Kafka, .NET MAUI, VS Code, Semantic Kernal

CODE

SEP
OCT
2024

codemag.com - THE LEADING INDEPENDENT DEVELOPER MAGAZINE - US $ 8.95 Can $ 11.95

CODE 30 YEARS

The Art of Orchestrating Kubernetes

Create Custom Scripting Languages

Explore the Outbox Pattern with Kafka

Secrets of VS Code

0 9>
7 25274 56130 8

Container Orchestration Using Kubernetes

In my previous article on Docker ("Introduction to Containerization Using Docker," Mar/Apr 2021 issue), I explained what containerization is and how Docker simplifies application deployment. Docker has revolutionized software development, deployment, and management, especially in the context of microservices. In the realm of microservices, Docker plays a crucial role by providing a consistent and isolated environment for each service. This isolation ensures that services don't interfere with each other, thereby facilitating the development, testing, and deployment of independent application components. Docker's containerization also simplifies scaling microservices, as containers can be easily replicated and orchestrated using tools like Kubernetes.

Wei-Meng Lee
weimenglee@learn2develop.net
http://www.learn2develop.net
@weimenglee

Wei-Meng Lee is a technologist and founder of Developer Learning Solutions (www.learn2develop.net), a technology company specializing in hands-on training on the latest technologies. Wei-Meng has many years of training experiences and his training courses place special emphasis on the learning-by-doing approach. His hands-on approach to learning programming makes understanding the subject much easier than reading books, tutorials, and documentation. His name regularly appears in online and print publications such as DevX.com, MobiForge.com, and CODE Magazine.

For large-scale deployments, Kubernetes is often used to manage and scale containers. Although Kubernetes is incredibly powerful and versatile, it's also known for its steep learning curve. Many developers and system administrators find the complexity of Kubernetes daunting because it's challenging to understand and implement effectively. However, tools like minikube offer a simplified way to get started with Kubernetes. Minikube allows you to run a single-node Kubernetes cluster on your local machine, providing a practical environment to learn and experiment without the overhead of a full-fledged cluster.

In this article, I'll guide you through the process of creating a simple Kubernetes cluster using minikube. I'll cover essential Kubernetes components such as Pods, Deployments, Services, NodePort, and ConfigMaps. By the end of this article, you'll have a functional local Kubernetes cluster and a better understanding of how these core elements work together to deploy and manage applications in Kubernetes.

Introducing Minikube

Before deploying applications on a Kubernetes cluster, it's crucial for developers to become familiar with the system. Rather than starting with a live Kubernetes cluster, it's much easier to test on a local test cluster. This is where minikube comes in.

Minikube is a tool that allows developers to run a single-node Kubernetes cluster locally on their personal computers. It's lightweight, easy to install, and effectively simulates how a Kubernetes cluster operates.

Installing Minikube

To install minikube, go to https://minikube.sigs.k8s.io/docs/start/ and select the OS and architecture of the platform you are installing minikube on (see **Figure 1**).

Follow the instructions on the page to begin your installation. For example, if you're using an Apple M-series

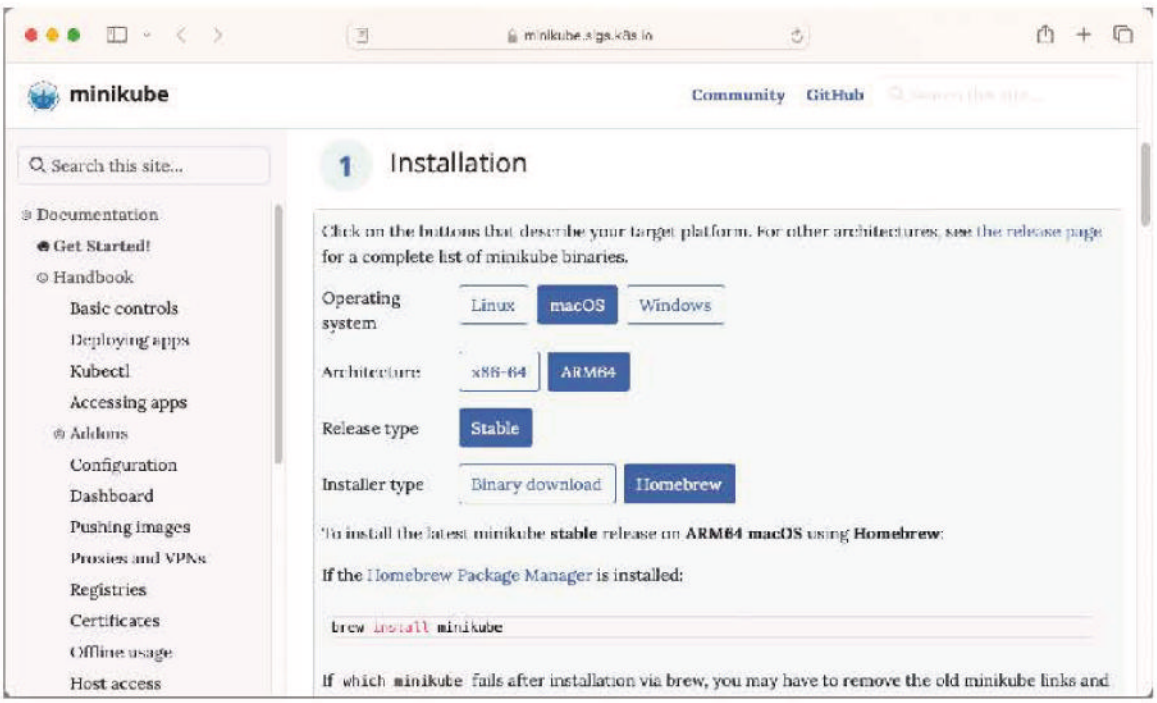

Figure 1: Installing minikube: Choose your OS, architecture, release, and installer type

Mac and have HomeBrew installed, use the following command to install minikube:

```
$ brew install minikube
```

For this article, I'm assuming that you already have Docker installed on your system.

