1. Utwórz nowy Pod o nazwie nginx z obrazem nginx:1.17.10. Ujawnij port kontenera 80. Pod powinien znajdować się w przestrzeni nazw o nazwie ckad.

2. Uzyskaj szczegółowe informacje o Podze, w tym jego adres IP.

3. Utwórz tymczasowy Pod, który używa obrazu busybox do wykonania polecenia wget wewnątrz kontenera. Polecenie wget powinno uzyskać dostęp do punktu końcowego udostępnionego przez kontener Nginx. Powinieneś zobaczyć treść odpowiedzi HTML wyrenderowaną w terminalu.

4. Pobierz dzienniki kontenera Nginx.

5. Dodaj zmienne środowiskowe

DB_URL=postgresql://mydb:5432 i

DB_USERNAME=admin do kontenera nginx. Pod

6. Otwórz powłokę kontenera nginx i sprawdź zawartość bieżącego katalogu ls -l.

7. Utwórz manifest YAML dla pętli o nazwie Pod, która uruchamia obraz busybox w kontenerze. Kontener powinien uruchomić następującą komendę: for i in {1..10}; wykonaj echo "Witamy $i razy"; zrobione. Utwórz pod z manifestu YAML. Jaki jest status kapsuły?

8. Edytuj pętlę nazwaną Pod. Zmień polecenie, aby działało w nieskończonej pętli. Każda iteracja powinna odzwierciedlać bieżącą datę.

9. Sprawdź zdarzenia i stan pętli Poda.

10. Usuń przestrzeń nazw ckad i jej Pody.

Odpowiedzi

1. Można zastosować podejście imperatywne lub podejście deklaratywne. Najpierw przyjrzymy się tworzeniu przestrzeni nazw za pomocą podejścia imperatywnego:

$ kubectl create namespace ckad

Utwórz poda:

$ kubectl run nginx --image=nginx:1.17.10 --port=80 -- namespace=ckad

Alternatywnie możesz zastosować podejście deklaratywne. Utwórz nowy plik YAML o nazwie ckad-namespace.yaml z następującą zawartością:

apiVersion: v1

kind: Namespace

metadata:

name: ckad

Utwórz przestrzeń nazw z pliku YAML:

$ kubectl create -f ckad-namespace.yaml

Utwórz nowy plik YAML o nazwie nginx-pod.yaml z następującą zawartością:

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx:1.17.10

ports:

- containerPort: 80

Utwórz Poda z pliku YAML:

$ kubectl create -f nginx-pod.yaml --namespace=ckad

2. Możesz użyć opcji wiersza poleceń -o wide, aby pobrać adres IP Poda:

$ kubectl get pod nginx --namespace=ckad -o wide

Te same informacje są dostępne w przypadku zapytania o szczegóły kapsuły:

$ kubectl describe pod nginx --namespace=ckad | grep IP:

3. Możesz użyć opcji wiersza poleceń --rm i -it, aby uruchomić tymczasowy Pod. Poniższe polecenie zakłada, że adres IP kapsuły o nazwie nginx to 10.1.0.66:

$ kubectl run busybox --image=busybox --restart=Never - -rm -it -n ckad \ -- wget -O-
10.1.0.66:80

4. Aby pobrać logi użyj prostego polecenia logs:

$ kubectl logs nginx --namespace=ckad

5. Zabroniona jest edycja obiektu żywego. Jeśli spróbujesz dodać zmienne środowiskowe, pojawi się komunikat o błędzie:

$ kubectl edit pod nginx --namespace=ckad

Będziesz musiał odtworzyć obiekt ze zmodyfikowanym plikiem YAML, ale najpierw będziesz musiał usunąć istniejący obiekt:

$ kubectl delete pod nginx --namespace=ckad

Edytuj istniejący plik YAML nginx-pod.yaml:

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx:1.17.10

ports:

- containerPort: 80

env:

- name: DB_URL

value: postgresql://mydb:5432

- name: DB_USERNAME

value: admin

Zastosuj zmiany:

$ kubectl create -f nginx-pod.yaml --namespace=ckad

6. Użyj polecenia exec, aby otworzyć interaktywną powłokę kontenera:

$ kubectl exec -it nginx --namespace=ckad -- /bin/sh # ls -l
v

7. Połącz opcje wiersza poleceń -o yaml i --dryrun= klient, aby zapisać wygenerowany YAML do pliku. Pamiętaj, aby uniknąć znaków cudzysłowu w ciągu renderowanym przez polecenie echo:

$ kubectl run loop --image=busybox -o yaml --dryrun= client \ --restart=Never -- /bin/sh -c 'for i in 1 2 3 4 5 6 7 8 9 10; \

do echo "Welcome $i times"; done' \

> pod.yaml

Utwórz Poda z pliku YAML:

$ kubectl create -f pod.yaml --namespace=ckad

Status Poda będzie wskazywał Zakończono, ponieważ wykonane polecenie w kontenerze nie działa w nieskończonej pętli:

$ kubectl get pod loop --namespace=ckad

8. Polecenie dotyczące kontenera nie może zostać zmienione w przypadku istniejących Podów. Usuń Pod, aby móc zmodyfikować plik manifestu i ponownie utworzyć obiekt:

$ kubectl delete pod loop --namespace=ckad

Zmień zawartość pliku YAML:

apiVersion: v1

kind: Pod

metadata:

creationTimestamp: null

labels:

run: loop

name: loop

spec:

containers:

- args:

- /bin/sh

- -c

- while true; do date; sleep 10; done

image: busybox

name: loop

resources: {}

dnsPolicy: ClusterFirst

restartPolicy: Never

status: {}

Utwórz Poda z pliku YAML:

$ kubectl create -f pod.yaml --namespace=ckad

9. Możesz opisać zdarzenia Poda, zaznaczając termin:

$ kubectl describe pod loop --namespace=ckad | grep –C 10 Events:

10. Możesz po prostu usunąć przestrzeń nazw, co spowoduje usunięcie wszystkich obiektów w przestrzeni nazw:

$ kubectl delete namespace ckad

Konfiguracja

1. Utwórz katalog o nazwie config. W katalogu utwórz dwa pliki. Pierwszy plik powinien mieć nazwę db.txt i zawierać parę klucz-wartość hasło=mypwd. Drugi plik nosi nazwę ext-service.txt i powinien definiować parę klucz-wartość api_key=LmLHbYhsgWZwNifiqaRorH8T.

2. Utwórz sekret o nazwie ext-service-secret, który używa katalogu jako źródła danych i sprawdź reprezentację obiektu YAML.

3. Utwórz kapsułę o nazwie Consumer z obrazem nginx i zamontuj Sekret jako wolumin ze ścieżką montowania /var/app. Otwórz interaktywną powłokę i sprawdź wartości Sekretu.

4. Użyj podejścia deklaratywnego, aby utworzyć ConfigMap o nazwie ext-service-configmap. Podaj pary klucz-wartość api_endpoint=https://myapp.com/api i username=bot jako literały.

