CI/CD - Helm sobre Openshift utilizando Tekton & Kubeseal

Antonio Rafael Marchán Posada

1. Introducción al proyecto.

Durante las prácticas hemos estado utilizando la plataforma Openshift, lo cual está especializada en dar servicio a pods de Kubernetes, otorgándonos diferentes utilidades que mostraré durante la documentación del proyecto.

1.1 Objetivo de la integración contínua con Helm sobre Openshift.

El objetivo a conseguir es la integración contínua y el despliegue contínuo de un chart de Helm sobre Openshift, con pods totalmente operativos y una URL externa con la que podamos visualizar la aplicación, esto sumado a un webhook que enlazaremos a nuestro repositorio de git. Es útil cuando necesitamos hacer cambios en nuestro chart y evitar despliegues y configuraciones repetitivas.

1.2 Es necesario este proyecto?

El proyecto nace de mi interés sobre Helm, ya que es una manera fácil de desplegar unos pods en un clúster con opciones preconfiguradas, al tener Openshift la posibilidad de integrar charts de helm, surgió la idea de si podría realizar integración y despliegue contínuo del mismo, ya que sirve tanto para mejorar mis skills sobre la plataforma como para dar un punto de vista diferente a los despliegues dentro de la plataforma.

Objetivos fundamentales a conseguir en este proyecto:

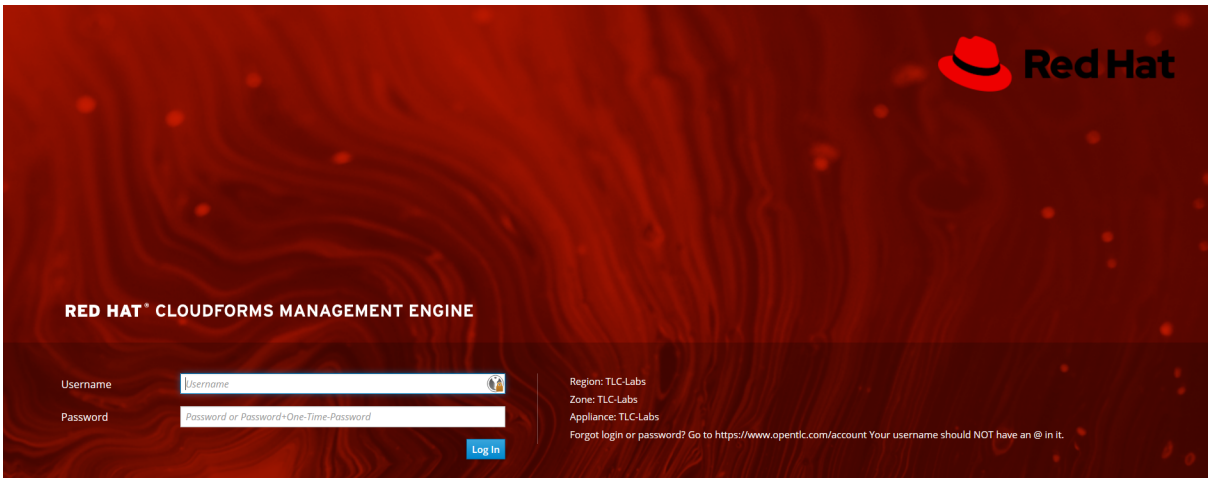
- Mejor control sobre la plataforma Openshift
- Establecer un ciclo CI/CD con Tekton ya que es propio de Kubernetes.
- Aprender a manipular los valores de un chart.
- Aprender a instalar un Operador.
- Uso de Kubeseal, el cual es muy beneficioso para nuestros secrets.