Starting Minikube

Once minikube is installed on your machine, you can simply start it by typing the following command:

```
$ minikube start
```

Figure 2 shows minikube starting up and creating a single-node cluster.

Minikube, by default, uses Docker as the container runtime. You're free to use other container runtimes (e.g., containerd, CRI-O, etc.) with minikube if you wish to. You can check the drivers that minikube is using via the following command:

```
$ minikube profile list
```

Figure 3 shows that minikube is using Docker for both the **VM driver** and the **Runtime**.

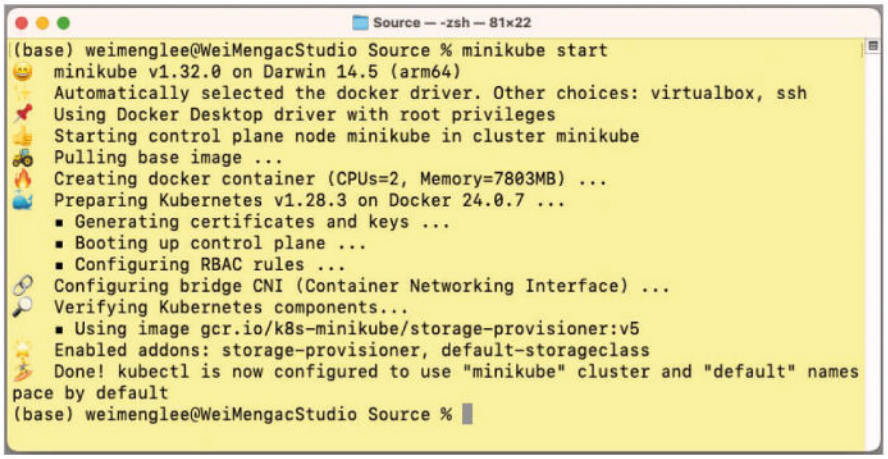

Figure 2: Minikube starting up

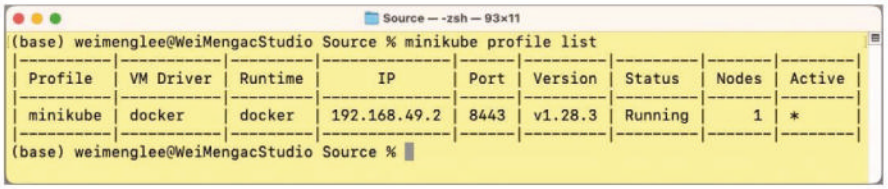

Figure 3: Viewing the drivers that minikube is using

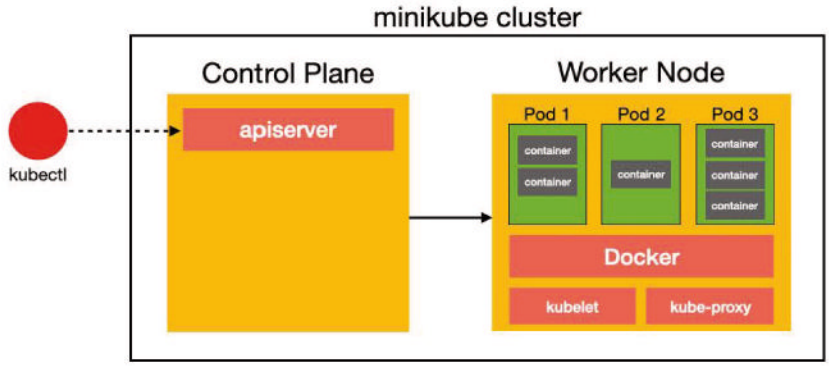

Figure 4: The minikube cluster contains both the control plane and the worker node in a single node (machine).

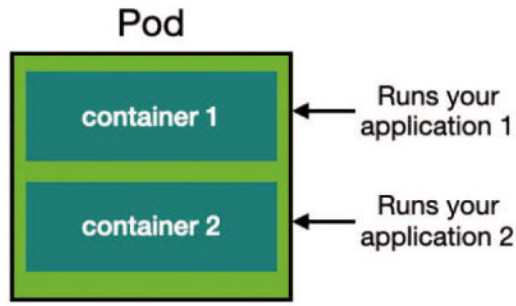

Figure 5: A pod can contain one or more applications.

In this example, using the Docker driver as the VM Driver means that minikube creates Docker containers to manage the various components in your Kubernetes cluster (think of it as minikube using Docker containers to simulate the various nodes in a Kubernetes cluster). Alternatively, you can change the driver to use VirtualBox if you prefer (note that VirtualBox is not supported on Apple's M-series processors):

```
$ minikube start --driver=virtualbox
```

The Runtime, on the other hand, indicates the software minikube uses to manage the containers running inside the Kubernetes cluster. By default, minikube uses Docker as the container runtime. This can be changed. For example, if you want to use **containerd** instead of Docker, you can use the following command:

```
$ minikube start --container-runtime=containerd
```

For this article, I'll be using Docker for both the VM Driver as well as the Runtime for Kubernetes.

