Zarządzanie Systemami Rozproszonymi
Laboratoria z Kubernetes – Kontynuacja

mgr inż. Jakub Woźniak

1 Wprowadzenie

Celem tych zajęć jest rozwinięcie wiedzy na temat Kubernetes, w szczególności mechanizmów monitorowania zdrowia aplikacji (liveness i readiness probes), zarządzania aplikacjami stanowymi (StatefulSet) oraz routingu HTTP/HTTPS za pomocą Ingress. Zadania będą stopniowo wprowadzać studentów w bardziej zaawansowane scenariusze, zakończone samodzielnym projektem.

2 Readiness i liveness probes

Kubernetes umożliwia monitorowanie zdrowia aplikacji za pomocą dwóch rodzajów mechanizmów sprawdzających: **livenessProbe** i **readinessProbe**.

- **livenessProbe** – Sprawdza, czy aplikacja jest "żywa" i działa poprawnie. Jeśli livenessProbe wykryje problem, Kubernetes automatycznie restartuje Pod.
- **readinessProbe** – Sprawdza, czy aplikacja jest gotowa do obsługiwania ruchu. W przypadku wykrycia problemu, ruch do tego Poda zostanie tymczasowo wstrzymany, ale Pod nie zostanie zrestartowany.

LivenessProbe jest stosowany do monitorowania stanu aplikacji, gdy niezbędne jest jej ponowne uruchomienie w razie awarii. ReadinessProbe używamy wtedy, gdy aplikacja może potrzebować więcej czasu na inicjalizację lub tymczasowo przestać być dostępna (np. w trakcie przetwarzania danych).

2.1 Zadanie: Konfiguracja livenessProbe dla Deploymentu z NGINX

Cel: Nauczyć się konfigurowania livenessProbe oraz zobaczyć, jak Kubernetes automatycznie zastępuje niezdrowe Pody.
Przykład YAML dla Deployment z livenessProbe:

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        livenessProbe:
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 10
          periodSeconds: 5
```

2.2 Symulacja awarii

Wprowadź awarię, usuwając plik `index.html`:

```
kubectl exec -it <pod-name> -- rm /usr/share/nginx/html/index.html
```

Obserwuj stan Podów:

```
kubectl get pods
kubectl describe pod <pod-name>
```

3 StatefulSet – Aplikacje stanowe

StatefulSet w Kubernetes różni się od Deployment tym, że zapewnia stabilne identyfikatory sieciowe oraz trwałe miejsca przechowywania danych dla Podów. Każdy Pod utworzony przez StatefulSet ma unikalną nazwę i może być połączony z PersistentVolume, aby przechowywać dane.

3.1 Porównanie StatefulSet i Deployment

- **Deployment** jest stosowany do aplikacji bezstanowych, gdzie wszystkie instancje są identyczne i można je łatwo skalować w górę lub w dół. Nie ma jednak gwarancji, że po ponownym uruchomieniu Pody zachowają swoje tożsamości.

- **StatefulSet** zapewnia stabilność identyfikatorów sieciowych (nazw Podów) oraz umożliwia przypisywanie unikalnych nazw Podom (np. `mysql-0`, `mysql-1`). Każdy Pod jest również skojarzony z PersistentVolume, co pozwala na trwałe przechowywanie danych.

StatefulSet jest szczególnie przydatny w przypadkach, gdy aplikacja wymaga:

- Zachowania porządku przy tworzeniu i usuwaniu Podów (kolejność).
- Stabilnych identyfikatorów sieciowych, które umożliwiają komunikację pomiędzy replikami.

3.2 Dlaczego StatefulSet jest lepszy dla MySQL?

W przypadku bazy danych MySQL, trwałość danych i stabilność identyfikatorów sieciowych są kluczowe. StatefulSet współpracuje z PersistentVolumeClaim (PVC), aby zapewnić przechowywanie danych pomiędzy restartami. Przewidywalność identyfikatorów jest kluczowa w sytuacji, gdzie poszczególne instancje aplikacji różnią się rolami, np. model master-slave.

3.2.1 Przykład YAML z użyciem `volumeClaimTemplates`

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: "mysql"
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:5.7
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: example
        volumeMounts:
        - name: mysql-data
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
```

```
- metadata:
    name: mysql-data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
```

3.2.2 Obserwacja

Aby zobaczyć, jak StatefulSet utrzymuje stabilność identyfikatorów i danych:

- Uruchom StatefulSet i sprawdź, jakie nazwy mają Pody (`kubectl get pods`).
- Zanotuj nazwę Poda, np. `mysql-0`, a następnie usuń ten Pod (`kubectl delete pod mysql-0`).
- Obserwuj, jak Kubernetes automatycznie odtwarza Pod o tej samej nazwie (`mysql-0`) i przypisuje mu ten sam PersistentVolume.
- Zwróć uwagę, że dane w bazie są nadal dostępne, co jest efektem użycia PVC, które zapewnia trwałość danych pomiędzy restartami.

