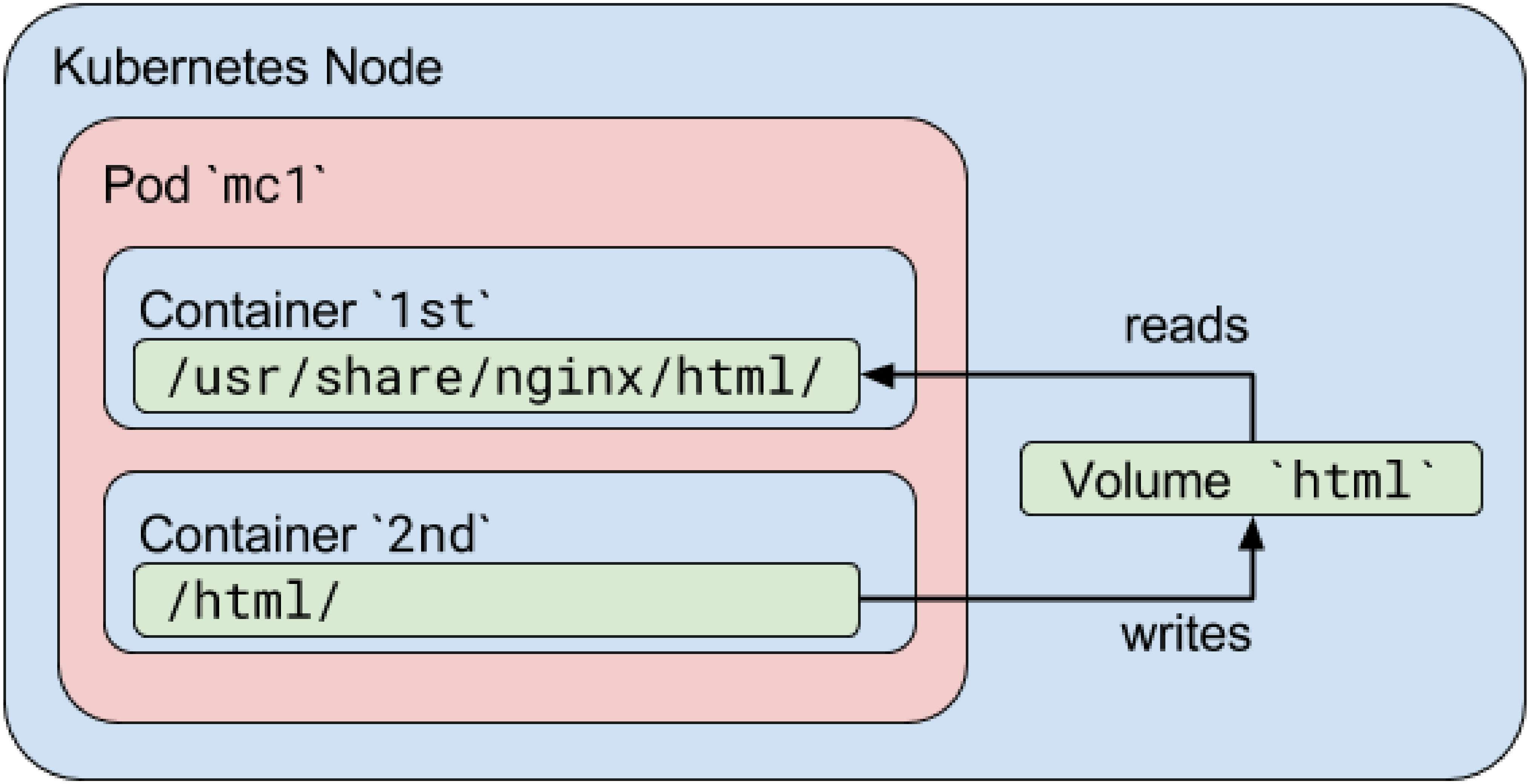
The Mirantis Blog

Your guide through the wilderness of Open Cloud

Multi-container pods and container communication in Kubernetes

Pavel Chekin - August 28, 2017 - kd200 | kubernetes | pods | training

Containers are often intended to solve a single, narrowly defined problem, such as a microservice, but in the real world, problems require multiple containers for a complete



solution. In this article, we're going to talk about combining multiple containers into a single Kubernetes Pod, and what it means for inter-container communication.