5. Wstrzyknij wartości ConfigMap do istniejącego Poda jako zmienne środowiskowe. Upewnij się, że klucze są zgodne z typowymi konwencjami nazewnictwa zmiennych środowiskowych.

6. Otwórz interaktywną powłokę i sprawdź wartości ConfigMap.

7. Zdefiniuj kontekst zabezpieczeń na poziomie kontenera nowego Poda o nazwie security-context-demo, który korzysta z obrazu alpine. Kontekst zabezpieczeń dodaje do kontenera funkcję systemu Linux CAP_SYS_TIME. Wyjaśnij, czy wartość tego kontekstu zabezpieczeń można ponownie zdefiniować w kontekście zabezpieczeń na poziomie kapsuły.

8. Zdefiniuj ResourceQuota dla przestrzeni nazw Projectfirebird. Reguły powinny ograniczać liczbę tajnych obiektów w przestrzeni nazw do 1.

9. Utwórz tyle obiektów Secret w przestrzeni nazw, aż zostanie osiągnięta maksymalna liczba wymuszona przez ResourceQuota.

10. Utwórz nowe Konto usługi o nazwie monitorowanie i przypisz je do nowego Poda z wybranym przez Ciebie obrazem. Otwórz interaktywną powłokę i zlokalizuj token uwierzytelniający przypisanego Konta usługi.

1. Najprostszym sposobem na utworzenie sekretu jest zastosowanie podejścia imperatywnego, ponieważ nie trzeba ręcznie kodować wartości w formacie Base64. Zacznij od utworzenia katalogu i odpowiednich plików. Poniższe polecenia umożliwiają osiągnięcie tego na platformach Unix, Linux i macOS. Oczywiście możesz także tworzyć pliki i treści ręcznie za pomocą edytora:

$ mkdir config

$ echo -e "password=mypwd" > config/db.txt

$ echo -e "api_key=LmLHbYhsgWZwNifiqaRorH8T" >

config/ext-service.txt

$ ls config

db.txt ext-service.txt

2. Użyj imperatywnego podejścia, aby utworzyć nowy sekret, wskazując go do katalogu konfiguracyjnego. Po sprawdzeniu aktywnego obiektu odkryjesz, że każdy klucz używa nazwy pliku konfiguracyjnego. Wartości zostały zakodowane w formacie Base64:

$ kubectl create secret generic ext-service-secret --

from-file=config

secret/ext-service-secret created

$ kubectl get secret ext-service-secret -o yaml

apiVersion: v1

data:

db.txt: cGFzc3dvcmQ9bXlwd2QK

ext-service.txt:

YXBpX2tleT1MbUxIYlloc2dXWndOaWZpcWFSb3JIOFQK

kind: Secret

metadata:

creationTimestamp: "2020-07-12T23:56:33Z"

managedFields:

- apiVersion: v1

fieldsType: FieldsV1

fieldsV1:

f:data:

.: {}

f:db.txt: {}

f:ext-service.txt: {}

f:type: {}

manager: kubectl

operation: Update

time: "2020-07-12T23:56:33Z"

name: ext-service-secret

namespace: default

resourceVersion: "1462456"

selfLink: /api/v1/namespaces/default/secrets/extservice-

secret

uid: b7f4faae-e624-4027-8bcf-af385019a8d8

type: Opaque

3. Na początek wygeneruj manifest YAML Poda.

$ kubectl run consumer --image=nginx --dry-run=client -

-restart=Never \

-o yaml > pod.yaml

Następnie zmodyfikuj manifest, montując sekret jako wolumin. Wynik końcowy może wyglądać jak poniższa definicja YAML:

apiVersion: v1

kind: Pod

metadata:

creationTimestamp: null

labels:

run: consumer

name: consumer

spec:

containers:

- image: nginx

name: consumer

volumeMounts:

- name: secret-volume

mountPath: /var/app

readOnly: true

resources: {}

volumes:

- name: secret-volume

secret:

secretName: ext-service-secret

dnsPolicy: ClusterFirst

restartPolicy: Never

status: {}

Teraz utwórz kapsułę. Wskocz do kapsuły, gdy tylko status wskaże Uruchomiony. Przejdź do katalogu /var/app. Każda para klucz-wartość Sekretu istnieje jako plik i obserwuje swoją wartość w postaci zwykłego tekstu jako treść:

$ kubectl create -f pod.yaml

pod/consumer created

$ kubectl get pod consumer

NAME READY STATUS RESTARTS AGE

consumer 1/1 Running 0 17s

$ kubectl exec consumer -it -- /bin/sh

cd /var/app

ls

db.txt ext-service.txt

cat db.txt

password=mypwd

cat ext-service.txt

api_key=LmLHbYhsgWZwNifiqaRorH8T

exit

4. Zwykle łatwiej i szybciej jest utworzyć ConfigMap, uruchamiając polecenie imperatywne. Tutaj będziemy chcieli przećwiczyć podejście deklaratywne. Manifest YAML dla ConfigMap z oczekiwanymi parami klucz-wartość może wyglądać następująco:

apiVersion: v1

kind: ConfigMap

metadata:

name: ext-service-configmap

data:

api_endpoint: https://myapp.com/api

username: bot

Korzystając z tej definicji, utwórz obiekt:

$ kubectl create -f configmap.yaml

configmap/ext-service-configmap created

$ kubectl get configmap ext-service-configmap

NAME DATA AGE

ext-service-configmap 2 36s

$ kubectl get configmap ext-service-configmap -o yaml

apiVersion: v1

data:

api_endpoint: https://myapp.com/api

username: bot

kind: ConfigMap

metadata:

creationTimestamp: "2020-07-13T00:17:43Z"

managedFields:

- apiVersion: v1

fieldsType: FieldsV1

fieldsV1:

f:data:

.: {}

f:api_endpoint: {}

f:username: {}

manager: kubectl

operation: Update

time: "2020-07-13T00:17:43Z"

name: ext-service-configmap

namespace: default

resourceVersion: "1465228"

selfLink: /api/v1/namespaces/default/configmaps/extservice-

configmap

uid: b1b51b17-2dad-4320-b7c2-6758feca3800

5. Klucze danych konfiguracyjnych ConfigMap nie są zgodne z typowymi konwencjami nazewnictwa zmiennych środowiskowych. Bez modyfikowania kluczy w ConfigMap, nadal możesz przypisać je do bardziej rozsądnej konwencji nazewnictwa podczas wstrzykiwania ich do kapsuły. Będziesz musiał ponownie utworzyć Pod, aby wprowadzić niezbędne zmiany, ponieważ Kubernetes nie pozwala na dodawanie nowych zmiennych środowiskowych do działającego kontenera. Wynikowy manifest YAML może wyglądać jak poniższy fragment kodu

apiVersion: v1

kind: Pod

metadata:

creationTimestamp: null

labels:

run: consumer

name: consumer

spec:

containers:

- image: nginx

name: consumer

volumeMounts:

- name: secret-volume

mountPath: /var/app

readOnly: true

env:

- name: API_ENDPOINT

valueFrom:

configMapKeyRef:

name: ext-service-configmap

key: api_endpoint

- name: USERNAME

valueFrom:

configMapKeyRef:

name: ext-service-configmap

key: username

volumes:

- name: secret-volume

secret:

secretName: ext-service-secret

dnsPolicy: ClusterFirst

restartPolicy: Always

status: {}

6. Powinieneś być w stanie znaleźć zmienną środowiskową o właściwej nazwie, uruchamiając komendę env z poziomu kontenera:

$ kubectl exec -it consumer -- /bin/sh

```
# env
...
API_ENDPOINT=https://myapp.com/api
USERNAME=bot
...
# exit
```

7. Możesz zacząć od utworzenia manifestu Poda za pomocą polecenia run:

```
$ kubectl run security-context-demo --image=alpine --
dry-run=client \
--restart=Never -o yaml > pod.yaml
```

Edytuj plik pod.yaml i dodaj kontekst zabezpieczeń. Możliwości systemu Linux nie można zastąpić na poziomie kapsuły z dwóch powodów. Z jednej strony możliwości Linuksa można zdefiniować jedynie na poziomie kontenera. Z drugiej strony definicja na poziomie podu nie definiuje na nowo kontekstu bezpieczeństwa na poziomie kontenera — jest odwrotnie:

```
apiVersion: v1
kind: Pod
metadata:
creationTimestamp: null
labels:
run: security-context-demo
name: security-context-demo
spec:
containers:
- image: alpine
name: security-context-demo
resources: {}
securityContext:
capabilities:
add: ["SYS_TIME"]
dnsPolicy: ClusterFirst
restartPolicy: Never
status: {}
```

8. Zacznij od utworzenia nowej przestrzeni nazw:

$ kubectl create namespace project-firebird

namespace/project-firebird created

$ kubectl get namespace project-firebird

NAME STATUS AGE

project-firebird Active 23s

Utwórz manifest YAML dla ResourceQuota. Możesz zdefiniować maksymalną liczbę sekretów w przestrzeni nazw za pomocą atrybutu spec.hard.secrets:

apiVersion: v1

kind: ResourceQuota

metadata:

name: firebird-quota

spec:

hard:

secrets: 1

Załóżmy, że zapisałeś manifest w pliku Resource-quota.yaml; możesz go utworzyć za pomocą następującego polecenia. Pamiętaj o podaniu przestrzeni nazw:

$ kubectl create -f resource-quota.yaml --

namespace=project-firebird

resourcequota/firebird-quota created

9. Zauważysz, że przestrzeń nazw zawiera już sekret należący do domyślnego konta usługi. W efekcie osiągnięto już maksymalną liczbę Sekretów:

$ kubectl get resourcequota firebird-quota --

namespace=project-firebird

NAME AGE REQUEST LIMIT

firebird-quota 39s secrets: 1/1

$ kubectl get secrets --namespace=project-firebird

NAME TYPE

DATA AGE

default-token-mdcd8 kubernetes.io/service-accounttoken

3 7m54s

Teraz śmiało stwórz Sekret. ResourceQuota wyświetli komunikat o błędzie i uniemożliwi utworzenie klucza tajnego:

$ kubectl create secret generic my-secret --fromliteral=

test=hello \

--namespace=project-firebird

Error from server (Forbidden): secrets "my-secret" is

forbidden: \

exceeded quota: firebird-quota, requested: secrets=1,

used: secrets=1, \

limited: secrets=1

10. Cały proces możesz przejść uruchamiając polecenia imperatywne. Zacznij od utworzenia niestandardowego konta usługi, następnie utwórz nowy Pod i użyj opcji wiersza poleceń --serviceaccount, aby przypisać konto usługi. Token uwierzytelniający znajdziesz w katalogu kontenera /var/run/secrets/kubernetes.io/serviceaccount/token:

$ kubectl create serviceaccount monitoring

serviceaccount/monitoring created

$ kubectl get serviceaccount monitoring

NAME SECRETS AGE

monitoring 1 12s

$ kubectl run nginx --image=nginx --restart=Never \

--serviceaccount=monitoring

pod/nginx created

$ kubectl exec -it nginx -- /bin/sh

cat

/var/run/secrets/kubernetes.io/serviceaccount/token

eyJhbGciO...rH4fkeYsw

Pody wielopojemnikowe

1. Utwórz manifest YAML dla kapsuły o nazwie complex-pod. Główny kontener aplikacji o nazwie app powinien używać obrazu nginx i udostępniać port kontenera 80. Zmodyfikuj manifest YAML tak, aby Pod definiował kontener początkowy o nazwie setup, który używa obrazu busybox. Kontener init uruchamia polecenie wget -O- google.com.

2. Utwórz Pod z manifestu YAML.

3. Pobierz dzienniki kontenera init. Powinieneś zobaczyć wynik polecenia wget.

4. Otwórz interaktywną powłokę głównego kontenera aplikacji i uruchom komendę ls. Wyjdź z kontenera.

5. Wymuś usunięcie Poda.

6. Utwórz manifest YAML dla Poda o nazwie data-exchange. Główny kontener aplikacji o nazwie main-app powinien używać obrazu busybox. Kontener uruchamia polecenie, które co 30 sekund zapisuje nowy plik w nieskończonej pętli w katalogu /var/app/data. Nazwa pliku jest zgodna ze wzorcem {counter++}-data.txt. Licznik zmiennych jest zwiększany co interwał i zaczyna się od wartości 1.

7. Zmodyfikuj manifest YAML, dodając kontener przyczepki o nazwie sidecar. Kontener wózka bocznego korzysta z busybox obrazu i uruchamia polecenie zliczające liczbę plików generowanych przez kontener aplikacji głównej co 60 sekund w nieskończonej pętli. Polecenie wypisuje liczbę plików na standardowe wyjście.

8. Zdefiniuj wolumin typu pustyDir. Zamontuj ścieżkę /var/app/data dla obu kontenerów.

9. Utwórz kapsułę. Ogonuj logi kontenera z wózkiem bocznym.

10. Usuń kapsułę.

1. Możesz zacząć od wygenerowania manifestu YAML w trybie próbnym. Wynikowy manifest skonfiguruje główny kontener aplikacji:

$ kubectl run complex-pod --image=nginx --port=80 --

restart=Never \

-o yaml --dry-run=client > complex-pod.yaml

Edytuj plik manifestu, dodając kontener init i zmieniając niektóre wygenerowane ustawienia domyślne. Ostateczny manifest może wyglądać następująco:

apiVersion: v1

kind: Pod

metadata:

name: complex-pod

spec:

initContainers:

- image: busybox

name: setup

command: ['sh', '-c', 'wget -O- google.com']

containers:

- image: nginx

name: app

ports:

- containerPort: 80

resources: {}

dnsPolicy: ClusterFirst

restartPolicy: Never

status: {}

2. Uruchom polecenie create, aby utworzyć instancję Poda. Sprawdź, czy Pod działa bez problemów:

$ kubectl create -f complex-pod.yaml

pod/complex-pod created

$ kubectl get pod complex-pod

NAME READY STATUS RESTARTS AGE

complex-pod 1/1 Running 0 27s

3. Użyj polecenia logs i wskaż kontener init, aby pobrać dane wyjściowe dziennika:

$ kubectl logs complex-pod -c setup

Connecting to google.com (172.217.1.206:80)

Connecting to www.google.com (172.217.2.4:80)

writing to stdout

...