2. Escenario sobre el que vamos a trabajar

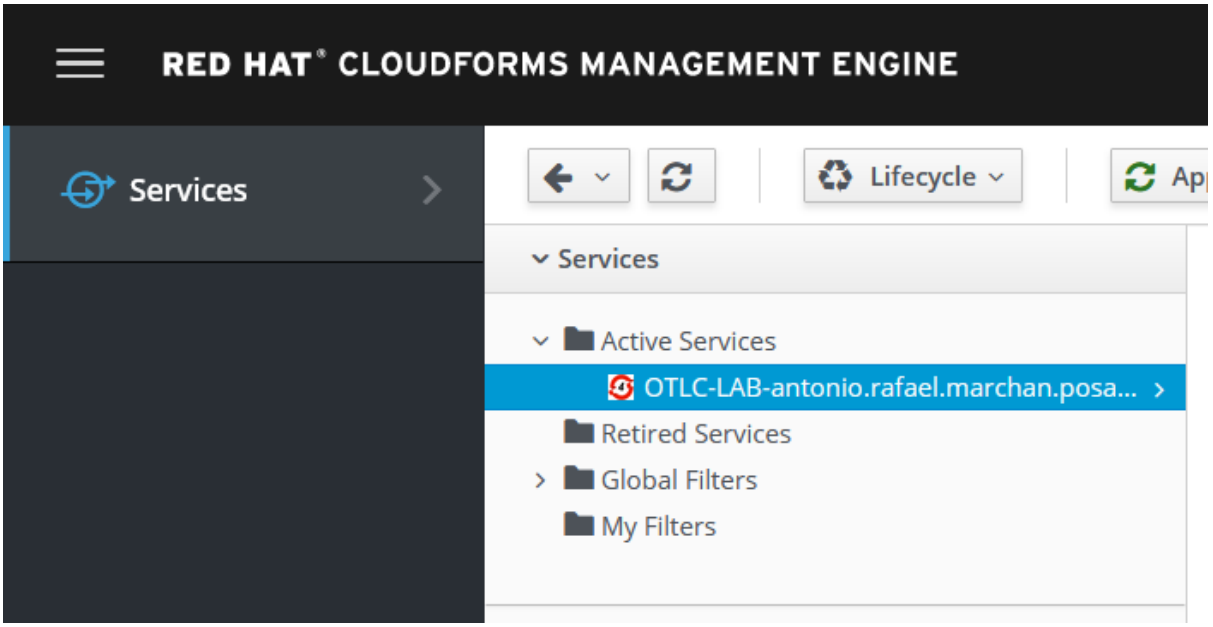
2.1 Openshift 4.10

El escenario que utilizaremos será Openshift 4.10, es una plataforma montada sobre un clúster proporcionado por redhat el cual es un laboratorio que simula un entorno real que podría tener un cliente.

Para ello entramos en **Red Hat Cloudforms Management Engine**:



Una vez dentro solicitamos el lab **Hands on Openshift 4.10**



Una vez tenemos este escenario, Openshift nos proporciona los recursos necesarios para poder integrar pipelines con Tekton a través del operador que debemos instalar.

2.2 Tekton

Tekton es una tecnología nativa de Kubernetes, el cual utilizaremos para implementar los charts de helm, podemos hacer esto de dos maneras, a través de la consola web de Openshift Container Platform o por la CLI.

En este proyecto utilizaremos la CLI.

2.3 Helm

Con Helm lo que tendremos es un repositorio git en el que hemos realizado un fork de los charts de Bitnami, esto lo haremos para realizar diferentes cambios para adaptarlo al entorno de Openshift como mostraré más adelante.

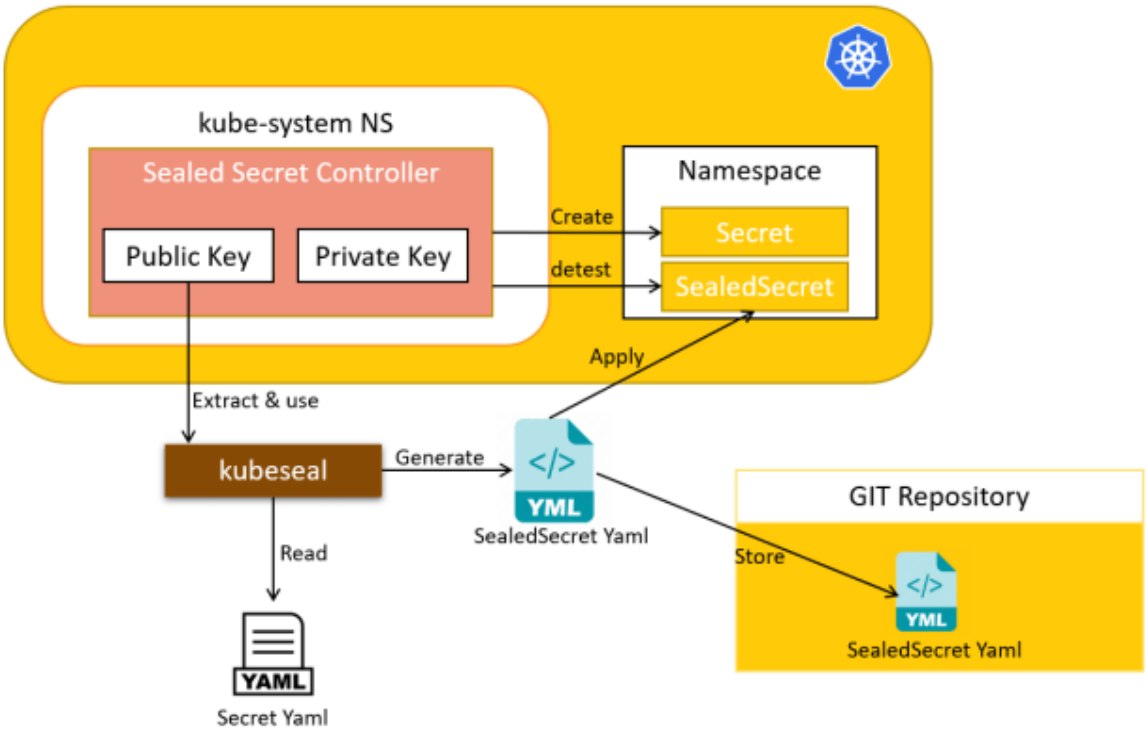
Helm es una herramienta de orquestación de paquetes para aplicaciones Kubernetes. Se utiliza para simplificar la implementación, actualización y gestión de aplicaciones en clústeres de Kubernetes.

