



UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY



FACULTAD DE
INGENIERÍA

Kubernetes y helm: deploy automático en tres capas

Informe presentado por

Martín Giachino

Supervisores

Edgar Magaña
Eduardo Grampín

Montevideo, 16 de diciembre de 2023



Kubernetes y helm: deploy automático en tres capas por
Martín Giachino tiene licencia [CC Atribución 4.0](#).

Resumen

El producto del presente trabajo es un artefacto que permite automatizar y replicar en iguales condiciones, una infraestructura completa de servicios que componen una aplicación en tres capas completa. Para esto se utilizan tecnologías de contenedores (*docker*), orquestación y parametrización de los mismos (*kubernetes*, *dockerDesktop* y *helm*). El resultado es un conjunto de archivos de configuración, los cuales son suficientes para configurar y levantar una infraestructura funcional, con el beneficio adicional de ser replicable en otro ambiente. Para evaluar que el resultado es el esperado, se eliminó el proyecto completamente y se pudo comprobar que fue posible replicar nuevamente toda la infraestructura con solo volver a ejecutar el orquestador, tomando como parámetro los archivos de configuración mencionados anteriormente.

Dada la naturaleza del curso, otro objetivo muy importante es poder explorar los conceptos y estructuras para armar estos despliegues.

Se concluye que efectivamente, *kubernetes* y *helm* son una combinación de herramientas muy beneficiosas, y que permiten hacer despliegue de infraestructura desde archivos de configuración de texto.

Palabras clave: kubernetes, helm, docker, dockerDesktop, containers

Introducción

Como trabajo final del curso de kubernetes se plantea generar un despliegue automático de toda la infraestructura necesaria de una aplicación que esté compuesta por tres capas (*front-end*, *back-end* y *storage* persistente). Este despliegue deberá estar contenido únicamente en archivos manifesto que serán leídos por una herramienta, que luego hará la instanciación e instalación de todos los componentes de la infraestructura. El producto del presente trabajo es un artefacto que permite automatizar y replicar en iguales condiciones, una infraestructura completa de servicios que componen una aplicación en tres capas completa.

Para esto se utilizan tecnologías de contenedores (*docker*¹), orquestación y parametrización de los mismos (*kubernetes*², *dockerDesktop*³ y *helm*⁴). El resultado es un conjunto de archivos de configuración, lo cuales son suficientes para configurar y levantar una infraestructura funcional, con el beneficio adicional de ser replicable en otro ambiente. Para evaluar que el resultado es el esperado, se eliminó el proyecto completamente y se pudo comprobar que fue posible replicar nuevamente toda la infraestructura con solo volver a ejecutar el orquestador, tomando como parámetro los archivos de configuración mencionados anteriormente.

Dada la naturaleza del curso, otro objetivo muy importante es poder explorar los conceptos y estructuras para armar estos despliegues. Particularmente este trabajo trata de explorar muchos componentes del software de base que se utiliza, para poder lograr variadas funcionalidades y por lo tanto lograr comprender en detalle las estructuras y componentes estudiados en el curso.

Se concluye que efectivamente este conjunto de herramientas son muy adecuadas para hacer despliegues de infraestructura desde archivos de configuración de texto, que son funcionales y que permiten replicar la misma infraestructura en otros ambientes. Adicionalmente se entiende que se logra explorar y lograr un uso amplio de conceptos de kubernetes y helm.

¹<https://www.docker.com/>

²<https://kubernetes.io/es/>

³<https://www.docker.com/products/docker-desktop/>

⁴<https://helm.sh/>

Despliegue propuesto

En esta sección se describe el diagrama que será configurado y desplegado. El objetivo principal del mismo es obligar a que la configuración deba hacer uso extenso de estructuras y conceptos, más que apuntar a la relevancia o complejidad de la aplicación que se instala. Se entiende que el objetivo del curso es aprender los conceptos de kubernetes, y por eso es el enfoque escogido para armar el esquema de arquitectura.

Herramientas utilizadas

Las herramientas utilizadas para desarrollar este proyecto son descritas en esta sección.

kubernetes

El orquestador de contenedores utilizado para todo el trabajo.

docker

Esta tecnología es utilizada de forma indirecta, ya que en realidad en esta solución son manejados por *kubernetes*.

docker-desktop

Es la aplicación que implementa *kubernetes*. Dado que este trabajo se realizó tanto en *Linux* como *Windows*, se pudo comprobar la interoperabilidad del software al pasar los deploy de uno a otro ambiente.

helm

El software que permite crear y gestionar los charts para automatizar despliegues automáticos y complejos.

GitHub

El manejador de versiones GitHub ⁵ es utilizado durante el desarrollo y para la publicación de los fuentes. Adicionalmente fue utilizado como repositorio *helm* para publicar el package generado.

GitHub Desktop

Se utilizó *GitHub Desktop* ⁶ para sincronizar los fuentes y manejar el control de versiones de lo generado en el proyecto.

