

PLATAFORMA DE BACKUP Y RECUPERACIÓN PARA ENTORNOS KUBERNETES CON VELERO Y MINIO



Pablo Martín Hidalgo

Índice

1. Objetivos que se quieren conseguir y se han conseguido.....	4
2. Escenario necesario para la realización del proyecto.....	5
3. Preparación entorno Kubernetes.....	6
4. Orden lógico de despliegue de servicios.....	10
5. MinIO como solución de almacenamiento compatible con S3.....	11
5.1 ¿Qué es MinIO y por qué usarlo?.....	11
5.2 Despliegue de MinIO en Kubernetes.....	11
5.2.1 Creación del Deployment y Service.....	12
5.2.2 Configuración del Ingress.....	13
5.2.3 Configuración del acceso HTTPS mediante mkcert.....	14
5.3 Configuración de MinIO para su uso con Velero.....	16
5.3.1 Object Browser.....	16
5.3.2 Creación y configuración del bucket.....	17
5.3.3 Creación de la política y del usuario para Velero.....	19
5.3.4 Generación del archivo de credenciales para Velero.....	20
5.3.5 Monitoring.....	21
5.3.6 Events.....	21
6. Copias de seguridad con Velero.....	23
6.1 Introducción a Velero.....	23
6.2 Instalación de Velero en el clúster.....	23
6.2.1 Solución de errores.....	26
6.2.2 Prueba de funcionamiento Velero.....	29
6.3 Comandos de Velero.....	33
6.4 Programación de copias de seguridad.....	36
6.5 Restauración selectiva de recursos.....	38
6.6 Uso de etiquetas y selectors en backups.....	41

6.7 Integración con Volume Snapshots.....	43
6.8 Restore Hooks (ganchos de restauración).....	45
6.8.1 ¿Qué es un Restore Hook?.....	45
6.9 ¿Velero sirve para copias fuera de Kubernetes?.....	46
7. Demo/s.....	47
7.1. Demo 1: Recuperación Deployment y Service de Nextcloud.....	47
7.2 Demo 2: Restore Hook.....	55
8. Conclusiones y propuestas.....	60
9. Dificultades que se han encontrado.....	61
10. Bibliografía.....	62

1. Objetivos que se quieren conseguir y se han conseguido

El objetivo principal de este proyecto es diseñar e implementar una plataforma que permita realizar copias de seguridad y restaurar información en un entorno gestionado mediante **Kubernetes**. Para lograrlo, se ha recurrido a **Velero**, una herramienta especializada en la gestión de backups de los recursos desplegados en **clústeres de Kubernetes**, y a **MinIO**, un sistema de almacenamiento local compatible con el estándar **S3**, que actúa como destino para esas copias de seguridad.

La finalidad del proyecto es simular una solución realista que podría implantarse en el entorno de una empresa, donde es muy importante garantizar **la integridad y disponibilidad de los datos**. En este contexto, se pretende ofrecer una alternativa que no dependa de plataformas externas ni de servicios en la nube, lo que permite una mayor autonomía, control del entorno y reducción de costes. Al usar **MinIO** como sistema de almacenamiento local, se consigue precisamente ese control, manteniendo los datos dentro de la infraestructura propia y sin comprometer la compatibilidad con herramientas modernas como **Velero**.

Otro de los objetivos propuestos y finalmente no alcanzados ha sido incorporar un **sistema de monitorización y análisis** del funcionamiento del sistema de backups. Para ello, se buscaba integrar una pila de observabilidad basada en **Elasticsearch** y **Kibana**. Esta integración permitía registrar eventos clave relacionados con las tareas que realiza **Velero**, visualizar el estado de las copias de seguridad, detectar errores, analizar tiempos de ejecución y tener una representación gráfica que facilita la comprensión y seguimiento del sistema. Por falta de tiempo y errores no he sido capaz de implementarlos.

La motivación personal detrás del desarrollo de este proyecto radica en el interés por **Kubernetes**, una tecnología que está adquiriendo cada vez más relevancia en el ámbito profesional. **Velero** ha resultado ser una herramienta especialmente interesante por su enfoque práctico en la protección de sistemas frente a fallos, mientras que **MinIO** ofrece la posibilidad de montar una infraestructura completa de manera local, lo cual ha sido clave para cumplir con la premisa de no depender de servicios de terceros.

2. Escenario necesario para la realización del proyecto

Para llevar a cabo este proyecto he preparado un entorno controlado que simula de forma realista un sistema de respaldo y recuperación de datos en Kubernetes. La base de todo es **Minikube**, una herramienta que permite ejecutar un **clúster de Kubernetes** de manera local en una única máquina. Gracias a **Minikube**, ha sido posible desplegar todos los servicios necesarios dentro del mismo entorno sin tener que recurrir a infraestructura externa o servicios en nube.

Dentro de este clúster se han desplegado varios **pods**, cada uno ejecutando un componente clave de la plataforma. Por un lado, **Velero** se ha instalado como el servicio encargado de realizar las copias de seguridad y restauraciones de los recursos del clúster. Este servicio se comunicará de forma directa con **MinIO**, el cual ha sido desplegado en un pod independiente y que actuará como sistema de almacenamiento compatible con el estándar **S3**.

Toda la gestión del clúster y de los distintos servicios se ha realizado a través de la herramienta de consola **kubectrl**, la cual permite interactuar con Kubernetes desde línea de comandos. Asimismo, **Docker** ha sido fundamental para la creación y ejecución de los contenedores necesarios para cada componente. Todo este entorno se ha ejecutado sobre una máquina física con el sistema operativo **Ubuntu 24.04**, el cual ha proporcionado una base estable y compatible para poder ejecutar Minikube y demás herramientas.

3. Preparación entorno Kubernetes

Antes de desplegar cualquier servicio o aplicación en un clúster Kubernetes, es importante preparar el escenario de trabajo con todas las herramientas que nos harán falta. En esta fase de la preparación de todo el entorno, configuraré un entorno local que simulará un clúster de Kubernetes real. Este entorno constará de tres pilares fundamentales:

Docker, que proporciona el motor de contenedores sobre el cual Kubernetes ejecuta sus cargas de trabajo; básicamente el que actuará como driver.

Minikube, una herramienta que crea un clúster Kubernetes de un solo nodo en la máquina local/física.

kubectl, este es el cliente oficial de Kubernetes, muy importante para interactuar con el clúster, desplegar recursos y gestionar el ciclo de vida de las aplicaciones.

Dicho esto pasemos a la instalación y configuración del entorno. Lo primero que necesitamos es tener **Docker** instalado en nuestro sistema, ya que **Minikube** lo utilizará como el **hipervisor** para crear máquinas virtuales y ejecutar **Kubernetes**.

Primero, actualizamos los repositorios de APT con el comando:

```
sudo apt update
```

Luego, instalamos los paquetes necesarios para permitir el uso de repositorios a través de HTTPS:

```
sudo apt install apt-transport-https ca-certificates curl  
software-properties-common -y
```

```
pablo@ubuntu:~$ sudo apt install apt-transport-https ca-certificates curl software-properties-common -y  
Leyendo lista de paquetes... Hecho  
Creando árbol de dependencias... Hecho  
Leyendo la información de estado... Hecho  
ca-certificates ya está en su versión más reciente (20240203).  
fijado ca-certificates como instalado manualmente.  
software-properties-common ya está en su versión más reciente (0.99.49.1).  
fijado software-properties-common como instalado manualmente.  
Se instalarán los siguientes paquetes NUEVOS:  
  apt-transport-https curl  
0 actualizados, 2 nuevos se instalarán, 0 para eliminar y 0 no actualizados.
```

A continuación, agregamos 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
```

```
pablo@ubuntu:~$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/trusted.gpg.d/docker.gpg
```

Después, añadimos el repositorio de Docker a los repositorios de APT:

```
sudo add-apt-repository "deb [arch=amd64]
```

```
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

```
pablo@ubuntu:~$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
Repositorio: «deb [arch=amd64] https://download.docker.com/linux/ubuntu noble stable»
Descripción:
Archive for codename: noble components: stable
Más información: https://download.docker.com/linux/ubuntu
Añadiendo repositorio.
Oprima [INTRO] para continuar o Ctrl+c para cancelar.
Se encontró la entrada deb existente en /etc/apt/sources.list.d/archive_uri-https_download_docker_com_linux_ubuntu-noble.list
Añadiendo la entrada deb a /etc/apt/sources.list.d/archive_uri-https_download_docker_com_linux_ubuntu-noble.list
Se encontró la entrada deb-src existente en /etc/apt/sources.list.d/archive_uri-https_download_docker_com_linux_ubuntu-noble.list
Añadiendo la entrada desactivada deb-src a /etc/apt/sources.list.d/archive_uri-https_download_docker_com_linux_ubuntu-noble.list
```

Actualizamos nuevamente los repositorios e instalamos Docker con:

```
sudo apt update
```

```
sudo apt install docker-ce docker-ce-cli containerd.io
```

```
pablo@ubuntu:~$ sudo apt install docker-ce docker-ce-cli containerd.io
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
Se instalarán los siguientes paquetes adicionales:
  docker-buildx-plugin docker-ce-rootless-extras docker-compose-plugin git git-man liberror-perl libslirp0 pigz slirp4netns
Paquetes sugeridos:
  cgroupfs-mount | cgroup-lite git-daemon-run | git-daemon-sysvinit git-doc git-email git-gui gitk gitweb git-cvs
  git-mediawiki git-svn
Se instalarán los siguientes paquetes NUEVOS:
  containerd.io docker-buildx-plugin docker-ce docker-ce-cli docker-ce-rootless-extras docker-compose-plugin git git-man
  liberror-perl libslirp0 pigz slirp4netns
0 actualizados, 12 nuevos se instalarán, 0 para eliminar y 199 no actualizados.
Se necesita descargar 125 MB de archivos.
Se utilizarán 464 MB de espacio de disco adicional después de esta operación.
¿Desea continuar? [S/n]
```

Una vez instalado Docker, arrancamos y habilitamos el servicio y luego verificamos:

```
sudo systemctl start docker
```

```
sudo systemctl enable --now docker
```

```
sudo systemctl status docker
```

```
pablo@ubuntu:~$ sudo systemctl start docker
pablo@ubuntu:~$ sudo systemctl enable --now docker
Synchronizing state of docker.service with SysV service script with /usr/lib/systemd/systemd-sysv-install.
Executing: /usr/lib/systemd/systemd-sysv-install enable docker
pablo@ubuntu:~$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; preset: enabled)
   Active: active (running) since Thu 2025-05-08 13:05:05 CEST; 1min 20s ago
 TriggeredBy: ● docker.socket
    Docs: https://docs.docker.com
   Main PID: 8724 (dockerd)
      Tasks: 12
   Memory: 22.3M (peak: 23.5M)
     CPU: 1.104s
    CGroup: /system.slice/docker.service
            └─8724 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

may 08 13:05:04 ubuntu dockerd[8724]: time="2025-05-08T13:05:04.423738546+02:00" level=info msg="detected 127.0.0.53 nameserv
may 08 13:05:04 ubuntu dockerd[8724]: time="2025-05-08T13:05:04.460001432+02:00" level=info msg="Creating a containerd client
may 08 13:05:04 ubuntu dockerd[8724]: time="2025-05-08T13:05:04.559847120+02:00" level=info msg="Loading containers: start."
may 08 13:05:05 ubuntu dockerd[8724]: time="2025-05-08T13:05:05.414660913+02:00" level=info msg="Loading containers: done."
may 08 13:05:05 ubuntu dockerd[8724]: time="2025-05-08T13:05:05.493023298+02:00" level=info msg="Docker daemon" commit=01f442
may 08 13:05:05 ubuntu dockerd[8724]: time="2025-05-08T13:05:05.497912264+02:00" level=info msg="Initializing buildkit"
may 08 13:05:05 ubuntu dockerd[8724]: time="2025-05-08T13:05:05.549375829+02:00" level=info msg="Completed buildkit initializ
may 08 13:05:05 ubuntu dockerd[8724]: time="2025-05-08T13:05:05.568920322+02:00" level=info msg="Daemon has completed initial
may 08 13:05:05 ubuntu dockerd[8724]: time="2025-05-08T13:05:05.569143191+02:00" level=info msg="API listen on /run/docker.sock
may 08 13:05:05 ubuntu systemd[1]: Started docker.service - Docker Application Container Engine.
lines 1-22/22 (END)
```

Ahora que Docker está correctamente instalado, podemos proceder con la instalación de Minikube.

Descargamos e instalamos el paquete de Minikube desde su repositorio oficial con los siguientes comandos:

```
curl -Lo minikube
https://storage.googleapis.com/minikube/releases/v1.35.0/minikube-linux-amd64
sudo chmod +x minikube
sudo mv minikube /usr/local/bin/
```

```
pablo@ubuntu:~$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/v1.35.0/minikube-linux-amd64
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left  Speed
100 119M  100 119M    0     0  19.7M      0  0:00:06  0:00:06 --:--:-- 27.8M
pablo@ubuntu:~$ sudo chmod +x minikube
pablo@ubuntu:~$ sudo mv minikube /usr/local/bin/
```

Verificamos que Minikube esté correctamente instalado ejecutando:

```
minikube version
```

```
pablo@ubuntu:~$ minikube version
minikube version: v1.35.0
commit: dd5d320e41b5451cdf3c01891bc4e13d189586ed-dirty
```

Para usar Docker como hipervisor en Minikube, configuramos el driver con:

```
minikube config set driver docker
```

```
pablo@ubuntu:~$ minikube config set driver docker
! These changes will take effect upon a minikube delete and then a minikube start
```

Esta configuración asegura que Minikube utilizará Docker para crear las máquinas virtuales que ejecutarán Kubernetes.

Ahora añadimos nuestro usuario al grupo docker, pues de lo contrario no nos dejará iniciar Minikube por un error de permisos.

```
sudo usermod -aG docker $USER && newgrp docker
```

```
pablo@ubuntu:~$ sudo usermod -aG docker $USER && newgrp docker
[sudo] contraseña para pablo:
```

A continuación, iniciamos Minikube con:

```
minikube start
```

```
pablo@ubuntu:~$ minikube start
🐹 minikube v1.35.0 en Ubuntu 24.04 (vbox/amd64)
🌟 Using the docker driver based on user configuration
🔧 Using Docker driver with root privileges
👍 Starting "minikube" primary control-plane node in "minikube" cluster
📦 Pulling base image v0.0.46 ...
📦 Descargando Kubernetes v1.32.0 ...
> preloaded-images-k8s-v18-v1...: 333.57 MiB / 333.57 MiB 100.00% 5.81 Mi
> gcr.io/k8s-minikube/kicbase...: 500.15 MiB / 500.31 MiB 99.97% 6.62 MiB
🔥 Creating docker container (CPUs=2, Memory=2200MB) ...
🔧 Preparando Kubernetes v1.32.0 en Docker 27.4.1...
  ■ Generando certificados y llaves
  ■ Iniciando plano de control
  ■ Configurando reglas RBAC...
🔗 Configurando CNI bridge CNI ...
🔍 Verifying Kubernetes components...
  ■ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟 Complementos habilitados: storage-provisioner, default-storageclass
💡 kubectl not found. If you need it, try: 'minikube kubectl -- get pods -A'
🏁 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Minikube descargará las imágenes necesarias de Kubernetes y las configurará en Docker.

Este proceso puede tardar algunos minutos, dependiendo de la velocidad de la red.

Verificamos el estado del clúster con:

```
minikube status
```

```
pablo@ubuntu:~$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Cuando Minikube esté en funcionamiento, instalamos **kubectl**, que es la herramienta de línea de comandos para interactuar con Kubernetes. Usamos Snap para instalar **kubectl**:

```
sudo snap install kubectl --classic
```

```
pablo@ubuntu:~$ sudo snap install kubectl --classic
kubectl 1.32.4 from Canonical✓ installed
```

Por último, verificamos que **kubectl** se haya instalado correctamente con:

```
kubectl version --client
```

```
pablo@ubuntu:~$ kubectl version --client
Client Version: v1.32.4
Kustomize Version: v5.5.0
```

4. Orden lógico de despliegue de servicios

Antes de comenzar con las instalaciones y configuraciones individuales de cada uno de los servicios, es necesario establecer un orden lógico de despliegue que respete las dependencias entre los distintos componentes del clúster. Dado que este proyecto busca simular un entorno empresarial realista, este tipo de escenarios es muy habitual que ciertos servicios deban estar operativos antes que otros. Dicho esto, el orden de despliegue que se ha adoptado ha sido el siguiente:

MinIO se desplegará en primer lugar, ya que va a ser el que actúe como sistema de almacenamiento. Este es un componente esencial para que Velero pueda funcionar, ya que éste requiere un **backend de almacenamiento** donde poder guardar sus copias. Antes de continuar con el resto del entorno, hay que verificar que MinIO esté funcionando correctamente, que sea accesible por **consola web**, que tenga creado un **bucket** específico para Velero y que tenga unas **credenciales configuradas**.

