Publicado en Mejor programación · Seguir

nic verde · Seguir
27 de enero · 7 minutos de lectura ★

Cómo crear una canalización de CI/CD para aplicaciones con estado de Kubernetes

Parte 1: Empecemos

Foto de Mourizal Zativa en Unsplash

Esta serie de publicaciones lo guiará a través de una experiencia práctica centrada en la creación de una aplicación con estado simple (pero genial) que ha sido diseñada para ejecutarse en Kubernetes. La parte 1 prepara el escenario.

casi el caso para todas las aplicaciones", entonces tiene razón, pero el problema es que, en general, el backend no vive en Kubernetes.

Además, su arquitectura generalmente se distribuye en plataformas heterogéneas. Por lo general, el componente frontend ya se encuentra en Kubernetes y se puede aprovisionar, ampliar y reducir a pedido, ya que generalmente no tiene estado. Al mismo tiempo, los conjuntos de datos se pueden alojar en varios formatos y factores de forma: máquina virtual local, servicios PaaS en la nube pública, plataforma Big Data o incluso máquinas físicas.

Esta serie explora la experiencia de DevOps al *desarrollar aplicaciones con estado diseñadas para ejecutarse en Kubernetes y crear servicios de datos bajo demanda.* Puede que se pregunte por qué ejecutar todos los componentes en Kubernetes. El objetivo es que los propietarios de aplicaciones aceleren el lanzamiento de aplicaciones críticas para el negocio, aprovechando los beneficios nativos de Kubernetes.

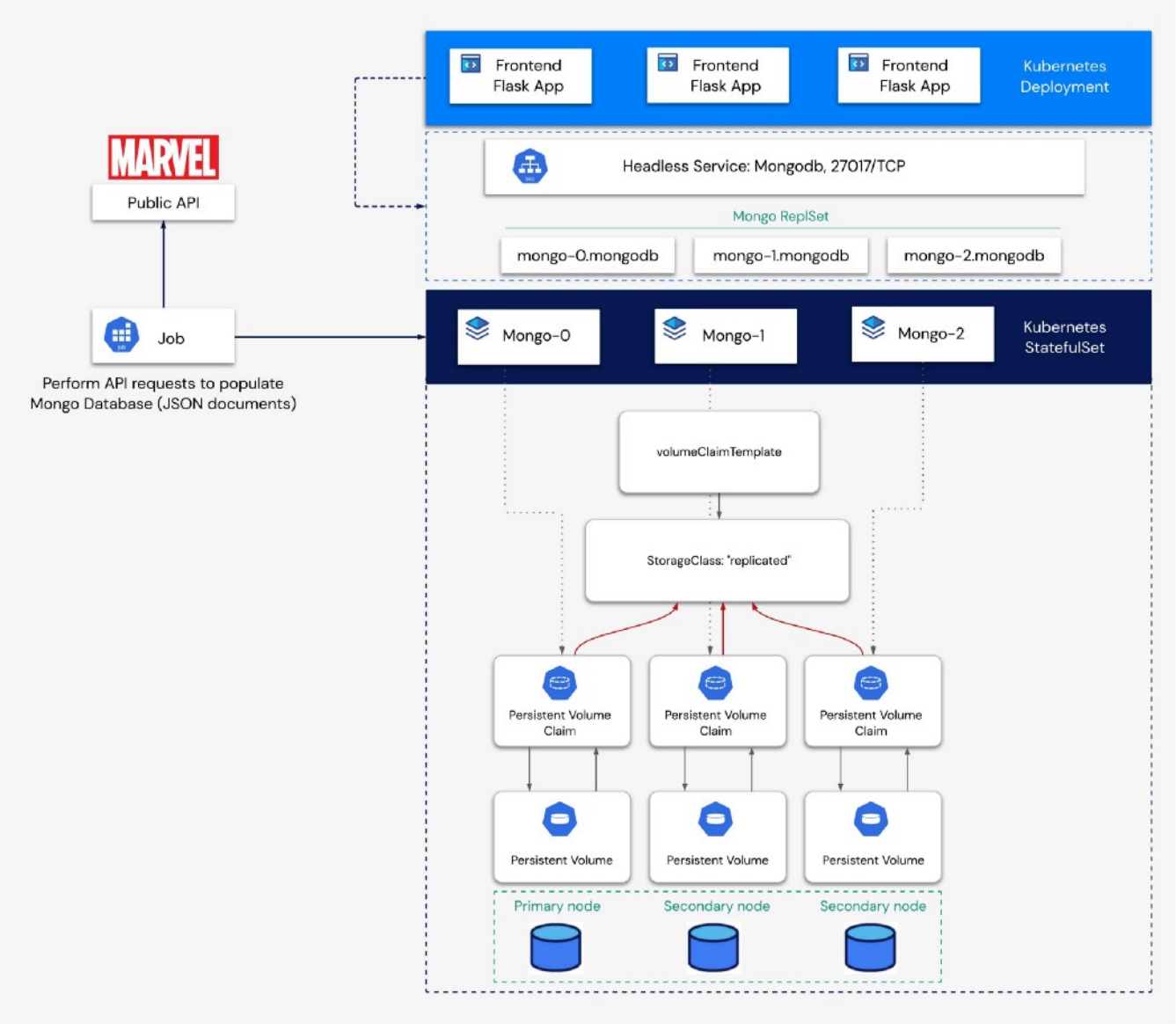
Nos vamos a centrar en el conjunto de herramientas que permite a los desarrolladores navegar por el ciclo de vida de desarrollo de estas aplicaciones de manera eficiente. Es decir, vamos a profundizar en:

- personalizar
- patín
- Volúmenes persistentes de Ondat
- Base de datos MongoDB y conjunto de réplicas
- Operador de Kubernetes de la comunidad MongoDB
- pymongo

Si esta no es la configuración más DevOps para una aplicación con estado, ya no sé qué significa DevOps 😆.

Conoce la aplicación Marvel

de aplicaciones. ¡Con suerte, esto será mucho más divertido que otra aplicación "Hello World"! La arquitectura de la aplicación se representa a continuación:



El código está disponible aquí: https://github.com/vfiftyfive/FlaskMarvelApp .

El componente FE (Frontend) es una aplicación Python Flask. Su función es proporcionar una capa de visualización para los datos incorporados en la base de datos MongoDB. Se ejecuta en Kubernetes como un archivo `Deployment` .

El BE (Backend) es una base de datos MongoDB implementada como un clúster de 3 nodos y administrada por MongoDB Community Kubernetes Operator. Se ejecuta en

Como se muestra en la imagen, un `StatefulSet` aprovecha un controlador de Kubernetes para garantizar que cada uno `Pod` tenga acceso a su propio almacén de datos al reclamar volúmenes persistentes (`PV`) vinculados a archivos individuales `PVCs` . Para más información `StatefulSets` y sus casos de uso, puedes echar un vistazo aquí en la web de Ondat o en la documentación oficial de Kubernetes .

Como parte de la implementación de la aplicación, también se aprovisiona un Kubernetes `Job` . Su función es obtener la información de los personajes de Marvel de la API de Marvel disponible aquí y almacenarla en la base de datos de MongoDB. Solo se ejecuta una vez, en el `Job` momento de la implementación de la aplicación. Kubernetes programará `Pods` hasta que el trabajo se complete con éxito (dentro de un número limitado de reintentos). `Pods` puede fallar, esperando que se establezca la conexión con la base de datos, pero una vez que se logra, nunca se vuelve a ejecutar. La FE puede entonces mostrar tarjetas de caracteres aleatorios directamente recuperadas de la base de datos.

