 3000:3000



🌍 Acceder al panel web

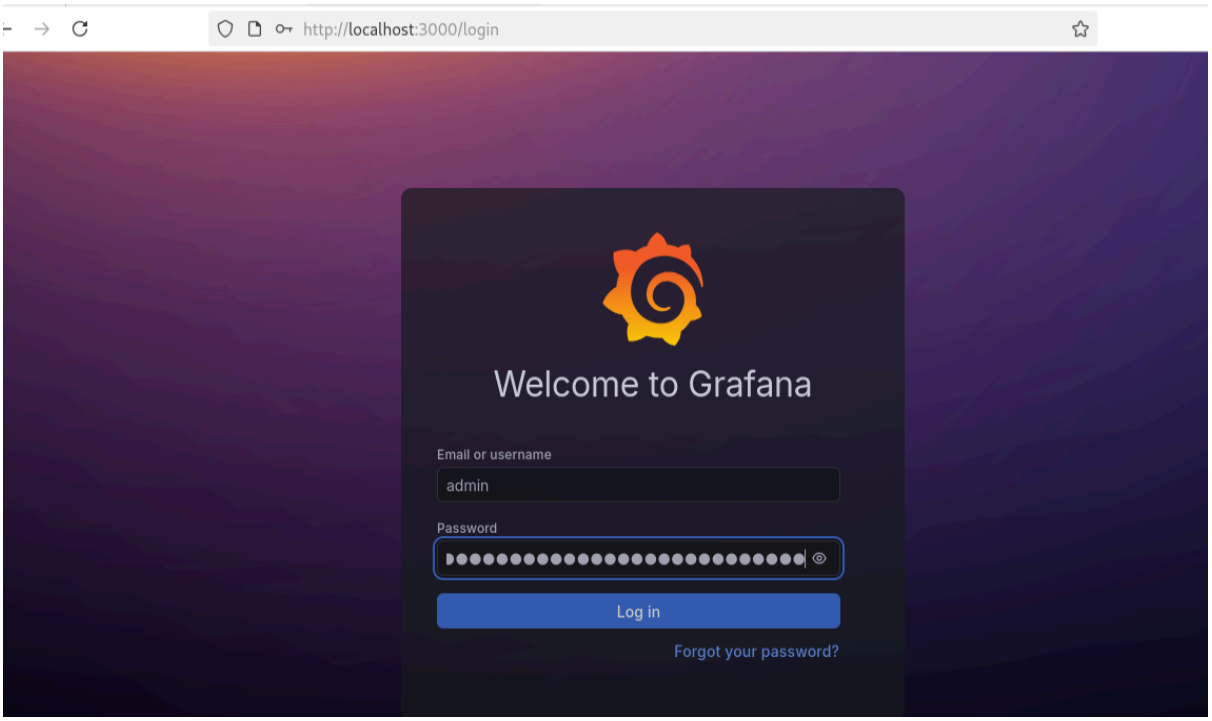
Abre en el navegador:

http://localhost:3000

🔑 Credenciales de acceso

Usuario: admin

Contraseña: la obtenida en el paso anterior



🚀 Despliegue de la aplicación en Kubernetes

Con el clúster operativo y la monitorización activa, ahora desplegaremos los componentes de la aplicación dentro de Kubernetes.

Este proceso crea:

✔ base de datos PostgreSQL

✔ backend API

✔ frontend web

✔ NGINX como punto de entrada

⚠️ Importante: orden de despliegue

Los manifiestos deben aplicarse **en orden**, esperando unos segundos entre cada paso.

Esto es necesario porque:

- el **backend depende de la base de datos**
- el **frontend depende del backend**
- el **Ingress NGINX enruta tráfico hacia el frontend**

Aunque los componentes se ejecuten en namespaces aislados, sus dependencias siguen siendo críticas.