What is a Kubernetes Pod?

Let’s start by explaining what a Pod is in the first place. A Pod is is the smallest unit that can be deployed and managed by Kubernetes. In other words, if you need to run a single container in Kubernetes, then you need to create a Pod for that container. At the same time, a Pod can contain more than one container, usually because these containers are relatively tightly coupled. How tightly coupled? Well, think of it this way: the containers in a pod represent processes that would have run on the same server in a pre-container world.

And that makes sense, because in many respects, a Pod aces like a single server. For example, each container can access the other containers in the pod as different ports on localhost.

Why does Kubernetes use a Pod as the smallest deployable unit, and not a single container?

While it would seem simpler to just deploy a single container directly, there are good reasons to add a layer of abstraction represented by the Pod. A container is an existing entity, which refers to a specific thing. That specific thing might be a Docker container, but it might also be a rkt container, or a VM managed by Virtlet. Each of these has different requirements.

What’s more, to manage a container, Kubernetes needs additional information, such as a restart policy, which defines what to do with a container when it terminates, or a liveness probe, which defines an action to detect if a process in a container is still alive from the application’s perspective, such as a web server responding to HTTP requests.

Instead of overloading the existing “thing” with additional properties, Kubernetes architects have decided to use a new entity, the Pod, that logically contains (wraps) one or more containers that should be managed as a single entity.

Why does Kubernetes allow more than one container in a Pod?

Containers in a Pod run on a “logical host”; they use the same network namespace (in other words, the same IP address and port space), and the same IPC namespace. They can also use shared volumes. These properties make it possible for these containers to efficiently communicate, ensuring data locality. Also, Pods enable you to manage several tightly coupled application containers as a single unit.

So if an application needs several containers running on the same host, why not just make a single container with everything you need? Well first, you’re likely to violate the “one process per container” principle. This is important because With multiple processes in the same container, it is harder to troubleshoot the container because logs from different processes will be mixed together, and it is harder manage the processes lifecycle, for example to take care of “zombie” processes when their parent process dies. Second, using several containers for an application is simpler, more transparent, and enables decoupling software dependencies. Also, more granular containers can be reused between teams.

Use Cases for Multi-Container Pods

The primary purpose of a multi-container Pod is to support co-located, co-managed helper processes for a primary application. There are some general patterns for using helper processes in Pods:

- **Sidecar containers** “help” the main container. Some examples include log or data change watchers, monitoring adapters, and so on. A log watcher, for example, can be built once by a different team and reused across different applications. Another example of a sidecar container is a file or data loader that generates data for the main container.
- **Proxies, bridges, and adapters** connect the main container with the external world. For example, Apache HTTP server or nginx can serve static files. It can also act as a reverse proxy to a web application in the main container to log and limit HTTP requests. Another example is a helper container that re-routes requests from the main container to the external world. This makes it possible for the main container to connect to localhost to access, for example, an external database, but without any service discovery.

While you can host a multi-tier application (such as WordPress) in a single Pod, the recommended way is to use separate Pods for each tier, for the simple reason that you can scale tiers up independently and distribute them across cluster nodes.

Communication between containers in a Pod

Having multiple containers in a single Pod makes it relatively straightforward for them to communicate with each other. They can do this using several different methods.

Shared volumes in a Kubernetes Pod

In Kubernetes, you can use a shared Kubernetes Volume as a simple and efficient way to share data between containers in a Pod. For most cases, it is sufficient to use a directory on the host that is shared with all containers within a Pod.