4 Ingress – Zarządzanie ruchem HTTP/HTTPS

Ingress w Kubernetes pozwala na zarządzanie ruchem HTTP/HTTPS do aplikacji w klastrze. Jest to mechanizm, który umożliwia wystawienie usług na zewnątrz klastra, definiując reguły routingu oraz terminację SSL. Ingress działa jako punkt wejścia do klastra, przekierowując ruch z zewnętrznego świata do odpowiednich usług w klastrze.

4.1 Dlaczego używamy Ingress?

W Kubernetes mamy kilka sposobów na eksponowanie usług:

- **ClusterIP** – Domyślny typ Service, który udostępnia usługę tylko wewnątrz klastra.
- **NodePort** – Pozwala na dostęp do usługi z zewnątrz poprzez stały port na każdym węźle klastra.
- **LoadBalancer** – Używany w środowiskach chmurowych do integracji z load balancerami dostawcy chmury.
- **Ingress** – Umożliwia definiowanie bardziej zaawansowanych zasad routingu, takich jak kierowanie ruchu na podstawie nazw domen lub ścieżek URL.

Ingress różni się od Service tym, że pozwala na centralne zarządzanie ruchem HTTP/HTTPS w klastrze oraz oferuje terminację SSL. Ingress nie jest wbudowanym zasobem obsługującym ruch w Kubernetes – wymaga wdrożenia tzw. Ingress Controller.

4.2 Jak działa Ingress?

Ingress Controller to aplikacja działająca w klastrze Kubernetes, która interpretuje zasoby Ingress i przekierowuje ruch zgodnie z ich konfiguracją. Popularnym kontrolerem jest NGINX Ingress Controller, ale dostępne są też inne, takie jak Traefik, HAProxy, czy Contour.
W rzeczywistości Ingress Controller składa się z zestawu Podów (np. NGINX), które nasłuchują na porcie 80/443 i dynamicznie konfigurują serwer NGINX na podstawie zasobów Ingress w klastrze.

4.3 Przygotowanie Minikube i Ingress Controller

Aby używać Ingress w Minikube, musimy włączyć dodatek Ingress:

```
minikube addons enable ingress
```

Dodatek ten instaluje NGINX Ingress Controller w klastrze, który obsługuje ruch zgodnie z konfiguracją Ingress.

4.4 Przygotowanie aplikacji NGINX i echoserver

W pierwszej kolejności należy stworzyć Deploymenty oraz Services dla aplikacji, które będą obsługiwane przez Ingress.

4.4.1 Deployment i Service dla NGINX

Najpierw skonfiguruj Deployment dla serwera NGINX:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
```

```
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

Następnie utwórz Service typu ClusterIP, który udostępnia aplikację NGINX w klastrze:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: ClusterIP
```

4.4.2 Deployment i Service dla echoserver

Podobnie jak dla NGINX, skonfiguruj Deployment dla echoserver:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: echoserver-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: echoserver
  template:
    metadata:
      labels:
        app: echoserver
    spec:
      containers:
      - name: echoserver
        image: k8s.gcr.io/echoserver:1.10
        ports:
        - containerPort: 8080
```

Stwórz również Service typu ClusterIP dla echoserver:

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: echoserver-service
spec:
  selector:
    app: echoserver
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 8080
  type: ClusterIP
```

4.5 Konfiguracja Ingress

Po przygotowaniu Deploymentów i Services, możemy przystąpić do skonfigurowania Ingress, aby obsługiwał ruch HTTP do obu aplikacji.
Przykład YAML dla Ingress:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
  - host: example.local
    http:
      paths:
      - path: /nginx
        pathType: Prefix
        backend:
          service:
            name: nginx-service
            port:
              number: 80
      - path: /echo
        pathType: Prefix
        backend:
          service:
            name: echoserver-service
            port:
              number: 8080
```

4.6 Testowanie konfiguracji

Aby przetestować konfigurację, dodaj wpis w pliku `/etc/hosts`, aby przekierować domenę `example.local` na adres IP Minikube:

```
sudo echo "$(minikube ip) example.local" | sudo tee -a /etc/hosts
```

Teraz możesz przetestować dostęp do obu aplikacji, wchodząc na adresy:

- `http://example.local/nginx`
- `http://example.local/echo`

5 Zadanie główne: Ingress dla Redmine i Ghost

Ostatnie zadanie polega na skonfigurowaniu Ingress, który obsłuży dwie aplikacje – Ghost i Redmine – dostępne pod różnymi domenami.
Cel: Samodzielna konfiguracja Ingress oraz użycie zdobytej wiedzy w praktycznym scenariuszu.
Kroki:

- Skonfiguruj Ingress dla aplikacji Ghost (`ghost.local`) oraz Redmine (`redmine.local`).
- Dodaj odpowiednie wpisy w `/etc/hosts`.
- Zweryfikuj, czy aplikacje działają poprawnie, a ruch jest kierowany zgodnie z konfiguracją.