4. Możesz także kierować reklamy na główną aplikację. Tutaj otworzysz interaktywną powłokę i uruchomisz polecenie ls:

$ kubectl exec complex-pod -it -c app -- /bin/sh

ls

bin dev docker-entrypoint.sh home lib64 mnt proc

run \

srv tmp var boot docker-entrypoint.d etclib media

opt \

root sbin sys usr

exit

5. Uniknij płynnego usunięcia Poda, dodając opcje -- Grace-period=0 i --force:

$ kubectl delete pod complex-pod --grace-period=0 --

force

warning: Immediate deletion does not wait for

confirmation that the \

running resource has been terminated. The resource may

continue to run \

on the cluster indefinitely.

pod "complex-pod" force deleted

6. Możesz zacząć od wygenerowania manifestu YAML w trybie próbnym. Wynikowy manifest skonfiguruje główny kontener aplikacji:

$ kubectl run data-exchange --image=busybox --

restart=Never -o yaml \

--dry-run=client > data-exchange.yaml

Edytuj plik manifestu, dodając kontener przyczepki i zmieniając niektóre wygenerowane ustawienia domyślne. Ostateczny manifest może wyglądać następująco:

apiVersion: v1

kind: Pod

metadata:

name: data-exchange

spec:

containers:

- image: busybox

name: main-app

command: ['sh', '-c', 'counter=1; while true; do

touch \

"/var/app/data/$counter-data.txt";

counter=$((counter+1)); \

sleep 30; done']

resources: {}

dnsPolicy: ClusterFirst

restartPolicy: Never

status: {}

7. Po prostu dodaj kontener przyczepy bocznej obok głównego kontenera aplikacji, używając odpowiedniego polecenia. Dodaj do istniejącego manifestu YAML:

```
apiVersion: v1
kind: Pod
metadata:
name: data-exchange
spec:
containers:
- image: busybox
name: main-app
command: ['sh', '-c', 'counter=1; while true; do
touch \
"/var/app/data/$counter-data.txt";
counter=$((counter+1)); \
sleep 30; done']
resources: {}
- image: busybox
name: sidecar
command: ['sh', '-c', 'while true; do ls -dq
/var/app/data/*-data.txt \
| wc -l; sleep 30; done']
dnsPolicy: ClusterFirst
restartPolicy: Never
status: {}
```

8. Zmodyfikuj manifest tak, aby do wymiany plików pomiędzy głównym kontenerem aplikacji a kontenerem przyczepy używany był wolumen:

```
apiVersion: v1
kind: Pod
metadata:
name: data-exchange
spec:
containers:
- image: busybox
```

```
name: main-app
command: ['sh', '-c', 'counter=1; while true; do
touch \
"/var/app/data/$counter-data.txt";
counter=$((counter+1)); \
sleep 30; done']
volumeMounts:
- name: data-dir
mountPath: "/var/app/data"
resources: {}
- image: busybox
name: sidecar
command: ['sh', '-c', 'while true; do ls -d
/var/app/data/*-data.txt \
| wc -l; sleep 30; done']
volumeMounts:
- name: data-dir
mountPath: "/var/app/data"
volumes:
- name: data-dir
emptyDir: {}
dnsPolicy: ClusterFirst
restartPolicy: Never
status: {}
```

9. Utwórz kapsułę, sprawdź jej istnienie i zapisz dzienniki kontenera przyczepy bocznej. Liczba plików będzie się zwiększać z biegiem czasu:

```
$ kubectl create -f data-exchange.yaml
pod/data-exchange created
$ kubectl get pod data-exchange
NAME READY STATUS RESTARTS AGE
data-exchange 2/2 Running 0 31s
```

$ kubectl logs data-exchange -c sidecar -f

1

2

...

10. Usuń kapsułę:

$ kubectl delete pod data-exchange

pod "data-exchange" deleted

Obserwowalność

1. Zdefiniuj nowy Pod o nazwie serwer WWW z obrazem nginx w manifeście YAML. Odsłoń port kontenera 80. Nie twórz jeszcze kapsuły.

2. Dla kontenera zadeklaruj sondę startową typu httpGet. Sprawdź, czy można wywołać punkt końcowy kontekstu głównego. Użyj domyślnej konfiguracji sondy.

3. Dla kontenera zadeklaruj sondę gotowości typu httpGet. Sprawdź, czy można wywołać punkt końcowy kontekstu głównego. Przed pierwszym sprawdzeniem należy odczekać pięć sekund.

4. Dla kontenera zadeklaruj sondę na żywo typu httpGet. Sprawdź, czy można wywołać punkt końcowy kontekstu głównego. Przed pierwszym sprawdzeniem należy odczekać 10 sekund. Sonda powinna przeprowadzać kontrolę co 30 sekund.

5. Utwórz Poda i postępuj zgodnie z fazami cyklu życia Poda podczas tego procesu.

6. Sprawdź szczegóły czasu działania sond kapsuły.

7. Pobierz metryki Poda (np. procesor i pamięć) z serwera metryk.

8. Utwórz kapsułę o nazwie niestandardowe-cmd z obrazem busybox. Kontener powinien uruchomić polecenie top-analyzer z flagą wiersza polecenia --all.

9. Sprawdź status. W jaki sposób można dalej rozwiązywać problemy z kapsułą, aby zidentyfikować pierwotną przyczynę awarii?

1. Możesz zacząć od wygenerowania manifestu YAML w trybie próbnym. Wynikowy manifest utworzy kontener z odpowiednim obrazem:

$ kubectl run web-server --image=nginx --port=80 --

restart=Never \

-o yaml --dry-run=client > probed-pod.yaml

2. Edytuj manifest, definiując sondę startową. Ostateczny manifest może wyglądać następująco:

apiVersion: v1

kind: Pod

metadata:

creationTimestamp: null

labels:

run: web-server

name: web-server

spec:

containers:

- image: nginx

name: web-server

ports:

- containerPort: 80

name: nginx-port

startupProbe:

httpGet:

path: /

port: nginx-port

resources: {}

dnsPolicy: ClusterFirst

restartPolicy: Never

status: {}

3. Dalsza edycja manifestu poprzez zdefiniowanie sondy gotowości. Ostateczny manifest może wyglądać następująco:

apiVersion: v1

kind: Pod

metadata:

creationTimestamp: null

labels:

run: web-server

name: web-server

spec:

containers:

- image: nginx

name: web-server

ports:

- containerPort: 80

name: nginx-port

startupProbe:

httpGet:

path: /

port: nginx-port

readinessProbe:

httpGet:

path: /

port: nginx-port

initialDelaySeconds: 5

resources: {}

dnsPolicy: ClusterFirst

restartPolicy: Never

status: {}

4. Kontynuuj edycję manifestu, definiując sondę na żywo. Ostateczny manifest może wyglądać następująco:

apiVersion: v1

kind: Pod

metadata:

creationTimestamp: null

labels:

run: web-server

name: web-server

spec:

containers:

- image: nginx

name: web-server

ports:

- containerPort: 80

name: nginx-port

startupProbe:

httpGet:

path: /

port: nginx-port

readinessProbe:

httpGet:

path: /

port: nginx-port

initialDelaySeconds: 5

livenessProbe:

httpGet:

path: /

port: nginx-port

initialDelaySeconds: 10

periodSeconds: 30

resources: {}

dnsPolicy: ClusterFirst

restartPolicy: Never

status: {}

5. Utwórz kapsułę, a następnie sprawdź jej kolumny GOTOWY i STATUS. Kontener przejdzie z trybu ContainerCreating do Running. W pewnym momencie dostępny będzie kontener 1/1:

$ kubectl create -f probed-pod.yaml

pod/probed-pod created

$ kubectl get pod web-server

NAME READY STATUS RESTARTS AGE

web-server 0/1 ContainerCreating 0 7s

$ kubectl get pod web-server

NAME READY STATUS RESTARTS AGE

web-server 0/1 Running 0 8s

$ kubectl get pod web-server

NAME READY STATUS RESTARTS AGE

web-server 1/1 Running 0 38s

6. Po wykonaniu polecenia opisu powinieneś znaleźć konfigurację sond:

$ kubectl describe pod web-server

...

Containers:

web-server:

...

Ready: True

Restart Count: 0

Liveness: http-get http://:nginx-port/

delay=10s timeout=1s \

period=30s #success=1 #failure=3

Readiness: http-get http://:nginx-port/

delay=5s timeout=1s \

period=10s #success=1 #failure=3

Startup: http-get http://:nginx-port/

delay=0s timeout=1s \

period=10s #success=1 #failure=3

...

7. Uruchom polecenie top, aby pobrać metryki monitorowania z serwera metryk:

$ kubectl top pod web-server

NAME CPU(cores) MEMORY(bytes)

web-server 0m 2Mi

8. Możesz użyć polecenia run i podać polecenie uruchomienia jako argument. Status Poda zmieni się na Error:

$ kubectl run custom-cmd --image=busybox --

restart=Never \

-- /bin/sh -c "top-analyzer --all"

pod/custom-cmd created

$ kubectl get pod custom-cmd

NAME READY STATUS RESTARTS AGE

custom-cmd 0/1 Error 0 71s

9. Użyj polecenia logs, aby znaleźć bardziej przydatne informacje o czasie wykonywania. Z komunikatu o błędzie dowiesz się, że narzędzie Top-Analizer nie jest dostępne dla obrazu:

$ kubectl logs custom-cmd

/bin/sh: top-analyzer: not found

Projektowanie Pod

1. Utwórz trzy Pody korzystające z obrazu nginx. Nazwy kapsuł powinny brzmieć: pod-1, pod-2 i pod-3. Przypisz etykietę tier=frontend do pod-1 i etykietę tier=backend do pod-2 i pod-3. Do wszystkich podów należy także przypisać etykietę team=artemidis.

2. Przypisz adnotację z kluczem wdrażającym do zasobnika-1 i zasobnika-3. Użyj własnego imienia jako wartości.

3. W wierszu poleceń wybierz etykietę, aby znaleźć wszystkie Pody z zespołem artemidis lub aircontrol i które są uważane za usługę zaplecza.

4. Utwórz nowe wdrożenie o nazwie serwer-wdrożenie. Wdrożenie powinno kontrolować dwie repliki przy użyciu obrazu grand-server: 1.4.6.

5. Sprawdź wdrożenie i znajdź podstawową przyczynę jego niepowodzenia.

6. Rozwiąż problem, przypisując zamiast tego obraz nginx. Sprawdź historię wdrażania. Ile poprawek byś się spodziewał?

7. Utwórz nowy CronJob o nazwie google-ping. Po wykonaniu zadanie powinno uruchomić polecenie curl dla google.com. Wybierz odpowiedni obraz. Wycięcie powinno odbywać się co dwie minuty.

8. Zapisz dzienniki zadania CronJob w czasie wykonywania. Sprawdź opcje wiersza poleceń odpowiedniego polecenia lub zapoznaj się z dokumentacją Kubernetes.

9. Skonfiguruj ponownie CronJob, aby zachować historię siedmiu wykonań. 10. Skonfiguruj ponownie zadanie CronJob, aby uniemożliwić nowe wykonanie, jeśli bieżące wykonanie nadal trwa. Aby uzyskać więcej informacji, zapoznaj się z dokumentacją Kubernetes.

1. Zacznij od utworzenia Podów. Możesz przypisać etykiety w momencie tworzenia:

$ kubectl run pod-1 --image=nginx --restart=Never \

--labels=tier=frontend,team=artemidis

pod/pod-1 created

$ kubectl run pod-2 --image=nginx --restart=Never \

--labels=tier=backend,team=artemidis

pod/pod-2 created

$ kubectl run pod-3 --image=nginx --restart=Never \

--labels=tier=backend,team=artemidis

pod/pod-3 created

$ kubectl get pods --show-labels

NAME READY STATUS RESTARTS AGE LABELS

pod-1 1/1 Running 0 30s

team=artemidis,tier=frontend

pod-2 1/1 Running 0 24s

team=artemidis,tier=backend

pod-3 1/1 Running 0 16s

team=artemidis,tier=backend

2. Możesz edytować aktywne obiekty, aby dodać adnotację, lub użyć polecenia adnotacja. Użyjemy tutaj polecenia rozkazującego:

$ kubectl annotate pod pod-1 pod-3 deployer='Benjamin

Muschko'

pod/pod-1 annotated

pod/pod-3 annotated

$ kubectl describe pod pod-1 pod-3 | grep Annotations:

Annotations: deployer: Benjamin Muschko

Annotations: deployer: Benjamin Muschko

3. Wybór etykiety wymaga połączenia kryteriów równości i kryteriów, aby znaleźć Pody

$ kubectl get pods -l tier=backend,'team in

(artemidis,aircontrol)' \

--show-labels

NAME READY STATUS RESTARTS AGE LABELS

pod-2 1/1 Running 0 6m38s

team=artemidis,tier=backend

pod-3 1/1 Running 0 6m30s

team=artemidis,tier=backend

4. Polecenie tworzenia wdrożenia tworzy stanowisko, ale nie pozwala na podanie liczby replik w opcji wiersza poleceń. Następnie będziesz musiał uruchomić polecenie skalowania:

$ kubectl create deployment server-deployment --

image=grand-server:1.4.6

deployment.apps/server-deployment created

$ kubectl scale deployment server-deployment --

replicas=2

deployment.apps/server-deployment scaled

5. Przekonasz się, że wdrożenie nie udostępnia żadnego z Podów nawet po pewnym czasie oczekiwania. Problem polega na tym, że przypisany obraz nie istnieje. Spojrzenie na jeden z podów ujawni problem w dzienniku zdarzeń:

$ kubectl get deployments

NAME READY UP-TO-DATE AVAILABLE

AGE

server-deployment 0/2 2 0

69s

$ kubectl get pods

NAME READY STATUS

RESTARTS \

AGE

server-deployment-779f77f555-q6tq2 0/1

ImagePullBackOff 0 \

4m31s

server-deployment-779f77f555-sxtnc 0/1

ImagePullBackOff 0 \

3m45s

$ kubectl describe pod server-deployment-779f77f555-

q6tq2

...