To check the status of the minikube cluster, use the following command:

```
$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Here's the breakdown of each of the components in the cluster:

1. **Control Plane:** Manages the Kubernetes cluster. It makes global decisions about the cluster (e.g., scheduling) and detects and responds to cluster events (e.g., starting up new pods when containers fail).
2. **Host:** The container that hosts the Kubernetes cluster.
3. **Kubelet:** The primary node agent that runs on each node. It ensures that containers are running in a pod (more on this later).
4. **Apiserver:** A component of the Kubernetes control plane that exposes the Kubernetes API. It allows clients to connect to the Kubernetes cluster.
5. **Kubeconfig:** A configuration file used by **kubectl** (a client utility) to interact with the cluster. It contains cluster information, user credentials, namespaces, and contexts.

> In real-life, a Kubernetes cluster often has multiple control planes and worker nodes. Each worker node is typically on a separate physical machine or virtual machine. This design allows Kubernetes to distribute workloads across multiple physical or virtual resources, ensuring better resource utilization, fault tolerance, and scalability.

To stop the cluster, use the **stop** option:

```
$ minikube stop
```

To delete the cluster, use the **delete** option:

```
$ minikube delete
```

Using the Various Components in Kubernetes

Now that the minikube cluster is up and running, let's explore the various key components in Kubernetes and learn how they work. You'll progressively build a cluster that contains the following:

1. Pods
2. Deployments
3. Services
4. NodePort
5. ConfigMaps

Pods

A pod is the smallest deployable and manageable unit in Kubernetes. It represents a single instance of a running process in a cluster and can contain one or more containers (usually Docker containers) that share the same network namespace and storage volumes. In short, you run your application within a container hosted in a pod. A pod can contain one or more containers, but it usually contains one.

> A pod represents a single instance of a running process in your cluster, which could be a containerized application, such as Docker container, or multiple tightly coupled containers that need to share resources.

Figure 5 shows a Pod running multiple applications.

Let's now learn how to run an application using a pod. For this example, I'm going to run the **nginx** web server inside a pod. To get Kubernetes to run nginx using a pod, you need to create a YAML configuration file. Let's create a file named **nginxpod.yaml** and populate it with the following statements:

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

Here's what the configuration is doing:

- **apiVersion: v1:** Specifies the API version used to create the pod. For basic pod creation, v1 is commonly used.
- **kind: Pod:** Indicates the type of Kubernetes object being created. In this case, it's a pod.
- **metadata:** Information about information.
  - **name: nginx:** Defines the name of the pod. This name must be unique within the namespace.
  - **labels: name: nginx:** Labels are key-value pairs that are attached to Kubernetes objects. They can be used to organize and select subsets of objects. Here, a label **name: nginx** is added.
- **spec:** Specifies the desired state of the pod.
  - **containers:** A list of containers that will run inside the pod. In this example, there is one container.
  - **name: nginx:** The name of the container within the pod.
  - **image: nginx:** The Docker image (**nginx**) to use for the container. By default, it pulls from Docker Hub.
  - **ports: containerPort: 80:** Exposes port 80 of the container to the network. This is typically where the nginx server listens for HTTP traffic.
  - **imagePullPolicy: Always:** Ensures that Kubernetes always pulls the latest version of the image. This can be useful for development and testing to ensure that the latest image changes are always used.

To apply this configuration to the cluster, use the **kubectl** command:

```
$ kubectl apply -f nginxpod.yaml
pod/nginx created
```

Your pod should now be created. You can verify this by typing the following command:

```
$ kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          27s
```

To verify that the nginx web server is functioning correctly, you can execute a command in a container running within a pod by using the following command:

```
$ kubectl exec -it nginx -- bash
```

The **exec** command is to execute a command in a container and the **-it** option indicates that you want to run the command in interactive (**-i**) mode with a TTY (terminal, **-t**). The **--** indicates that the following argument (which is **bash**) should be passed to the command executed in the container (and not to **kubectl**).

You should now see the following:

```
root@nginx:/#
```

You are now inside the container running the nginx web server (because the pod only contains one container). To test that the web server works, use the **curl** command as shown in **Listing 1**. If you're able to see the output as shown, your pod running the nginx web server is now working.

Besides going into the nginx container to test the web server, you can also use port forwarding to forward one

Listing 1: Using curl to test the nginx web server

```
root@nginx:/# curl localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is
 successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please
refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

or more local ports to a port on a pod. The following command forwards the local port 8080 to the port 80 on the pod:

```
$ kubectl port-forward nginx 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

You can now test the nginx web server locally on your machine (open a new Terminal to test this):

```
$ curl localhost:8080
```

It's worth noting at this point that the minikube cluster itself has its own IP address, and so does the pod running within the worker node.

To get the IP address of the minikube cluster, use the following command:

```
$ minikube ip
192.168.49.2
```

To get the IP address of the Pod running within the worker node in the minikube cluster:

```
$ kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP
NODE       NOMINATED NODE   READINESS GATES
nginx   1/1     Running   0          15m
10.244.0.4   minikube   <none>
<none>
```

Figure 6 shows the IP addresses of the minikube cluster and the pod.

When you're done with a pod, you can delete it using the **delete** option and specifying the pod name:

```
$ kubectl delete pod nginx
pod "nginx" deleted
```

You can also delete the pod using the configuration file that you used earlier to start the pod:

```
$ kubectl delete -f nginxpod.yaml
```

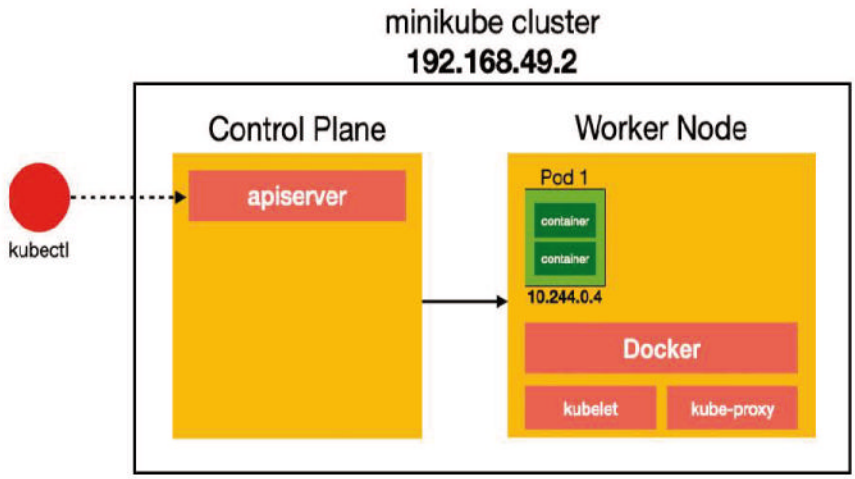

Figure 6: The minikube cluster and the pod each have their own IP addresses.

Deployments

Now that you understand what a pod is, let's move on to the next important component in Kubernetes – **Deployment**. A deployment in Kubernetes is a resource object used to manage the lifecycle of pods. It provides declarative updates to applications, such as rolling out new features or changes, updating existing ones, and rolling back to previous versions if necessary.

> Deployments are a higher-level of abstraction built on top of pods and replica sets, providing more flexibility and ease of management.

In Kubernetes, deployments are often used to manage applications like web servers. Instead of creating a single pod to run nginx, deployments enable the creation of multiple nginx pods. This strategy distributes the work-

Listing 2: Creating a deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver-deployment
  labels:
    app: webserver
spec:
  replicas: 5
  selector:
    matchLabels:
      app: webserver
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        resources:
          requests:
            memory: "150Mi"
          limits:
            memory: "300Mi"
        livenessProbe:
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 3
          periodSeconds: 3
        readinessProbe:
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 10
```

load across multiple nodes (physical machines), allowing for scalability by dynamically adjusting the number of web server instances in response to changing demand. Additionally, deployments automatically restart failed pods, ensuring high availability of the application.

To see the use of deployments, let's create a new file named **deployment.yaml** and populate it with the following statements, as shown in **Listing 2**.

Here are the key attributes in this configuration file:

- **replicas: 5:** Specifies that you want five replicas (instances) of the pod managed by this Deployment.
- **resources:** Defines resource requests and limits for the container.
  - **requests:** Requests 150MiB of memory.
  - **limits:** Limits memory usage to 300MiB.
- **livenessProbe:** The liveness probe determines if the container is alive and healthy. If it fails, Kubernetes restarts the container.
  - **initialDelaySeconds: 3:** Sets a delay of three seconds before the first liveness probe is performed after the container starts.
  - **periodSeconds: 3:** Sets the frequency at which the liveness probe should be performed, in this case, every three seconds.
- **readinessProbe:** The readiness probe checks if the container is ready to serve traffic. If it fails, Kubernetes removes the container from load balancing until it passes the probe.
  - **initialDelaySeconds: 5:** Sets a delay of five seconds before the first readiness probe is performed after the container starts.
  - **periodSeconds: 10:** Sets the frequency at which the readiness probe should be performed, in this case, every 10 seconds.

This Deployment definition deploys and manages five replicas of the nginx container, ensuring that each pod has defined resource constraints and probes for health checks. To apply this configuration to the cluster, use the following command:

```
$ kubectl apply -f deployment.yaml
deployment.apps/webserver-deployment created
```

You can verify the successful deployment using the following command:

```
$ kubectl get deployments
NAME                  READY   UP-TO-DATE
AVAILABLE     AGE
webserver-deployment   5/5     5
5            35s
```

You can see that five out of five pods (5/5 under the READY column) are ready. To view the pods created by this deployment, use the following command:

```
$ kubectl get pods
NAME                                    READY
STATUS    RESTARTS   AGE
webserver-deployment-55f8f4f47c-4lq7r   1/1
Running   0          40s
webserver-deployment-55f8f4f47c-98jvk   1/1
Running   0          40s
webserver-deployment-55f8f4f47c-klntj   1/1
Running   0          40s
webserver-deployment-55f8f4f47c-tqhk8   1/1
Running   0          40s
webserver-deployment-55f8f4f47c-v4kl7   1/1
Running   0          40s
```

From the output, you can see that five pods were indeed created for this deployment. The name of each pod is prefixed with the deployment name, **webserver-deployment**, followed by a unique identifier (**55f8f4f47c** in this case) and additional suffixes to ensure each pod has a unique name (e.g., **4lq7r**).

Each pod has its own assigned IP address. To view the IP address of each pod, use the **-o wide** option:

```
$ kubectl get pods -o wide
```

Figure 7 shows the IP address of each pod.

Figure 8 summarizes the state of the worker node containing the five pods.

