Part 21: Kubernetes Real-Time Troubleshooting

Introduction 🌐

Welcome to the world of Kubernetes troubleshooting, where every challenge is an opportunity to sharpen your skills and emerge victorious. Join us as we embark on a journey through common real-time scenarios, unraveling mysteries, and uncovering solutions along the way.

PART 21 - KUBERNETES REAL-TIME TROUBLESHOOTING

- Creating a Namespace and Resource-Limited Pod
- Managing Kubernetes Contexts
- Deploying a Pod on a Specific Node
- Scaling StatefulSets
- Implementing Readiness Probes

Scenario 101: Creating a Namespace and Resource-Limited Pod in Kubernetes
Scenario: You need to create a new namespace called limit and deploy a pod named resource-checker with specific CPU and memory requests and limits.
Solution:

1. **Create the Namespace:**
   `kubectl create ns limit`

- **Define the Pod:**

```
apiVersion: v1
kind: Pod
metadata:
  namespace: limit
  labels:
    run: resource-checker
  name: resource-checker
```

```
spec:
  containers:
  - image: httpd:alpine
    name: my-container
    resources:
      requests:
        memory: "30Mi"
        cpu: "30m"
      limits:
        memory: "30Mi"
        cpu: "300m"
  dnsPolicy: ClusterFirst
  restartPolicy: Always
```

- **Deploy the Pod:**
  ```
  `kubectl apply -f <pod-definition-file>.yaml`
  ```

Outcome: This setup ensures that the resource-checker pod has controlled resource usage, preventing it from consuming excessive CPU or memory.

Scenario 102: Managing Kubernetes Contexts and Current Context Retrieval
Scenario: You need to list all Kubernetes contexts and retrieve the current context using both kubectl commands and shell scripts.
Solution:

1. **List All Contexts:**
   ```
   `kubectl config get-contexts`
   ```

- **Save Contexts to a File:**
  ```
  `kubectl config get-contexts -o name > /root/filesystem/tmp`
  ```

- **Script to Get Current Context Using kubectl:**
  ```
  `echo "kubectl config current-context" > current_context_using_kubectl.sh`
  ```

- **Script to Get Current Context Without kubectl:**
  ```
  `echo "cat ~/.kube/config | grep current-context | sed 's/current-context: //'" > current_context_without_kubectl.sh`
  ```

Outcome: These scripts provide flexibility in retrieving the current Kubernetes context, useful for automation and monitoring.

Scenario 103: Deploying a Pod on a Specific Node with Taints
Scenario: You need to deploy a pod on a specific node and ensure it remains scheduled there by managing taints.
Solution:

1. **Switch Context:**
   ```
   `kubectl config use-context k8s-c1-H`
   ```

- **Create the Pod Definition:**
  ```
  apiVersion: v1
  ```

```
kind: Pod
metadata:
  labels:
    run: pod1
  name: pod1
spec:
  nodeName: controlplane
  containers:
  - image: httpd:2.4.41-alpine
    name: pod1-container
  dnsPolicy: ClusterFirst
  restartPolicy: Always
```

- **Remove Taint from the Node:**
  `kubectl taint nodes controlplane node-role.kubernetes.io/control-plane:NoSchedule-`

- **Deploy the Pod:**
  `kubectl apply -f <pod-definition-file>.yaml`

Outcome: The pod is successfully deployed and scheduled on the specified node, ensuring consistent resource allocation.

Scenario 104: Scaling StatefulSets in Kubernetes
Scenario: You need to create and scale StatefulSets in a Kubernetes namespace to manage stateful applications efficiently.
Solution:

1. **Create the Namespace:**
   `kubectl create ns project`

- **Define the StatefulSet:**

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-statefulset
  namespace: project
spec:
  serviceName: "my-service"
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: nginx:latest
        ports:
```

```
        - containerPort: 80
```

- **Scale Down the StatefulSet:**
  `kubectl scale statefulset my-statefulset -n project --replicas=1`

Outcome: StatefulSets provide stable network identities and persistent storage, crucial for stateful applications.

Scenario 105: Implementing Readiness Probes with Service Dependencies
Scenario: You need to ensure a pod becomes ready only when a dependent service is available.
Solution:

1. **Create the First Pod with Readiness Probe:**

```
apiVersion: v1
kind: Pod
metadata:
  name: ready-if-service-ready
spec:
  containers:
  - name: nginx
    image: nginx:1.16.1-alpine
    readinessProbe:
      exec:
        command:
        - wget
        - -T2
        - -O-
        - http://service-am-i-ready:80
```

- **Create the Second Pod with Labels:**

```
apiVersion: v1
kind: Pod
metadata:
  name: am-i-ready
  labels:
    id: cross-server-ready
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

- **Patch the Service to Select the Second Pod:**
  `kubectl patch service service-am-i-ready --type='json' -p='[{"op": "add", "path": "/spec/selector", "value": {"id": "cross-server-ready"}}]'`

Outcome: The first pod becomes ready only when the dependent service is available, ensuring proper service dependencies.

In the up-coming parts, we will discussion on more troubleshooting steps for the different Kubernetes based scenarios. So, stay tuned for the and follow @Prasad Suman Mohan for more such posts.