Velero se instalará una vez MinIO esté disponible. Durante su instalación se conecta con el bucket creado en MinIO, utilizando un plugin S3 genérico, y se configurará para evitar snapshots.

5. MinIO como solución de almacenamiento compatible con S3

5.1 ¿Qué es MinIO y por qué usarlo?

MinIO es una solución de almacenamiento de objetos de alto rendimiento que implementa la **API de Amazon S3**, permitiendo que herramientas y servicios compatibles con dicha API puedan integrarse sin modificaciones extras. Al estar diseñado con una **arquitectura ligera y orientado a la nube**, MinIO puede ejecutarse de manera eficiente tanto en entornos locales como en despliegues en contenedores, especialmente sobre plataformas como Kubernetes.

A diferencia de otras soluciones de almacenamiento más complejas o cerradas, MinIO destaca por su sencillez de despliegue, una interfaz intuitiva y fácil de manejar, compatibilidad total con **S3** y facilidad para integrarse con sistemas de copias de seguridad, distribución de contenido o gestión de datos en general. Todo esto lo convierte en una gran opción para entornos empresariales que requieran de un almacenamiento **seguro, distribuido y escalable** sin tener que depender de forma directa de proveedores cloud.

En cuanto a este proyecto, MinIO se utiliza como **sistema de almacenamiento persistente** donde Velero puede guardar los respaldos del clúster de Kubernetes. Dado que Velero está diseñado para interactuar con distintos servicios compatibles con **S3**, MinIO se presenta como una alternativa perfecta para poder simular entornos empresariales sin la necesidad de recurrir a **Amazon Web Services** u otras **plataformas de pago**.

Su capacidad para funcionar sobre una **infraestructura local**, su compatibilidad con **herramientas nativas** de Kubernetes y la posibilidad de aplicar **políticas de acceso** lo convierten en una herramienta clave para un sistema de copias de seguridad.

5.2 Despliegue de MinIO en Kubernetes

El despliegue de MinIO en el clúster de Kubernetes es el primer paso funcional del entorno, éste se ha desplegado en modo **standalone** (independiente) utilizando un único **Pod**. Aunque MinIO permite configuraciones en **alta disponibilidad**, para este escenario se ha optado por una solución más simple y directa, suficiente para cumplir con los objetivos del proyecto.

Comentar que se realizará la instalación tanto de la interfaz web como de la consola a través de la terminal, pues para un determinado paso se necesitará sí o sí la consola.

5.2.1 Creación del Deployment y Service

Para desplegar MinIO en el clúster de Kubernetes se ha definido un fichero en formato **YAML** que contiene los dos recursos principales: un **Deployment** y un **Service**. Este fichero está disponible públicamente en el repositorio del proyecto:

 [minio-deployment.yaml](#)

En primer lugar, el **Deployment** se encarga de garantizar que el contenedor de MinIO se ejecute de forma continua, permitiendo que Kubernetes lo reinicie si falla, y facilitando su escalabilidad en **próximas versiones** del entorno. Se ha utilizado una imagen oficial de MinIO (`minio/minio:RELEASE.2025-04-08T15-41-24Z`), lanzando el servicio en modo independiente. Dentro del **Deployment** se define:

- **Contenedor**: ejecutando el binario de MinIO con el comando `server /data`, que indica que el directorio `/data` será usado para almacenar objetos.
- **Variables de entorno**: se definen `MINIO_ROOT_USER` y `MINIO_ROOT_PASSWORD` con las credenciales por defecto `minioadmin/minioadmin`, necesarias para acceder a la zona de administración.
- **Puertos expuestos**:
 - Puerto **9000** para operaciones S3.
 - Puerto **9001** para el panel web de administración.

Por otro lado, el recurso **Service** tiene como objetivo exponer el **Deployment** dentro del clúster. Se ha definido como un **ClusterIP**, lo que significa que será accesible solo desde **dentro del clúster** de Kubernetes, ideal para conectar otros **Pods** (como Velero) directamente a MinIO sin exponerlo públicamente.

Una vez creado el fichero `minio-deployment.yaml`, lo ejecutamos con el comando habitual de **kubectl**:

```
kubectl apply -f minio-deployment.yaml
```

```
pablo@ubuntu:~/proyecto$ kubectl apply -f minio-deployment.yaml
persistentvolumeclaim/minio-pvc created
deployment.apps/minio created
service/minio-service created
```

Tras ello, se puede verificar que el Pod y el servicio están en ejecución con:

```
kubectl get pods
kubectl get svc
```

```
pablo@ubuntu:~/proyecto$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
minio-59dd689b54-7xdf6             1/1     Running   0           50s
pablo@ubuntu:~/proyecto$ kubectl get svc
NAME                TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
kubernetes          ClusterIP   10.96.0.1    <none>        443/TCP                  3m20s
minio-service       ClusterIP   10.104.98.44 <none>        9000/TCP,9001/TCP       57s
```

Es importante comprobar que el Pod de MinIO esté en estado “*Running*”, pues de lo contrario significaría que está en estado “*CrashLoopBackOff*”, lo que puede indicar **errores** en las variables de entorno, los volúmenes, o la ejecución del binario.

De esa otra forma podemos ver todo el contenido actual del nodo del clúster:

```
kubectl get all
```

```
pablo@ubuntu:~/proyecto$ kubectl get all
NAME                                READY   STATUS    RESTARTS   AGE
pod/minio-59dd689b54-7xdf6         1/1     Running   0           4m

NAME                TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
service/kubernetes  ClusterIP   10.96.0.1    <none>        443/TCP                  6m24s
service/minio-service ClusterIP   10.104.98.44 <none>        9000/TCP,9001/TCP       4m1s

NAME                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/minio 1/1      1             1           4m1s

NAME                DESIRED   CURRENT   READY   AGE
replicaset.apps/minio-59dd689b54 1          1         1       4m1s
```

5.2.2 Configuración del Ingress

Para exponer la interfaz web de MinIO (por defecto en el puerto **9001**) de forma accesible, he utilizado un recurso **Ingress**. Este tipo de recurso permite definir reglas de acceso HTTP(S) y enrutar tráfico hacia los servicios internos del clúster. Antes de crear el Ingress, es necesario que el clúster tenga desplegado un controlador Ingress, que se encargue de gestionar esas reglas:

```
minikube addons enable ingress
```

```
pablo@ubuntu:~$ minikube addons enable ingress
💡 ingress is an addon maintained by Kubernetes. For any concerns contact minikube on GitHub.
You can view the list of minikube maintainers at: https://github.com/kubernetes/minikube/blob/master/OWNERS
  ■ Using image registry.k8s.io/ingress-nginx/kube-webhook-certgen:v1.4.4
  ■ Using image registry.k8s.io/ingress-nginx/controller:v1.11.3
  ■ Using image registry.k8s.io/ingress-nginx/kube-webhook-certgen:v1.4.4
🔍 Verifying ingress addon...
🌟 The 'ingress' addon is enabled
```

Este comando despliega el controlador Ingress como un conjunto de **Pods**, junto con un **Service de tipo NodePort**, accesible desde el host.

Una vez habilitado, crearé el archivo `ingress-minio.yaml` que definirá el Ingress de MinIO, el cual solo enrutará el **tráfico HTTPS** inicialmente. Su contenido es el siguiente:

 [ingress-minio.yaml](#)

Este recurso define dos entradas DNS:

- `minio-console.pablominio.local` → para la consola web (puerto 9001)
- `minio-api.pablominio.local` → para la API S3 (puerto 9000)

Lo aplicamos:

```
kubectl apply -f ingress-minio.yaml
```

```
pablo@ubuntu:~/proyecto$ kubectl apply -f ingress-minio.yaml
ingress.networking.k8s.io/minio-ingress created
```

Después de aplicar el recurso, tendremos que editar el archivo `/etc/hosts` en el sistema anfitrión para que ambos dominios apunten a la IP del clúster Minikube. Para ello primero debemos saber la IP:

```
minikube ip
```

```
pablo@ubuntu:~$ minikube ip
192.168.49.2
```

para seguidamente añadirla al fichero mencionado:

```
pablo@ubuntu:~/proyecto$ cat /etc/hosts
127.0.0.1 localhost
127.0.1.1 ubuntu

# The following lines are desirable for IPv6 capable hosts
::1    ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

192.168.49.2 minio-console.pablominio.local
192.168.49.2 minio-api.pablominio.local
```

5.2.3 Configuración del acceso HTTPS mediante mkcert

Como medida extra de seguridad, se ha configurado el Ingress definido anteriormente para servir el contenido cifrado mediante **HTTPS**. Dado a que se trata de un entorno local, he optado por usar la herramienta **mkcert**, la cual permite generar certificados TLS autofirmados.

Primero instalamos mkcert en nuestro sistema siguiendo estos pasos:

```
sudo apt install libnss3-tools
curl -JLO https://dl.filippo.io/mkcert/latest?for=linux/amd64
```

```
chmod +x mkcert-v*-linux-amd64
sudo mv mkcert-v*-linux-amd64 /usr/local/bin/mkcert
mkcert -install
```

Una vez instalado, hay que generar los certificados para los dominios utilizados en el Ingress:

```
mkcert minio-api.pabloginio.local minio-console.pabloginio.local
```

```
pablo@ubuntu:~/proyecto$ mkcert minio-api.pabloginio.local minio-console.pabloginio.local
Created a new certificate valid for the following names
- "minio-api.pabloginio.local"
- "minio-console.pabloginio.local"
The certificate is at "./minio-api.pabloginio.local+1.pem" and the key at "./minio-api.pabloginio.local+1-key.pem"
It will expire on 1 September 2027
```

Esto generó dos archivos:

```
pablo@ubuntu:~/proyecto$ ls -lh
total 16K
-rw-rw-r-- 1 pablo pablo 788 jun  1 19:42 ingress-minio.yaml
-rw----- 1 pablo pablo 1,7K jun  1 19:44 minio-api.pabloginio.local+1-key.pem
-rw-r--r-- 1 pablo pablo 1,6K jun  1 19:44 minio-api.pabloginio.local+1.pem
-rw-rw-r-- 1 pablo pablo 1,3K jun  1 18:32 minio-deployment.yaml
```

- *minio-api.pabloginio.local+1.pem* → certificado.
- *minio-api.pabloginio.local+1-key.pem* → clave privada.

Con los certificados generados, crearé un **Secret** de tipo TLS con kubectl para que el Ingress pueda utilizarlo al establecer conexiones HTTPS:

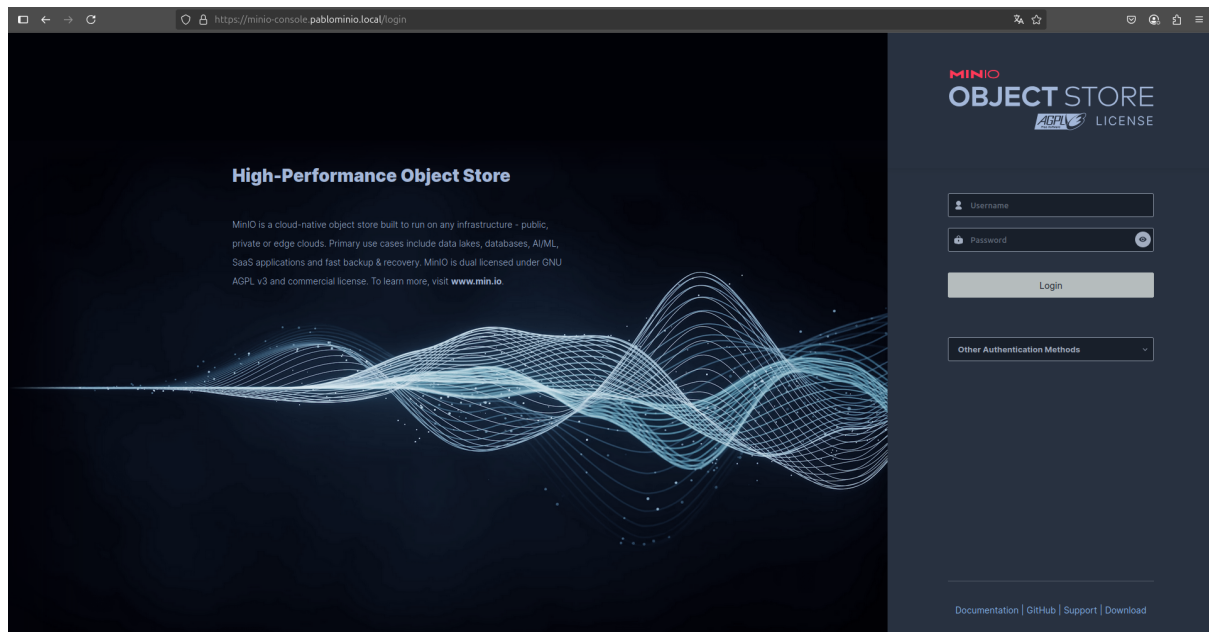
```
kubectl create secret tls minio-tls \
  --cert=minio-api.pabloginio.local+1.pem \
  --key=minio-api.pabloginio.local+1-key.pem
```

```
pablo@ubuntu:~/proyecto$ kubectl create secret tls minio-tls \
  --cert=minio-api.pabloginio.local+1.pem \
  --key=minio-api.pabloginio.local+1-key.pem
secret/minio-tls created
```

Por último, he modificado el fichero YAML del Ingress para que pueda soportar TLS y hecho esto, aplico de nuevo el fichero:

```
kubectl apply -f ingress-minio.yaml
```

De esta forma ya podremos acceder sin problemas a la consola web, en la cual me identificaré con las credenciales ya mencionadas anteriormente.



Ahora continuamos con la instalación de la consola a través de la terminal, para ello:

```
wget https://dl.min.io/client/mc/release/linux-amd64/mc
chmod +x mc
sudo mv mc /usr/local/bin/
mc --version
```

Con esto ya lo tendríamos instalado, pero tendremos que registrar un alias para poder interactuar con el servidor MinIO, para ello:

```
mc alias set minio-k8s https://minio-api.pablominio.local minioadmin minioadmin
```

```
pablo@ubuntu:~/proyecto$ mc alias set minio-k8s https://minio-api.pablominio.local minioadmin minioadmin
Added 'minio-k8s' successfully.
```

De esta forma ya habríamos acabado ambas instalaciones.

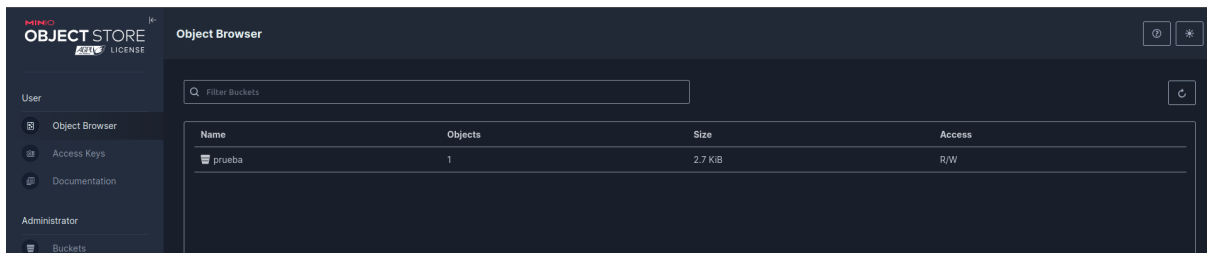
5.3 Configuración de MinIO para su uso con Velero

En esta sección voy a tratar los aspectos más relevantes que he tenido en cuenta para preparar MinIO como backend S3 compatible con Velero. Explicaré cómo configuramos el bucket de almacenamiento, los permisos de acceso, la creación del usuario dedicado y las credenciales necesarias para establecer la conexión desde Velero.

5.3.1 Object Browser

Nada más acceder a la consola web de MinIO, se nos sitúa directamente en el apartado “Object Browser”. En esta sección se nos muestran todos los **buckets** que existen en el servidor, cada uno con información importante como el número de objetos contenidos, el tamaño total, y la fecha de creación.

Además, disponemos de una barra de búsqueda en la parte superior que nos permite filtrar buckets por nombre de forma rápida, lo que resulta especialmente útil si trabajamos con múltiples servicios o backups. Desde aquí, con tan solo hacer click sobre uno de los buckets, accedemos a su contenido y funcionalidades adicionales, como subir archivos, crear carpetas, establecer políticas de acceso o incluso gestionar el versionado.