Let's try deleting a pod in the deployment using its name:

```
$ kubectl delete pod webserver-deployment-
55f8f4f47c-4lq7r
```

Once that's done, let's check the status of the pod again:

```
$ kubectl get pods
```

You'll observe that a new pod is created to replace the deleted pod (see **Figure 9**). The ability to automatically

```
NAME                                    READY   STATUS    RESTARTS   AGE   IP
webserver-deployment-55f8f4f47c-4lq7r   1/1     Running   0          84s   10.244.0.7
webserver-deployment-55f8f4f47c-98jvk   1/1     Running   0          84s   10.244.0.9
webserver-deployment-55f8f4f47c-klntj   1/1     Running   0          84s   10.244.0.8
webserver-deployment-55f8f4f47c-tqhk8   1/1     Running   0          84s   10.244.0.6
webserver-deployment-55f8f4f47c-v4kl7   1/1     Running   0          84s   10.244.0.5
```

Figure 7: Viewing the IP addresses of each pod

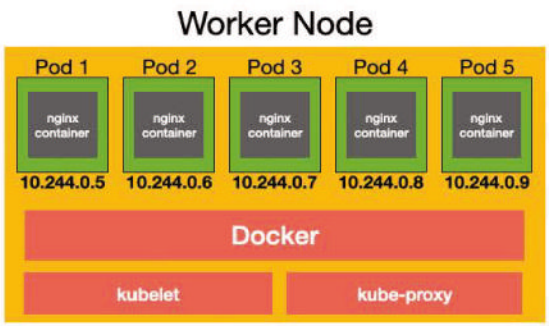

Figure 8: The state of the cluster with the five replicas running nginx

```
NAME                                    READY   STATUS              RESTARTS   AGE
webserver-deployment-55f8f4f47c-98jvk   1/1     Running             0          28m
webserver-deployment-55f8f4f47c-klntj   1/1     Running             0          28m
webserver-deployment-55f8f4f47c-r42kj   0/1     ContainerCreating   0          1s
webserver-deployment-55f8f4f47c-tqhk8   1/1     Running             0          28m
webserver-deployment-55f8f4f47c-v4kl7   1/1     Running             0          28m
```

Figure 9: A new pod is created and started to replace the deleted one

restart a pod that has gone down is one of the advantages of using deployments in Kubernetes.

You can adjust the scale of a deployment as well. For instance, if your deployment currently has five replicas but demand has decreased, you can scale it down to three replicas:

```
$ kubectl scale deployment webserver-
deployment --replicas=3
deployment.apps/webserver-deployment scaled
```

When you now get the pods, you'll see that only three remain:

```
$ kubectl get pods
NAME                                  READY
STATUS    RESTARTS   AGE
webserver-deployment-55f8f4f47c-98jvk   1/1
Running   0          30m
webserver-deployment-55f8f4f47c-klntj   1/1
Running   0          30m
webserver-deployment-55f8f4f47c-v4kl7   1/1
Running   0          30m
```

So now you have three pods running the nginx web server. How do you test them? You can use the port-forwarding technique I discussed in the previous section to port-forward a port on your local server to a particular pod in the worker node:

```
$ kubectl port-forward webserver-deployment-
55f8f4f47c-98jvk 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

In the above command, I forwarded the local port 8080 to the pod named **webserver-deployment-55f8f4f47c-98jvk** listening at port 80. You can now use another terminal to access the nginx web server using the following command:

```
$ curl localhost:8080
```

Services

Up to this point, you've learned that a deployment lets you define how many replicas (or pods) to deploy. When running five pods concurrently, users shouldn't need to interact with a specific pod directly. Instead, users only need to know the website URL, and Kubernetes manages routing requests to the correct pod. This is where **Services** play a crucial role.

A Service in Kubernetes is an abstraction that defines a logical set of Pods and a policy by which to access them. Services enable communication between different parts of the application and can be accessed internally within the cluster or externally depending on the type of service.

Let's now apply a service to the deployment to our nginx deployment so that users don't need to access a specific pod directly. Create a new file named **service.yaml** and populate it with the statements shown in **Listing 3**.

The YAML definition you provided describes a Kubernetes Service named **web-service** that exposes your **webserver** deployment on port 80 using TCP. To apply the configuration to the minikube cluster, use the following command:

```
$ kubectl apply -f service.yaml
service/web-service created
```

You can verify that the **web-service** service is created using the following command:

```
$ kubectl get services
NAME          TYPE        CLUSTER-IP
EXTERNAL-IP   PORT(S)   AGE
kubernetes    ClusterIP   10.96.0.1
<none>        443/TCP   26h
web-service   ClusterIP   10.108.174.191
<none>        80/TCP    24s
```

To view more detailed information about the service, use the following command:

```
$ kubectl describe service web-service
Name:              web-service
Namespace:         default
Labels:            svc=web-service
Annotations:       <none>
Selector:          app=webserver
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.108.174.191
IPs:               10.108.174.191
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.244.0.5:80,
```

Listing 3: The service configuration file

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
  labels:
    svc: web-service
spec:
  selector:
    app: webserver
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

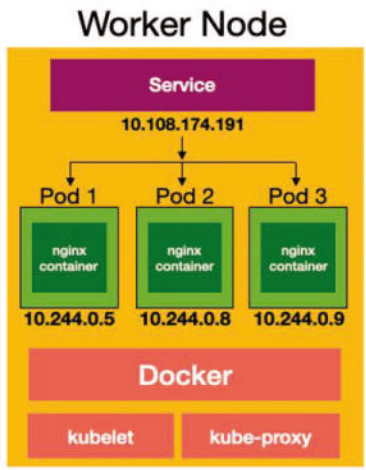

Figure 10: The service has an IP address that all the pods within the node will use to access the nginx web servers.