Los paquetes Helm, también conocidos como "charts", son colecciones de archivos YAML que describen una aplicación de Kubernetes. Estos archivos YAML incluyen información como las imágenes de contenedor que se utilizan, la configuración de los pods y los servicios, entre otras configuraciones como los volúmenes persistentes.

2.4 Kubeseal

Kubeseal es una herramienta de código abierto desarrollada por Bitnami que nos permitirá encriptar o desencriptar los secretos para así poder subirlos a un repositorio sin peligro a que el secreto sea descifrado. Funciona a través del principio clave pública-privada el cual al ejecutar el comando sobre un secreto .yaml o .json lo encripta, siendo solo este operador el que será capaz de descifrarlo ya que es el que posee la clave privada.

Esta herramienta la utilizaremos para generar un secreto encriptado para guardarlo en github, ya que al tener un volumen persistente este necesita la contraseña para poder usar la base de datos de nuevo.



3. Configuraciones

3.1 Openshift

Ahora estableceremos las configuraciones que nos brindará poder realizar despliegues en la plataforma de Openshift

3.1.1 Teoría

En Openshift, tendremos las siguientes ventajas sobre las que establecer nuestras configuraciones:

- Operador de clúster: permite la administración y el monitoreo de clústeres de OpenShift a través de una interfaz de usuario unificada.
- Integración con Kubernetes: OpenShift 4.10 se basa en Kubernetes 1.18, lo que significa que hereda todas las mejoras y características de Kubernetes.
- Arquitectura de múltiples nubes: OpenShift 4.10 está diseñado para ejecutarse en cualquier nube pública o privada, lo que brinda a los equipos de TI la flexibilidad para implementar aplicaciones en cualquier entorno.
- Pipelines de CI/CD: OpenShift 4.10 incluye herramientas integradas para la implementación continua y la entrega continua, lo que permite a los equipos de desarrollo automatizar el proceso de implementación de aplicaciones.
- Servicio de aplicaciones nativo en la nube: OpenShift 4.10 está diseñado para admitir aplicaciones nativas en la nube, lo que significa que las aplicaciones pueden ejecutarse de manera más eficiente en la nube.

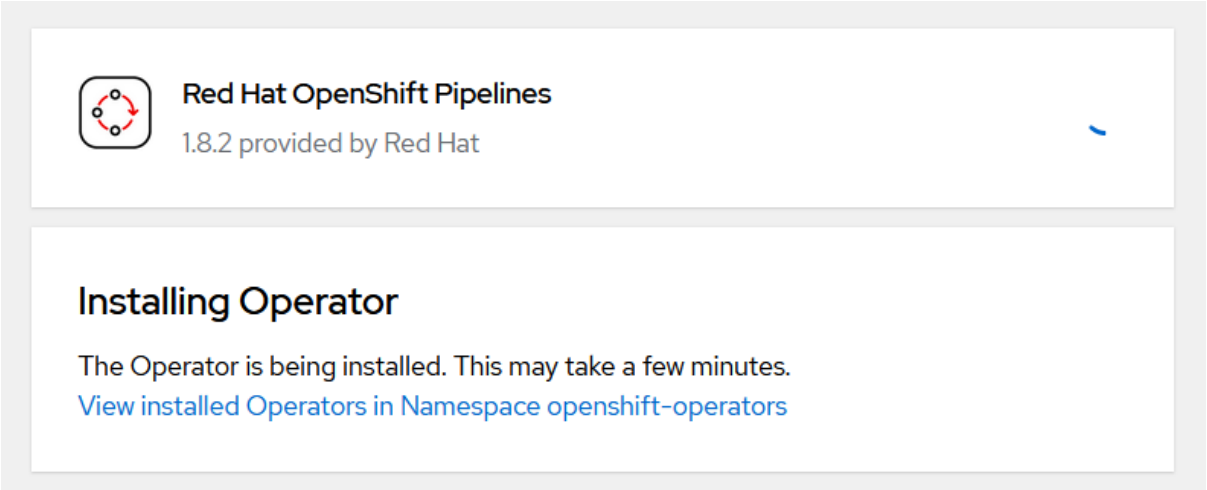
- OpenShift cuenta con soporte en RHCOS y RHEL.
- Solución de nube híbrida de OpenShift 4.
- Con soporte en nubes públicas y privadas, así como en servidores físicos (bare metal).
- Automatización de pila completa (IPI) con soporte en:
    - AWS, Microsoft Azure, Google Cloud Platform
    - Red Hat Virtualization, Red Hat OpenStack® Platform
- Infraestructura preexistente (UPI) con soporte en:
    - AWS, Microsoft Azure, Google Cloud Platform
    - VMware vSphere, Red Hat OpenStack Platform, IBM Z
    - Power Systems de IBM, servidor físico (bare metal)
- Futuros lanzamientos para soportar incluso más infraestructuras.