La aplicación comprende el FE +BE y un paso de inicialización de datos que llena el BE con información relevante. Se puede implementar en Kubernetes mediante manifiestos YAML generados dinámicamente por una herramienta de línea de comandos o como una canalización de CI/CD. Kustomize es nuestra herramienta preferida en este artículo para generar manifiestos. También puede actualizar objetos relacionados con una iteración de compilación particular. Con Kustomize, puede actualizar fácilmente las referencias de imágenes de contenedores o, para nuestro caso de uso, implementar la aplicación con una nueva versión de Mongo o aprovisionar un nuevo Ondat `StorageClass` .

¡Crea plantillas y automatiza todas las cosas!

Todo comienza con la capacidad de iterar las pruebas y la implementación de aplicaciones en varias etapas de desarrollo, como localmente en una computadora portátil, en un clúster de prueba remoto o durante las etapas de preparación, prueba de aceptación del usuario (UAT) y producción. Kustomize se puede usar como una `kubectl` opción (`-k`) para aplicar la personalización al generar manifiestos de aplicación. Esta herramienta permite a los desarrolladores adaptar dinámicamente los

El principio de Kustomize es crear una superposición especificando los elementos de los manifiestos base que desea modificar. En este artículo, nos enfocamos en la fase de desarrollo de la aplicación, donde codificamos y probamos la aplicación dentro de un clúster local de Kubernetes K3s. *En ese caso de uso, Kustomize* necesita 2 carpetas, una para los manifiestos base y otra para la superposición de desarrollo:

```
base\
    job.yaml
    kustomization.yaml
    marvel_deploy.yaml
    marvel_svc.yaml
    mongodbcommunity_cr.yaml
    ondat_sc.yaml
overlay\
    dev\
        job.yaml
        kustomization.yaml
        marvel_deploy.yaml
        mongodbcommunity_cr.yaml
        name_reference.yaml
        ondat_sc.yaml
```

Se puede acceder al contenido de estos archivos *aquí* . Uno puede simplemente ejecutar Kustomize con la `build` opción de generar los manifiestos de Kubernetes apropiados. Por ejemplo, para modificar los manifiestos base con la *superposición* de desarrollo, puede ejecutar `kustomize build overlay/dev` , suponiendo que se encuentra en la carpeta principal del directorio "superposición". La salida es un conjunto de manifiestos que se muestran directamente en su terminal, por lo que si desea guardar el resultado como YAML, simplemente redirija la salida a un archivo. Otra opción es usar la salida de `kustomize build` como entrada para `kubectl apply` esto:

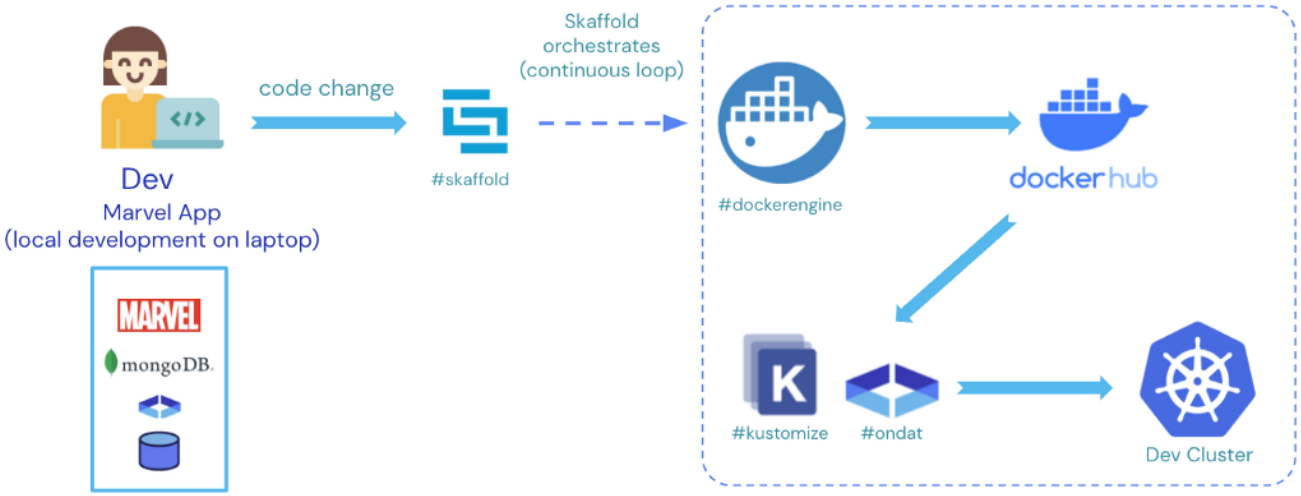
```
personalizar la superposición de compilación/desarrollo | kubectl
aplicar -f -
```