Events:

Type Reason Age From \

Message

---- ------ ---- ---- \

Normal Scheduled <unknown> defaultscheduler

\

Successfully assigned default/server-deployment-

779f77f555-q6tq2 \

to minikube

Normal Pulling 3m17s (x4 over 4m54s) kubelet,

minikube \

Pulling image "grand-server:1.4.6"

Warning Failed 3m16s (x4 over 4m53s) kubelet,

minikube \

Failed to pull image "grand-server:1.4.6": rpc error:

code = \

Unknown desc = Error response from daemon: pull

access denied \

for grand-server, repository does not exist or may

require \

'docker login': denied: requested access to the

resource is denied

Warning Failed 3m16s (x4 over 4m53s) kubelet,

minikube \

Error: ErrImagePull

Normal BackOff 3m5s (x6 over 4m53s) kubelet,

minikube \

Back-off pulling image "grand-server:1.4.6"

Warning Failed 2m50s (x7 over 4m53s) kubelet,

minikube \

Error: ImagePullBackOff

6. Polecenie set image to przydatny skrót umożliwiający przypisanie nowego obrazu do stanowiska. Po zmianie historia wdrożenia powinna zawierać dwie wersje: jedną wersję dotyczącą początkowego utworzenia Wdrożenia i drugą dotyczącą zmiany obrazu:

$ kubectl set image deployment server-deployment grandserver=

nginx

deployment.apps/server-deployment image updated

$ kubectl rollout history deployments server-deployment

deployment.apps/server-deployment

REVISION CHANGE-CAUSE

1 <none>

2 <none>

7. Możesz użyć obrazu nginx, który ma zainstalowane narzędzie wiersza poleceń curl. Wyrażenie cron uniksowe dla tego zadania to */2 * * * *:

$ kubectl create cronjob google-ping --schedule="*/2 *

* * *" \

--image=nginx -- /bin/sh -c 'curl google.com'

cronjob.batch/google-ping created

8. Możesz sprawdzić, kiedy CronJob jest wykonywany, używając opcji wiersza poleceń -w:

$ kubectl get cronjob -w

NAME SCHEDULE SUSPEND ACTIVE LAST

SCHEDULE AGE

google-ping */2 * * * * False 0 115s

2m10s

google-ping */2 * * * * False 1 6s

2m21s

google-ping */2 * * * * False 0 16s

2m31s

google-ping */2 * * * * False 1 6s

4m21s

google-ping */2 * * * * False 0 16s

4m31s

9. Jawnie przypisz wartość 7 do atrybutu spec.successfulJobsHistoryLimit aktywnego obiektu. Wynikowy manifest YAML powinien mieć następującą konfigurację:

...

spec:

successfulJobsHistoryLimit: 7

10. Edytuj domyślną wartość spec.concurrencyPolicy aktywnego obiektu. Wynikowy manifest YAML powinien mieć następującą konfigurację:

...

spec:

concurrencyPolicy: Forbid

Usługi i sieci

1. Utwórz nowy Pod o nazwie frontend, który używa obrazu nginx. Przypisz etykiety tier=frontend i app=nginx. Odsłoń port kontenera 80.

2. Utwórz nowy Pod o nazwie backend, który używa obrazu nginx. Przypisz etykiety tier=backend i app=nginx. Odsłoń port kontenera 80.

3. Utwórz nową usługę o nazwie nginx-service typu ClusterIP. Przypisz port 9000 i port docelowy 80. Selektor etykiet powinien używać kryteriów warstwa=backend i wdrożenie=aplikacja.

4. Spróbuj uzyskać dostęp do zestawu Podów za pośrednictwem Usługi z poziomu klastra. Które Pody wybiera Usługa?

5. Napraw przypisanie usługi, aby prawidłowo wybrać moduł zaplecza i przypisać właściwy port docelowy.

6. Udostępnij usługę tak, aby była dostępna spoza klastra. Zadzwoń do Serwisu.

7. Załóżmy, że stos aplikacji definiuje trzy różne warstwy: frontend, backend i bazę danych. Każda z warstw działa w kapsule. Definicję można znaleźć w stosie aplikacji pliku YAML. yaml:

kind: Pod

apiVersion: v1

metadata:

name: frontend

namespace: app-stack

labels:

app: todo

tier: frontend

spec:

```
containers:
- name: frontend
image: nginx
---
kind: Pod
apiVersion: v1
metadata:
name: backend
namespace: app-stack
labels:
app: todo
tier: backend
spec:
containers:
- name: backend
image: nginx
---
kind: Pod
apiVersion: v1
metadata:
name: database
namespace: app-stack
labels:
app: todo
tier: database
spec:
containers:
- name: database
image: mysql
env:
- name: MYSQL_ROOT_PASSWORD
```

value: example

Utwórz przestrzeń nazw i kapsuły, korzystając z pliku appstack.yaml.

8. Utwórz politykę sieciową w pliku app-stack-networkpolicy.yaml. Polityka sieciowa powinna zezwalać na ruch przychodzący z backendu do bazy danych, ale nie zezwalać na ruch przychodzący z frontendu.

9. Skonfiguruj ponownie zasady sieciowe, aby zezwalać na ruch przychodzący do bazy danych tylko na porcie TCP 3306, a nie na innym porcie.