📦1️⃣Desplegar PostgreSQL

kubectl apply -f k8s/postgres/

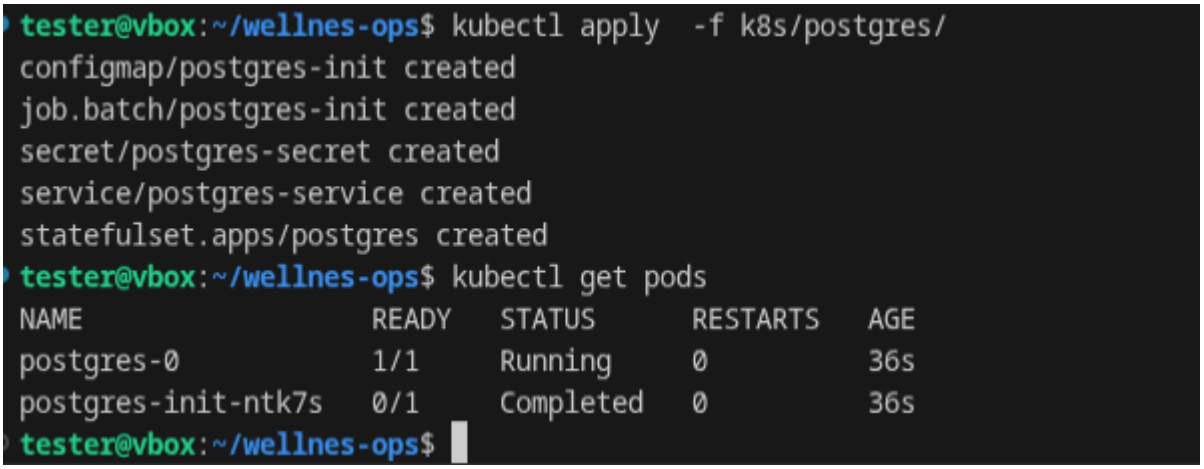
⏳ Esperar 30 segundos

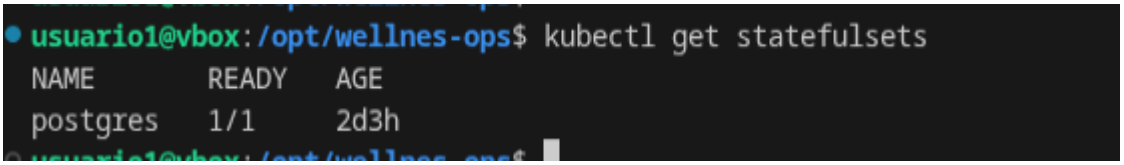
Esto permite que:

✔ el contenedor inicie correctamente

✔ el volumen persistente se monte

✔ la base de datos quede lista





📦2️⃣ Desplegar Backend

kubectl apply -f k8s/backend/

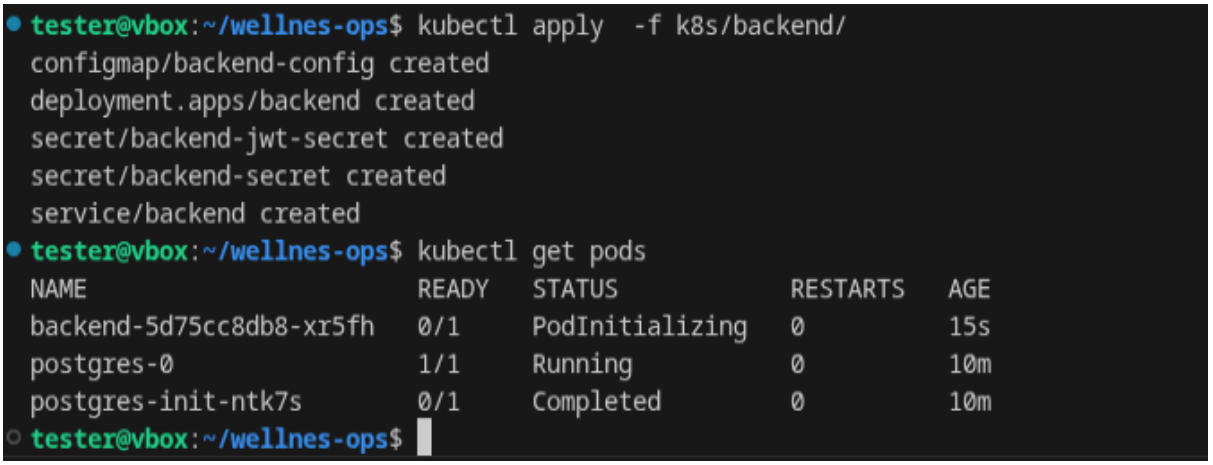
⏳ **Esperar 30 segundos**

El backend necesita:

✔ conectarse a PostgreSQL

✔ inicializar variables de entorno

✔ abrir el puerto del servicio



📦3️⃣Desplegar Frontend

kubectl apply -f k8s/frontend/

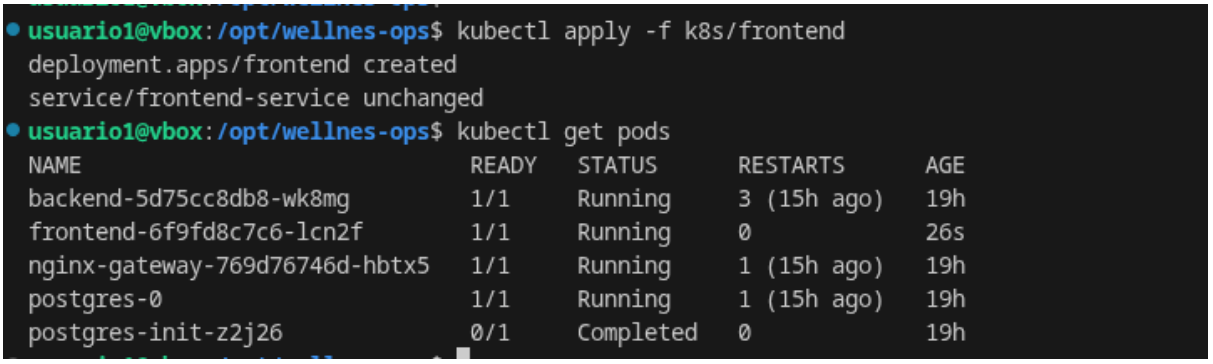
⏳ Esperar 30 segundos

Esto permite:

✔ cargar la aplicación web

✔ conectar con el backend

✔ iniciar el servidor web



📦4️⃣ Desplegar NGINX

kubectl apply -f k8s/nginx/

⏳ Esperar 30 segundos

👉 Este paso habilita el acceso externo a la aplicación.

El Ingress se encargará de enrutar el tráfico HTTP/HTTPS hacia el frontend.

```
tester@vbox:~/wellnes-ops$ kubectl apply -f k8s/nginx/
configmap/nginx-gateway-config created
deployment.apps/nginx-gateway created
service/nginx-gateway created
tester@vbox:~/wellnes-ops$
```

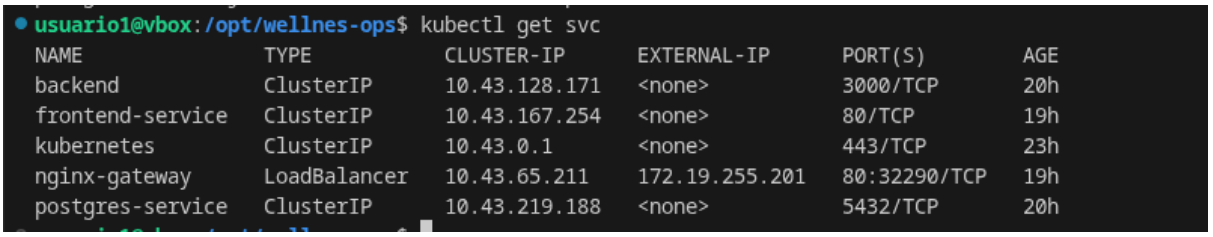
🔍 Verificar que todos los componentes están activos

```
tester@vbox:~/wellnes-ops$ kubectl get pods
NAME                             READY   STATUS      RESTARTS      AGE
backend-5d75cc8db8-xr5fh         1/1     Running     0             82s
frontend-6f9fd8c7c6-zqj9l        1/1     Running     0             11s
nginx-gateway-769d76746d-dwvgw   1/1     Running     2 (26s ago)   33s
postgres-0                       1/1     Running     0             11m
postgres-init-ntk7s              0/1     Completed   0             11m
```