5.3.2 Creación y configuración del bucket

Como en cualquier servicio S3, un bucket es una unidad lógica de almacenamiento muy parecido a un directorio raíz, en donde se guardan los objetos (archivos) que subimos al sistema. Todos los objetos que maneja MinIO se almacenan en buckets, cada uno de ellos puede tener configuraciones específicas que afecten al comportamiento de los distintos objetos.

Para crear el bucket, accedo a la consola de MinIO y dentro del apartado “**Buckets**”, selecciono “**Create Bucket**”. En el formulario que aparece para rellenar le indico el nombre `velero` y procedo a configurar las opciones avanzadas que aparecen, las cuales son muy importantes para la integración con Velero. A continuación explico cada una de ellas y porqué se las he seleccionado:

- **Versioning**: activo esta opción ya que permite mantener versiones anteriores de los objetos dentro del bucket. Es decir, si Velero sube una nueva copia de seguridad con el mismo nombre que otra anterior, no se sobrescribe, sino que se almacenan ambas versiones. Esto es muy útil si se quiere restaurar el sistema a un estado anterior o comparar distintas copias.
- **Object Locking**: esta opción también la activo, pues es una característica que impide la modificación o eliminación de objetos durante un tiempo determinado. Esto se convierte en una medida de seguridad muy útil en el contexto de backups, ya que evita que un fallo humano borre datos críticos. A diferencia de la opción anterior, Velero no necesariamente requiere esta opción, pero considero que es una buena práctica habilitarla para darle más fortaleza.

- **Quota**: esta opción la desactivo, pues permite imponer un límite máximo de almacenamiento, lo cual no me interesa en este entorno.
- **Retention**: esta opción aparece al activar **Object Locking**, aquí configuro los siguientes parámetros: **modo Governance**, en este modo los objetos están protegidos contra borrados, salvo que se tengan permisos específicos para ello, por otro lado, **periodo de retención de 30 días**.

MINIO OBJECT STORE
M771 LICENSE

← Buckets

User

- Object Browser
- Access Keys
- Documentation

Administrator

- Buckets
- Policies
- Identity
- Monitoring
- Events
- Configuration
- License

Create Bucket

Bucket Name*

View Bucket Naming Rules

Features

Versioning OFF ☒ ON

Object Locking OFF ☒ ON

Quota OFF ☐ ON

Retention OFF ☒ ON

Mode ☐ Compliance ☒ Governance

Validity* days

Una vez creado el Bucket podemos ver toda la información relevante a modo resumen en el panel:

← Buckets

velero
Access: Private

Summary

Access Policy: ☒ Private

Replication: ☒ Disabled

Tags:

Versioning

Current Status: ☒ Versioned

Retention

Retention: ☒ Enabled

Validity: 30 Days

Encryption: ☒ Disabled

Object Locking: ☒ Enabled

Quota: ☒ Disabled

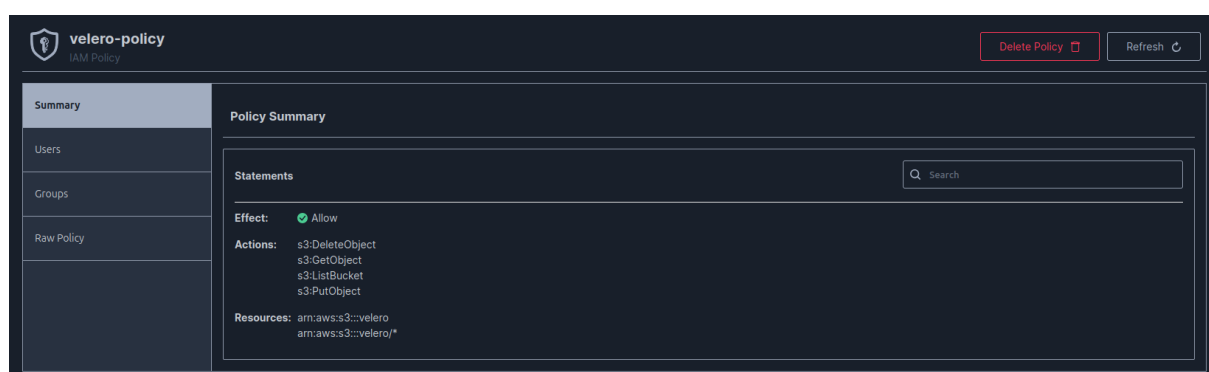
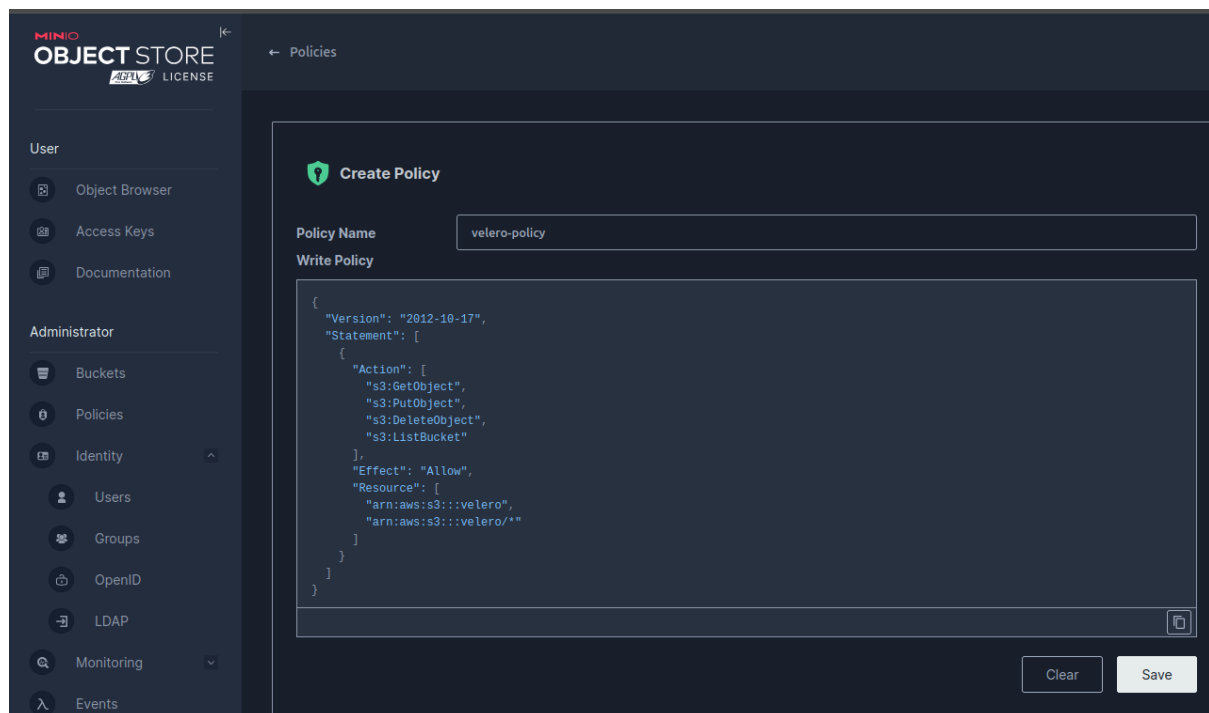
Reported Usage: 0.0 B

Mode: Governance

5.3.3 Creación de la política y del usuario para Velero

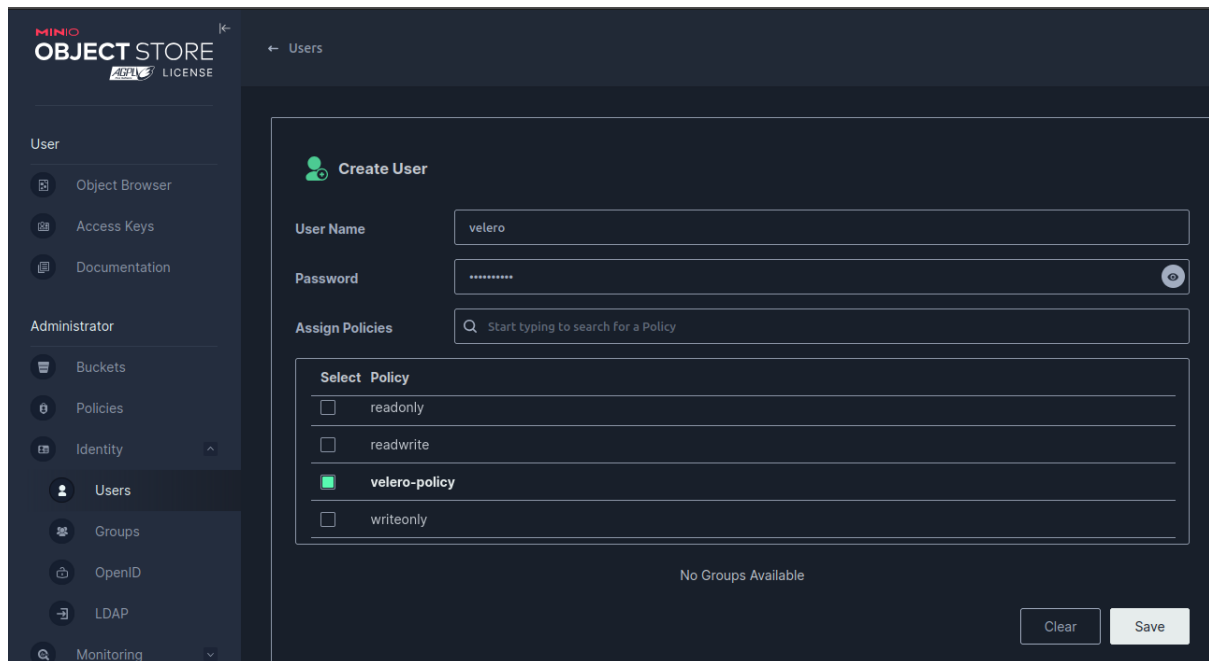
Antes de instalar Velero, creo una política de acceso personalizada que limite los permisos exclusivamente al bucket `velero`, ya que este será el espacio donde se almacenarán las copias de seguridad.

Desde el panel de MinIO accedo a **"Policies"** y creo una nueva política llamada `velero-policy` con el siguiente contenido:



Esta política concede permisos básicos: leer, escribir, eliminar objetos y listar el contenido del bucket.

Una vez creada, paso al apartado **"Users"** y creo el usuario `velero`. En el momento de su creación, asigno únicamente esta política, restringiendo así su acceso al bucket correspondiente.



5.3.4 Generación del archivo de credenciales para Velero

Después de haber creado el usuario **velero** en MinIO y asignado la política que le permite acceder al bucket correspondiente, es necesario generar un par de credenciales especiales que Velero utilizará para autenticarse y operar con MinIO.

Estas credenciales consisten en una **Access Key** y una **Secret Key**, que funcionan como un identificador y una contraseña segura para que Velero pueda interactuar con MinIO mediante la API compatible con S3. Para obtener estas credenciales:

```
mc admin user svcacct add minio-k8s velero
```

```
pablo@ubuntu:~/proyecto/Velero$ mc admin user svcacct add minio-k8s velero
Access Key: H4KBFYE5VS85YJR2M8PB
Secret Key: nQrvc+uYzGAZICgSA+Mb0pw+4LcTaeAFLNC08y3i
Expiration: no-expiry
```

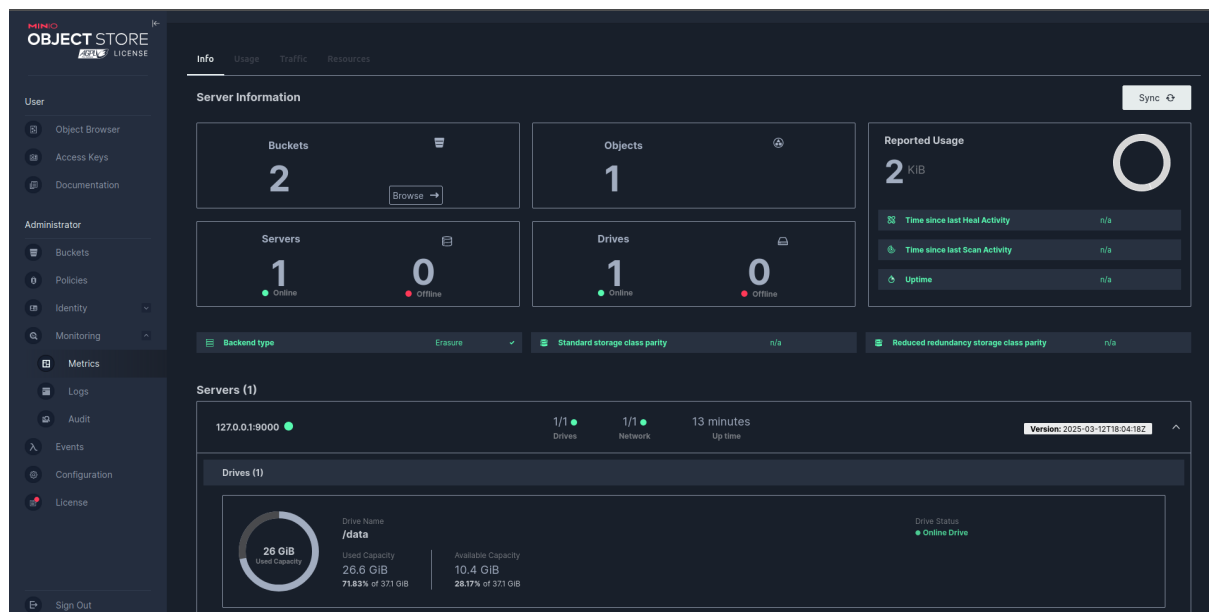
Este comando genera un archivo en formato JSON con la Access Key y la Secret Key correspondientes al usuario **velero**. Este archivo es el que Velero requiere para conectarse correctamente a MinIO y realizar las operaciones de respaldo y restauración.

Por lo tanto, dentro del directorio donde trabajaré con Velero, creo un fichero con las credenciales, el cual tendrá el siguiente contenido:

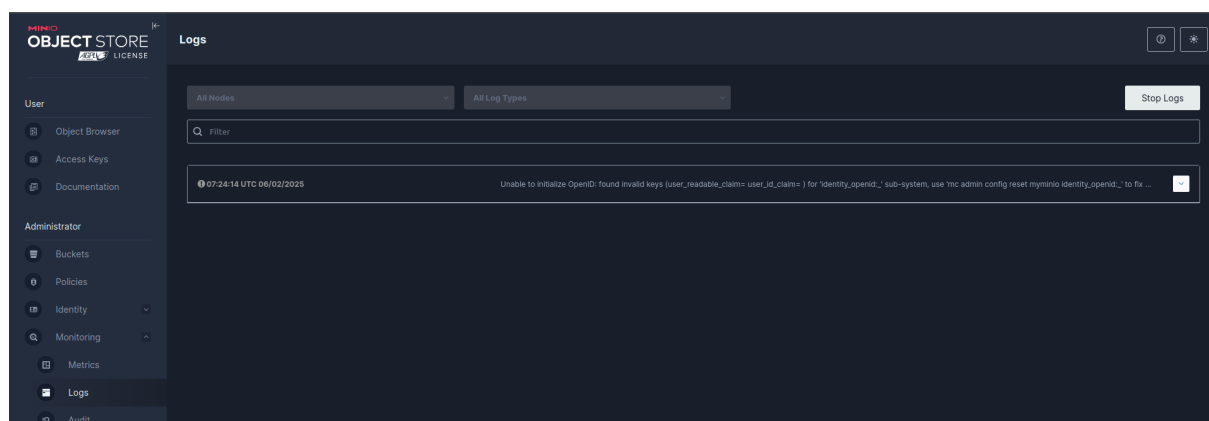
```
pablo@ubuntu:~/proyecto/Velero$ cat credenciales-velero  
[default]  
aws_access_key_id = H4KBFYE5VS85YJR2M8PB  
aws_secret_access_key = nQrvc+uYzGAZICgSA+Mb0pw+4LcTaeAfLNC08y3i
```

5.3.5 Monitoring

En esta parte vamos a ver las herramientas que nos da MinIO para poder controlar y revisar cómo está funcionando nuestro servicio. La sección llamada “**Metrics**” nos muestra un panel con información en tiempo real sobre el estado del sistema, como por ejemplo el uso de recursos o el tráfico. Esto nos ayuda a saber si todo va bien o si hay algún problema.