⁵<https://github.com/>

⁶<https://desktop.github.com/>

Diagramas del despliegue

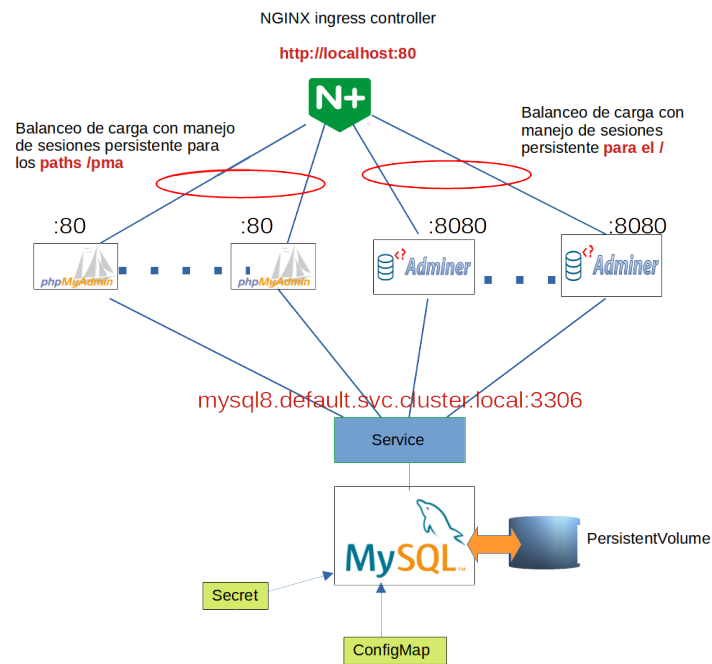
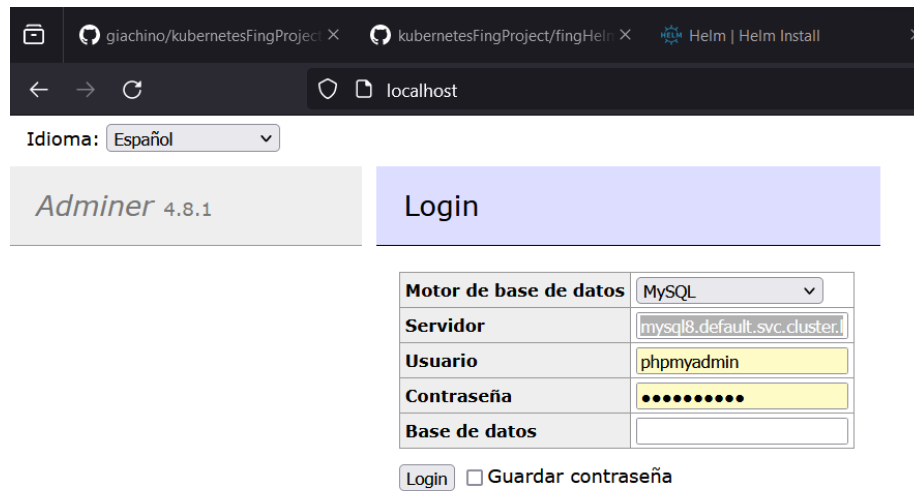


Figura 1: Despliegue de la arquitectura propuesta



Idioma: Español

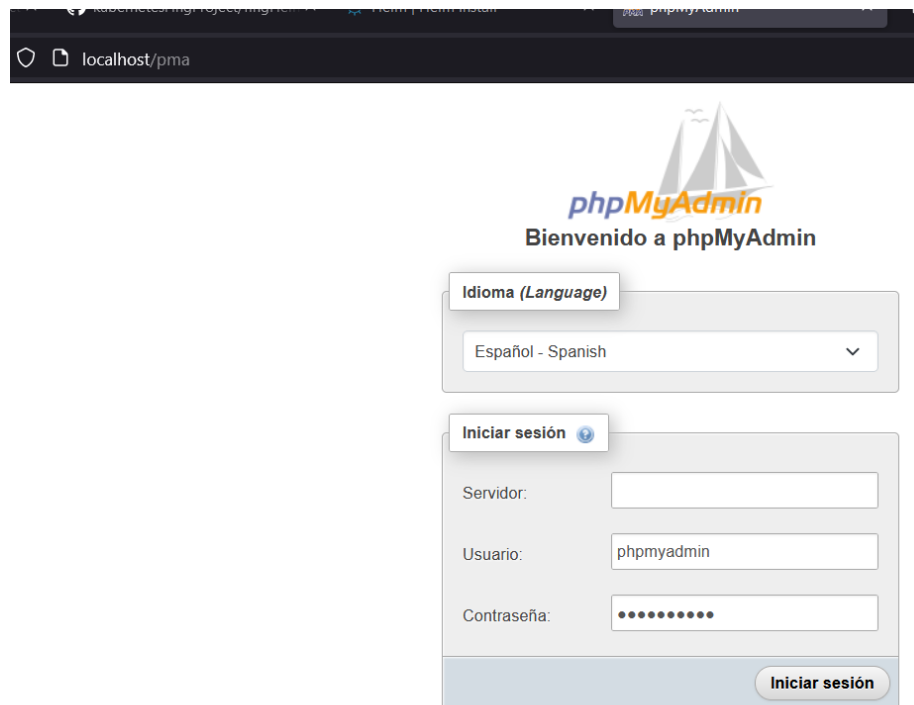
Adminer 4.8.1

Login

Motor de base de datos	MySQL
Servidor	mysql8.default.svc.cluster
Usuario	phpmyadmin
Contraseña	••••••••
Base de datos	