```
                  10.244.0.8:80,
                  10.244.0.9:80
Session Affinity:  None
Events:            <none>
```

In particular, take note of the following for this example:

- The service has an IP address of **10.108.174.191**. This is the address that all pods within the node will use to access the nginx web servers.
- The **Endpoints** section contains the IP addresses of all the pods that it is connected to. The service will route traffic to the correct pods.

Figure 10 shows the state of the worker node at this moment.

With the service that's created, how do you test the nginx web servers? The first way I want to show you is to create another pod in the node and access the nginx web servers through the service. For this, let's create a new file named **curlpod.yaml** and populate it with the following statements:

```
apiVersion: v1
kind: Pod
metadata:
  name: curl-pod
spec:
  containers:
  - name: curl-container
    image: curlimages/curl:latest
    command:
    - sleep
    - "3600"  # Sleep to keep the
              # container running
```

The above configuration file creates a Pod that contains the curl utility. Using this pod, you can use the curl utility to access the web servers.

Let's apply the configuration to the minikube cluster:

```
$ kubectl apply -f curlpod.yaml
pod/curl-pod created
```

You can now verify that the new pod is created (named **curl-pod**):

```
$ kubectl get pods -o wide
```

Figure 11 shows the curl-pod, along with the three nginx pods.

Figure 12 shows the current state of the worker node.

You can now try to use the **curl-pod** to access the **web-service** service:

```
$ kubectl exec -it curl-pod  -- curl
web-service
```

The above command uses the **curl-pod** to issue a curl request to the nginx web server through the **web-service** service.

```
NAME                                READY   STATUS    RESTARTS   AGE   IP
curl-pod                            1/1     Running   0          53s   10.244.0.12
webserver-deployment-55f8f4f47c-98jvk   1/1     Running   0          23h   10.244.0.9
webserver-deployment-55f8f4f47c-klntj   1/1     Running   0          23h   10.244.0.8
webserver-deployment-55f8f4f47c-v4kl7   1/1     Running   0          23h   10.244.0.5
```

Figure 11: The curl-pod along with the three nginx pods

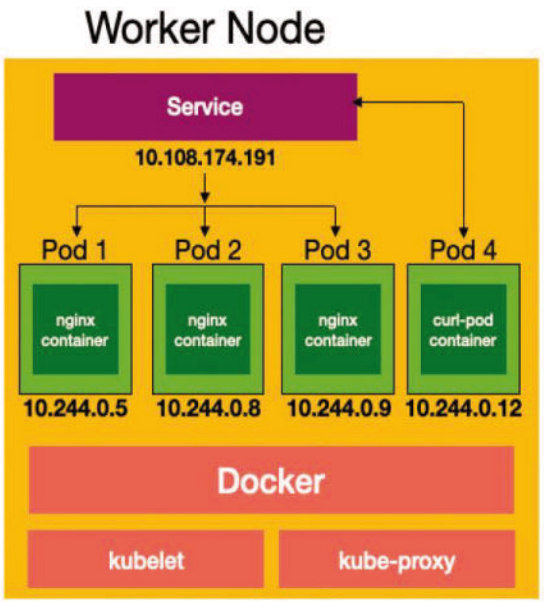

Figure 12: The addition of the curl-pod to the worker node

Alternatively, you can also use the IP address of the **web-service** service:

```
$ kubectl exec -it curl-pod  -- curl 10.108.174.191
```

If you wish to access the web-service service from your local machine, you can use port forwarding. The following command is used to create a port-forwarding tunnel from your local machine to the **web-service** service:

```
$ kubectl port-forward service/web-service
8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

You can now use another terminal to access the nginx web server using the following command:

```
$ curl localhost:8080
```

NodePort

A Service in Kubernetes allows pods in a cluster to communicate with each other, regardless of whether they're running on the same node or different nodes. What happens if you want to expose the service so that it's accessible from outside the cluster? This is where **NodePort** comes into the picture. NodePort is a specific type of Service that exposes the Service on a static port (the NodePort) on each Node's IP address. When you create a NodePort Service, a **ClusterIP Service**, to which the NodePort Service routes, is automatically created. The NodePort Service makes it possible to access the application using **<NodeIP>:<NodePort>** from outside the cluster.