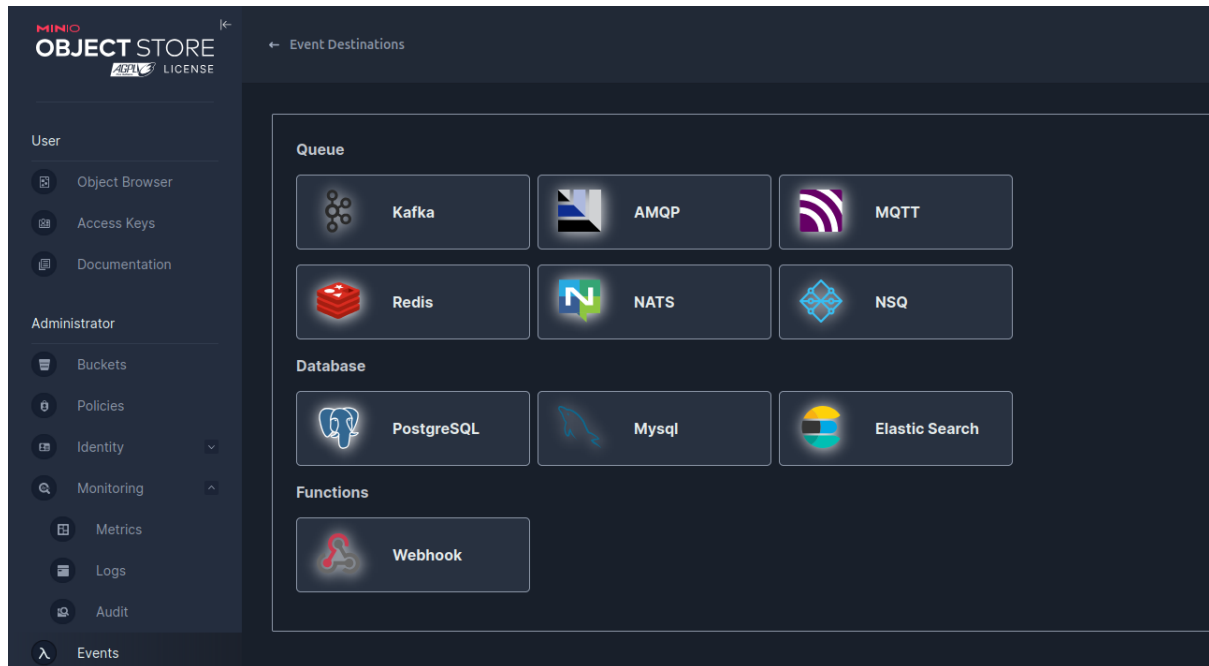
También podemos recopilar logs del servicio en la ventana de “**Logs**” y auditarlos posteriormente en la ventana “**Audit**”.



5.3.6 Events

También podemos configurar notificaciones para que se activen cuando ocurran ciertos eventos dentro del sistema, y esto se hace desde la sección llamada “**Events**”. MinIO permite

enviar estas notificaciones a varios destinos diferentes, como servicios externos o aplicaciones que estén escuchando esos avisos, lo que nos da muchas opciones para integrar el sistema con otras herramientas. En mi caso utilizaré Elastic Search, pero eso será algo que configuraré más tarde.



6. Copias de seguridad con Velero

6.1 Introducción a Velero

Velero es una herramienta de código abierto que se utiliza para realizar copias de seguridad y restauraciones de clústeres Kubernetes. Es especialmente útil en entornos donde se quiere proteger la configuración de los recursos del clúster (como **Deployments**, **Services**, **ConfigMaps**, etc.) y también los volúmenes persistentes que utilizan las aplicaciones.

En lugar de hacer copias de seguridad a nivel del sistema operativo como haríamos en una máquina tradicional, Velero trabaja directamente con los objetos de Kubernetes. Esto significa que puede guardar el estado completo de un namespace, una aplicación concreta o incluso todo el clúster, y luego restaurarlo si ocurre un fallo o si simplemente queremos moverlo a otro entorno.

Una de las ventajas de Velero es que permite almacenar las copias de seguridad en sistemas compatibles con el protocolo S3, como por ejemplo MinIO, el cual ya he configurado previamente en este proyecto. Gracias a esto, podemos tener nuestros backups guardados de forma remota y segura, incluso aunque se borre completamente el clúster.

Además de las copias de seguridad y restauración, Velero también permite hacer migraciones de aplicaciones entre diferentes clústeres Kubernetes, lo cual es muy útil si queremos replicar un entorno de pruebas o mover una aplicación a producción.

En los siguientes apartados se detallará cómo instalar Velero en nuestro clúster, cómo conectarlo con MinIO para guardar las copias de seguridad, y cómo realizar una copia y restaurarla, todo con ejemplos prácticos.

6.2 Instalación de Velero en el clúster

Para poder hacer copias de seguridad y restauraciones en Kubernetes, necesitamos tener tanto el **cliente Velero** instalado en la máquina desde la que trabajamos, como el **componente servidor de Velero** desplegado dentro del clúster.

Para instalar el **cliente Velero**, descargamos la versión que funciona correctamente en nuestro entorno y la colocamos en un directorio accesible desde cualquier parte del sistema. Los pasos realizados han sido los siguientes:

```
wget
https://github.com/vmware-tanzu/velero/releases/download/v1.16.1/velero-v1.16.1-
linux-amd64.tar.gz
tar xfv velero-v1.16.1-linux-amd64.tar.gz
sudo mv velero-v1.16.1-linux-amd64/velero /usr/local/bin
```

```
pablo@ubuntu:~/proyecto/Velero$ wget https://github.com/vmware-tanzu/velero/releases/download/v1.16.1/velero-v1.16.1-linux-amd64.tar.gz
```

```
pablo@ubuntu:~/proyecto/Velero$ ls -lh
total 53M
-rw-rw-r-- 1 pablo pablo 53M may 19 05:53 velero-v1.16.1-linux-amd64.tar.gz
pablo@ubuntu:~/proyecto/Velero$ tar xvf velero-v1.16.1-linux-amd64.tar.gz
velero-v1.16.1-linux-amd64/LICENSE
velero-v1.16.1-linux-amd64/examples/minio/00-minio-deployment.yaml
velero-v1.16.1-linux-amd64/examples/nginx-app/README.md
velero-v1.16.1-linux-amd64/examples/nginx-app/base.yaml
velero-v1.16.1-linux-amd64/examples/nginx-app/with-pv.yaml
velero-v1.16.1-linux-amd64/velero
```

```
pablo@ubuntu:~/proyecto/Velero$ sudo mv velero-v1.16.1-linux-amd64/velero /usr/local/bin/
[sudo] contraseña para pablo:
```

Una vez movido el ejecutable, se puede comprobar su funcionamiento con:

```
pablo@ubuntu:~/proyecto/Velero$ velero version
Client:
  Version: v1.16.1
  Git commit: 2eb97fa8b187f9ed0aeb49f216565eddf93a0b08
<error getting server version: no matches for kind "ServerStatusRequest" in version "velero.io/v1">
```

Este comando debería devolver la versión del cliente instalada correctamente. Si aún no se ha desplegado el servidor en Kubernetes, es normal que aparezca un mensaje de error indicando que no se puede conectar con el servidor Velero, ya que todavía no está desplegado en el clúster.

A continuación, voy a desplegar el **servidor de Velero** dentro de Kubernetes. Para ello, utilizamos el propio cliente Velero, que permite hacerlo mediante un único comando que incluye todos los parámetros necesarios. Este comando incluye la ubicación del bucket en MinIO, el proveedor (en este caso compatible con S3), la URL del endpoint, las credenciales y otros parámetros importantes.

El comando completo para desplegar el servidor Velero es el siguiente:

```
velero install \
  --provider aws \
  --plugins velero/velero-plugin-for-aws:latest \
  --bucket velero \
  --secret-file ./credenciales-velero \
  --backup-location-config
region=minio,s3ForcePathStyle="true",s3Url=https://minio-api.pablominio.local \
  --use-volume-snapshots=false \
  --namespace velero
```

Donde:

- `--provider aws`: Indica que el proveedor de almacenamiento es compatible con AWS S3.
- `--plugins velero/velero-plugin-for-aws:latest`: Especifica el plugin que Velero usará para interactuar con el almacenamiento S3. Se usa la última versión disponible del plugin oficial para AWS.
- `--bucket velero`: El nombre del bucket (o contenedor) en MinIO donde se almacenarán los backups.
- `--secret-file ./credenciales-velero`: Ruta al archivo con las credenciales para acceder a MinIO (acceso y secreto).
- `--backup-location-config`
`region=minio,s3ForcePathStyle="true",s3Url=https://minio-api.pablominio.local`: Configuración específica para el almacenamiento. Aquí se indica que la región es arbitraria (se usa "minio"), se fuerza el estilo de ruta compatible con MinIO (`s3ForcePathStyle="true"`) y la URL para acceder al servicio MinIO.
- `--use-volume-snapshots=false`: Se indica que no se usarán snapshots de volúmenes, porque con MinIO no se soporta este método de snapshot nativo.
- `--namespace velero`: Define el namespace en Kubernetes donde se desplegarán los recursos de Velero.

```
pablo@ubuntu:~/proyecto/Velero$ velero install \
  --provider aws \
  --plugins velero/velero-plugin-for-aws:latest \
  --bucket velero \
  --secret-file ./credenciales-velero \
  --backup-location-config region=minio,s3ForcePathStyle="true",s3Url=https://minio-api.pablominio.local \
  --use-volume-snapshots=false \
  --namespace velero
```

```
ServiceAccount/velero: attempting to create resource
ServiceAccount/velero: attempting to create resource client
ServiceAccount/velero: created
Secret/cloud-credentials: attempting to create resource
Secret/cloud-credentials: attempting to create resource client
Secret/cloud-credentials: created
BackupStorageLocation/default: attempting to create resource
BackupStorageLocation/default: attempting to create resource client
BackupStorageLocation/default: created
Deployment/velero: attempting to create resource
Deployment/velero: attempting to create resource client
Deployment/velero: created
Velero is installed! 🚀 Use 'kubectl logs deployment/velero -n velero' to view the status.
```

Una vez ejecutado el comando vemos como se instala a la perfección. Para asegurarme de que todo está funcionando correctamente, compruebo que el pod de Velero está en estado **Running**, lo que indica que el contenedor está activo y sin errores. Para ello:

```
pablo@ubuntu:~/proyecto/Velero$ kubectl get pods -n velero
NAME                                READY   STATUS    RESTARTS   AGE
velero-77749f5c98-btk5q            1/1     Running   0           58s
```

Otra verificación que se puede hacer es usar el propio comando de Velero para ver la versión instalada y verificar que se conecta correctamente al servidor en Kubernetes:

```
pablo@ubuntu:~/proyecto/Velero$ velero version
Client:
  Version: v1.16.1
  Git commit: 2eb97fa8b187f9ed0aeb49f216565eddf93a0b08
Server:
  Version: v1.16.1
```

6.2.1 Solución de errores

Y tal como se puede observar, ahora sí sale la versión del servidor correctamente. Aunque llegado a este punto me encuentro con un problema, y es que al hacer una pequeña prueba para verificar que funcione todo me da el siguiente error:

```
pablo@ubuntu:~/proyecto/Velero$ velero backup describe respaldo-prueba
Name:          respaldo-prueba
Namespace:     velero
Labels:        velero.io/storage-location=default
Annotations:   velero.io/resource-timeout=10m0s
               velero.io/source-cluster-k8s-gitversion=v1.32.0
               velero.io/source-cluster-k8s-major-version=1
               velero.io/source-cluster-k8s-minor-version=32

Phase:  Failed (run `velero backup logs respaldo-prueba` for more information)
```

También puede verse el error en los logs:

```
time="2025-06-02T17:54:31Z" level=info msg="BackupStorageLocation is invalid, marking as unavailable" backup-storage-location=velero/default controller=backup-storage-location logSource="pkg/controller/backup_storage_location_controller.go:122"
time="2025-06-02T17:54:31Z" level=error msg="Current BackupStorageLocations available/unavailable/unknown: 0/1/0, BackupStorageLocation \"default\" is unavailable: rpc error: code = Unknown desc = operation error S3: ListObjectsV2, exceeded maximum number of attempts, 3, https response error StatusCode: 0, RequestID: , HostID: , request send failed, Get \"https://minio-api.pablominio.local/velero?delimiter=%2F&list-type=2&prefix=\": tls: failed to verify certificate: x509: certificate signed by unknown authority)" controller=backup-storage-location logSource="pkg/controller/backup_storage_location_controller.go:188"
time="2025-06-02T17:54:31Z" level=info msg="plugin process exited" backup-storage-location=velero/default cmd=/plugins/velero-plugin-for-aws controller=backup-storage-location id=461 logSource="pkg/plugin/clientmgmt/process/logrus_adapter.go:88" plugin=/plugins/velero-plugin-for-aws
time="2025-06-02T17:54:53Z" level=error msg="Error listing backups in backup store" backupLocation=velero/default controller=backup-sync error="rpc error: code = Unknown desc = operation error S3: ListObjectsV2, exceeded maximum number of attempts, 3, https response error StatusCode: 0, RequestID: , HostID: , request send failed, Get \"https://minio-api.pablominio.local/velero?delimiter=%2F&list-type=2&prefix=backups%2F\": tls: failed to verify certificate: x509: certificate signed by unknown authority" error.file="/go/src/velero-plugin-for-aws/velero-plugin-for-aws/object_store.go:361" error.function="main.(*ObjectStore).ListCommonPrefixes" logSource="pkg/controller/backup_sync_controller.go:118"
time="2025-06-02T17:54:53Z" level=info msg="plugin process exited" backupLocation=velero/default cmd=/plugins/velero-plugin-for-aws controller=backup-sync id=477 logSource="pkg/plugin/clientmgmt/process/logrus_adapter.go:88" plugin=/plugins/velero-plugin-for-aws
```

El error es bastante claro y apunta a un problema con la verificación del certificado TLS que usa MinIO para asegurar la conexión HTTPS. La parte más importante es la que dice que Velero no puede verificar el certificado porque es un certificado firmado por una autoridad desconocida (x509: certificate signed by unknown authority).

Esto ocurre cuando MinIO está configurado para usar HTTPS con un certificado autofirmado o con una autoridad certificadora (CA) que no está reconocida ni confiable para Velero (o más exactamente para el plugin S3 que Velero usa).

Por lo tanto, para solucionarlo he tenido que configurar Velero para confiar en el certificado TLS de MinIO.

En primer lugar, extraigo el certificado de la autoridad certificadora que firmó el certificado TLS de MinIO:

```
openssl s_client -connect minio-api.pablominio.local:443 -showcerts </dev/null
2>/dev/null | openssl x509 -outform PEM > minio-ca.crt
```

```
pablo@ubuntu:~/proyecto/Velero$ openssl s_client -connect minio-api.pablominio.local:443 -showcerts </dev/null 2>/dev/null | openssl x509 -outform PEM > minio-ca.crt
pablo@ubuntu:~/proyecto/Velero$ ls -lh
total 8,0K
-rw-rw-r-- 1 pablo pablo 116 jun  1 21:13 credenciales-velero
-rw-rw-r-- 1 pablo pablo 1,6K jun  2 20:01 minio-ca.crt
```

Con este archivo, lo subo al clúster como un ConfigMap que Velero pueda montar:

```
kubectl -n velero create configmap minio-ca-cert --from-file=cert=./minio-ca.crt
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl -n velero create configmap minio-ca-cert --from-file=cert=./minio-ca.crt
configmap/minio-ca-cert created
```

Ahora necesito modificar el Deployment de Velero para que:

- Monte ese ConfigMap como un volumen dentro del pod.

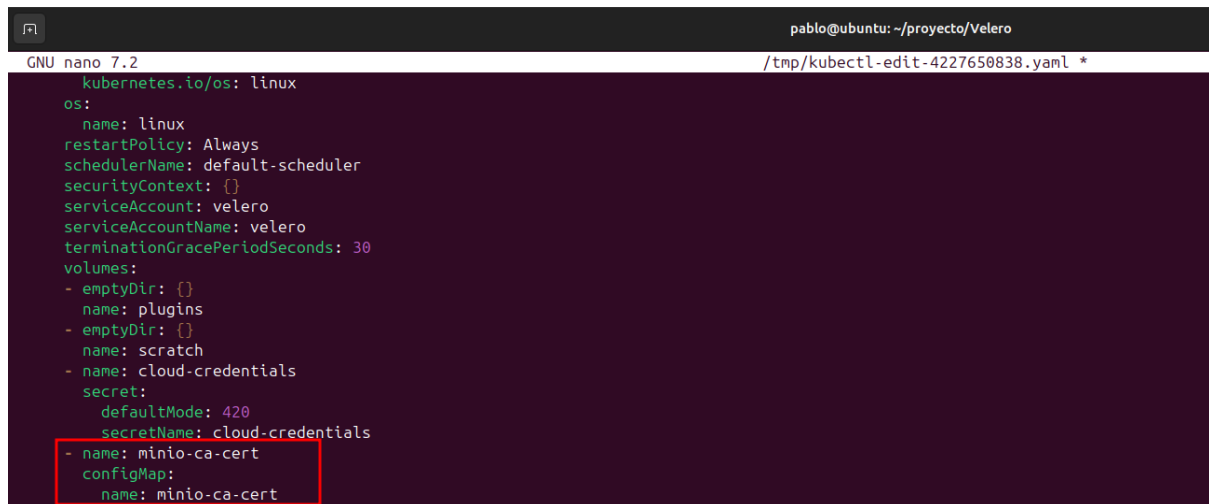
- Configure la variable de entorno `SSL_CERT_FILE` en el contenedor plugin para que apunte a ese certificado.