- OpenShift se ejecuta en RHCOS y RHEL.
- OpenShift tiene dos tipos de nodos:
    - Nodos trabajadores
    - Nodos master
- Los nodos son:
    - Instancias de RHEL o RHCOS con OpenShift instalado
    - Nodos trabajadores: donde se ejecutan las aplicaciones del usuario final.
    - Nodos master: administran el clúster.
- El daemon del nodo de OpenShift y otro software se ejecutan en todos los nodos.

3.1.1 Práctica

Comenzaremos configurando Openshift instalando el operador de Openshift Pipelines:

Iremos a Administrator -> Operators -> OperatorHub y en el buscador ingresaremos Red Hat Openshift Pipelines, el cual por dentro será Tekton el que esté proporcionando el servicio.



Ahora dentro de la terminal, vamos a crear un ClusterRole y se lo vamos asignar a la pipeline que podrá de esta manera hacer un deploy del sealedsecret que usaremos más adelante, de modo que si aplicamos estos .yaml, crearemos una regla que nos permita instalar sealedsecrets a través de la instalación del chart de helm:

`clusterrole-sealedsecrets.yaml`

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: sealedsecrets-role
rules:
- apiGroups: ["bitnami.com"]
  resources: ["sealedsecrets"]
  verbs: ["get", "create", "update", "delete"]
```

`clusterrolebinding-sealedsecrets.yaml`

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: sealedsecrets-rolebinding
subjects:
- kind: ServiceAccount
  name: pipeline
  namespace: pipeline-helm
roleRef:
  kind: ClusterRole
  name: sealedsecrets-role
  apiGroup: rbac.authorization.k8s.io
```

3.2 Cambiar valores del archivo de configuración del Chart de Bitnami

Helm es un gestor de paquetes de código abierto para aplicaciones de Kubernetes. Ofrece una manera de empaquetar, compartir y gestionar el ciclo de vida de las aplicaciones de Kubernetes.

Un Helm Chart es una colección de archivos y plantillas que definen una aplicación Helm. Estos archivos se empaquetan posteriormente para su distribución mediante repositorios Helm.

El comando helm create crea los archivos necesarios en la estructura correcta. Los archivos creados por este comando son los básicos necesarios para un Helm Chart, que incluyen las plantillas mínimas requeridas para que funcione una aplicación.

Un Helm chart consta principalmente de dos archivos YAML y una lista de plantillas.

Los archivos YAML son los siguientes:

- Chart.yaml: contiene la información de definición del gráfico.
- values.yaml: contiene los valores que Helm usa en las plantillas predeterminadas y creadas por el usuario.

Además de estos dos archivos, un Helm Chart contiene varios archivos de plantillas; estos archivos son la base para los recursos de Kubernetes que las conforman.

3.2.1 Teoría

Ahora vamos a proceder con la configuración del chart para que al ser procesado por Tekton no tengamos ningún tipo de problemas de seguridad en Openshift, esto es debido a que Openshift viene con un parámetro de seguridad llamado

PodSecurityContext el cual es un grupo de seguridad que viene por defecto en Openshift el cual evita que el pod se ejecute internamente a través de un usuario privilegiado, esto es así para que un atacante que posea acceso al pod no pueda realizar cambios significativos en el clúster.

3.2.1.1 Creación de un Helm Chart

En la creación de un chart de helm puede tener la siguiente estructura, la cual podremos ir cambiando en cuanto a las necesidades que tengamos:

```
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── serviceaccount.yaml
│   ├── service.yaml
│   └── tests
│       └── test-connection.yaml
└── values.yaml
```

Chart.yaml: al crear la estructura, se genera este archivo que indica la versión y el nombre que se le va a proporcionar al chart, entre otras posibles configuraciones.
Values.yaml: núcleo importante del chart de helm, donde se establecen los valores para cada tipo de variable que pueda contener un chart, un ejemplo de ello podría ser desde el puerto por el que va a comunicarse, hasta la imagen que va a utilizar.
template: En el template encontraremos los archivos de kubernetes necesarios para lanzar la aplicación, entre ellos podemos encontrar:

- **deployment.yaml**: Este archivo describe cómo crear y configurar un conjunto de réplicas de la aplicación. Define cómo se ejecutará la aplicación en el clúster, cuántas réplicas se deben ejecutar y cómo se debe escalar la aplicación en función de la carga de trabajo.