🔍 Verificar que todos los componentes están activos

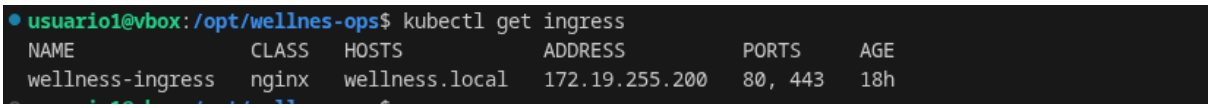
Verificar servicios

kubectl get svc



Verificar Ingress

kubectl get ingress



🔐 Cert-Manager y TLS automático en Kubernetes

Para habilitar HTTPS dentro del clúster utilizamos Cert-Manager, que automatiza:

✔ emisión de certificados

✔ renovación automática

✔ gestión de autoridades certificadoras

✔ integración con Ingress

🚀 Instalación de Cert-Manager

Crear namespace dedicado

kubectl create namespace cert-manager

Separar el componente mejora organización y seguridad.

Añadir repositorio Helm oficial

helm repo add jetstack https://charts.jetstack.io
helm repo update

Instalar Cert-Manager (incluyendo CRDs)

```
helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --set crds.enabled=true
```

🔍 Qué hace esta instalación

✔ instala controladores de certificados
✔ añade CRDs necesarios
✔ habilita gestión automática de TLS
✔ permite emitir certificados dentro del cluster

```
tester@vbox:~/wellnes-ops$ kubectl create namespace cert-manager
namespace/cert-manager created
tester@vbox:~/wellnes-ops$
tester@vbox:~/wellnes-ops$ helm repo add jetstack https://charts.jetstack.io
helm repo update
"jetstack" has been added to your repositories
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "ingress-nginx" chart repository
...Successfully got an update from the "jetstack" chart repository
...Successfully got an update from the "prometheus-community" chart repository
Update Complete. ⎈Happy Helming!⎈
tester@vbox:~/wellnes-ops$ helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --set crds.enabled=true
NAME: cert-manager
LAST DEPLOYED: Fri Feb 13 17:12:37 2026
NAMESPACE: cert-manager
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
```

⏳ Verificar que Cert-Manager está funcionando

kubectl get pods -n cert-manager

✔ Resultado esperado: Pods Running.

```
tester@vbox:~/wellnes-ops$ kubectl get pods -n cert-manager
kubectl get all -n cert-manager
NAME                                       READY   STATUS    RESTARTS       AGE
cert-manager-675d667c9-sgpfk               1/1     Running   1 (9m1s ago)   64m
cert-manager-cainjector-6674494d8-dfkm4    1/1     Running   1 (9m1s ago)   64m
cert-manager-webhook-8566bcbc98-rk28g      1/1     Running   1 (9m1s ago)   64m
NAME                                           READY   STATUS    RESTARTS       AGE
pod/cert-manager-675d667c9-sgpfk               1/1     Running   1 (9m1s ago)   64m
pod/cert-manager-cainjector-6674494d8-dfkm4    1/1     Running   1 (9m1s ago)   64m
pod/cert-manager-webhook-8566bcbc98-rk28g      1/1     Running   1 (9m1s ago)   64m

NAME                           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/cert-manager           ClusterIP   10.43.17.125    <none>        9402/TCP   64m
service/cert-manager-webhook   ClusterIP   10.43.134.240   <none>        443/TCP    64m

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/cert-manager               1/1     1            1           64m
```

🔐 Aplicar configuración TLS

Ahora configuraremos la autoridad certificadora interna y los certificados.

⚠️Es importante aplicar los manifiestos en orden.