Edito el deployment manualmente con:

```
export EDITOR=nano
kubectl -n velero edit deployment velero
```

En donde añadiré los siguientes bloques dentro del YAML:

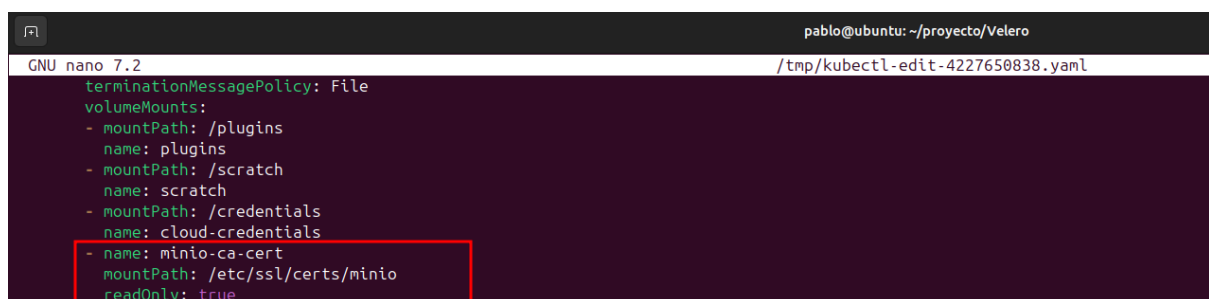
a) En el Volumen:



```

kubernetes.io/os: linux
os:
  name: linux
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: velero
  serviceAccountName: velero
  terminationGracePeriodSeconds: 30
volumes:
- emptyDir: {}
  name: plugins
- emptyDir: {}
  name: scratch
- name: cloud-credentials
  secret:
    defaultMode: 420
    secretName: cloud-credentials
- name: minio-ca-cert
  configMap:
    name: minio-ca-cert
```

b) En el contenedor del plugin (el que tiene la imagen del plugin de AWS), agrego el volumeMount:



```

terminationMessagePolicy: File
volumeMounts:
- mountPath: /plugins
  name: plugins
- mountPath: /scratch
  name: scratch
- mountPath: /credentials
  name: cloud-credentials
- name: minio-ca-cert
  mountPath: /etc/ssl/certs/minio
  readOnly: true
```

c) También en ese contenedor, añado la variable de entorno `SSL_CERT_FILE`:

```
pablo@ubuntu: ~/proyecto/Velero
GNU nano 7.2 /tmp/kubectrl-edit-4227650838.yaml *
env:
- name: VELERO_SCRATCH_DIR
  value: /scratch
- name: VELERO_NAMESPACE
  valueFrom:
    fieldRef:
      apiVersion: v1
      fieldPath: metadata.namespace
- name: LD_LIBRARY_PATH
  value: /plugins
- name: GOOGLE_APPLICATION_CREDENTIALS
  value: /credentials/cloud
- name: AWS_SHARED_CREDENTIALS_FILE
  value: /credentials/cloud
- name: AZURE_CREDENTIALS_FILE
  value: /credentials/cloud
- name: ALIBABA_CLOUD_CREDENTIALS_FILE
  value: /credentials/cloud
- name: SSL_CERT_FILE
  value: /etc/ssl/certs/minio/cert
image: velero/velero:v1.16.1
imagePullPolicy: IfNotPresent
name: velero
```

Una vez editado, guardo el fichero Deployment y lo reinicio:

```
kubectrl rollout restart deployment velero -n velero
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectrl rollout restart deployment velero -n velero
deployment.apps/velero restarted
```

Aquí se ve claramente que el Deployment de Velero está funcionando correctamente: pues hay un pod en estado Running con 1 réplica lista y disponible. Esto quiere decir que Velero ya está activo en el clúster y listo para operar.

```
pablo@ubuntu:~/proyecto/Velero$ kubectrl get all -n velero
NAME                                READY    STATUS    RESTARTS   AGE
pod/velero-986b946d6-98kzk         1/1     Running   0           42s

NAME                                READY    UP-TO-DATE    AVAILABLE   AGE
deployment.apps/velero             1/1     1              1           29m

NAME                                DESIRED    CURRENT    READY    AGE
replicaset.apps/velero-5cf5c75887   0          0          0        95s
replicaset.apps/velero-77749f5c98   0          0          0        29m
replicaset.apps/velero-986b946d6    1          1          1        42s
```

6.2.2 Prueba de funcionamiento Velero

Ahora sí podré hacer una prueba para verificar que todo esté funcionando, comienzo creando un namespace en el cual lanzaré un recurso:

```
kubectl create namespace prueba-velero
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl create namespace prueba-velero
namespace/prueba-velero created
```

Dentro de ese namespace, despliego un pod como por ejemplo nginx:

```
kubectl run nginx --image=nginx -n prueba-velero
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl run nginx --image=nginx -n prueba-velero
pod/nginx created
```

Compruebo que se haya creado:

```
kubectl get pods -n prueba-velero
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl get pods -n prueba-velero
NAME      READY   STATUS    RESTARTS   AGE
nginx     1/1     Running   0           17s
```

Ahora voy a crear el respaldo de ese namespace:

```
velero backup create respaldo-bueno-prueba --include-namespaces prueba-velero
```

```
pablo@ubuntu:~/proyecto/Velero$ velero backup create respaldo-bueno-prueba --include-namespaces prueba-velero
Backup request "respaldo-bueno-prueba" submitted successfully.
Run 'velero backup describe respaldo-bueno-prueba' or 'velero backup logs respaldo-bueno-prueba' for more details.
```

Y como podemos observar se ha creado correctamente, pues ha mostrado "**Phase: Completed**":

```
velero backup describe respaldo-bueno-prueba
```

```
pablo@ubuntu:~/proyecto/Velero$ velero backup describe respaldo-bueno-prueba
Name:      respaldo-bueno-prueba
Namespace: velero
Labels:    velero.io/storage-location=default
Annotations: velero.io/resource-timeout=10m0s
             velero.io/source-cluster-k8s-gitversion=v1.32.0
             velero.io/source-cluster-k8s-major-version=1
             velero.io/source-cluster-k8s-minor-version=32

Phase: Completed

Namespaces:
  Included: prueba-velero
  Excluded: <none>

Resources:
  Included: *
  Excluded: <none>
  Cluster-scoped: auto

Label selector: <none>
Or label selector: <none>

Storage Location: default

Velero-Native Snapshot PVs: auto
Snapshot Move Data: false
Data Mover: velero

TTL: 720h0m0s

CSISnapshotTimeout: 10m0s
ItemOperationTimeout: 4h0m0s

Hooks: <none>

Backup Format Version: 1.1.0

Started:    2025-06-02 20:20:31 +0200 CEST
Completed:  2025-06-02 20:20:32 +0200 CEST
Expiration: 2025-07-02 20:20:31 +0200 CEST
```

Para comprobar que el restore funciona, borro el namespace:

```
kubectl delete namespace prueba-velero
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl delete namespace prueba-velero
namespace "prueba-velero" deleted
```

Lanzo la restauración:

```
velero restore create --from-backup respaldo-bueno-prueba
```

```
pablo@ubuntu:~/proyecto/Velero$ velero restore create --from-backup respaldo-bueno-prueba
Restore request "respaldo-bueno-prueba-20250602202714" submitted successfully.
Run 'velero restore describe respaldo-bueno-prueba-20250602202714' or 'velero restore logs respaldo-bueno-prueba-20250602202714' for more details.
```

Y compruebo el estado:

```
velero restore describe <nombre-del-restore>
```

```
pablo@ubuntu:~/proyecto/Velero$ velero restore describe respaldo-bueno-prueba-20250602202714
Name:      respaldo-bueno-prueba-20250602202714
Namespace: velero
Labels:    <none>
Annotations: <none>

Phase: Completed
Total items to be restored: 4
Items restored: 4

Started:    2025-06-02 20:27:14 +0200 CEST
Completed:  2025-06-02 20:27:14 +0200 CEST

Warnings:
  Velero:    <none>
  Cluster:   <none>
  Namespaces:
    prueba-velero: could not restore, ConfigMap:kube-root-ca.crt already exists. Warning: the in-cluster version is different than the backed-up version
Backup: respaldo-bueno-prueba

Namespaces:
  Included: all namespaces found in the backup
  Excluded: <none>

Resources:
  Included: *
  Excluded: nodes, events, events.events.k8s.io, backups.velero.io, restores.velero.io, resticrepositories.velero.io, csinodes.storage.k8s.io, volumeattachments.storage.k8s.io, backuprepositories.velero.io
  Cluster-scoped: auto

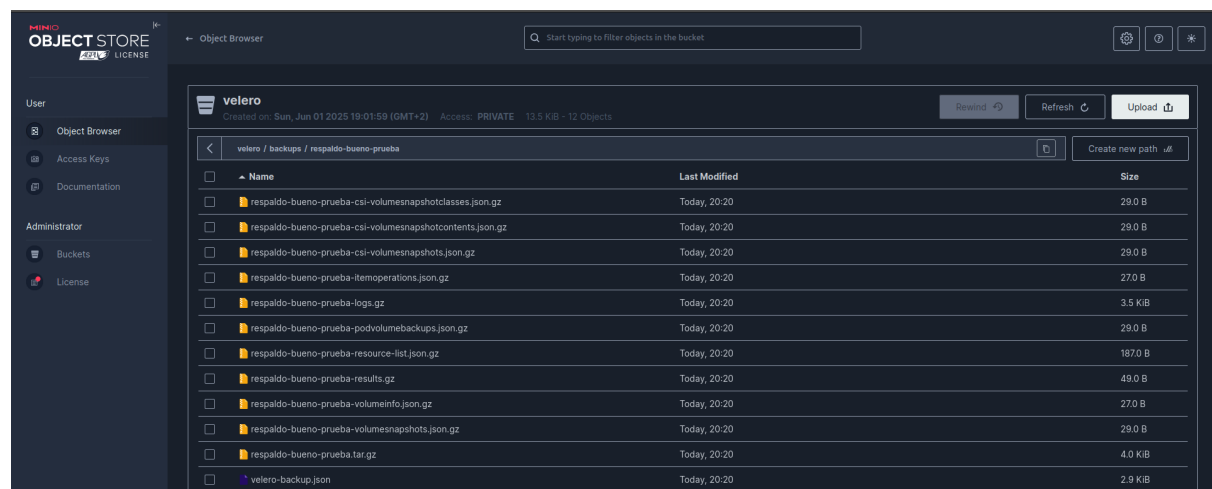
Namespace mappings: <none>
Label selector: <none>
Or label selector: <none>
Restore PVs: auto
CSI Snapshot Restores: <none included>
Existing Resource Policy: <none>
ItemOperationTimeout: 4h0m0s
Preserve Service NodePorts: auto
```

Por último, me aseguro que el namespace y el pod hayan vuelto y como se puede observar es así:

```
kubectl get all -n prueba-velero
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl get all -n prueba-velero
NAME          READY   STATUS    RESTARTS   AGE
pod/nginx     1/1     Running   0           47s
```

Por supuesto todo esto está conectado a la consola de la interfaz web, en la cual podemos ver los distintos ficheros que se generan:



6.3 Comandos de Velero

Velero proporciona una interfaz de línea de comandos muy completa, a través de la cual se pueden realizar todas las operaciones relacionadas con las copias de seguridad, restauraciones, configuración y observabilidad del sistema. Esta CLI se invoca mediante el comando `velero`, seguido de subcomandos específicos. A continuación se exponen los comandos más relevantes, agrupados según su funcionalidad:

Comandos generales

El comando base de Velero es:

```
velero [comando] [subcomando] [flags]
```

Algunos comandos comunes para obtener ayuda o información general son:

```
velero help          # Muestra ayuda general
velero version       # Muestra la versión del cliente y del servidor Velero
```

Gestión de backups

Para trabajar con copias de seguridad (respaldos), Velero proporciona una serie de comandos que permiten crear, listar, describir, ver logs y eliminar backups:

```
velero backup create NOMBRE_BACKUP \
  --include-namespaces NAMESPACE1,NAMESPACE2 \
  --exclude-namespaces NAMESPACE3 \
  --include-resources pods,services \
  --selector etiqueta=valor

velero backup get          # Lista todos los backups
velero backup describe NOMBRE # Muestra detalles de un backup
velero backup logs NOMBRE  # Muestra los logs del proceso de backup
velero backup delete NOMBRE # Elimina un backup existente
```

También es posible incluir o excluir volúmenes persistentes y snapshots con:

```
--snapshot-volumes=true|false  
--default-volumes-to-restic
```

Gestión de restauraciones

Para restaurar el estado del clúster o parte del mismo, Velero ofrece los siguientes comandos:

```
velero restore create --from-backup NOMBRE_BACKUP  
velero restore get           # Lista todas las restauraciones  
velero restore describe NOMBRE # Detalla una restauración  
velero restore logs NOMBRE   # Muestra los logs de una restauración  
velero restore delete NOMBRE # Elimina una restauración
```

Es posible incluir o excluir recursos en la restauración con flags como:

```
--include-namespaces  
--exclude-resources  
--restore-volumes=true|false
```

Backups programados (schedules)

Para automatizar la creación de copias de seguridad periódicas:

```
velero schedule create NOMBRE \  
  --schedule "0 4 * * *" \  
  --include-namespaces NAMESPACE \  
  --ttl 240h  
  
velero schedule get           # Lista las tareas programadas  
velero schedule describe NOMBRE # Muestra detalles de una programación  
velero schedule delete NOMBRE  # Elimina una programación
```

La opción `--ttl` indica el tiempo de vida del backup antes de su eliminación automática.

Configuración del entorno

Velero permite gestionar configuraciones como el proveedor de almacenamiento (S3/MinIO), los plugins instalados y las ubicaciones de backup:

```
velero install ... # Instalación inicial del entorno
velero plugin get # Lista los plugins activos
velero backup-location get # Muestra las ubicaciones de
almacenamiento
velero backup-location describe default # Detalla la ubicación "default"
```

Otros comandos útiles

También se puede trabajar con logs globales de Velero:

```
kubectl logs deployment/velero -n velero # Logs del pod Velero
```

Y comprobar la salud del sistema o recursos relacionados:

```
kubectl get pods -n velero
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl get pods -n velero
NAME                                READY   STATUS    RESTARTS   AGE
velero-986b946d6-98kzk             1/1     Running   1 (2d ago)  2d2h
```

```
kubectl describe pod NOMBRE -n velero
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl describe pod velero-986b946d6-98kzk -n velero
Name:                                velero-986b946d6-98kzk
Namespace:                           velero
Priority:                              0
Service Account:                      velero
Node:                                 minikube/192.168.49.2
Start Time:                           Mon, 02 Jun 2025 20:11:43 +0200
Labels:                               component=velero
                                      deploy=velero
                                      pod-template-hash=986b946d6
Annotations:                          kubectrl.kubernetes.io/restartedAt: 2025-06-02T20:11:43+02:00
                                      prometheus.io/path: /metrics
                                      prometheus.io/port: 8085
                                      prometheus.io/scrape: true
Status:                               Running
IP:                                   10.244.0.25
IPs:                                  10.244.0.25
Controlled By:                        ReplicaSet/velero-986b946d6
Init Containers:
  velero-velero-plugin-for-aws:
    Container ID:  docker://201ed56752acda88072c7425cf026ef5926c16d573fe5d0161e383a3f9c6be14
    Image:         velero/velero-plugin-for-aws:latest
    Image ID:      docker-pullable://velero/velero-plugin-for-aws@sha256:b9735c9d08c3244c462bb81263ff5f4ad4e24b96865338c14733a59e3624dfaf
    Port:         <none>
    Host Port:    <none>
    State:        Terminated
      Reason:     Completed
      Exit Code:   0
      Started:    Wed, 04 Jun 2025 21:57:34 +0200
      Finished:   Wed, 04 Jun 2025 21:57:34 +0200
    Ready:        True
    Restart Count: 1
    Environment:  <none>
    Mounts:
      /target from plugins (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-s6bnh (ro)
Containers:
  velero:
    Container ID:  docker://68237764e12994badd2829ce291015cb4ffeb38671d8d0704e5da29cf42fb627
    Image:         velero/velero:v1.16.1
    Image ID:      docker-pullable://velero/velero@sha256:c790429fcd543f0a5eed3a490e85a2c39bf9aefb8ce7ddbcb7a158557745ab33f
    Port:         8085/TCP
    Host Port:    0/TCP
    Command:
      /usr/bin/velero
      --velero-plugin=velero-plugin-for-aws
```

6.4 Programación de copias de seguridad

Una de las funcionalidades más útiles que tiene Velero es la posibilidad de **programar copias de seguridad automáticas**. Esto permite olvidarnos de tener que ejecutar manualmente los backups cada día, ya que se pueden definir reglas para que se ejecuten en el momento que uno quiera. Esta característica se gestiona con el comando `velero schedule`.