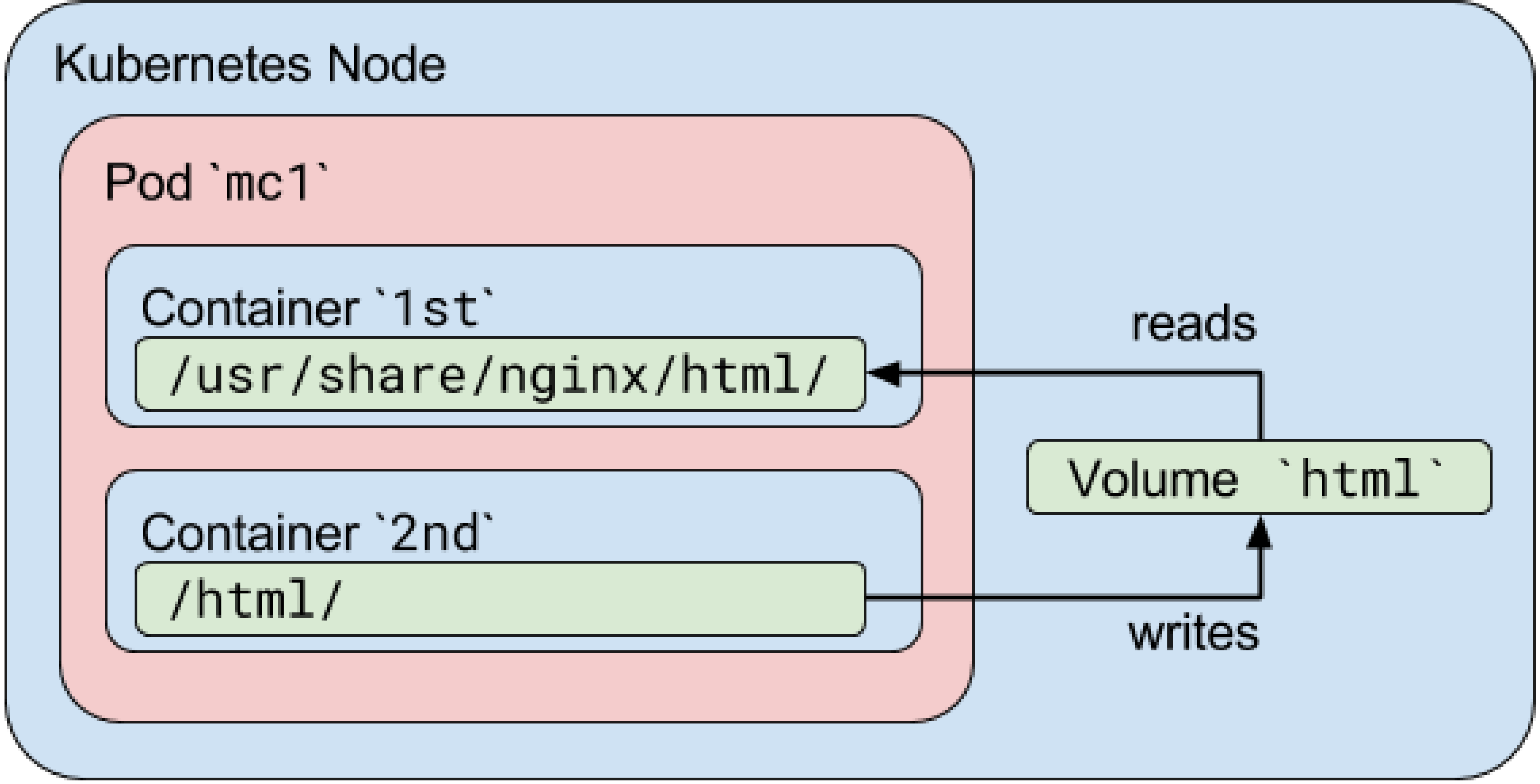
Kubernetes Volumes enables data to survive container restarts, but these volumes have the same lifetime as the Pod. That means that the volume (and the data it holds) exists exactly as long as that Pod exists. If that Pod is deleted for any reason, even if an identical replacement is created, the shared Volume is also destroyed and created anew.