Esto asegura la correcta inicialización de:

- **Issuers**
- **Certificates**
- **Secrets TLS**

1️⃣Crear ClusterIssuer (autoridad certificadora)

👉 Este recurso actúa como autoridad emisora.

kubectl apply -f k8s/tls/ca-clusterissuer.yml

```
tester@vbox:~/wellnes-ops$ kubectl apply -f k8s/tls/ca-clusterissuer.yml
clusterissuer.cert-manager.io/selfsigned-issuer created
tester@vbox:~/wellnes-ops$ kubectl get clusterissuer
NAME                READY   AGE
selfsigned-issuer   True    15s
```

2️⃣Crear certificado raíz

kubectl apply -f k8s/tls/ca-certificate.yml

Verificar:
kubectl get certificate -n cert-manager
kubectl get secret -n cert-manager

👉 Se genera el secreto con la CA.

```
tester@vbox:~/wellnes-ops$ kubectl apply -f k8s/tls/ca-certificate.yml
Warning: spec.privateKey.rotationPolicy: In cert-manager >= v1.18.0, the default value changed from `Never` to `Always`.
certificate.cert-manager.io/wellness-ca created
tester@vbox:~/wellnes-ops$ kubectl get certificate -n cert-manager
NAME          READY   SECRET              AGE
wellness-ca   True    wellness-ca-secret   11s
tester@vbox:~/wellnes-ops$ kubectl get secret -n cert-manager
NAME                                 TYPE                 DATA   AGE
cert-manager-webhook-ca              Opaque               3      69m
sh.helm.release.v1.cert-manager.v1   helm.sh/release.v1   1      70m
wellness-ca-secret                   kubernetes.io/tls    3      23s
```

3️⃣ Crear Issuer basado en la CA

kubectl apply -f k8s/tls/ca-issuer.yml

Verificar:

kubectl get clusterissuer

👉 Este emisor será usado por la aplicación.

```
tester@vbox:~/wellnes-ops$ kubectl apply -f k8s/tls/ca-issuer.yml
clusterissuer.cert-manager.io/wellness-ca-issuer created
tester@vbox:~/wellnes-ops$ kubectl get clusterissuer
NAME                READY   AGE
selfsigned-issuer   True    116s
wellness-ca-issuer  True    8s
```

4️⃣ Crear certificado TLS para la aplicación

kubectl apply -f k8s/tls/wellness-tls.yml

Verificar:

kubectl get certificate
kubectl get secret

👉 Aquí se genera el **certificado TLS real** que usará el Ingress.

```
tester@vbox:~/wellnes-ops$ kubectl apply -f k8s/tls/wellness-tls.yml
Warning: spec.privateKey.rotationPolicy: In cert-manager >= v1.18.0, the default value changed from `Never` to `Always`.
certificate.cert-manager.io/wellness-tls created
tester@vbox:~/wellnes-ops$ kubectl get certificate
NAME           READY   SECRET         AGE
wellness-tls   True    wellness-tls   10s
tester@vbox:~/wellnes-ops$ kubectl get secret
NAME                TYPE                DATA   AGE
backend-jwt-secret  Opaque              1      80m
backend-secret      Opaque              2      80m
postgres-secret     Opaque              3      80m
wellness-tls        kubernetes.io/tls   3      17s
```

🧠 Qué ocurre ahora

Cert-Manager:

✔ genera certificados
✔ los almacena como Secrets
✔ los renueva automáticamente
✔ los hace disponibles para Ingress

👉 Ahora el Ingress puede servir HTTPS.

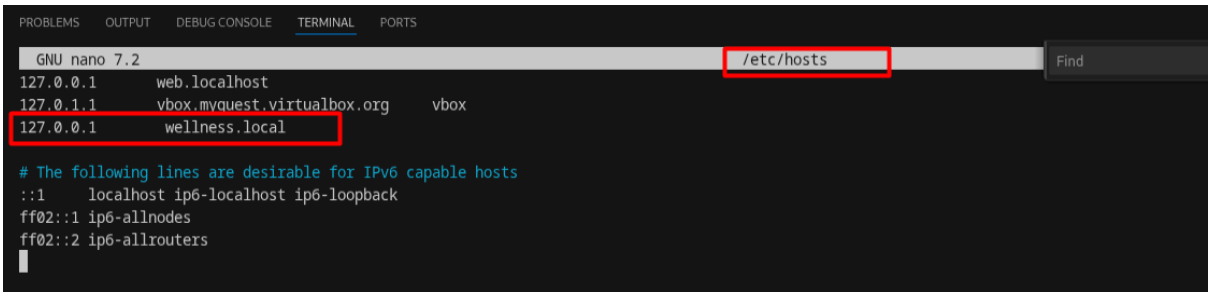
🌐 Editar /etc/hosts

El archivo:

/etc/hosts

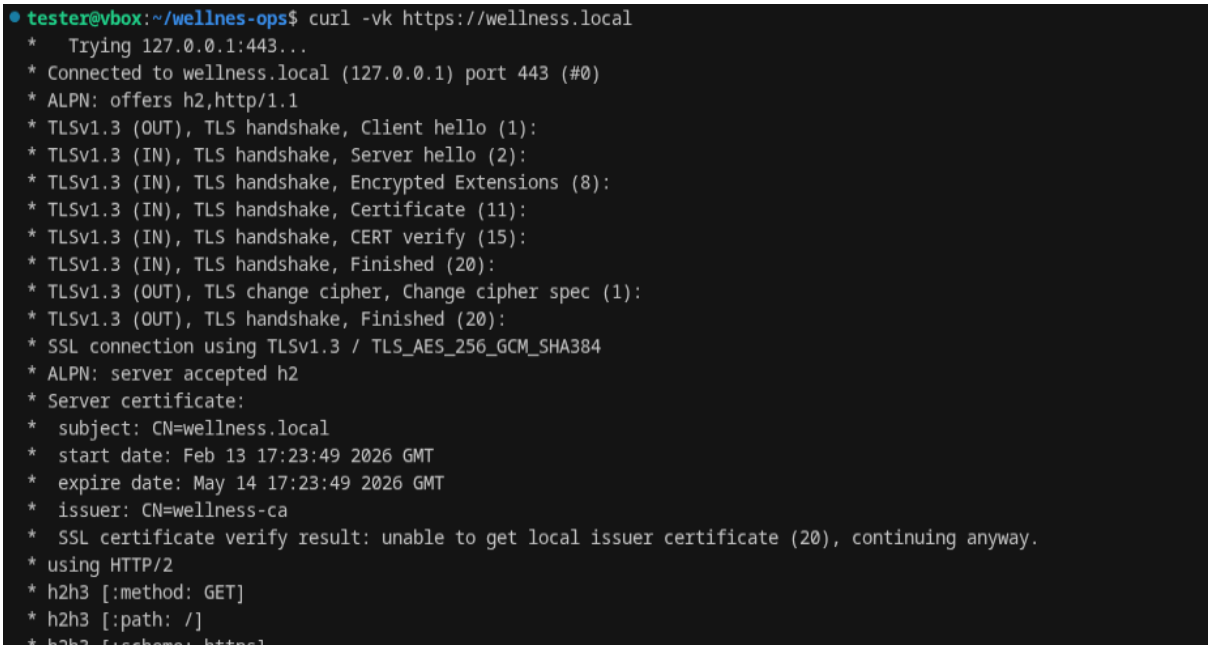
permite mapear manualmente dominios → IP.

127.0.0.1 wellness.local



🌐 Verificar acceso HTTPS

curl -vk https://wellness.local



✅ Verificación final del acceso a la aplicación

Una vez desplegados todos los componentes y configurado TLS correctamente, es necesario confirmar que la aplicación es accesible de forma segura a través del endpoint HTTPS.

Este paso valida que toda la arquitectura funciona de extremo a extremo.

Abrir en el navegador:

https://wellness.local

🔐 Verificar el certificado TLS

Al acceder:

✔ la conexión debe mostrarse como **segura (HTTPS)**
✔ el certificado debe estar presente
✔ no deben aparecer errores críticos de TLS

⚠️El navegador puede mostrar advertencias.

Flujo confirmado:

Cliente → Ingress NGINX → Frontend → Backend → PostgreSQL