Para programar una copia de seguridad automática, se utiliza la siguiente sintaxis:

```
velero schedule create NOMBRE \  
  --schedule "CRON" \  
  [más opciones...]
```

Lo más importante de este comando es la opción `--schedule`, que se basa en el formato **cron**. Este formato sirve para definir cuándo quiero que se haga la copia. Por ejemplo:

```
velero schedule create backup-diario-prueba \  
  --schedule "0 2 * * *" \  
  --ttl 72h0m0s
```

Con este comando he creado una programación llamada `backup-diario-prueba` que lanza un backup todos los días a las 2:00 de la madrugada. El parámetro `--ttl` indica cuánto tiempo quiero que se conserve la copia antes de que Velero la borre automáticamente. En este caso, el valor `72h0m0s` significa que cada backup durará 3 días.

Velero permite también filtrar los recursos que quiero incluir en los backups programados. Por ejemplo, si solo quiero hacer una copia del namespace `prueba-velero`, usaría:

```
velero schedule create backup-prueba-velero \  
  --schedule "0 1 * * *" \  
  --include-namespaces prueba-velero \  
  --ttl 168h
```

Esto hace que todos los días a la 1:00 se cree un backup solo del namespace mencionado, y que dicho backup se mantenga durante 7 días (168 horas).

```
pablo@ubuntu:~/proyecto/Velero$ kubectl get namespaces
NAME                STATUS   AGE
default             Active   28d
ingress-nginx       Active   3d4h
kube-node-lease     Active   28d
kube-public         Active   28d
kube-system         Active   28d
prueba-velero       Active   2d1h
velero              Active   2d2h
```

```
pablo@ubuntu:~/proyecto/Velero$ velero schedule create backup-prueba-velero \
--schedule "0 1 * * *" \
--include-namespaces prueba-velero \
--ttl 168h
Schedule "backup-prueba-velero" created successfully.
```

Una vez creado, si quiero consultar qué tareas programadas tengo en el sistema, simplemente ejecuto:

```
velero schedule get
```

```
pablo@ubuntu:~/proyecto/Velero$ velero schedule get
NAME                STATUS   CREATED                SCHEDULE   BACKUP TTL   LAST BACKUP   SELECTOR   PAUSED
backup-prueba-velero Enabled  2025-06-04 22:18:21 +0200 CEST 0 1 * * *   168h0m0s    n/a          <none>     false
```

Y si quiero ver más detalles sobre alguna de ellas, como por ejemplo cuándo fue el último backup que se hizo o si hubo algún error, utilizo:

```
velero schedule describe <nombre_tarea>
```

```
pablo@ubuntu:~/proyecto/Velero$ velero schedule describe backup-prueba-velero
Name:                backup-prueba-velero
Namespace:           velero
Labels:              <none>
Annotations:         <none>

Phase: Enabled

Paused: false

Schedule: 0 1 * * *

Backup Template:
  Namespaces:
    Included: prueba-velero
    Excluded: <none>

  Resources:
    Included: *
    Excluded: <none>
    Cluster-scoped: auto

  Label selector: <none>

  Or label selector: <none>

  Storage Location:
```

6.5 Restauración selectiva de recursos

Una de las cosas que más me llamó la atención de Velero es que no solo me permite restaurar una copia de seguridad completa, sino que también puedo **restaurar únicamente ciertos recursos**, lo que se conoce como restauración selectiva. Esto es especialmente útil cuando, por ejemplo, quiero recuperar solo un *namespace*, un *deployment* o una *configmap* concreta sin necesidad de traer de vuelta todo el contenido de un backup.

Para hacer este tipo de restauración, hay que usar el comando `velero restore create`, al que puedo añadir diferentes opciones para limitar lo que quiero restaurar. Para la realización de estos ejemplos he montado un pod de wordpress en un namespace aparte, son los mismos pasos que realicé en el apartado [6.2.2](#):

```
pablo@ubuntu: ~/proyecto/Velero$ kubectl create namespace wordpress
namespace/wordpress created
pablo@ubuntu: ~/proyecto/Velero$ kubectl run wordpress --image=wordpress -n wordpress
pod/wordpress created
pablo@ubuntu: ~/proyecto/Velero$ kubectl get pods -n wordpress
NAME      READY   STATUS    RESTARTS   AGE
wordpress 0/1     ContainerCreating 0          10s
pablo@ubuntu: ~/proyecto/Velero$ kubectl get pods -n wordpress
NAME      READY   STATUS    RESTARTS   AGE
wordpress 1/1     Running   0           39s
pablo@ubuntu: ~/proyecto/Velero$ velero backup create respaldo-wordpress --include-namespaces wordpress --wait
Backup request "respaldo-wordpress" submitted successfully.
Waiting for backup to complete. You may safely press ctrl-c to stop waiting - your backup will continue in the background.
Backup completed with status: Completed. You may check for more information using the commands 'velero backup describe respaldo-wordpress' and 'velero backup logs respaldo-wordpress'.
pablo@ubuntu: ~/proyecto/Velero$ velero backup get
NAME      STATUS    ERRORS   WARNINGS   CREATED              EXPIRES   STORAGE LOCATION   SELECTOR
respaldo-bueno-prueba Completed 0         0          2025-06-02 20:20:31 +0200 CEST  27d       default             <none>
respaldo-prueba      Deleting  0         0          2025-06-02 19:52:29 +0200 CEST  27d       default             <none>
respaldo-wordpress    Completed 0         0          2025-06-04 22:22:29 +0200 CEST  29d       default             <none>
```

Una vez montado, ejecuto el siguiente comando para crear la restauración:

```
velero restore create restaurar-wordpress \
  --from-backup respaldo-wordpress \
  --include-namespaces wordpress
```

```
pablo@ubuntu: ~/proyecto/Velero$ velero restore create restaurar-wordpress \
  --from-backup respaldo-wordpress \
  --include-namespaces wordpress
Restore request "restaurar-wordpress" submitted successfully.
Run 'velero restore describe restaurar-wordpress' or 'velero restore logs restaurar-wordpress' for more details.
```

Con este comando le estoy diciendo a Velero que cree una restauración llamada `restaurar-wordpress`, utilizando como origen el backup `respaldo-wordpress`, pero solo quiero restaurar el namespace `wordpress`. Es decir, aunque ese backup tenga datos de otros namespaces, únicamente se recuperará el que yo he indicado.

También puedo restaurar recursos concretos dentro de un namespace. Por ejemplo, si quiero restaurar solo los `persistentvolumeclaims` del namespace `wordpress`, haría algo como esto:

```
velero restore create pvc-wordpress \  
  --from-backup respaldo-wordpress \  
  --include-resources persistentvolumeclaims \  
  --include-namespaces wordpress
```

```
pablo@ubuntu:~/proyecto/Velero$ velero restore create pvc-wordpress \  
  --from-backup respaldo-wordpress \  
  --include-resources persistentvolumeclaims \  
  --include-namespaces wordpress  
Restore request "pvc-wordpress" submitted successfully.  
Run 'velero restore describe pvc-wordpress' or 'velero restore logs pvc-wordpress' for more details.
```

En este caso, estoy filtrando tanto por tipo de recurso como por namespace, y únicamente se restauran los volúmenes persistentes.

Algo que también he probado y me ha parecido útil es el uso de **etiquetas**. Si durante el backup se guardaron objetos que tenían ciertas etiquetas, puedo decirle a Velero que solo restaure los que coincidan con una etiqueta concreta. Por ejemplo:

```
velero restore create restaurar-etiquetado \  
  --from-backup respaldo-wordpress \  
  --selector app=wordpress
```

```
pablo@ubuntu:~/proyecto/Velero$ velero restore create restaurar-etiquetado \  
  --from-backup respaldo-wordpress \  
  --selector app=wordpress  
Restore request "restaurar-etiquetado" submitted successfully.  
Run 'velero restore describe restaurar-etiquetado' or 'velero restore logs restaurar-etiquetado' for more details.
```

Este comando busca dentro del backup completo todos los recursos que tengan la etiqueta `app=wordpress` y solamente restaura esos.

Por último, también tengo la opción de **excluir recursos**. Esto es lo contrario a lo anterior: puedo decir qué *no* quiero restaurar. Por ejemplo:

```
velero restore create sin-servicios \  
  --from-backup respaldo-wordpress \  
  --exclude-resources services
```

```
pablo@ubuntu:~/proyecto/Velero$ velero restore create sin-servicios \
--from-backup respaldo-wordpress \
--exclude-resources services
Restore request "sin-servicios" submitted successfully.
Run 'velero restore describe sin-servicios' or 'velero restore logs sin-servicios' for more details.
```

Con esto, todos los recursos se restaurarán menos los `services`, lo cual puede ser útil si quiero reconfigurarlos manualmente o ya existen en el clúster y no quiero que los sobrescriba.

Para ver si se han creado correctamente las restauraciones, utilizo:

```
velero restore get
```

```
pablo@ubuntu:~/proyecto/Velero$ velero restore get
```

NAME	BACKUP	STATUS	STARTED	COMPLETED	ERRORS	WARNINGS	CREATED	SELECTOR
pvc-wordpress	respaldo-wordpress	Completed	2025-06-04 22:23:18 +0200 CEST	2025-06-04 22:23:18 +0200 CEST	0	0	2025-06-04 22:23:18 +0200 CEST	<none>
restaurar-etiquetado	respaldo-wordpress	Completed	2025-06-04 22:24:57 +0200 CEST	2025-06-04 22:24:57 +0200 CEST	0	0	2025-06-04 22:24:57 +0200 CEST	app=wordpress
restaurar-wordpress	respaldo-wordpress	Completed	2025-06-04 22:23:01 +0200 CEST	2025-06-04 22:23:01 +0200 CEST	0	1	2025-06-04 22:23:01 +0200 CEST	<none>
sin-servicios	respaldo-wordpress	Completed	2025-06-04 22:25:14 +0200 CEST	2025-06-04 22:25:15 +0200 CEST	0	1	2025-06-04 22:25:14 +0200 CEST	<none>

Como se puede observar dos de ellos han dado una advertencia, por lo tanto, para profundizar y ver mejor cual es el problema, se usa el siguiente comando:

```
velero restore describe restaurar-wordpress
```

```
pablo@ubuntu:~/proyecto/Velero$ velero restore describe restaurar-wordpress
```

Name: **restaurar-wordpress**
Namespace: **velero**
Labels: <none>
Annotations: <none>

Phase: **Completed**
Total items to be restored: 3
Items restored: 3

Started: 2025-06-04 22:23:01 +0200 CEST
Completed: 2025-06-04 22:23:01 +0200 CEST

Warnings:
Velero: <none>
Cluster: <none>
Namespaces:
wordpress: could not restore, Pod:wordpress already exists. Warning: the in-cluster version is different than the backed-up version

Backup: respaldo-wordpress

Namespaces:
Included: wordpress
Excluded: <none>

Según he estado buscando, este tipo de advertencia es completamente normal cuando se restauran recursos que ya existen en el clúster, como en este caso el Pod dentro del namespace `wordpress`.

Esto significa que durante la restauración, Velero detectó que ya existía un recurso con el mismo nombre (en este caso, un `Pod` llamado `wordpress`) y no lo sobrescribió para evitar

posibles problemas. Además, avisa de que la versión actual de ese recurso en el clúster no es exactamente la misma que la que estaba en el backup.

¿Qué puedo hacer para evitar este warning?

Dependiendo del escenario obviamente, pero estas son las opciones:

- **Eliminar previamente los recursos del clúster:** Si supiera que voy a restaurar un recurso y quiero que se aplique tal y como estaba en la copia de seguridad, puedo eliminar el recurso manualmente antes de lanzar la restauración. Por ejemplo:

```
kubectl delete pod wordpress -n wordpress
```

Y después se hace la restauración.

- **Restaurar en un namespace diferente:** Velero permite restaurar los datos en otro namespace, usando la opción `--namespace-mappings`. Así se pueden evitar colisiones:

```
velero restore create restaurar-wordpress-nuevo \  
  --from-backup respaldo-wordpress \  
  --namespace-mappings wordpress=wordpress-nuevo
```

Con esto, el contenido del backup que estaba en `wordpress` se restaurará en `wordpress-nuevo`.

6.6 Uso de etiquetas y selectors en backups

Velero es muy flexible a la hora de elegir **qué recursos quiero incluir** en una copia de seguridad. No siempre necesito hacer un backup de todo el clúster o de un namespace completo. A veces solo me interesa una aplicación concreta, un despliegue específico o incluso un tipo de recurso muy puntual. Aquí es donde entran en juego las **etiquetas** y los **selectors**.

¿Qué son las etiquetas en Kubernetes?

En Kubernetes, las etiquetas (labels) son pares clave-valor que se añaden a los recursos para clasificarlos o agruparlos. Por ejemplo, un Pod, un Deployment o un Service pueden tener algo como esto:

```
labels:  
  app: wordpress  
  entorno: produccion
```

Estas etiquetas me permiten, por ejemplo, seleccionar todos los recursos que pertenecen a la aplicación `wordpress`, sin importar el tipo de recurso o incluso el namespace.

Selección por etiquetas en Velero

Velero permite utilizar selectores de etiquetas con la opción `--selector` cuando hago un backup. Esto me da la posibilidad de realizar una copia **solo de los recursos que tienen ciertas etiquetas**. Esta funcionalidad está documentada claramente en la guía oficial y se basa en los selectores de Kubernetes.

Por ejemplo, si quiero hacer un backup únicamente de los recursos etiquetados con `app=wordpress`, puedo ejecutar:

```
velero backup create backup-etiquetado-wordpress --selector app=wordpress
```

```
pavlo@ubuntu:~$ velero backup create backup-etiquetado-wordpress --selector app=wordpress  
Backup request "backup-etiquetado-wordpress" submitted successfully.  
Run 'velero backup describe backup-etiquetado-wordpress' or 'velero backup logs backup-etiquetado-wordpress' for more details.
```

Velero va a recorrer todos los namespaces buscando recursos que tengan esa etiqueta y los incluirá en el backup. Es útil cuando quiero hacer una copia únicamente de una aplicación sin tocar el resto del entorno.

También puedo combinar varias etiquetas, por ejemplo:

```
velero backup create backup-produccion --selector 'entorno=produccion,app=wordpress'
```

En este caso, el backup solo incluirá recursos que tengan ambas etiquetas.

Una vez creado el backup, puedo comprobar qué contiene con:

```
velero backup describe backup-etiquetado-wordpress --details
```

```
pavlo@ubuntu:~$ velero backup describe backup-etiquetado-wordpress --details
Name:      backup-etiquetado-wordpress
Namespace: velero
Labels:    velero.io/storage-location=default
Annotations: velero.io/resource-timeout=10m0s
            velero.io/source-cluster-k8s-gitversion=v1.32.0
            velero.io/source-cluster-k8s-major-version=1
            velero.io/source-cluster-k8s-minor-version=32

Phase: Completed

Namespaces:
  Included:  *
  Excluded:  <none>

Resources:
  Included:  *
  Excluded:  <none>
  Cluster-scoped: auto

Label selector: app=wordpress
Or label selector: <none>

Storage Location: default

Velero-Native Snapshot PVs: auto
Snapshot Move Data:         false
Data Mover:                  velero

TTL: 720h0m0s
```

Y ahí veré los recursos que se incluyeron gracias a ese selector.

Ventajas de usar etiquetas y selectors

Utilizar etiquetas como filtro para backups tiene varias ventajas:

- Me permite **controlar mejor qué guardo** y evitar copias innecesarias.
- Es muy útil para **automatizar backups** de aplicaciones concretas.
- Me ayuda a **restaurar de forma más precisa** solo lo que necesito.

Además, esto se puede combinar con backups programados, como vimos en apartados anteriores.