- **_helpers.tpl**: Este es un archivo de plantilla que contiene funciones de ayuda que se utilizan en otros archivos de configuración. Proporciona abstracciones y automatización para simplificar la configuración y la gestión del clúster.
- **hpa.yaml**: Este archivo describe cómo se debe escalar automáticamente la aplicación en función de la carga de trabajo. Se define un objeto Horizontal Pod Autoscaler (HPA) que controla el número de réplicas de la aplicación en función de la utilización de recursos.
- **ingress.yaml**: Este archivo describe cómo se debe enrutar el tráfico hacia la aplicación desde fuera del clúster. Se define un objeto Ingress que gestiona el acceso a la aplicación y la seguridad de la red.
- **NOTES.txt**: Este archivo contiene información adicional sobre la implementación, como la configuración de la aplicación y los comandos útiles para su gestión.
- **serviceaccount.yaml**: Este archivo describe cómo se debe crear una cuenta de servicio para la aplicación. Esta cuenta se utiliza para que la aplicación pueda acceder a los recursos del clúster de Kubernetes de forma segura.
- **service.yaml**: Este archivo describe cómo se debe exponer la aplicación dentro del clúster. Se define un objeto Service que expone la aplicación a otros servicios del clúster.
- **tests/test-connection.yaml**: Este archivo describe cómo se pueden probar las conexiones a la aplicación desde otros servicios en el clúster. Define un objeto de prueba que verifica que la aplicación responda correctamente a las solicitudes de prueba.

3.2.2 Práctica

Entonces vamos a ir a nuestro repositorio de git:
https://github.com/Evanticks/charts-bitnami/tree/main/bitnami/wordpress
Y procederemos a modificar el archivo values.yaml

“values.yaml establece las diferentes variables que va a poseer el chart de Helm, en el resto de configuraciones se hace referencias a los valores que se establecen en este archivo, así que podríamos decir que es el núcleo del chart”

nos fijamos en lo siguiente, lo cual estableceremos a false

```
podSecurityContext:
  enabled: false
  fsGroup: 1001
...
containerSecurityContext:
  enabled: false
  runAsUser: 1001
...
 primary:
     podSecurityContext:
     enabled: false
     containerSecurityContext:
     enabled: false
```

Una vez hecho esto podremos desplegar la aplicación sin que falle la pipeline debido a que estos parámetros ejecutan el pod en modo sin privilegios.

De modo alternativo podríamos desplegar **Argocd**, que es un operador que nos permite desplegar en **Openshift** imágenes que le proporcionemos, el cual no será nuestro caso, ya que **Tekton** tiene la posibilidad de desplegar el Chart por sí mismo.

Tras esto lo siguiente que debemos hacer será mencionar el secret que crearemos posteriormente. de tal manera que necesitaremos crear la última línea haciendo referencia al secret que crearemos posteriormente:

```
mariadb:
  architecture: standalone
  auth:
    rootPassword: ""
    database: bitnami_wordpress
    username: bn_wordpress
    password: ""
    existingSecret: "mysecret"
```

gracias a esto, el chart esperará al secreto **mysecret** para obtener las credenciales de la base de datos.

Tras esto, nos conectaremos al bastión de nuestro clúster de Openshift e instalaremos helm para poder utilizarlo a través de línea de comandos y no solo a través de la consola web:

```
wget https://get.helm.sh/helm-v3.11.0-linux-amd64.tar.gz
tar zxvf helm-v3.11.0-linux-amd64.tar.gz
sudo install -m 755 helm /usr/local/bin/helm
```

3.3 Configurar e implantar secrets cifrados con Kubeseal

3.3.1 Teoría

Kubeseal es una herramienta de Kubernetes que se utiliza para cifrar secretos y protegerlos de accesos no autorizados.

Kubeseal utiliza la criptografía asimétrica para cifrar los secretos y garantizar que solo los usuarios autorizados tengan acceso a ellos. En lugar de almacenar los secretos en texto plano en Kubernetes, Kubeseal cifra los secretos utilizando una clave pública y solo puede descifrarse utilizando la clave privada correspondiente.

El secreto cifrado puede entonces ser almacenado de manera segura en Kubernetes o un repositorio Git o similar.

Cuando Kubernetes necesita hacer uso de ese secreto utiliza la clave privada de Kubeseal para descifrarlo.

3.3.2 Práctica

Primero vamos a proceder a instalar Kubeseal en nuestro cúster, de moto que si ya anteriormente instalamos helm, ahora necesitaremos instalar el chart de Kubeseal, así que procedemos a ello:

```
helm repo add sealed-secrets https://bitnami-labs.github.io/sealed-secrets
helm repo update
helm install sealed-secrets-controller sealed-secrets/sealed-secrets -n kube-system
```

Y su respectivo binario:

```
wget https://github.com/bitnami-labs/sealed-secrets/
releases/download/v0.20.5/kubeseal-0.20.5-linux-amd64.tar.gz
```

```
tar -xvzf kubeseal-0.20.5-linux-amd64.tar.gz kubeseal
sudo install -m 755 kubeseal /usr/local/bin/kubeseal
```

Ahora procedemos a crear un secreto en Kubernetes, como vamos a utilizarlo para la base de datos de mariadb, lo crearemos con sus propias credenciales:

```
kubectl create secret generic mysecret \
--from-literal=mariadb-root-password=hola123 \
--from-literal=mariadb-replication-password=hola987 \
--from-literal=mariadb-password=hola933 \
--dry-run=client -o yaml > secretonuevo.yaml
```

Una vez realizada esta acción, se nos generará el archivo secretonuevo.yaml, aquí es donde entrará kubeseal:

```
kubeseal < secretonuevo.yaml > mysealedsecret.yaml
```

