Python with Kubernetes - Yatri Cloud

Creator: Yatharth Chauhan

Kubernetes is a powerful open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. When combined with Python, developers can leverage various libraries and tools to interact with Kubernetes clusters, manage resources, and deploy applications programmatically.

1. **Cluster**: A set of machines (nodes) that run containerized applications.
2. **Pod**: The smallest deployable unit in Kubernetes, which can contain one or more containers.
3. **Service**: An abstraction that defines a logical set of pods and a policy to access them.
4. **Deployment**: A resource that manages a set of identical pods, ensuring the desired state.
5. **Namespace**: A way to divide cluster resources between multiple users or applications.

Interacting with Kubernetes Using Python

Python can be used to interact with Kubernetes through the **Kubernetes Python client**. This client allows you to manage Kubernetes resources programmatically, enabling automation and integration into your DevOps workflows.

Setting Up Kubernetes with Python

1. Installing the Kubernetes Python Client

You can install the Kubernetes Python client using pip:

```
pip install kubernetes
```

2. Configuring Access to Your Kubernetes Cluster

To interact with a Kubernetes cluster, you need to configure the client to access the cluster. This can be done by loading the kubeconfig file, typically located at `~/.kube/config`.

```
from kubernetes import client, config

# Load kubeconfig
config.load_kube_config()
```

Example Use Cases with Python and Kubernetes

Example 1: Listing Pods in a Namespace

This example demonstrates how to list all pods in a specific namespace.

```python
from kubernetes import client, config

# Load kubeconfig
config.load_kube_config()

# Create a Kubernetes API client
v1 = client.CoreV1Api()

# Define the namespace to query
namespace = 'default'

# List pods in the specified namespace
pods = v1.list_namespaced_pod(namespace)

print(f"Pods in namespace '{namespace}':")
for pod in pods.items:
    print(f"- {pod.metadata.name} (Status: {pod.status.phase})")
```

Explanation:

- **CoreV1Api**: An API class to interact with core Kubernetes resources like pods.
- **list_namespaced_pod(namespace)**: Retrieves all pods in the specified namespace.

Example 2: Creating a Deployment

This example shows how to create a deployment in Kubernetes using Python.

```python
from kubernetes import client, config

# Load kubeconfig
config.load_kube_config()

# Create a Kubernetes API client
apps_v1 = client.AppsV1Api()

# Define the deployment
deployment = client.V1Deployment(
    api_version="apps/v1",
    kind="Deployment",
    metadata=client.V1ObjectMeta(name="Yatri Cloud-deployment"),
    spec=client.V1DeploymentSpec(
        replicas=3,
        selector={'matchLabels': {'app': 'Yatri Cloud-app'}},
        template=client.V1PodTemplateSpec(
            metadata=client.V1ObjectMeta(labels={'app': 'Yatri Cloud-app'}),
            spec=client.V1PodSpec(containers=[
                client.V1Container(
                    name="Yatri Cloud-container",
                    image="nginx:latest",
                    ports=[client.V1ContainerPort(container_port=80)],
```

```
                )
            ])
        )
    )
)

# Create the deployment
namespace = 'default'
response = apps_v1.create_namespaced_deployment(
    namespace=namespace,
    body=deployment,
)

print(f"Deployment created. Name: {response.metadata.name}")
```

Explanation:

- **V1Deployment**: Represents the deployment object in Kubernetes.
- **spec.replicas**: Specifies the number of pod replicas.
- **template.spec.containers**: Defines the containers to run within the pods.

Example 3: Scaling a Deployment

You can easily scale an existing deployment up or down with the following example.

```
from kubernetes import client, config

# Load kubeconfig
config.load_kube_config()

# Create a Kubernetes API client
apps_v1 = client.AppsV1Api()

# Define the deployment name and namespace
deployment_name = "Yatri Cloud-deployment"
namespace = "default"

# Scale the deployment
scale = client.V1Scale(
    spec=client.V1ScaleSpec(replicas=5)  # Set the desired number of replicas
)

response = apps_v1.patch_namespaced_deployment_scale(
    name=deployment_name,
    namespace=namespace,
    body=scale,
)

print(f"Scaled deployment '{deployment_name}' to {response.spec.replicas}
replicas.")
```

Explanation:

- **V1Scale**: Represents the scaling configuration.
- **patch_namespaced_deployment_scale()**: Updates the number of replicas for the specified deployment.

Example 4: Deleting a Pod

To delete a specific pod, you can use the following code.

```
from kubernetes import client, config

# Load kubeconfig
config.load_kube_config()

# Create a Kubernetes API client
v1 = client.CoreV1Api()

# Define the pod name and namespace
pod_name = "Yatri Cloud-pod"
namespace = "default"

# Delete the pod
v1.delete_namespaced_pod(name=pod_name, namespace=namespace)

print(f"Pod '{pod_name}' deleted from namespace '{namespace}'.")
```

Explanation:

- **delete_namespaced_pod()**: Deletes the specified pod in the given namespace.

Best Practices for Using Python with Kubernetes

1. **Error Handling**: Implement proper error handling to manage exceptions and failures while interacting with the Kubernetes API.
2. **Logging**: Use logging to keep track of actions and errors in your scripts.
3. **Testing**: Test your scripts in a development environment before running them in production.
4. **Use Environment Variables**: Manage sensitive information (like API tokens) using environment variables instead of hardcoding them.

Kubernetes is a powerful tool for orchestrating containerized applications, and using Python to interact with Kubernetes APIs can greatly enhance automation and operational efficiency. The Kubernetes Python client provides a rich set of functionalities to manage resources such as pods, deployments, and services programmatically. By leveraging Python scripts, organizations can automate deployment processes, monitor cluster health, and streamline their DevOps workflows, leading to more efficient and reliable application delivery.