1. Najszybszym sposobem na utworzenie kapsuły jest użycie polecenia run. W poniższym poleceniu możesz zobaczyć, że możesz przypisać port i etykiety w momencie tworzenia Poda:

$ kubectl run frontend --image=nginx --restart=Never --

port=80 \

-l tier=frontend,app=nginx

pod/frontend created

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

frontend 1/1 Running 0 21s

2. Użyj tej samej metody, aby utworzyć moduł backend pod:

$ kubectl run backend --image=nginx --restart=Never --

port=80 \

-l tier=backend,app=nginx

pod/backend created

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

backend 1/1 Running 0 19s

frontend 1/1 Running 0 3m53s

3. Do wygenerowania usługi możesz użyć polecenia create service. Niestety nie można od razu przypisać etykiet. Dlatego zapiszesz wynik polecenia w pliku YAML, a następnie dokonasz edycji definicji selektora etykiet:

$ kubectl create service clusterip nginx-service --

tcp=9000:8081 \

--dry-run=client -o yaml > nginx-service.yaml

Edytuj manifest YAML, aby zmodyfikować selektor etykiet. Wynik powinien wyglądać podobnie do następującego manifestu YAML:

apiVersion: v1

kind: Service

metadata:

creationTimestamp: null

labels:

app: nginx-service

name: nginx-service

spec:

ports:

- port: 9000

protocol: TCP

targetPort: 8081

selector:

tier: backend

deployment: app

type: ClusterIP

status:

loadBalancer: {}

Teraz utwórz usługę z pliku YAML. Lista usługi powinna pokazywać prawidłowy typ i odsłonięty port:

$ kubectl create -f nginx-service.yaml

service/nginx-service created

$ kubectl get services

NAME TYPE CLUSTER-IP EXTERNALIP

PORT(S) AGE

nginx-service ClusterIP 10.110.127.205 <none>

9000/TCP 20s

4. Próba połączenia z podstawowymi Podami usługi nie będzie działać. Na przykład upłynął limit czasu polecenia wget. Takie zachowanie ma miejsce, ponieważ konfiguracja Usługi nie wybiera żadnych Podów z dwóch powodów. Po pierwsze, selektor etykiet nie pasuje do żadnego z istniejących Podów. Po drugie, port docelowy nie jest dostępny w żadnym z istniejących Podów:

$ kubectl run busybox --image=busybox --restart=Never -

it --rm -- /bin/sh

/ # wget --spider --timeout=1 10.110.127.205:9000

Connecting to 10.110.127.205:9000 (10.110.127.205:9000)

wget: download timed out

/ # exit

pod "busybox" deleted

5. Edytuj aktywny obiekt Usługi, aby wyglądał następująco. W poniższym fragmencie kodu widać, że selektor etykiet został zmieniony, a także port docelowy:

apiVersion: v1

kind: Service

metadata:

creationTimestamp: null

labels:

app: nginx-service

name: nginx-service

spec:

ports:

- port: 9000

protocol: TCP

targetPort: 80

selector:

tier: backend

app: nginx

type: ClusterIP

status:

loadBalancer: {}

W wyniku zmiany możliwe będzie połączenie się z backendem Podem:

$ kubectl run busybox --image=busybox --restart=Never -

it --rm -- /bin/sh

/ # wget --spider --timeout=1 10.110.127.205:9000

Connecting to 10.110.127.205:9000 (10.110.127.205:9000)

remote file exists

/ # exit

pod "busybox" deleted

6. Możesz bezpośrednio modyfikować obiekt aktywny usługi nginx, wprowadzając żądane zmiany YAML. Tutaj przełączasz się z typu ClusterIP na typ NodePort. Możesz teraz połączyć się z nim spoza klastra, korzystając z adresu IP węzła i przypisanego portu statycznego:

$ kubectl patch service nginx-service -p \

'{ "spec": {"type": "NodePort"} }'

service/nginx-service patched

$ kubectl get services

NAME TYPE CLUSTER-IP EXTERNALIP

PORT(S) \

AGE

nginx-service NodePort 10.110.127.205 <none>

9000:32682/TCP \

141m

$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

minikube Ready master 102d v1.18.3

$ kubectl describe node minikube | grep InternalIP:

InternalIP: 192.168.64.2

$ wget --spider --timeout=1 192.168.64.2:32682

Spider mode enabled. Check if remote file exists.

--2020-09-26 15:59:12-- http://192.168.64.2:32682/

Connecting to 192.168.64.2:32682... connected.

HTTP request sent, awaiting response... 200 OK

Length: 612 [text/html]

7. Zacznij od utworzenia przestrzeni nazw o nazwie app-stack. Skopiuj zawartość podanej definicji YAML do pliku appstack. yaml i zastosuj go. Powinieneś otrzymać trzy Pody:

$ kubectl create namespace app-stack

namespace/app-stack created

$ kubectl apply -f app-stack.yaml

pod/frontend created

pod/backend created

pod/database created

$ kubectl get pods -n app-stack

NAME READY STATUS RESTARTS AGE

backend 1/1 Running 0 105s

database 1/1 Running 0 105s

frontend 1/1 Running 0 105s

8. Utwórz nowy plik o nazwie app-stack-network-policy.yaml. Poniższe reguły opisują pożądany ruch przychodzący i wychodzący dla Poda bazy danych:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: app-stack-network-policy

namespace: app-stack

spec:

podSelector:

matchLabels:

app: todo

tier: database

policyTypes:

- Ingress

- Egress

ingress:

- from:

- podSelector:

matchLabels:

app: todo

tier: backend

Zastosuj plik YAML za pomocą następującego polecenia:

$ kubectl create -f app-stack-network-policy.yaml

networkpolicy.networking.k8s.io/app-stack-networkpolicy

created

$ kubectl get networkpolicy -n app-stack

NAME POD-SELECTOR AGE

app-stack-network-policy app=todo,tier=database 7s

9. Możesz dodatkowo ograniczyć porty, stosując następującą definicję:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: app-stack-network-policy

namespace: app-stack

spec:

podSelector:

matchLabels:

app: todo

tier: database

policyTypes:

- Ingress

- Egress

ingress:

- from:

- podSelector:

matchLabels:

app: todo

tier: backend

ports:

- protocol: TCP

port: 3306

Polecenie opisu może sprawdzić, czy zastosowano poprawną regułę ruchu przychodzącego na port:

$ kubectl describe networkpolicy app-stack-networkpolicy

-n app-stack

Name: app-stack-network-policy

Namespace: app-stack

Created on: 2020-09-27 16:22:31 -0600 MDT

Labels: <none>

Annotations: <none>

Spec:

PodSelector: app=todo,tier=database

Allowing ingress traffic:

To Port: 3306/TCP

From:

PodSelector: app=todo,tier=backend

Allowing egress traffic:

<none> (Selected pods are isolated for egress

connectivity)

Policy Types: Ingress, Egress

Trwałość państwa

1. Utwórz plik Pod YAML z dwoma kontenerami korzystającymi z obrazu alpine:3.12.0. Podaj polecenie dla obu kontenerów, które sprawi, że będą działać wiecznie.

2. Zdefiniuj wolumin typu pustyDir dla kapsuły. Kontener 1 powinien zamontować wolumin do ścieżki /etc/a, a kontener 2 powinien zamontować wolumen do ścieżki /etc/b.

3. Otwórz powłokę interaktywną dla kontenera 1 i utwórz katalog danych w ścieżce podłączenia. Przejdź do katalogu i utwórz plik hello.txt z zawartością „Hello World". Wyjdź z kontenera.

4. Otwórz interaktywną powłokę kontenera 2 i przejdź do katalogu /etc/b/data. Sprawdź zawartość pliku hello.txt. Wyjdź z kontenera.