Implementará directamente los objetos en su clúster de Kubernetes.



a medida que confirma el código nuevo, la actualización de los manifiestos de Kubernetes y la implementación de una nueva versión de su pila de aplicaciones en el clúster de prueba de Kubernetes para que pueda ejecutar sus pruebas.

Nuestro objetivo es proporcionar una canalización que cree automáticamente las imágenes del contenedor, actualice los manifiestos en consecuencia e implemente *continuamente* la pila de aplicaciones tan pronto como modifiquemos el código fuente. El flujo de trabajo se ve así:



Skaffold es una de las herramientas que te permite hacer precisamente eso. Es un proyecto de código abierto de Google que proporciona una CLI para administrar el ciclo de vida de su aplicación dentro de varias etapas de la canalización de CI/CD. Puede ayudar con las fases de desarrollo, compilación e implementación de su aplicación. En nuestro caso de uso, estamos interesados en la etapa inicial de desarrollo. La función de Skaffold será crear una nueva imagen a partir de FE Dockerfile cada vez que el código se guarde localmente antes de ejecutarlo `git commit` e implementarlo en el clúster de desarrollo de Kubernetes mediante Kustomize. Como resultado, no necesita confirmar o enviar código a su repositorio de git para probarlo.

Comencemos con el contenido del FE Dockerfile

```
COPY ./requirements.txt /code/requirements.txt EJECUTAR

pip install --no-cache-dir --upgrade -r /code/requirements.txt

COPY ./app /code/app

CMD [ "gunicorn", "--conf", "app/gunicorn_conf.py", "--bind",
"0.0.0.0:80", "app.main:app"]
```

Aquí realizamos operaciones muy estándar:

- Instale las dependencias requeridas por la aplicación Flask
- Copie el código fuente en el contenedor.
- Ejecute el servidor web usando `gunicorn` WSGI (como la aplicación no sirve páginas HTML estáticas, no hay necesidad de `nginx` otro servidor HTTP).

El modo de desarrollo de Skaffold le permite detectar cualquier cambio que ocurra en tiempo real en el código fuente de la aplicación, crear automáticamente una nueva imagen de contenedor utilizando ese Dockerfile e implementarlo en el clúster de desarrollo de Kubernetes. No es necesario realizar `git commit` o `git push` desencadenar este proceso a través de un webhook. En ese modo, el binario Skaffold se ejecuta como un demonio que detecta cambios en el código. Skaffold puede implementar los componentes de la aplicación usando diferentes herramientas. Hemos elegido Kustomize, pero Docker, kubectl y Helm también son opciones disponibles. De manera similar, la fase de compilación de Skaffold puede aprovechar Dockerfiles, buildpacks y otras herramientas mencionadas en la documentación , así como scripts personalizados.

Dado que usamos una arquitectura basada en ARM para el desarrollo, necesitamos un script personalizado para realizar compilaciones multiplataforma de Docker. Aquí se proporciona un ejemplo de un script de este tipo . El `build.sh` script que usamos se encuentra en la raíz del repositorio de aplicaciones de Marvel . Contiene el mismo código que en el ejemplo. Skaffold utiliza este script para construir el artefacto de imagen en la fase de construcción. Luego, Kustomize genera dinámicamente los

La próxima vez, nos ensuciaremos las manos siguiendo cada uno de los pasos necesarios para construir esta canalización. ¡También profundizaremos en el Operador MongoDB y explicaremos por qué lo necesita!