Una vez hecho esto, se generará el cifrado de cada valor del secreto.


```
    "encryptedData": {
      "mariadb-password": "AgCqbVBvL2l8MBmi1H5qX7kSvRjWwfaf4A4tYzonISXijZH4AglPkBmoQGUgTvunLe3I1RS95lTlqM0inzOzcIUiR/mxfP/QKPaitNV1YDfi
htgizVZ22jJHmq9no8Ymp7yC1uBmo1ZS+BGyJrH8MlbOFAN003y7MC+QFpUiSDl5IyqIMIO22tfij4ETreU8NWT5mvk+GjvFsqX9UL4aPQOdvhXSTTX9Pius9hx/thKT1UXX/6X
fSy+5HBtJHt4L5TXDMwNY7pEpfB/aRFfJANMtgK2RxkJi8bJzxgQwWyHicofQODAJ3AGyvh/705Q39U1ysK9WRHOkD8/1vup2bIlMIn6Hh9WFgL1CV6PApBeMH7/rpOWR7sAwW7
TNhfFHiV+mO9sUSdyBSPXcVjOmwKTb7swj4ZoLYT1z2RqDTVHUyp8HKQfDOx/jQnRFWnXmTiN+BjFZIaw1kSrArOXaSR81myqVwcLFiSSd11Fk3taPsFiE39hyrB9SWHriVWhR4
ibtaiDr3lRNYfzH515EKrK3QlzS65VHW7ECFWslWcj5Yay9CH5UgP4Cil5vKXCYG0+JUwtxjoQgmw5kc/EstgPPcNeWbu66JpnM/KpgE+BorU8RpwEhbZx+bKsh4NH7/q3HTHqq
6BQ/2a4eS2RDOctcifLaGa7VEc1WkXXY5W/f9RbP76NeERofZD2a/EXvVh2clh20obdT",
      "mariadb-replication-password": "AgCNNE9K/N9kJyD3lN7TDjxnkf++f/sk9CDhswj2vkTjcpcp/28CMind4Y4P406dR7xMq5qYsg0mHIbIkMgr7h/CLuHiljah
gKbQWRnqfbUbMWj9UrG2+p52qphoI60YD/gym9Of0H8oBKBBKNo/Uej9iBU0dS85eycplegUOFFXq4WV0ra3oaFY4IiFG3yKxoQSmBSY9QOdAEoCuzBhRGKJUSfvOhLxkg9BtQ/
NcjbV6PUbCSEwOZCIuSIsKZoaThQmIBS7FD5ecDv6ohgazbO4gO2A5mwehDUreZxqTr4kCwEI54W8Tfdz48s1fxM42GrWkNdgTmHbv0KNe8naZU1XvZULnf7HIy8VQfGmVANjE7
YXa+xYb3mAggKh5IF0mt4anqep0CTQKp2Wdd5JOI+Em4b7Q23TTX17XT9nf0URp/b8rrbLCPGUlxz7g5HEOowUUailg4B4RoLQ2v6fLs2uQnVQLTjrPNY2N2lrY6W1jnTToYGpQ
f4Eh9R3GaluMPkG9FlNa8wm4WbsV3T72yBF0OOJE+9L5m8iwxW1tZmZsLyNSQiDROLbJYoQ6okjiqYtrHbpjfCWR2HFUJRtT5HiE+HPIK3Ypo+EyTaOJlQTuhAEvnHipQMm4OCg
+dCiddzSLNAe150AHgvf3lLjuDVLqurw4c3+Tu6WIvErfJyTMxOYgTT/EkD474FUAdb73xo93I0huh9T",
      "mariadb-root-password": "AgCZ1e2oJ9hwQrDfOiDxRQTXTdTXZ/Tr/cpf/AK8V1y4bwYD0dw8mRg6XStq76ApyhIq3iJnJr9aEijCQ1f24S5FV7tAY8obSWEclMr
kfw1Zjidc/tjbjF26hglq43MV2rHb64tX7BiLleC8cMi3ZxSWJYyruc/BQrbTIgICkesRA1K+xrG7p7su6eIzrFBKJi/byJxSYiVGx1ZmXVxbw+G5prjNzfadu9IRKloOEIrhTR
wlFN4wdI7zDoffQWoKWhBf06ajG/XtUUgk3u27JmLLKsuEWKDIjfdjj4ebV+gIXaew5GhTty7Xr1lWlI8Ht08DJYiUE53w7ixPqLOBJcK6KmtcJFhZhLf2gkyVBof7SihSu5Ep+
XNJwIvaCg1RXmkoh8SIdXz35+W6mdesm6LyQYprJi7p6kozNEniDy4E3Q8ATKpwbwLlldT2MnhwRnmcl3lDkY4dSO2wRWz4ZR3VuwOHHcSi71KbkTVOMrCqUOrJEGa68mqhby/h
FaA/XlgpLGlD46NfXwKbqaYH1hnQc94Jj2OiFEjufBz+sf4dSAd9Am+7+TLP8YgabUU1WmLgak/Wlstia8VU6JHsuXhE+Nm5BH1GUiABRMvfT1wjsAuk++Prq5hQtxPJvbyvdL+
2LD8zYKyT1rckEhn0CrhUC4Uqrssrgytyyiy0r36n6B1fb1UNq8SeOKpXhvav6z/Y+CQEUNTo"
    }
```


Ya con esto solo nos quedará aplicarlo al clúster

```
oc create -f mysealedsecret.yaml
```

O bien podremos subirlo a cualquier repositorio sin problema, ya que será sólo nuestro clúster el que sea capaz de descifrarlo a través de las claves asimétricas.