A standard use case for a multi-container Pod with a shared Volume is when one container writes logs or other files to the shared directory, and the other container reads from the shared directory. For example, we can create a Pod like so:

WATCH NOW

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mc1
spec:
  volumes:
  - name: html
    emptyDir: {}
  containers:
  - name: 1st
    image: nginx
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
  - name: 2nd
    image: debian
    volumeMounts:
    - name: html
      mountPath: /html
    command: ["/bin/sh", "-c"]
    args:
      - while true; do
          date >> /html/index.html;
          sleep 1;
        done
```

In this example, we define a volume named html. Its type is emptyDir, which means that the volume is first created when a Pod is assigned to a node, and exists as long as that Pod is running on that node. As the name says, it is initially empty. The 1st container runs nginx server and has the shared volume mounted to the directory /usr/share/nginx/html. The 2nd container uses the Debian image and has the shared volume mounted to the directory /html. Every second, the 2nd container adds the current date and time into the index.html file, which is located in the shared volume. When the user makes an HTTP request to the Pod, the Nginx server reads this file and transfers it back to the user in response to the request.



You can check that the pod is working either by exposing the nginx port and accessing it using your browser, or by checking the shared directory directly in the containers:

```
$ kubectl exec mc1 -c 1st -- /bin/cat /usr/share/nginx/html/index.html
 ...
 Fri Aug 25 18:36:06 UTC 2017

