# Zarządzanie Systemami Rozproszonymi
# Laboratoria z Kubernetes

mgr inż. Jakub Woźniak

## 1 Wprowadzenie

Kubernetes to platforma do automatyzacji zarządzania kontenerami. W ramach tych laboratoriów studenci będą mieli okazję pracować z Minikube – lokalną instancją Kubernetes – aby poznać podstawowe zasoby systemu, takie jak Pody, Deploymenty, PersistentVolume, ConfigMap i Secret.

### 1.1 Instalacja Minikube

Minikube to narzędzie, które umożliwia uruchomienie klastra Kubernetes lokalnie na komputerze. Aby zainstalować Minikube na openSUSE, użyj poniższych komend:

```
sudo zypper install -y virtualbox
curl -LO
   https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
minikube start --driver=virtualbox
```

Oprogramowanie `kubectl` jest wbudowane w pakiet `minikube`, dlatego polecany jest alias w `bash`, który można dodać do `.bashrc` lub podobnego.

```
alias kubectl="minikube␣kubectl␣--"
```

Po instalacji warto sprawdzić, czy klaster Kubernetes został poprawnie uruchomiony, za pomocą:

```
kubectl get nodes
```

Komenda ta wyświetla listę węzłów w klastrze. Jeśli wszystko działa poprawnie, powinien być widoczny węzeł `Ready`.

## 2 Podstawy Kubernetes

### 2.1 Pod – Co to jest i dlaczego jest ważny?

Pod to najmniejsza jednostka zarządzania w Kubernetes, która uruchamia kontenery. Pody grupują jeden lub więcej kontenerów, współdzieląc te same zasoby,

takie jak sieć i woluminy. Każdy Pod działa z własnym adresem IP, a Kubernetes zarządza jego cyklem życia.
Przykładowy YAML do uruchomienia poda Nginx:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

**Wyjaśnienie elementów YAML:**

- `apiVersion: v1` – Wersja API Kubernetes używana do definiowania zasobu. Wersja v1 to stabilna wersja API dla podstawowych zasobów.
- `kind: Pod` – Określa rodzaj zasobu. W tym przypadku jest to Pod, który uruchamia jeden lub więcej kontenerów.
- `metadata:` – Sekcja, która zawiera metadane, takie jak nazwa zasobu. W tym przypadku Pod nosi nazwę `nginx`.
- `spec:` – Specyfikacja opisuje, jakie kontenery zostaną uruchomione w Podzie. Zawiera listę kontenerów, w tym ich obraz (`nginx:latest`) oraz porty (`80`).

**Kroki wdrożenia i sprawdzenia:**

- Uruchom pod:

  ```
  kubectl apply -f nginx-pod.yaml
  ```

- Sprawdź status poda:

  ```
  kubectl get pods
  ```

  Powinieneś zobaczyć pod o nazwie `nginx` w statusie `Running`.

- Aby zobaczyć logi kontenera:

  ```
  kubectl logs nginx
  ```

## 2.2 Deployment – Dlaczego go potrzebujemy?

Deployment to zasób Kubernetes, który automatycznie zarządza Podami, zapewniając, że określona liczba replik (kopii) aplikacji jest zawsze uruchomiona. Deployment umożliwia również bezpieczne wdrażanie nowych wersji aplikacji poprzez tzw. rolling update. Kubernetes monitoruje stan Podów i automatycznie odtwarza je w przypadku awarii.
Przykład YAML dla Deploymentu Nginx:

```
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
        ports:
        - containerPort: 80
```

**Wyjaśnienie elementów YAML:**

- `replicas: 3` – Definiuje liczbę replik podów, które Kubernetes ma utrzymywać.
- `selector: matchLabels:` – Wybiera Pody na podstawie etykiety (`app: nginx`).
- `template:` – Szablon dla nowych Podów tworzonych przez ten Deployment. Zawiera specyfikację, która definiuje kontenery i ich ustawienia.

**Kroki wdrożenia i sprawdzenia:**

- Uruchom Deployment:

  ```
  kubectl apply -f nginx-deployment.yaml
  ```

- Sprawdź status Deploymentu:

  ```
  kubectl get deployments
  kubectl get pods
  ```

Powinieneś zobaczyć trzy pody `nginx` uruchomione i gotowe.