3.4 Establecer un ciclo CI/CD con Tekton

Este apartado es el más denso, ya que tekton se encarga de las configuraciones, las integraciones y procesos de despliegue, así que vamos a proceder a explicar qué es Tekton

3.4.1 Teoría

Tekton es una plataforma de orquestación de pipelines de CI/CD (Integración Continua/Entrega Continua) que se ejecuta en un clúster de Kubernetes. Permite a los equipos de desarrollo automatizar el proceso de construcción, prueba y entrega de aplicaciones.

Tekton utiliza una arquitectura basada en contenedores para ejecutar los pasos del pipeline. Los pipelines se definen mediante archivos YAML que describen los pasos del pipeline, como la construcción de la aplicación, la ejecución de pruebas y la implementación en un entorno de producción.

Tekton proporciona varios recursos para la creación de pipelines, como tareas, recursos y pipelines. Las tareas representan un trabajo específico que se realiza en el pipeline, mientras que los recursos son elementos externos utilizados por las tareas, como el código fuente o las imágenes de Docker. Los pipelines combinan tareas y recursos para crear un flujo de trabajo completo de CI/CD.

Tekton también incluye una biblioteca de componentes reutilizables de tareas y pipelines, lo que hace que sea fácil para los equipos de desarrollo construir y personalizar sus pipelines de CI/CD.

Para poder realizar un CI/CD con Tekton necesitamos de varios tipos de pipelines, que relacionados entre sí encajan como **si piezas de un puzzle** se tratase, estos archivos son:

- **Pipeline.yaml**: se encarga de nombrar las **task** que se van a utilizar, ya que es necesarios porque podemos almacenar archivos .yaml de task de cosas totalmente diferentes, así que la función del propio pipeline es importante.
- **Tasks.yaml**: se encarga de realizar algún tipo de acción sobre la que vamos a realizar, las **tasks** son consideradas las más importantes de **Tekton** y posee una **comunidad** detrás que implementa plantillas de tasks para que los demás usuarios podamos utilizarlos o modificarlos a nuestro antojo. https://hub.tekton.dev/
- **PipelineRun.yaml**: es el archivo iniciador del **pipeline**, aquí se indica el pipeline que se va a utilizar y lo va a poner en producción, además es el que nos indica el estado del pipeline ejecutado, cada vez que queramos ejecutar el pipeline, debemos ejecutar PipelineRun.
- **EventListener.yaml**: como su nombre indica es un servicio que establece un puerto de escucha el cual si lo asociamos a **github**, cada vez que haya un cambio en el repositorio lanza el **PipelineRun**.

3.4.2 Práctica

Para poder crear una pipeline vamos a empezar a crear la estructura por pasos, y el primer paso será crear el volumen que vamos a solicitar para que en el espacio compartido donde se ejecute la pipeline las tareas puedan enlazarse:

PersistentVolumeClaim.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pipelinerun-vol
spec:
  accessModes:
       - ReadWriteOnce
  resources:
       requests:
       storage: 1Gi
```

Ahora procederemos a instalar cada tarea, las cuales serán dos, una que realiza un git clone y otra que he modificado a mano para que haga un despliegue de la aplicación:

task-git-clone.yaml se encuentra en mi repositorio git al ser un texto demasiado extenso, pero de task-install-chart he modificado a manos unas líneas de código para que me permita realizar el despliegue haciendo CI/CD:

`task-install-chart.yaml`

```
helm dependency update bitnami/wordpress

helm list --namespace "$(params.release_namespace)"
       echo current installed helm releases
helm list --namespace "$(params.release_namespace)"

if helm ls | grep "helm-release"
then
        echo "WordPress de Bitnami está instalado. Desinstalando..."
        helm uninstall helm-release
        echo "WordPress de Bitnami ha sido desinstalado exitosamente."
fi

helm upgrade --install --values
"$(params.charts_dir)/$(params.values_file)" --namespace
"$(params.release_namespace)" --version "$(params.release_version)"
"$(params.release_name)" "$(params.charts_dir)" --debug  --set
"$(params.overwrite_values)" $(params.upgrade_extra_params) --set
service.type=ClusterIP
```

Con todos estos parámetros, la task detectará un chart del helm del cual vamos a realizar la demo, si está instalada la desinstala, tras esto procederá a instalarla con un servicio ClusterIP.

Tenemos el volumen, luego las tareas, ahora vamos a definir el pipeline el cual volveremos a elegir un segmento a analizar, lo cual sintetizamos el funcionamiento del mismo.
En Tekton, el pipeline se utiliza para definir las tareas y el orden con el que estas se van a ejecutar, aunque existe la posibilidad de que esto no sea así, pudiendo ejecutar tareas a la par, en nuestro caso una tarea depende de la otra así que quedaría la pipeline de la siguiente manera:

`pipeline-helm.yaml`