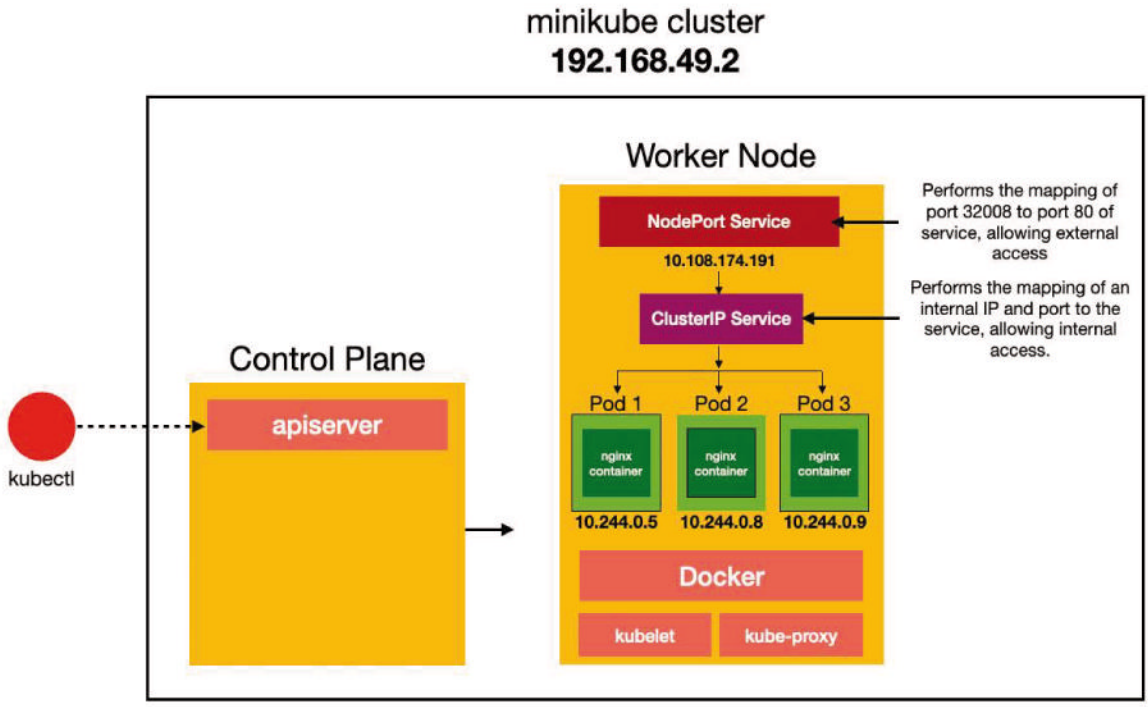

Figure 13: The cluster with the NodePort listening at port 32008

```
|-----------|-------------|-------------|---------------------------|
| NAMESPACE |     NAME    | TARGET PORT |            URL            |
|-----------|-------------|-------------|---------------------------|
| default   | web-service |          80 | http://192.168.49.2:32008 |
|-----------|-------------|-------------|---------------------------|
    Starting tunnel for service web-service.
|-----------|-------------|-------------|------------------------|
| NAMESPACE |     NAME    | TARGET PORT |          URL           |
|-----------|-------------|-------------|------------------------|
| default   | web-service |             | http://127.0.0.1:56554 |
|-----------|-------------|-------------|------------------------|
    Opening service default/web-service in default browser...
    Because you are using a Docker driver on darwin, the terminal needs
to be open to run it.
```

Figure 14: Creating a tunnel to the service

Let's see this in action. First, create a file named **service_nodeport.yaml** and populate it with the following statements:

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
  labels:
    svc: web-service
spec:
  type: NodePort
  selector:
    app: webserver
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 32008
```

This configuration defines a NodePort service. Similar to a regular service, it includes an additional attribute called **nodePort,** which specifies the port (e.g., 32008) on each node where the service will be accessible. To apply this configuration, use the following command, as usual:

```
$ kubectl apply -f service_nodeport.yaml
service/web-service configured
```

You can now verify that the NodePort is created correctly:

```
$ kubectl get svc web-service
NAME          TYPE       CLUSTER-IP
EXTERNAL-IP   PORT(S)        AGE
web-service   NodePort   10.108.174.191
<none>        80:32008/TCP   9h
```

Figure 13 shows the current state of the cluster.

You should now be able to access the nginx web server through the node's IP address which, in the case of minikube, is also the minikube IP address. However, in the case of minikube, you have to create a tunnel to the service via ports published by the minikube container, which will then perform a port forwarding for you:

```
$ minikube service web-service
```

Figure 14 shows that result returned by the above command.

Figure 15 shows that to access the node's web server, you have to use the address **127.0.0.1:56554**, which will then redirect the traffic to the minikube cluster's IP address (port 32008 as specified by the node port), which will then direct to the **ClusterIP Service**.

Figure 16 shows the page returned by the nginx web server.

Listing 4: The content of the ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: index-html-configmap
data:
  index.html: |
    <!DOCTYPE html>
    <html lang="en">
    <head>
      <meta charset="UTF-8">
      <meta name="viewport"
        content="width=device-width,
                 initial-scale=1.0">
      <title>Hello Kubernetes</title>
      <style>
        .tray {
          display: inline-block;
          background-color: LightCoral;
          color: white;
          padding: 10px;
          border-radius: 5px;
        }
      </style>
    </head>
    <body>
      <center>
        <h1 class="tray">Hello, Kubernetes!</h1>
      </center>
    </body>
    </html>
```

ConfigMaps

You now have the nginx web servers ready to serve from within a Kubernetes cluster, with built-in redundancies. The web servers are accessible through a NodePort Service, which redirects traffic from outside the cluster to the pods within the cluster. But how do you change the content that nginx is serving? One way is to use a **ConfigMap**.

A ConfigMap in Kubernetes is an API object used to store non-confidential configuration data in key-value pairs. ConfigMaps allow you to decouple configuration artifacts from container images, enabling you to keep application configuration separate from your containerized application code. They're often used to store configuration data such as application settings, environment variables, command-line arguments, and configuration files. They're commonly used when deploying your database servers in Kubernetes.

For this example, I'll use ConfigMap to set the **index.html** page that nginx will serve. Let's create a new file named **configmap.yaml** and populate it with the statements shown in **Listing 4**.

This ConfigMap (named **index-html-configmap**) contains an **index.html** file with the specified HTML content. You'll configure your nginx web server to serve this file (**index.html**) when it's accessed.

You need to apply this ConfigMap to the minikube cluster using the following command:

```
$ kubectl apply -f configmap.yaml
configmap/index-html-configmap created
```

Next, modify the **deployment.yaml** file by appending the statements as shown in **Listing 5**.

In the above, the deployment configuration defines a volume named **nginx-index-file** that obtains its data from the **index-html-configmap** ConfigMap (defined in the **configmap.yaml** file). The **volumeMounts**: section mounts the **nginx-index-file** volume at **/usr/share/nginx/html/index.html**.

Figure 17 shows the relationship between the ConfigMap and the Deployment.