- Usuń pod, aby zobaczyć, jak Kubernetes automatycznie go odtworzy:

  ```
  kubectl delete pod <nazwa-poda>
  ```

## 2.3 Service – Umożliwienie dostępu do aplikacji

Service to zasób Kubernetes, który umożliwia komunikację między Podami oraz udostępnienie aplikacji na zewnątrz klastra. Service zapewnia stabilny adres IP i port, nawet jeśli Pody są usuwane i tworzone na nowo.
Przykład YAML dla Service typu NodePort:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30007
```

**Wyjaśnienie elementów YAML:**

- `type: NodePort` – Umożliwia dostęp do aplikacji spoza klastra na określonym porcie (`30007`).
- `selector:` – Łączy Service z Podami oznaczonymi etykietą `app: nginx`.
- `ports:` – Definiuje porty, przez które aplikacja będzie dostępna (port 80 w Podzie, a port 30007 na nodzie).

**Kroki wdrożenia i sprawdzenia:**

- Uruchom Service:

  ```
  kubectl apply -f nginx-service.yaml
  ```

- Sprawdź, na jakim porcie usługa jest dostępna:

  ```
  kubectl get svc
  ```

- Aby uzyskać dostęp do usługi, użyj Minikube do wygenerowania URL:

  ```
  minikube service nginx-service --url
  ```

# 3 PersistentVolume i PersistentVolumeClaim – Trwałe przechowywanie danych

PersistentVolume (PV) to zasób klastra, który reprezentuje fizyczną przestrzeń dyskową do trwałego przechowywania danych. PersistentVolumeClaim (PVC) to żądanie aplikacji do korzystania z zasobu PV. Dzięki PV i PVC aplikacje mogą przechowywać dane, które przetrwają ponowne uruchomienie Podów.
Przykład YAML dla PersistentVolume (PV):

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Przykład YAML dla PersistentVolumeClaim (PVC):

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

Następnie tak przygotowany wolumen możemy zastosować przy wdrożeniu bazy danych typu MySQL (lub dowolnej innej), gdzie trwałość danych jest kluczowa. Tak przygotowany Pod przypomina bezstanowy, gdyż stan (dane) są przechowywane w osobnym typie zasobu. Oczywiście, bazy danych (szczególnie w przypadku więcej niż jednego węzła) są dużo trudniejszym zagadnienie.
Oto przykład definicji Deploymentu dla bazy danych MySQL

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  replicas: 1
  selector:
```

```
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:5.7
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: "password"
        volumeMounts:
        - mountPath: "/var/lib/mysql"
          name: mysql-storage
      volumes:
      - name: mysql-storage
        persistentVolumeClaim:
          claimName: mysql-pvc
```

W powyższej definicji Deploymentu aplikacja MySQL korzysta z PersistentVolumeClaim o nazwie `mysql-pvc`, który został zamontowany w ścieżce `/var/lib/mysql`, gdzie baza danych przechowuje swoje dane. Dzięki temu dane nie zostaną utracone po restarcie Poda, gdyż są przechowywane na trwałym wolumenie.

**Kroki wdrożenia i sprawdzenia:**

- Uruchom PersistentVolume i PersistentVolumeClaim:

  ```
  kubectl apply -f pv.yaml
  kubectl apply -f pvc.yaml
  kubectl apply -f mysql.yaml
  ```

- Sprawdź status PV i PVC:

  ```
  kubectl get pv
  kubectl get pvc
  kubectl get deployment
  ```

  Pomyślne przyznanie zasobów oznacza, że PVC zostało poprawnie podłączone do PV, a deployment MySQL uruchomił się bez problemów.

## 3.1 Różnice między PersistentVolume i PersistentVolumeClaim

- **PersistentVolume (PV)**: Jest to zasób przydzielony na poziomie klastra, który zapewnia fizyczną przestrzeń dyskową. PV mogą być tworzone

ręcznie przez administratora lub dynamicznie przez system na podstawie żądań użytkowników.

- **PersistentVolumeClaim (PVC)**: Jest to żądanie na poziomie aplikacji, które określa wymagania dotyczące przechowywania, takie jak pojemność i tryb dostępu. PVC jest automatycznie wiązane z dostępnym PV, który spełnia zadane kryteria.

## 3.2 Typowe tryby dostępu w PersistentVolume

- **ReadWriteOnce (RWO)**: Wolumen może być zamontowany tylko na jednym węźle i być używany do odczytu i zapisu.
- **ReadOnlyMany (ROX)**: Wolumen może być zamontowany na wielu węzłach, ale tylko do odczytu.
- **ReadWriteMany (RWX)**: Wolumen może być zamontowany na wielu węzłach i używany do odczytu oraz zapisu.