 $ kubectl exec mc1 -c 2nd -- /bin/cat /html/index.html
 ...
 Fri Aug 25 18:36:06 UTC 2017
 Fri Aug 25 18:36:07 UTC 2017
```

Inter-process communications (IPC)

Containers in a Pod share the same IPC namespace, which means they can also communicate with each other using standard inter-process communications such as SystemV semaphores or POSIX shared memory.

In the following example, we define a Pod with two containers. We use the same Docker image for both. The first container, producer, creates a standard Linux message queue, writes a number of random messages, and then writes a special exit message. The second container, consumer, opens that same message queue for reading and reads messages until it receives the exit message. We also set the restart policy to ‘Never’, so the Pod stops after termination of both containers.

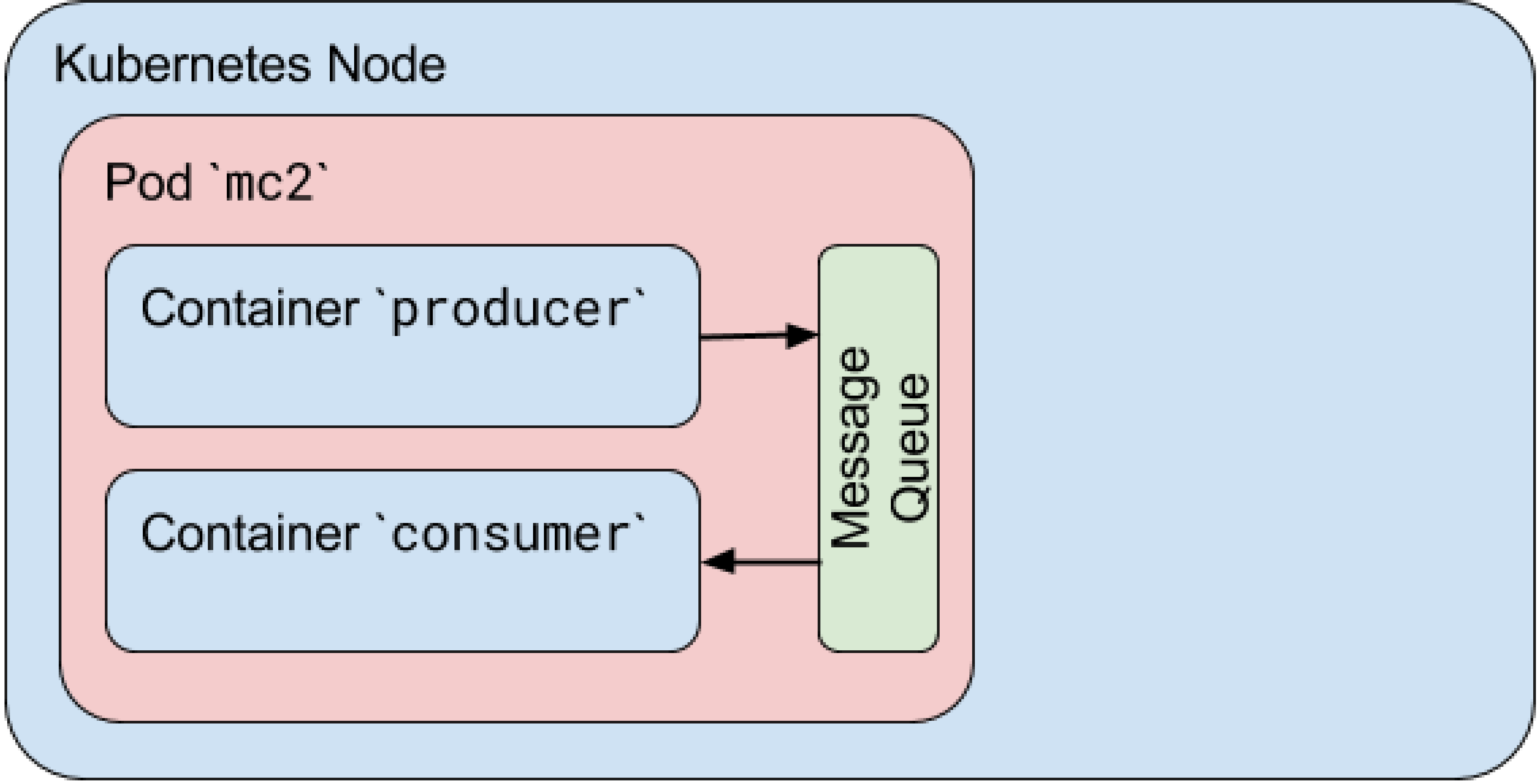
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mc2
spec:
  containers:
  - name: producer
    image: allingeek/ch6_ipc
    command: ["./ipc", "-producer"]
  - name: consumer
    image: allingeek/ch6_ipc
    command: ["./ipc", "-consumer"]
  restartPolicy: Never
```

To check this out, create the pod using kubectl create and watch the Pod status:

```
$ kubectl get pods --show-all -w
NAME      READY     STATUS              RESTARTS   AGE
mc2       0/2       Pending             0          0s
mc2       0/2       ContainerCreating   0          0s
mc2       0/2       Completed           0          29
```

Now you can check logs for each container and verify that the 2nd container received all messages from the 1st container, including the exit message:

```
$ kubectl logs mc2 -c producer
...
Produced: f4
Produced: 1d
Produced: 9e
Produced: 27
$ kubectl logs mc2 -c consumer
...
Consumed: f4
Consumed: 1d
Consumed: 9e
Consumed: 27
Consumed: done
```



There is one major problem with this Pod, however, and it has to do with how containers start up.

Container dependencies and startup order

Currently, all containers in a Pod are being started in parallel and there is no way to define that one container must be started after other container. For example, in the IPC example, there is a chance that the second container might finish starting before the first one has started and created the message queue. In this case, the second container will fail, because it expects that the message queue already exists.