```
...
  tasks:
       - name: task-install-chart
       runAfter:
       - git-clone
       taskRef:
       kind: Task
       name: task-install-chart
...
       - name: git-clone
       taskRef:
       kind: Task
       name: git-clone
       params:
       - name: url
       value: 'https://github.com/Evanticks/charts-bitnami.git'
       - name: revision
       value: main
...
```

De esta manera tendremos definido tanto la tarea en la pipeline como parámetros los cuales luego el task puede utilizar para recibir la información como el propio github que ingreso.

Una vez hecho esto vamos a ejecutar un pipelineRun, el cual se encargará de coger el pipeline de helm y ejecutarlo, vamos a analizar la parte más importante del código:

```
pipelineRef:
     name: "pipeline-helm"
serviceAccountName: pipeline
workspaces:
     - name: compartido
     persistentVolumeClaim:
     claimName: pipelinerun-vol
```

Podemos ver el pipelineRef, el cual será al que hace la referencia para ejecutarlo, el serviceAccountName es la identidad del pod generado, ya que cuando se ejecuta una Run realmente se crea un pod que lo ejecuta y luego muere tras finalizar su función.

El workspace es el nombre del espacio en el que se compartirá el recurso de la pipeline, por tanto debe tener el mismo nombre en los diferentes .yaml

Por último hacemos referencia al recurso del persistentVolumeClaim para que el clúser cree ese recurso que habíamos definido en el primer paso.

4. Demostración final

Para la demostración final una vez hechas las configuraciones entraremos en bastión dentro de la terminal de Openshift, bajaremos el repositorio donde se encuentra la pipeline de tekton.

```
git clone https://github.com/Evanticks/wordpress-helm-tekton.git
cd wordpress-helm-tekton
oc create -f .
```

Una vez creados los recursos, se estará ejecutando la pipelineRun, por tanto la pipeline en openshift está corriendo, podremos verlo entrando en la consola web.

Una vez desplegado el pod con frontend y su respectivo backend persistente, veremos que no posee una ruta a la cual acceder, por tanto la añadiremos en nuestro entorno local y haremos un push:

```
cp wordpress-ruta.yaml charts-bitnami/bitnami/wordpress/templates/
git commit -am "añado ruta"
git push
```

A través de un event listener, Github notificará a Openshift para que se vuelva a ejecutar el pipelineRun, por tanto se volverá a desplegar el chart de Helm esta vez utilizando la ruta.

Finalmente mostraré el interior del sealedsecret que ingresé dentro del chart de helm para mostrar la fiabilidad de la operación.

5. Dificultades encontradas

- En primer lugar encontré el problema de los privilegios de los pods, Openshift no corre pods privilegiados, entonces se deben ejecutar desde el usuario 1000, bien especificándolo en el Dockerfile o bien en el values.yaml del chart de Bitnami.

- En segundo lugar me encontré los problemas con las pipelines, tekton como tal tiene una curva de aprendizaje más elevada que Jenkins, ya que debemos aprender para qué se utiliza cada yaml.

- En tercer lugar el volumen persistente, ya que cada vez que se va a desplegar la aplicación genera una nueva contraseña, y si posee ese volumen persistente las contraseñas serán las del primer despliegue, entonces al hacer CI/CD el pod de Mariadb no funcionaba, por eso opté por la opción de Kubeseal ya que me facilita mucho los despliegues.

- En cuarto lugar aprender sobre los roles del clúster de Openshift para otorgar el rol de que pueda desplegarse el chart de bitnami con un SealedSecret dentro de la template del chart a través de la pipeline.

- Por último, tuve problemas con el LoadBalancer que me otorga AWS que es sobre donde está montado Openshift

6. Conclusión

En este proyecto he podido profundizar en Openshift como plataforma, adentrarme dentro del mundo de Devops con Kubernetes a través de las pipelines, crearlas y modificarlas a mi antojo para adaptarlo lo mejor posible al proyecto.

He aprendido a crear un event listener y relacionarlo con los webhooks de Github.

He aprendido a crear un chart de Helm, y una vez hecho esto he procedido a aprender sobre los charts hecho por profesionales de Bitnami, y a modificarlos para adaptar los valores al proyecto, así como hacer que se ejecute de forma no privilegiada, haciendo que cree un volumen persistente, y haciendo que tome los valores de la contraseña de la base de datos a través de un secret que yo guardo en la template del chart.

He aprendido a encriptar los secretos con Kubeseal e integrarlos en el cúster para poder utilizarlos posteriormente.

En definitiva ha sido un proyecto muy didáctico que me ha permitido seguir impulsándome en el mundo de la administración de sistemas.