☐ Guardar contraseña

Figura 2: Acceso a adminer por el puerto 80 al /



phpMyAdmin

Bienvenido a phpMyAdmin

Idioma (Language)

Español - Spanish

Iniciar sesión

Servidor:

Usuario: phpmyadmin

Contraseña: ••••••••

Iniciar sesión

Figura 3: Acceso a phpmyadmin por el puerto 80 al /pma

Conceptos aplicados

Dado el objetivo de poder explorar muchos conceptos, intentando ampliar los conocimientos adquiridos en el curso, es que se toman decisiones de características a implementar para obligar al uso de ciertas configuraciones y características especiales.

Persistencia de los datos

Para esto se definen un *PersistentVolume* y un *PersistentVolumeClaim*. En este caso la clase de storage es local, y es mapeado a un directorio del host. La idea es que, dado que el mysql permite definirle el directorio de datos de forma parametrizada, se hace que el *PersistentVolume* sea montado en el contenedor en el directorio correspondiente a donde se guardan las bases en mysql.

Limitación: dado que es una prueba de concepto, que corremos todos en un solo nodo y que usamos almacenamiento de tipo local, esto hace que estemos restringidos a que los datos están en ese lugar y no pueden ser accedidos de forma remota (como por ejemplo se podría si se usaba almacenamiento de tipo *nfs*).

Uso de FQDN

Para resolver el acceso a la base de datos para los frontends, dada la característica de dinámica de las direcciones IP asignadas a los servicios entre reinicios, es que se revisa la forma de identificar al contendor mysql con el FQDN. Siguiendo las definiciones de kubernetes sabemos que el FQDN se arma como *nombre contenedor.namespace.svc.cluster.local*

Limitación: al igual que en el caso anterior, el ser local y en un solo nodo puede limitar que la solución funcione en un ambiente con múltiples nodos.

Uso de LoadBalancer

Se utilizó para balancear y generar accesibilidad desde fuera del POD, habilitando acceso desde el host a través de *localhost*. Al usarlo para mysql esto habilita también el uso del *FQDN*.

Uso de ConfigMap

Su uso fue simplemente para probar otra funcionalidad común. En este caso era sencillo haber utilizado los valores dentro del `values.yaml`, pero la idea era explorar y probar funcionalidades y estructuras.

Uso de Secret

Igual que en el caso de *ConfigMap*, pero en este caso se decide el uso de *Secret* ya que se pasaban valores de contraseñas. Es cierto que no era el mecanismo más seguro para las passwords, pero es más adecuado que un *ConfigMap*.

Persistencia de sesiones HTTP

Este fue un punto bien interesante y desafiante para resolver. El LoadBalancer balancea en *Round Robin* cada una de los *HTTP Request*. Por este motivo en la primer prueba que se hizo con *LoadBalancer* lo que se obtenía era que los frontends no se podían utilizar ya que cada pedido iba a un backend diferente, y el flujo de trabajo no funcionaba.

Para resolverlo se utilizó el *NGINX ingress controller*, que permite hacer balanceo de carga de forma *stateful* en base a una cookie que se define dentro mismo de *NGINX* de esta forma, las sesiones se balancean de forma persistente en base a esa cookie (hasta que expiran las cookies, que es algo configurable), y permite que el mismo cliente con esa cookie de sesión continúe con el mismo frontend, y se pueda trabajar con normalidad.

Selección de backends según URL PATH

Para poner un paso más de complejidad es que se toma el camino de tener dos pools de frontends con aplicaciones diferentes. esto obliga a generar dos configuraciones para *nginx*. Es interesante notar que fue necesario entender la sintaxis y las reglas de evaluaciones de como se procesan y como son las reglas de preferencia si hay mas de una opción para la misma URL PATH.

En este caso fue necesario entender bien el mecanismo de elección, y se estudió la configuración de expresiones regulares y rewrite de URL PATH previo al reenvío.

Uso de NodePort

El uso de NodePort fue como prueba de concepto para confirmar que al tener el *nginx ingress* delante, no era necesario el uso de un LoadBalancer y que el routing hacia el frontend se hacía en base al selector del POD, que en nuestro caso se correspondía con el nombre de la aplicación.

Conclusiones

Las pruebas realizadas fueron satisfactorias funcionalmente, y se logró generar el deploy en tres capas. Se logró generar un chart helm, empaquetarlo y hasta crear un repositorio para instalaciones automáticas y remotas. Se logró hacer un despliegue con cierta variedad conceptual y de componentes, lo que obligó a resolver algunos problemas interesantes.

Se logró exponer dos pools de aplicaciones que mantienen estado, pero que el estado es por cada POD y no global. Eso generó un problema extra, y fue la necesidad de balancear en base a que una sesión una vez establecida, debía mantenerse en el mismo frontend.