Some efforts to provide some measure of control over how containers start, such as Kubernetes Init Containers, which start first (and sequentially), are under development, but in a cloud native environment, it's always better to plan for failures outside of your immediate control. For example, one way to to fix this issue would be to change the application to wait for the message queue to be created.

Inter-container network communication

Containers in a Pod are accessible via "localhost"; they use the same network namespace. Also, for containers, the observable host name is a Pod's name. Because containers share the same IP address and port space, you should use different ports in containers for incoming connections. In other words, applications in a Pod must coordinate their usage of ports.

In the following example, we will create a multi-container Pod where nginx in one container works as a reverse proxy for a simple web application running in the second container.

Step 1. Create a ConfigMap with the nginx configuration file. Incoming HTTP requests to port 80 will be forwarded to port 5000 on localhost:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mc3-nginx-conf
data:
  nginx.conf: |-
    user  nginx;
    worker_processes  1;

    error_log  /var/log/nginx/error.log warn;
    pid        /var/run/nginx.pid;

    events {
        worker_connections  1024;
    }

    http {
        include       /etc/nginx/mime.types;
        default_type  application/octet-stream;
        sendfile        on;
        keepalive_timeout  65;

        upstream webapp {
            server 127.0.0.1:5000;
        }

        server {
            listen 80;

            location / {
                proxy_pass         http://webapp;
                proxy_redirect     off;
            }
        }
    }
```

Step 2. Create a multi-container Pod with the simple web app and nginx in separate containers. Note that for the Pod, we define only nginx port 80. Port 5000 will not be accessible outside of the Pod.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mc3
  labels:
    app: mc3
spec:
  containers:
  - name: webapp
    image: training/webapp
  - name: nginx
    image: nginx:alpine
    ports:
    - containerPort: 80
    volumeMounts:
    - name: nginx-proxy-config
      mountPath: /etc/nginx/nginx.conf
      subPath: nginx.conf
  volumes:
  - name: nginx-proxy-config
    configMap:
      name: mc3-nginx-conf
```

Step 3. Expose the Pod using the NodePort service:

```
$ kubectl expose pod mc3 --type=NodePort --port=80
service "mc3" exposed
```