Apply the configuration to the cluster using the following command:

```
$ kubectl apply -f deployment.yaml
deployment.apps/webserver-deployment created
```

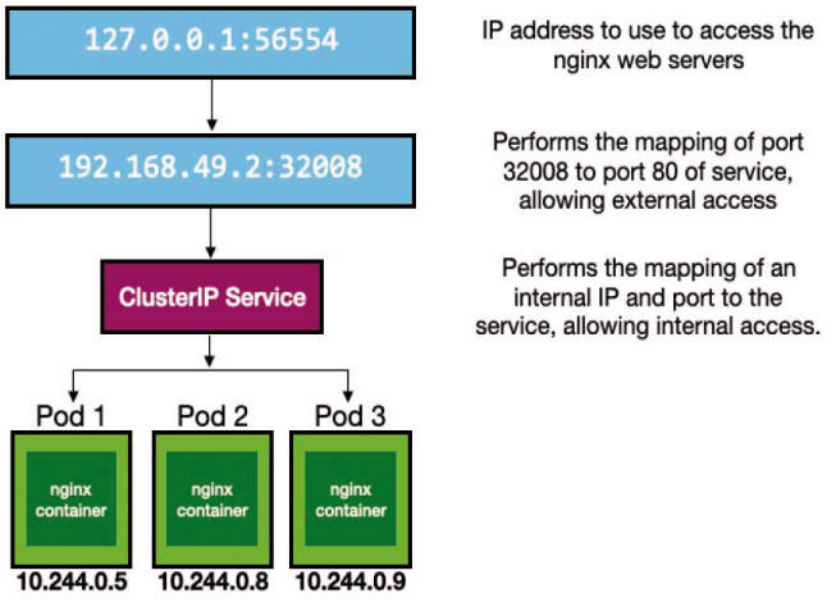

Figure 15: Accessing the nginx web servers using the local IP address

Figure 16: The web page returned by the nginx web server

Listing 5: Adding the volumes: and volumeMounts: to the deployment file

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver-deployment
  labels:
    app: webserver
spec:
  replicas: 5
  selector:
    matchLabels:
      app: webserver
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        resources:
          requests:
            memory: "150Mi"
          limits:
            memory: "300Mi"
        livenessProbe:
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 3
          periodSeconds: 3
        readinessProbe:
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 10
        # add the following statements
        volumeMounts:
        - name: nginx-index-file
          mountPath: "/usr/share/nginx/html/"
      volumes:
      - name: nginx-index-file
        configMap:
          name: index-html-configmap
```

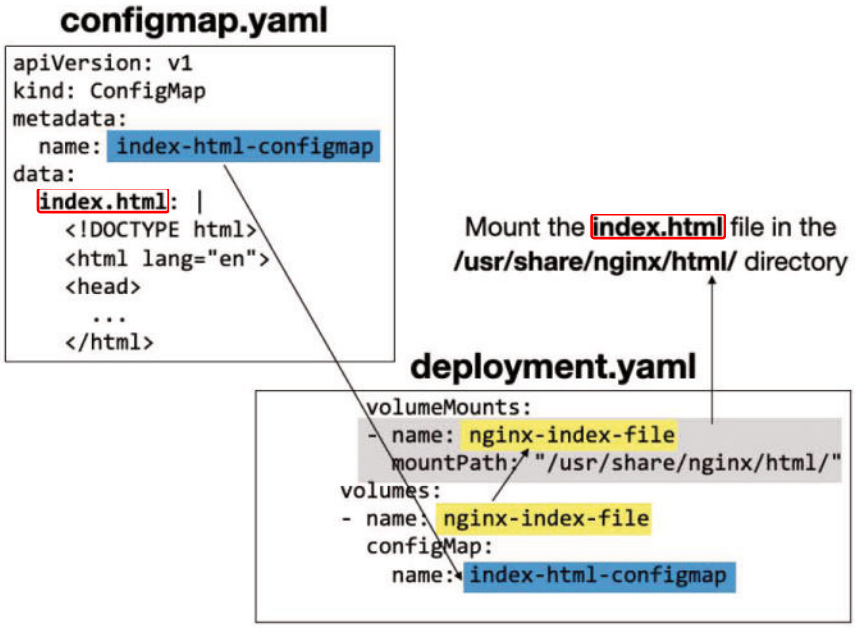

Figure 17: The relationship between the ConfigMap and Deployment

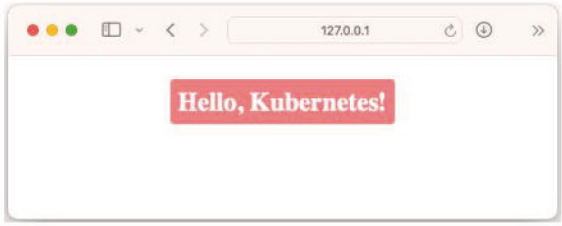

Figure 18: The nginx web servers now serving the index.html file that you've created in the ConfigMap

You can now create a tunnel to the service via ports published by the minikube container, which will then perform a port forwarding for you:

```
$ minikube service web-service
```

Figure 18 shows the page now served by the nginx web servers.

Summary

In this article, I explored the foundational concepts of Kubernetes, focusing on its role in modern application deployment and management.

To ease the learning curve associated with Kubernetes, you can use minikube, a tool that simplifies setting up and running Kubernetes clusters locally. Through a step-by-step guide, you learned how to deploy a basic Kubernetes cluster using minikube. Key Kubernetes components such as Pods, Deployments, Services, NodePort, and ConfigMaps were covered in detail, showcasing their essential roles in application deployment and management.

Hopefully you have now gained some useful insights into leveraging minikube to experiment with Kubernetes.

Wei-Meng Lee