6.7 Integración con Volume Snapshots

Cuando comencé a investigar sobre Velero, me encontré con que, además de realizar copias de seguridad de los recursos de Kubernetes (como pods, deployments, servicios, etc.),

también es capaz de hacer **copias de los volúmenes persistentes (PVCs)** mediante una funcionalidad muy interesante: las **snapshots de volúmenes**.

¿Qué es una snapshot de volumen?

Una snapshot es básicamente una **captura exacta del estado de un volumen en un momento determinado**. A diferencia de una copia de seguridad tradicional, que podría tardar más tiempo en transferir datos, la snapshot se realiza de forma casi instantánea a nivel de almacenamiento. Esto es especialmente útil para bases de datos o volúmenes que están en uso constante, ya que permite hacer un backup consistente sin necesidad de detener nada.

¿Cómo gestiona esto Velero?

Velero puede utilizar estas snapshots si el proveedor de almacenamiento en la nube (como AWS, Azure o GCP) lo permite y está correctamente configurado. Por ejemplo, en AWS, Velero usa las snapshots de EBS (Elastic Block Store), que se almacenan directamente en el backend de Amazon.

Pero en mi caso, **desactivé esta opción** en la instalación de Velero. Lo hice porque mi entorno de pruebas es local, no está en un proveedor cloud como AWS, y por lo tanto **no tengo acceso a un sistema de almacenamiento que soporte snapshots de forma nativa**.

¿Por qué son importantes las snapshots?

Aunque no las estoy usando ahora mismo, considero que es importante entender como funcionan:

- Son **más rápidas** que los backups por archivos.
- Permiten **copiar grandes volúmenes de datos** sin transferencias pesadas.
- Se almacenan **en el sistema de almacenamiento del proveedor**, lo que facilita su restauración directa.
- Son **consistentes a nivel de bloque**, ideal para bases de datos o volúmenes que cambian constantemente.

¿Y cómo se activarían?

Si en algún momento usara Velero en un entorno cloud con soporte de snapshots, bastaría con **activar la funcionalidad durante la instalación** y configurar correctamente el `volumeSnapshotLocation`. Por ejemplo, en AWS esto incluiría especificar la región y permitir a Velero crear snapshots EBS:

```
velero install \
  --provider aws \
  --plugins velero/velero-plugin-for-aws:v1.16.1 \
  --bucket mi-bucket \
  --backup-location-config region=eu-west-1 \
  --snapshot-location-config region=eu-west-1 \
  --use-volume-snapshots=true
```

Después, durante una restauración, Velero detectaría si un PVC fue respaldado con snapshot o con Restic y lo restauraría con el método correspondiente.

6.8 Restore Hooks (ganchos de restauración)

Una de las cosas que más me han gustado de Velero es que no solo puede restaurar recursos tal como estaban en un backup, sino que también me deja ejecutar comandos personalizados **inmediatamente después** de restaurarlos. A estos comandos adicionales se les llama **Restore Hooks**. En esta sección voy a explicar qué son, para qué sirven y luego mostraré un ejemplo práctico paso a paso para que quede claro cómo funcionan.

6.8.1 ¿Qué es un Restore Hook?

Un **Restore Hook** es un comando o conjunto de comandos que se ejecutan dentro de un contenedor específico, justo después de restaurar su pod correspondiente. Esto es muy útil cuando después de recuperar un recurso, necesito realizar alguna tarea extra. Por ejemplo:

- Ajustar permisos de archivos dentro del pod restaurado.
- Escribir datos en un archivo de log para hacer saber que la restauración ha terminado.
- Ejecutar scripts de inicialización o limpieza que solo se deben ejecutar después de que todo el recurso esté disponible.

En esta versión de Velero (v1.16), se definen los restore hooks dentro del objeto `Restore` bajo la sección `spec.hooks.resources[].postHooks`. Cada elemento de `postHooks` indica un comando que se ejecutará en el contenedor que yo elija. Para que esto funcione correctamente hay que especificar:

- **El recurso** que queremos que active el hook: por ejemplo, un Deployment o un Pod.
- **El namespace** donde existe ese recurso.
- **El contenedor** en el que se ejecutará el comando.
- **El/Los comando/s exacto/s** que quiero que se ejecuten.

Para realizar una demostración más visual de cómo funcionan los hooks, voy a realizar un caso práctico que tras restaurar un Deployment que levanta un pod con Nginx, deja un archivo en `/tmp/hooks/hook.log` dentro del contenedor nginx, de forma que puedo comprobar que el hook se ejecutó correctamente.

6.9 ¿Velero sirve para copias fuera de Kubernetes?

Velero funciona dentro del entorno de Kubernetes y solo interactúa con:

- Objetos de Kubernetes: deployments, services, configmaps, etc.
- Volúmenes persistentes montados como PVCs, si se activa el plugin de backup de volúmenes (Restic).
- El acceso a los datos se **realiza desde dentro de los pods**, no desde el host del nodo.

Por tanto, Velero **no puede acceder ni respaldar rutas arbitrarias del sistema anfitrión**, como `/home`, `/etc`, `/var` o cualquier otra, a menos que:

- Se monten esos directorios como volúmenes en pods (no recomendable).
- Crear un contenedor que los acceda (solución forzada).

7. Demo/s

7.1. Demo 1: Recuperación Deployment y Service de Nextcloud

Voy a hacer una demostración de cómo Velero me permite hacer copias de seguridad de una aplicación en Kubernetes y restaurarla después de una "catástrofe". Para ello usaré la aplicación Nextcloud, desplegada con su volumen persistente. La copia se almacenará en MinIO, que tengo previamente configurado como repositorio de backups para Velero.

Paso 1: Crear un namespace exclusivo

Primero creo un namespace donde desplegaré Nextcloud. Esto ayuda a tener todo más organizado y luego hacer backups y restauraciones fácilmente:

```
kubectl create namespace nextcloud
```

```
pablo@ubuntu:~/proyecto/demo$ kubectl get namespaces
NAME                STATUS    AGE
default              Active    27h
ingress-nginx        Active    27h
kube-node-lease      Active    27h
kube-public          Active    27h
kube-system          Active    27h
prueba-velero        Active    27h
velero                Active    4h3m
pablo@ubuntu:~/proyecto/demo$ kubectl create namespace nextcloud
namespace/nextcloud created
```

Verifico que el namespace existe:

```
kubectl create namespace nextcloud
```

```
pablo@ubuntu:~/proyecto/demo$ kubectl get namespaces
NAME                STATUS    AGE
default              Active    27h
ingress-nginx        Active    27h
kube-node-lease      Active    27h
kube-public          Active    27h
kube-system          Active    27h
nextcloud            Active    3s
prueba-velero        Active    27h
velero                Active    4h4m
```

Paso 2: Desplegar Nextcloud con YAML

Tengo preparado un fichero YAML con el [despliegue](#) de Nextcloud, que incluye:

- Un PVC para guardar los datos
- Un Deployment para lanzar el contenedor
- Un Service para exponer la aplicación

Lo aplico con:

```
kubectl apply -f nextcloud-deploy.yaml -n nextcloud
```

```
pablo@ubuntu:~/proyecto/demo$ ls -l
total 4
-rw-r--r-- 1 root root 895 jun  9 18:28 nextcloud-deployment.yaml
```

```
pablo@ubuntu:~/proyecto/demo$ kubectl apply -f nextcloud-deployment.yaml -n nextcloud
persistentvolumeclaim/nextcloud-pvc created
deployment.apps/nextcloud created
service/nextcloud created
```

Ahora compruebo que todo se ha creado correctamente:

```
kubectl get all -n nextcloud
kubectl get pvc -n nextcloud
```

```
pablo@ubuntu:~/proyecto/demo$ kubectl get all -n nextcloud
NAME                                READY   STATUS    RESTARTS   AGE
pod/nextcloud-685989f7bc-cz5cv      1/1     Running   0           9s

NAME                                TYPE          CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
service/nextcloud                   ClusterIP     10.108.170.181 <none>       80/TCP     9s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nextcloud           1/1     1             1           9s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/nextcloud-685989f7bc 1         1         1       9s
```

```
pablo@ubuntu:~/proyecto/demo$ kubectl get pvc -n nextcloud
NAME          STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
nextcloud-pvc Bound    pvc-98ccea76-adad-4609-886a-d4c2cfa6e3d5 1Gi        RWO            standard       <unset>                18s
```

Espero a que el pod de Nextcloud esté en estado **Running**.

Paso 3: Acceder a Nextcloud

Como no estoy usando Ingress, accedo a Nextcloud con un port-forward:

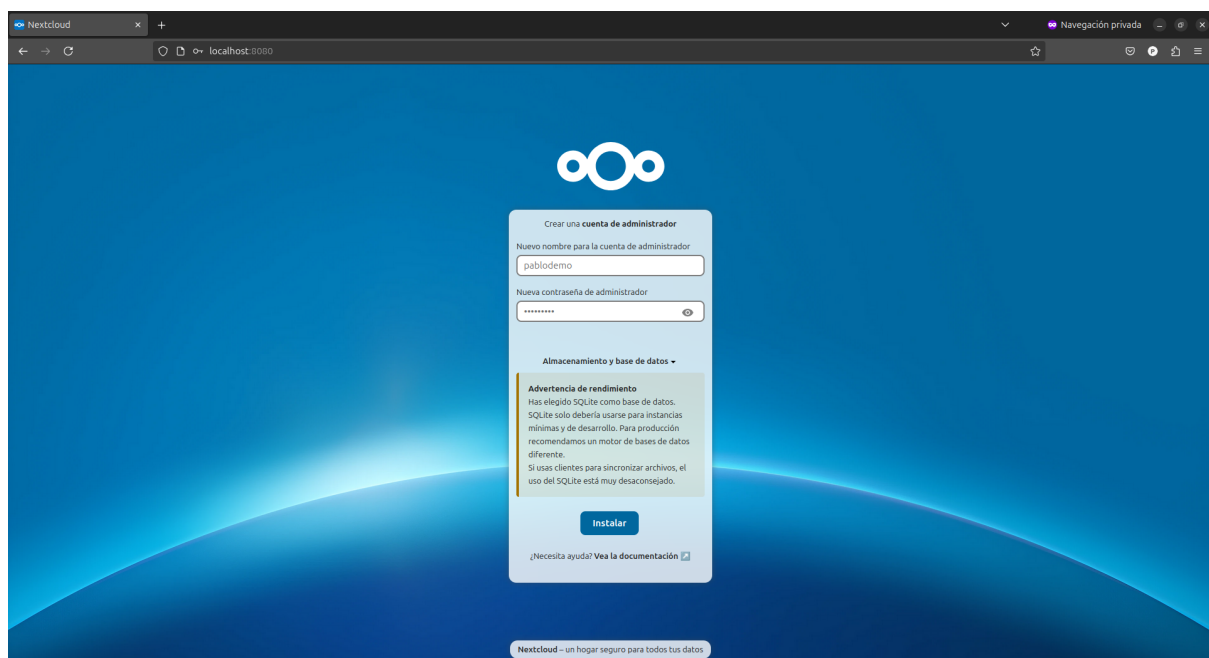
```
kubectl port-forward svc/nextcloud 8080:80 -n nextcloud
```

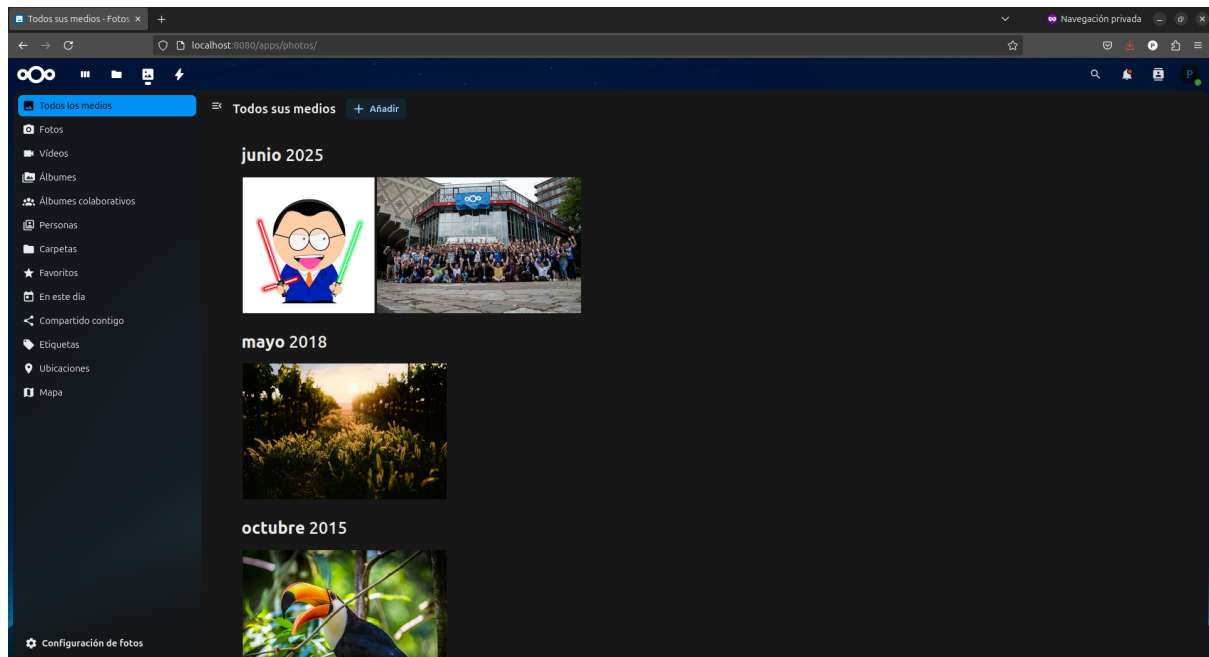
```
pablo@ubuntu:~/proyecto/demo$ kubectl port-forward svc/nextcloud 8080:80 -n nextcloud
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
```

Abro el navegador y entro en:

👉 <http://localhost:8080>

Aquí configuro Nextcloud por primera vez, creo un usuario y subo un archivo cualquiera como prueba (por ejemplo una imagen) para que luego cuando lo recupere, se pueda comprobar que sigue todo tal cual.





Paso 4: Hacer un backup de los recursos

Con la aplicación ya funcionando, hago una copia de seguridad usando Velero. Este backup solo guarda los recursos de Kubernetes, como el Deployment, el PVC y el Service. El contenido del volumen queda intacto en el PVC.

Lanzo el backup:

```
velero backup create nextcloud-backup --include-namespaces nextcloud
```

```
pablo@ubuntu:~/proyecto/demo$ velero backup create nextcloud-backup --include-namespaces nextcloud
Backup request "nextcloud-backup" submitted successfully.
Run 'velero backup describe nextcloud-backup' or 'velero backup logs nextcloud-backup' for more details.
```

Compruebo que se haya creado:

```
velero backup get
```

```
pablo@ubuntu:~/proyecto/demo$ velero backup get
NAME                STATUS    ERRORS    WARNINGS    CREATED                EXPIRES    STORAGE LOCATION    SELECTOR
nextcloud-backup    Completed 0          0           2025-06-09 23:12:23 +0200 CEST    29d        default              <none>
```

Verifico el estado:

```
velero backup describe nextcloud-backup --details
```

```
pablo@ubuntu:~/proyecto/demo$ velero backup describe nextcloud-backup --details
Name:          nextcloud-backup
Namespace:     velero
Labels:        velero.io/storage-location=default
Annotations:   velero.io/resource-timeout=10m0s
               velero.io/source-cluster-k8s-gitversion=v1.32.0
               velero.io/source-cluster-k8s-major-version=1
               velero.io/source-cluster-k8s-minor-version=32

Phase:  Completed

Namespaces:
  Included:  nextcloud
  Excluded:  <none>

Resources:
  Included:  *
  Excluded:  <none>
  Cluster-scoped:  auto

Label selector:  <none>

Or label selector:  <none>

Storage Location:  default

Velero-Native Snapshot PVs:  auto
Snapshot Move Data:         false
Data Mover:                 velero

TTL:  720h0m0s

CSISnapshotTimeout:  10m0s
ItemOperationTimeout: 4h0m0s
```

Aquí confirmo que Velero ha incluido los objetos esperados. Aunque no copia los datos internos del PVC, estos siguen almacenados en el clúster.