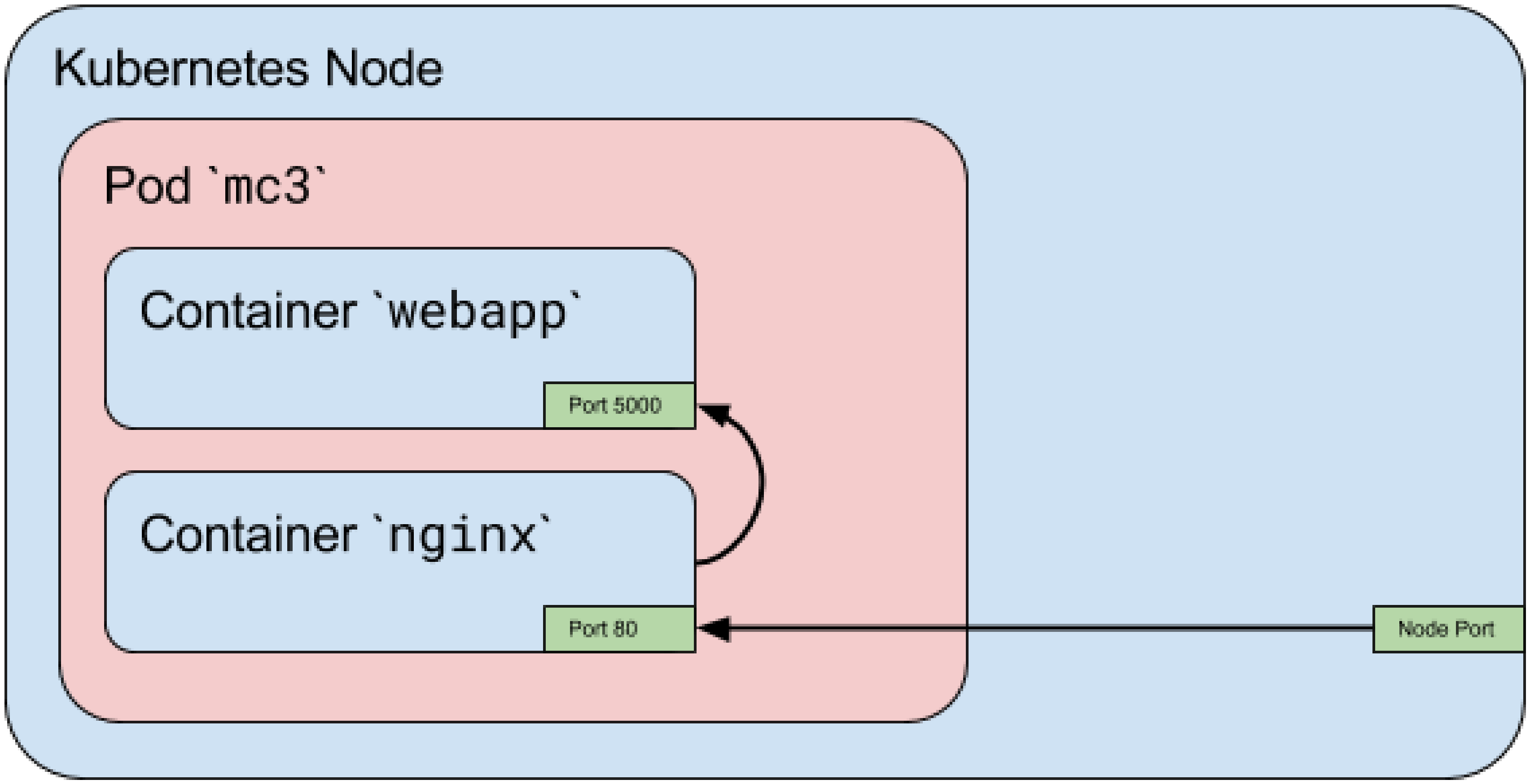
Step 4. Identify port on the node that is forwarded to the Pod:

```
$ kubectl describe service mc3
...
NodePort:       <unset> 31418/TCP
...
```

Now you can use your browser (or curl) to navigate to your node's port to access the web application through reverse proxy, as in:

http://myhost:31418

This request will then be forwarded to port 5000 of the webapp container.



Exposing multiple containers in a Pod

While this example shows how to use a single container to access other containers in the pod, it's quite common for several containers in a Pod to listen on different ports — all of which need to be exposed. To make this happen, you can either create a single service with multiple exposed ports, or you can create a single service for every poirt you're trying to expose.

Where to go from here

By creating pods, Kubernetes provides a great deal of flexibility for orchestrating how containers behave, and how they communicate with each other. They can share file volumes, they can communicate over the network, and they can even communicate using IPC.

That's just the beginning, of course. Interested in seeing what else you can do and getting more hands-on with Kubernetes? Check out our new advanced Kubernetes and Docker Bootcamp II (KD200) – the first course that prepares for the new Certified Kubernetes Administrator (CKA) exam. Hope to see you there!

4 3 2 1 : /

Featured Posts

How to build an edge cloud part 1: Building a simple facial recognition system

53 Things to look for in OpenStack Train

Tips for taking the new OpenStack COA (Certified OpenStack Administrator) exam – October 2019

13 RESPONSES TO “MULTI-CONTAINER PODS AND CONTAINER COMMUNICATION IN KUBERNETES”

Ryan says:

August 31, 2017 at 18:56

A Pod models an application-specific “logical host” and can contain different application containers which are relatively tightly coupled. Pods are the atomic unit on the Kubernetes platform. When we create a Deployment on Kubernetes, that Deployment creates Pods with containers inside them (as opposed to creating containers directly).