## 3.3 Dynamiczne przydzielanie zasobów

W Kubernetes istnieje możliwość dynamicznego przydzielania wolumenów dzięki wykorzystaniu klas przechowywania (StorageClass). Definiując StorageClass, możemy określić, jakiego rodzaju wolumeny mają być tworzone i jakie mają mieć parametry, takie jak typ dysku, szybkość odczytu i zapisu itp. Dzięki temu możemy automatyzować proces tworzenia PV na podstawie żądań PVC. Przykład definicji StorageClass:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
```

Podczas tworzenia PersistentVolumeClaim można wskazać nazwę StorageClass, co spowoduje dynamiczne utworzenie PV spełniającego zadane kryteria.

## 3.4 Podsumowanie

PersistentVolume i PersistentVolumeClaim są kluczowymi elementami zapewniającymi trwałość danych w klastrach Kubernetes. Dzięki nim aplikacje mogą niezawodnie przechowywać swoje dane, niezależnie od cyklu życia Podów, co jest szczególnie istotne w przypadku aplikacji stanowych, takich jak bazy danych. Poprawne zarządzanie zasobami przechowywania oraz zrozumienie mechanizmów ich przydzielania jest niezbędne do skutecznej pracy z Kubernetes.

# 4 ConfigMap i Secret – Przechowywanie konfiguracji i wrażliwych danych

## 4.1 ConfigMap

W tym zadaniu studenci będą korzystać z ConfigMap, aby zmienić stronę główną serwera NGINX. Skonfigurujemy Deployment, aby NGINX korzystał z pliku `index.html` przechowywanego w ConfigMap, oraz udostępnimy aplikację przy użyciu Service typu NodePort.

**ConfigMap YAML:**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  index.html: |
    <html>
    <head><title>NGINX</title></head>
    <body><h1>Witamy na serwerze NGINX</h1></body>
    </html>
```

**Deployment YAML:**

```
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:latest
        volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
      volumes:
      - name: html
        configMap:
```

```
      name: nginx-config
```

**Wyjaśnienie nowych elementów YAML:**

- `volumeMounts:` – Określa, gdzie w kontenerze ma być zamontowany wolumin z danymi. W tym przypadku plik `index.html` jest zamontowany w lokalizacji `/usr/share/nginx/html`, czyli w miejscu, gdzie NGINX przechowuje pliki stron.
- `volumes: configMap:` – Wolumin ConfigMap, który umożliwia montowanie danych z ConfigMap w kontenerze. W tym przypadku wolumin montuje `nginx-config` i używa go do zaktualizowania strony głównej NGINX.

**Service YAML:**

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30007
```

**Wyjaśnienie nowych elementów YAML:**

- `type: NodePort` – Umożliwia dostęp do aplikacji spoza klastra poprzez przypisanie portu (w tym przypadku `30007`) na wszystkich węzłach klastra.
- `selector:` – Wybiera pody na podstawie etykiety (`app: nginx`), co pozwala na kierowanie ruchu do odpowiednich instancji NGINX uruchomionych przez Deployment.

**Kroki wdrożenia i sprawdzenia:**

- Uruchom ConfigMap:

  ```
  kubectl apply -f nginx-config.yaml
  ```

- Uruchom Deployment:

  ```
  kubectl apply -f nginx-deployment.yaml
  ```

- Uruchom Service:

```
kubectl apply -f nginx-service.yaml
```

- Sprawdź, na jakim porcie usługa jest dostępna:

  ```
  kubectl get svc
  ```

- Uzyskaj dostęp do NGINX, używając Minikube do wygenerowania URL:

  ```
  minikube service nginx-service --url
  ```

  Otwórz stronę w przeglądarce i sprawdź, czy wyświetla się zmodyfikowana strona główna z ConfigMap.

## 4.2 Secret

Secret służy do bezpiecznego przechowywania wrażliwych informacji, takich jak hasła i klucze API. W tym zadaniu użyjemy Secret, aby przekazać klucz API do aplikacji, która będzie pobierać dane z serwisu `https://jsonplaceholder.typicode.com`.

**Secret YAML:**