Paso 5: Simular un desastre

Ahora voy a simular que algo ha fallado en Nextcloud, por ejemplo, el Deployment y el Service se han eliminado.

```
kubectl delete deployment nextcloud -n nextcloud
kubectl delete service nextcloud -n nextcloud
```

```
pablo@ubuntu:~/proyecto/demo$ kubectl delete deployment nextcloud -n nextcloud
deployment.apps "nextcloud" deleted
pablo@ubuntu:~/proyecto/demo$ kubectl delete service nextcloud -n nextcloud
service "nextcloud" deleted
```

Compruebo que ya no están:

```
kubectl get all -n nextcloud
```

```
pablo@ubuntu:~/proyecto/demo$ kubectl get all -n nextcloud
No resources found in nextcloud namespace.
```

Paso 6: Restaurar con Velero

Ahora simulo que ha pasado un rato y decido restaurar la aplicación. Uso Velero para volver a crear los recursos:

```
velero restore get
velero restore create --from-backup nextcloud-backup
```

```
pablo@ubuntu:~/proyecto/demo$ velero restore get
NAME                                BACKUP                                STATUS  STARTED                                COMPLETED                                ERRORS  WARNINGS  CREATED                                SELECT
OR
nextcloud-backup-20250609224231     nextcloud-backup                     Completed  2025-06-09 22:42:31 +0200 CEST          2025-06-09 22:42:32 +0200 CEST          0       1         2025-06-09 22:42:31 +0200 CEST          <none>
```

```
pablo@ubuntu:~/proyecto/demo$ velero restore create --from-backup nextcloud-backup
Restore request "nextcloud-backup-20250609231432" submitted successfully.
Run `velero restore describe nextcloud-backup-20250609231432` or `velero restore logs nextcloud-backup-20250609231432` for more details.
```

Espero unos segundos y consulto el estado:

```
velero restore get
```

```
pablo@ubuntu:~/proyecto/demo$ velero restore get
NAME                                BACKUP                                STATUS  STARTED                                COMPLETED                                ERRORS  WARNINGS  CREATED                                SELECT
OR
nextcloud-backup-20250609224231     nextcloud-backup                     Completed  2025-06-09 22:42:31 +0200 CEST          2025-06-09 22:42:32 +0200 CEST          0       1         2025-06-09 22:42:31 +0200 CEST          <none>
nextcloud-backup-20250609231432     nextcloud-backup                     Completed  2025-06-09 23:14:32 +0200 CEST          2025-06-09 23:14:33 +0200 CEST          0       1         2025-06-09 23:14:32 +0200 CEST          <none>
```

Luego compruebo que el Deployment y el Service han vuelto:

```
kubectl get all -n nextcloud
```

```
pablo@ubuntu:~/proyecto/demo$ kubectl get all -n nextcloud
NAME                                READY  STATUS   RESTARTS  AGE
pod/nextcloud-685989f7bc-cz5cv      1/1    Running  0          45s

NAME                                TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
service/nextcloud                   ClusterIP    10.97.15.13   <none>        80/TCP     44s

NAME                                READY  UP-TO-DATE  AVAILABLE  AGE
deployment.apps/nextcloud           1/1    1            1           44s

NAME                                DESIRED  CURRENT  READY  AGE
replicaset.apps/nextcloud-685989f7bc 1        1        1       44s
```

Y por último compruebo en detalle que tal ha ido la restauración.

```
velero restore describe nextcloud-backup-20250609231432
```

```
pablo@ubuntu: ~/proyecto/demo$ velero restore describe nextcloud-backup-20250609231432
Name:          nextcloud-backup-20250609231432
Namespace:     velero
Labels:        <none>
Annotations:   <none>

Phase:          Completed
Total items to be restored: 10
Items restored: 10

Started:    2025-06-09 23:14:32 +0200 CEST
Completed:  2025-06-09 23:14:33 +0200 CEST

Warnings:
  Velero:    <none>
  Cluster:   <none>
  Namespaces:
    nextcloud: could not restore, PersistentVolumeClaim:nextcloud-pvc already exists. Warning: the in-cluster version is different than the backed-up version
Backup: nextcloud-backup

Namespaces:
  Included:  all namespaces found in the backup
  Excluded:  <none>

Resources:
  Included:  *
  Excluded:  nodes, events, events.events.k8s.io, backups.velero.io, restores.velero.io, resticrepositories.velero.io, csinodes.storage.k8s.io, volumeattachments.storage.k8s.io, backup
repositories.velero.io
Cluster-scoped: auto
```

Paso 7: Comprobar que ha vuelto el servicio

Vuelvo a hacer port-forward:

```
kubectl port-forward svc/nextcloud 8080:80 -n nextcloud
```

```
pablo@ubuntu: ~/proyecto/demo$ kubectl port-forward svc/nextcloud 8080:80 -n nextcloud
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

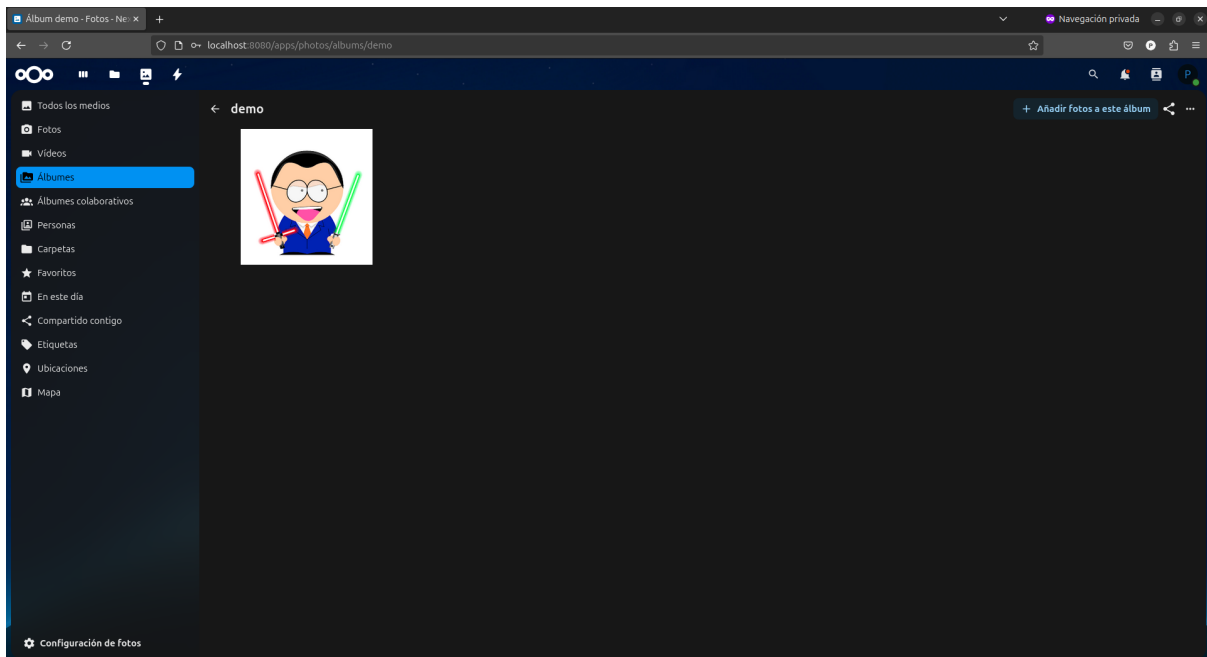
Abro de nuevo el navegador en <http://localhost:8080>.

Y aquí compruebo que:

- No aparece el asistente de instalación.
- El usuario que creé sigue estando.
- El archivo de prueba también está.

Esto significa que Velero ha restaurado correctamente los recursos necesarios para que Nextcloud vuelva a estar operativo como antes del desastre.

Aquí podemos observar que sigue funcionando a la perfección:



7.2 Demo 2: Restore Hook

A continuación describo cada paso que he hecho para probar un Restore Hook en mi entorno. Parto de cero dentro de los namespaces que ya tenía disponibles:

- **default:** aquí vamos a desplegar el Deployment de nginx.
- **velero:** aquí “habita” Velero y será donde crearé el objeto `Restore` con el hook.

```
pablo@ubuntu:~/proyecto/Velero$ kubectl get namespaces
NAME                STATUS    AGE
default             Active   29d
ingress-nginx       Active   4d1h
kube-node-lease     Active   29d
kube-public         Active   29d
kube-system         Active   29d
prueba-velero       Active   2d23h
velero              Active   2d23h
wordpress          Active   21h
```

1. Crear el Deployment de nginx en `default`

Primero, en mi directorio de trabajo (`~/proyecto/Velero`) creo un archivo llamado [deployment-nginx.yaml](#).

Este Deployment levanta un pod que ejecuta la imagen oficial de Nginx. Para aplicarlo:

```
kubectl apply -f deployment-nginx.yaml
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl apply -f deployment-nginx.yaml
deployment.apps/nginx-deployment created
```

Y compruebo que el pod esté listo con:

```
kubectl get pods -n default -l app=nginx
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl get pods -n default -l app=nginx
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-764dd87c46-v4qtv  1/1     Running   0           102s
```

Con esto confirmo que el pod de nginx funciona correctamente en el namespace `default`.

2. Crear un backup que incluya Deployment y Pod

Ahora, usando Velero, creo un backup que guarde tanto el Deployment como el pod asociado. Para ello ejecuto:

```
velero backup create backup-nginx --include-resources deployments,pods
```

```
pablo@ubuntu:~/proyecto/Velero$ velero backup create backup-nginx --include-resources deployments,pods
Backup request "backup-nginx" submitted successfully.
Run 'velero backup describe backup-nginx' or 'velero backup logs backup-nginx' for more details.
```

Espero a que finalice para verificar el estado del backup:

```
velero backup describe backup-nginx
```

```
pablo@ubuntu:~/proyecto/Velero$ velero backup describe backup-nginx
Name:      backup-nginx
Namespace: velero
Labels:    velero.io/storage-location=default
Annotations: velero.io/resource-timeout=10m0s
             velero.io/source-cluster-k8s-gitversion=v1.32.0
             velero.io/source-cluster-k8s-major-version=1
             velero.io/source-cluster-k8s-minor-version=32

Phase: Completed

Namespaces:
Included:  *
Excluded: <none>

Resources:
Included:  deployments, pods
Excluded:  <none>
Cluster-scoped: auto
```

Tal y como se puede ver en el parámetro “Phase” dice que se ha completado.

3. Eliminar el Deployment para simular una pérdida

Para probar la restauración, elimino el Deployment de nginx (lo cual también borra su pod):

```
kubectl delete deployment nginx-deployment -n default
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl delete deployment nginx-deployment -n default
deployment.apps "nginx-deployment" deleted
```

Compruebo que el pod haya desaparecido:

```
kubectl get pods -n default -l app=nginx
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl get pods -n default -l app=nginx
No resources found in default namespace.
```

El resultado muestra que el entorno quedó sin el Deployment ni el pod de nginx, simulando un escenario de desastre.

4. Definir el Restore con el hook

El siguiente archivo es el **Restore** que incluye el **Restore Hook**. Lo he guardado como [restore-nginx-hook.yaml](#):

Voy a desglosar cada sección:

- **apiVersion: velero.io/v1** y **kind: Restore**: defino un recurso de Velero de tipo **Restore**.
- **metadata.name: restore-nginx** y **metadata.namespace: velero**: el objeto se llama **restore-nginx** y se encuentra en el namespace **velero**, que es donde Velero busca los restores por defecto.
- **spec.backupName: backup-nginx**: indico que la restauración se base en el backup **backup-nginx** que creé antes.
- **spec.hooks.resources**: lista de recursos que activan hooks. En este caso, uso:
 - **name: deployment/nginx-deployment**: el hook se aplica al Deployment **nginx-deployment**.
 - **includedNamespaces: [default]**: limito el ámbito a los recursos que estaban en el namespace **default**.
 - **postHooks**: los comandos dentro de esta lista se ejecutan **después** de que el Deployment y su Pod hayan sido restaurados.
 - **exec**: significa que quiero ejecutar un comando dentro del contenedor.
 - **container: nginx**: nombre del contenedor dentro del Pod restaurado donde correrá el comando.
 - **command**: lista de strings que conforma el comando completo.

Con esta configuración, cada vez que se restaure el Deployment de nginx, Velero volverá a crear el Pod y luego ejecutará el comando anterior para dejar un registro dentro del contenedor.

5. Aplicar el Restore con hook

Para lanzar la restauración con el hook, ejecuto:

```
kubectl apply -f restore-nginx-hook.yaml
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl apply -f restore-nginx-hook.yaml
restore.velero.io/restore-nginx created
```

Esto indica que Kubernetes aceptó el objeto Restore y Velero comenzó a procesarlo.

Para comprobar el estado de la restauración, ejecuto el siguiente comando para abrirlo en detalle:

```
velero restore describe restore-nginx
```

```
pablo@ubuntu:~/proyecto/Velero$ velero restore describe restore-nginx
Name:          restore-nginx
Namespace:     velero
Labels:        <none>
Annotations:   kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"velero.io/v1","kind":"Restore","metadata":{"annotations":{"name":"restore-nginx","namespace":"velero"},"spec":{"backupName":"backup-nginx","hooks":{"resources":[{"includedNamespaces":["default"],"name":"deployment/nginx-deployment","postHooks":[{"exec":{"command":["/bin/sh","-c","mkdir -p /tmp/hooks\n\u0026\u0026\u0026 necho 'Restauración completa: Hook ejecutado' \u0026\u0026\u0026 /tmp/hooks/hook.log\n"],"container":"nginx"}}]}}}]}
Phase:         Completed
Total items to be restored: 18
Items restored: 18
Started:       2025-06-05 19:49:55 +0200 CEST
Completed:     2025-06-05 19:49:56 +0200 CEST
```

```
HooksAttempted: 1
HooksFailed:    0
```

De esta forma puedo confirmar que Velero completó la tarea de restaurar todos los ítems y trató un postHook que no falló (HooksFailed: 0).

7. Confirmar que el hook se ejecutó en el pod restaurado

Ya se ha restaurado el Deployment y se ha puesto en marcha el pod con la etiqueta `app=nginx`. Se muestra de la siguiente manera:

```
kubectl get pods -n default -l app=nginx
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl get pods -n default -l app=nginx
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-764dd87c46-v4qtv  1/1     Running   0           105s
```

Por último y no menos importante, ver cómo el mensaje se ha guardado justamente donde habíamos especificado.

```
kubectl exec -it nginx-deployment-764dd87c46-v4qtv -n default -- cat
/tmp/hooks/hook.log
```

```
pablo@ubuntu:~/proyecto/Velero$ kubectl exec -it nginx-deployment-764dd87c46-v4qtv -n default -- cat /tmp/hooks/hook.log
Restauración completa: Hook ejecutado
```

8. Conclusiones y propuestas

Personalmente, este proyecto me ha enseñado lo importante que es tener un buen sistema de copias de seguridad en Kubernetes. He podido comprobar que es posible proteger aplicaciones y datos de forma local, sin depender de servicios de pago ni de nubes externas. Me ha sorprendido lo sencillo que resulta recuperar un entorno completo con un solo comando, lo cual aporta mucha tranquilidad ante fallos o borrados accidentales.

Para futuros trabajos, me gustaría explorar cómo programar backups de forma automática, quizá usando cronjobs de Kubernetes o integrándolo con GitOps para que todo quede documentado en un repositorio. También sería interesante añadir cifrado de los datos en reposo y en tránsito, y probar a hacer backups incrementales para ahorrar espacio.

9. Dificultades que se han encontrado

Conexión TLS entre Velero y MinIO

Al principio Velero no era capaz de comunicarse con MinIO por el certificado autofirmado. Tardé un buen rato en descubrir que había que generar y añadir el certificado (mkcert) al Ingress de MinIO para que Velero aceptara la conexión HTTPS.

Hooks de restauración que no aparecían

En una de las pruebas tuve que configurar el Restore Hook de Velero, lo cual fue más complicado de lo que yo esperaba. Al principio usé la ruta `/var/log/nginx/access.log` y no funcionó porque la imagen de Nginx no escribía ahí. Tras intentarlo varias veces opté por hacerlo en el directorio `/tmp` que siempre existe en el contenedor y funcionó perfectamente.

Reintentos con Helm (Kibana/Elasticsearch)

Intenté instalar Kibana y Elasticsearch con Helm, pero colapsó el clúster por falta de recursos y tuve que eliminarlo, con todo el escenario montado. Helm quedó con releases “fantasma” y recursos huérfanos que tuve que borrar uno a uno. Al final decidí no incluirlos por falta de tiempo.

Errores de sintaxis en YAML

Varias veces un espacio o la indentación mal puesta en un archivo YAML de Kubernetes me dio problemas de “strict decoding” o recursos inválidos.

10. Bibliografía

- **Velero**
 - [Documentación oficial](#)
 - [GitHub Velero \(ejemplos y código\)](#)
- **MinIO**
 - [Guía de instalación en Kubernetes](#)
 - [Cliente mc \(comandos básicos\)](#)
- **mkcert (certificados locales)**
 - [mkcert en GitHub](#)
 - [Guía de instalación](#)