REPLY

Dee says:

September 6, 2017 at 23:54

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

REPLY

ganeshkumar pandian says:

November 9, 2017 at 22:58

multi container pod must be used only for tightly coupled containers.
not scenario like mentioned by you (web/app server combo). that will affect replicas performance.
because app containers must run in a separate deployment hence pod. the same way web container also must run in a separate deployment.
so you can scale like

```
kubectl edit deploy/app –replicas=N
kubectl autoscale deploy/app –min=1 –max=10
```

kubectl autoscale deploy/app –min=1 –max=10

don’t go multi container pod for web/app combo

REPLY

Nick Chase says:

November 10, 2017 at 08:22

Definitely something to think about, thank you!

—- Nick

REPLY

Paul K says:

January 11, 2018 at 09:48

I’m trying to follow “Inter-container network communication”.

Step 3 gives error :

Step 3 gives error :

```
$ kubectl expose pod mc3 –type=NodePort –port=80
Error from server (AlreadyExists): services “mc3” already exists
```

Any ideas?
Thanks

REPLY

Nick Chase says:

January 11, 2018 at 10:31

My guess is that htis is the second time you’ve run this, and the service stille xists from the first time. Try

```
kubectl delete service mc3
```

REPLY

Paul K says:

[illegible] says:

January 12, 2018 at 07:55

Bam! Thanks Nick, that did it.

Great tutorial and thanks again for the quick reply.

REPLY

Nick Chase says:

January 15, 2018 at 10:12

My pleasure!

REPLY

Nilesh says:

October 29, 2018 at 16:44

Thanks Nick for nice blog.

You have shown multiple example with different scenarios, but there is no scenario related

You have shown multiple example with different scenarios, but there is no scenario related to database.

Let's say you have an application with multiple microservices using mysql database. For now lets just talk about only one microservice A.
What I will need – one container(having mysql 5.7) and another container (for microservice A).

Two scenarios:

Scenario 1) I want to have one pod with these 2 containers communicating to each other, how do I do it? As there is database involved, how do I make sure data is not lost when pod is deleted or recreated?

Scenario 2) Deploy these two containers in different pods(Lets say pod X,Y) and manage communication between two pods. How do I do it? There is one post I saw exactly talking about same scenario: https://stackoverflow.com/questions/52846587/use-local-docker-image-having-mysql-with-minikube

REPLY

Mike says:

January 2, 2019 at 22:51

Hey, I'm no expert and still learning but Persistent Volumes seem to be the solution for stateful resources like Databases.

REPLY

Nick Chase says:

January 22, 2019 at 09:37

Agreed!

REPLY

Eva says:

May 21, 2019 at 13:22

Hello, I have an app running on a container, and I need to backup that app into S3. This can be done with a sidecar container? Any documentation I could look about it, because I cant be done with a sidecar container? Any documentation I could look about it, because I can't find it? Thank you!

REPLY

kumar gaurav says:

October 9, 2019 at 11:26

Hi,

I am unable to access http://myhost:portnumber

What host i should provide instead of myhost ,i tried with localhost and private ip it didn't work for inter container networking.

Please help.

Regards,
Gaurav

REPLY

LEAVE A REPLY

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

☐ Save my name, email, and website in this browser for the next time I comment.

☐ I'm not a robot

reCAPTCHA
Privacy - Terms

Post Comment

OpenStack

Ceph

Tungsten Fabric

DELIVERY APPROACH

Build

Operate

Transfer

MCP Support

K8s Support

RESOURCES

Customers

Blog

Webinars

Events

Brochures

Videos

Partners

APPLICATION PLATFORM

Overview

Request Beta Access

TRAINING

Overview

Instructor-led

Ondemand

Certification

Private

ABOUT

Contact

Company

Locations

Careers

Meet the Team

Board of Directors

Press Center

Mirantis Inc. 900 E Hamilton Avenue, Suite 650, Campbell, CA 95008 +1-650-963-9828