5. Utwórz wolumin PersistentVolume o nazwie logs-pv, który jest mapowany na ścieżkę hosta /var/logs. Tryb dostępu powinien mieć wartość ReadWriteOnce i ReadOnlyMany. Zapewnij pojemność pamięci 5Gi. Upewnij się, że stan PersistentVolume ma wartość Dostępny.

6. Utwórz żądanie PersistentVolumeClaim o nazwie logs-pvc. Dostęp, którego używa, to ReadWriteOnce. Poproś o pojemność 2Gi. Upewnij się, że stan PersistentVolume ma wartość Bound.

7. Zamontuj PersistentVolumeClaim w kapsule z uruchomionym obrazem nginx w ścieżce montowania /var/log/nginx.

8. Otwórz interaktywną powłokę kontenera i utwórz nowy plik o nazwie my-nginx.log w /var/log/nginx. Wyjdź z kapsuły.

9. Usuń Poda i utwórz go ponownie z tym samym manifestem YAML. Otwórz interaktywną powłokę Poda, przejdź do katalogu /var/log/nginx i znajdź wcześniej utworzony plik.

1. Zacznij od wygenerowania manifestu YAML za pomocą polecenia run w połączeniu z opcją --dry-run:

```
$ kubectl run alpine --image=alpine:3.12.0 --dryrun=
client \
--restart=Never -o yaml -- /bin/sh -c "while true; do
sleep 60; \
done;" > multi-container-alpine.yaml
$ vim multi-container-alpine.yaml
```

Po edycji Poda manifest może wyglądać następująco. Nazwy kontenerów to kontener1 i kontener2:

```
apiVersion: v1
kind: Pod
metadata:
creationTimestamp: null
labels:
run: alpine
name: alpine
spec:
containers:
- args:
- /bin/sh
- -c
- while true; do sleep 60; done;
image: alpine:3.12.0
name: container1
resources: {}
- args:
```

- /bin/sh

- -c

- while true; do sleep 60; done;

image: alpine:3.12.0

name: container2

resources: {}

dnsPolicy: ClusterFirst

restartPolicy: Always

status: {}

2. Dokonaj dalszej edycji pliku YAML, dodając wolumin i ścieżki montowania dla obu kontenerów. Ostatecznie definicja Poda mogłaby wyglądać następująco:

apiVersion: v1

kind: Pod

metadata:

creationTimestamp: null

labels:

run: alpine

name: alpine

spec:

volumes:

- name: shared-vol

emptyDir: {}

containers:

- args:

- /bin/sh

- -c

- while true; do sleep 60; done;

image: alpine:3.12.0

name: container1

volumeMounts:

- name: shared-vol

mountPath: /etc/a

resources: {}

- args:

- /bin/sh

- -c

- while true; do sleep 60; done;

image: alpine:3.12.0

name: container2

volumeMounts:

- name: shared-vol

mountPath: /etc/b

resources: {}

dnsPolicy: ClusterFirst

restartPolicy: Always

status: {}

Utwórz Poda i sprawdź, czy został poprawnie utworzony. Powinieneś zobaczyć Pod w stanie Uruchomiony z dwoma gotowymi kontenerami:

$ kubectl create -f multi-container-alpine.yaml

pod/alpine created

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

alpine 2/2 Running 0 18s

3. Użyj polecenia exec, aby wykonać powłokę w kontenerze o nazwie kontener1. Utwórz plik /etc/a/data/hello.txt z odpowiednią treścią:

$ kubectl exec alpine -c container1 -it -- /bin/sh

/ # cd /etc/a

/etc/a # ls -l

total 0

/etc/a # mkdir data

/etc/a # cd data/

/etc/a/data # echo "Hello World" > hello.txt

/etc/a/data # cat hello.txt

Hello World

/etc/a/data # exit

4. Użyj polecenia exec, aby wykonać powłokę w kontenerze o nazwie kontener2. Zawartość pliku /etc/b/data/hello.txt powinna brzmieć „Hello World”:

$ kubectl exec alpine -c container2 -it -- /bin/sh

/ # cat /etc/b/data/hello.txt

Hello World

/ # exit

5. Zacznij od utworzenia nowego pliku o nazwie logs-pv.yaml. Zawartość może wyglądać następująco:

kind: PersistentVolume

apiVersion: v1

metadata:

name: logs-pv

spec:

capacity:

storage: 5Gi

accessModes:

- ReadWriteOnce

- ReadOnlyMany

hostPath:

path: /var/logs

Utwórz obiekt PersistentVolume i sprawdź jego status:

$ kubectl create -f logs-pv.yaml

persistentvolume/logs-pv created

$ kubectl get pv

NAME CAPACITY ACCESS MODES RECLAIM POLICY

STATUS CLAIM \

STORAGECLASS REASON AGE

logs-pv 5Gi RWO,ROX Retain

Available \

18s

6. Utwórz plik logs-pvc.yaml, aby zdefiniować PersistentVolumeClaim. Poniższy manifest YAML pokazuje jego zawartość:

```
kind: PersistentVolumeClaim

apiVersion: v1

metadata:

name: logs-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 2Gi
```

Utwórz obiekt PersistentVolume i sprawdź jego status:

```
$ kubectl create -f logs-pvc.yaml

persistentvolumeclaim/logs-pvc created

$ kubectl get pvc

NAME STATUS VOLUME

CAPACITY \

ACCESS MODES STORAGECLASS AGE

logs-pvc Bound pvc-47ac2593-2cd2-4213-9e31-

450bc98bb43f 2Gi \

RWO standard 11s
```

7. Utwórz podstawowy manifest YAML, korzystając z opcji wiersza poleceń --dry-run:

```
$ kubectl run nginx --image=nginx --dry-run=client --

restart=Never \

-o yaml > nginx-pod.yaml
```

Teraz edytuj plik nginx-pod.yaml i powiąż z nim PersistentVolumeClaim:

```
apiVersion: v1

kind: Pod

metadata:

creationTimestamp: null
```

labels:

run: nginx

name: nginx

spec:

volumes:

- name: logs-volume

persistentVolumeClaim:

claimName: logs-pvc

containers:

- image: nginx

name: nginx

volumeMounts:

- mountPath: "/var/log/nginx"

name: logs-volume

resources: {}

dnsPolicy: ClusterFirst

restartPolicy: Never

status: {}

Utwórz Poda za pomocą następującego polecenia i sprawdź jego status:

$ kubectl create -f nginx-pod.yaml

pod/nginx created

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

nginx 1/1 Running 0 8s

8. Użyj polecenia exec, aby otworzyć interaktywną powłokę dla Poda i utwórz plik w zamontowanym katalogu:

$ kubectl exec nginx -it -- /bin/sh

cd /var/log/nginx

touch my-nginx.log

ls

access.log error.log my-nginx.log

exit

9. Po ponownym utworzeniu Poda plik przechowywany na PersistentVolume powinien nadal istnieć:

$ kubectl delete pod nginx

$ kubectl create -f nginx-pod.yaml

pod/nginx created

$ kubectl exec nginx -it -- /bin/sh

cd /var/log/nginx

ls

access.log error.log my-nginx.log

exit