```
apiVersion: v1
kind: Secret
metadata:
  name: api-secret
data:
  api-key: c2VjcmV0a2V5
```

**Wyjaśnienie nowych elementów YAML:**

- `kind: Secret` – Typ zasobu, który umożliwia bezpieczne przechowywanie wrażliwych informacji, takich jak hasła czy klucze API. Dane w `Secret` są zakodowane w formacie base64.
- `data: api-key` – Zawiera klucz API zakodowany w formacie base64. W tym przykładzie rzeczywista wartość klucza to `secretkey`.

Następnie aplikacja będzie korzystać z tego klucza API, aby pobierać dane JSON z `https://jsonplaceholder.typicode.com`.

**Pod YAML:**

```
apiVersion: v1
kind: Pod
metadata:
  name: json-app
spec:
  containers:
  - name: json-app
    image: curlimages/curl:8.10.1
```

```
env:
- name: API_KEY
  valueFrom:
    secretKeyRef:
      name: api-secret
      key: api-key
command: ["sh", "-c", "curl␣-d␣\"api-key=$API_KEY\"␣
   https://jsonplaceholder.typicode.com/posts"]
```

**Wyjaśnienie nowych elementów YAML:**

- `env:` – Używane do definiowania zmiennych środowiskowych w kontenerze. W tym przypadku klucz API jest przekazywany jako zmienna `API_KEY`.
- `valueFrom: secretKeyRef:` – Mechanizm pobierania wartości z `Secret`. W tym przypadku używamy klucza o nazwie `api-key` z zasobu `api-secret`.
- `command:` – Używane do uruchamiania polecenia `curl`, które pobiera dane JSON z serwisu `https://jsonplaceholder.typicode.com`, używając klucza API jako nagłówka autoryzacyjnego.

**Kroki wdrożenia i sprawdzenia:**

- Uruchom Secret:

  ```
  kubectl apply -f api-secret.yaml
  ```

- Uruchom Pod z aplikacją:

  ```
  kubectl apply -f json-app.yaml
  ```

- Sprawdź logi Poda, aby zobaczyć dane pobrane z `https://jsonplaceholder.typicode.com/posts`:

  ```
  kubectl logs json-app
  ```

# 5 Zadanie główne: Wdrożenie Ghosta z MySQL

## 5.1 Opis zadania

Celem zadania jest samodzielne wdrożenie aplikacji Ghost z bazą danych MySQL na Kubernetes, wykorzystując wiedzę zdobywaną podczas wcześniejszych ćwiczeń.

### 5.2 Wymagania

- Aplikacja Ghost musi być wdrożona z użyciem Deploymentu.
- Dane MySQL muszą być przechowywane w PersistentVolume, aby przetrwały ponowne uruchomienie Podów.
- Hasło do bazy MySQL musi być bezpiecznie przechowywane w zasobie Secret.
- Aplikacja Ghost musi być dostępna na zewnątrz klastra przez Service typu NodePort lub LoadBalancer.

### 5.3 Kryteria zaliczenia

- Aplikacja Ghost działa i jest dostępna przez przeglądarkę.
- Pierwsza konfiguracja Ghost jest dostępna pod ścieżką `/ghost/setup`.
- Dane w bazie MySQL są trwałe i przetrwały ponowne uruchomienie Poda.
- Deploymenty Ghost i MySQL automatycznie odtwarzają Pody w przypadku awarii.
- Hasło do bazy danych jest poprawnie przechowywane w zasobie Secret.
- Zastosowany PersistentVolume umożliwia trwałe przechowywanie danych MySQL.

### 5.4 Linki do dokumentacji

- Ghost: `https://ghost.org/docs/`
- MySQL: `https://dev.mysql.com/doc/`
- Kubernetes Secrets: `https://kubernetes.io/docs/concepts/configuration/secret/`

## 6 Horizontal Pod Autoscaler (HPA)

Na koniec, skonfigurujemy Horizontal Pod Autoscaler (HPA), który automatycznie skalibruje liczbę replik aplikacji Ghost w zależności od obciążenia CPU. Przykład YAML dla HPA:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: ghost-hpa
spec:
  scaleTargetRef:
```

```
    apiVersion: apps/v1
    kind: Deployment
    name: ghost
  minReplicas: 1
  maxReplicas: 5
  targetCPUUtilizationPercentage: 50
```

Aby przetestować HPA, można użyć narzędzia `Apache Benchmark` (ab), które wygeneruje ruch na aplikację Ghost:

```
ab -n 1000 -c 50 http://<ghost-service-url>/
```

**Weryfikacja działania:**

- Sprawdź, czy liczba replik aplikacji wzrasta:

  ```
  kubectl get hpa
  kubectl get pods
  ```