Multi-Container Pods in Kubernetes

What is a Pod

A Pod is is the smallest deployable 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, if these containers are relatively tightly coupled. In a pre-container world, they would have executed on the same server.

Motivation for Pods

Why does Kubernetes use a Pod as the smallest deployable unit and not a single container? A container is the existing entity, which denotes a specific thing, for example a Docker container. To manage a container, Kubernetes needs additional information, such as restart policy or live probe. A restart policy defines what to do with a container when it terminates. A live probe defines an action to detect if a process in a container still alive from the application perspective, for example if web server responds to HTTP requests. Instead of overloading the existing thing with additional properties, Kubernetes architects have decided to use a new entity - Pod - that logically contains (wraps) one or more containers, which should be managed as a single entity.

Why does Kubernetes allow more than one container in a Pod? Containers in a Pod runs on a "logical host": they use the same network namespace (same IP address and port space), IPC namespace and, optionally, they can use shared volumes. Therefore, these containers can efficiently communicate, ensuring data locality. Also, Pods allow managing several tightly coupled application containers as a single unit.

So, if an application needs several containers running on the same host, why not to make a single container with everything from those containers? First, putting many things into one container will probably violate the "one process per container" principle. Second, using several containers for an application is simpler to use, more transparent, and allows 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 main program. There are some general patterns of using helper processes in Pods:

Sidecar containers "help" the main container. For example, 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, adapters connect the main container with the external world. For example, Apache HTTP server or nginx can serve static files and act as a reverse proxy to a web application in the main container to log and limit HTTP request. Another example is a helper container that re-routes requests from the main container to the external world, so the main container connects to localhost to access, for example, external database without any service discovery.

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

Communication Between Containers in a Pod

Shared volumes

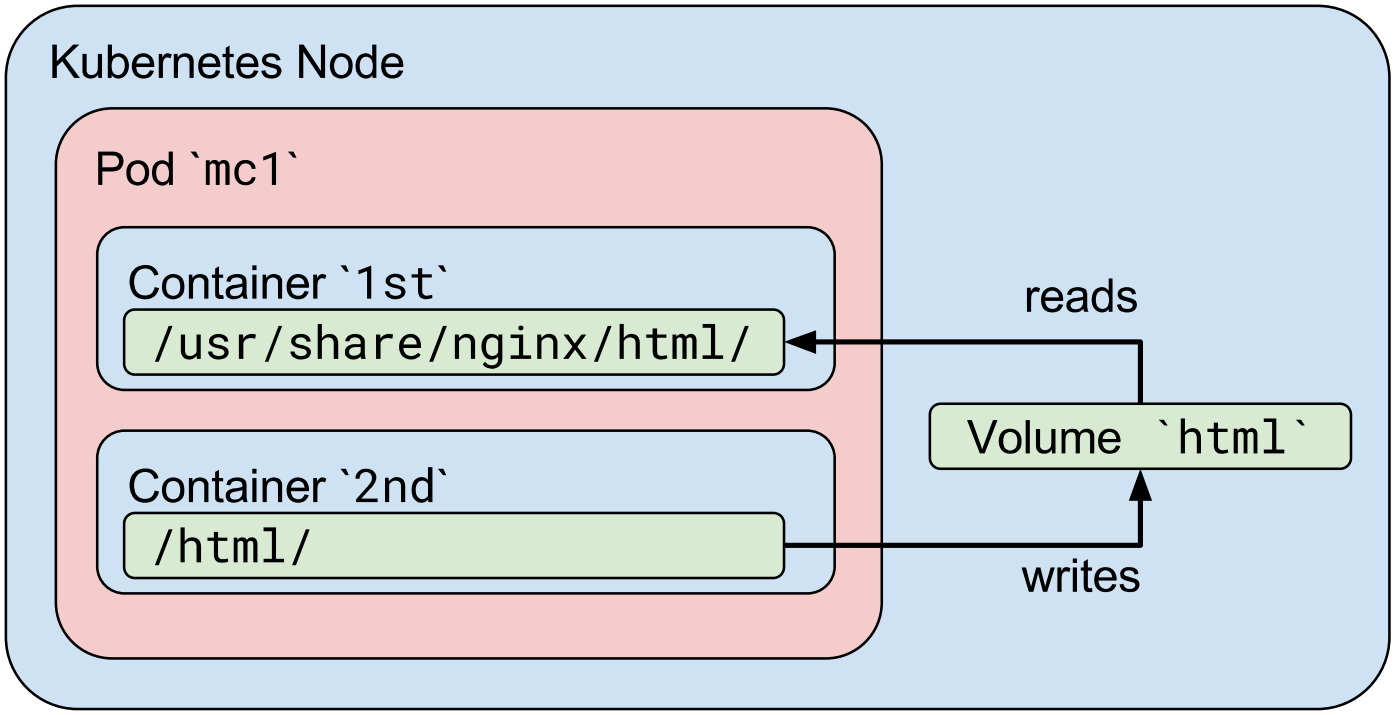
In Kubernetes, you can use a shared Kubernetes Volume as a simple and efficient way to share data between containers in a Pod. For most cases, it is sufficient to use a directory on the host that is shared with all containers within a Pod. Kubernetes Volumes enables data to survive container restarts. It has the same lifetime as a Pod. That means that it exists 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 shared Volume is when one container writes to the shared directory (logs or other files), and the other container reads from the shared directory. Example:

YAML

```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` and its type is `emptyDir` : volume that 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 first container runs nginx server and has the shared volume mounted to the directory `/usr/share/nginx/html` . The second container uses Debian image and has the shared volume mounted to the directory `/html` . The second container every second adds current date and time and into `index.html` that is located in the shared volume. Nginx servers reads this file and transfers it to the user for each HTTP request to the web server.



You can check that the pod is working either exposing nginx port and accessing it using your browser. Another way to check the shared directory directly in 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 and they can also communicate with each other using standard inter-process communications like 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, see https://github.com/allingeek/ch6_ipc. 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 the same message queue for reading and reads messages until it receives the exit message. We also set restart policy to 'Never', so the Pod stops after termination of both containers.

YAML

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mc2
spec:
  containers:
  - name: 1st
    image: allingeek/ch6_ipc
    command: ["./ipc", "-producer"]
  - name: 2nd
    image: allingeek/ch6_ipc
    command: ["./ipc", "-consumer"]
  restartPolicy: Never
```

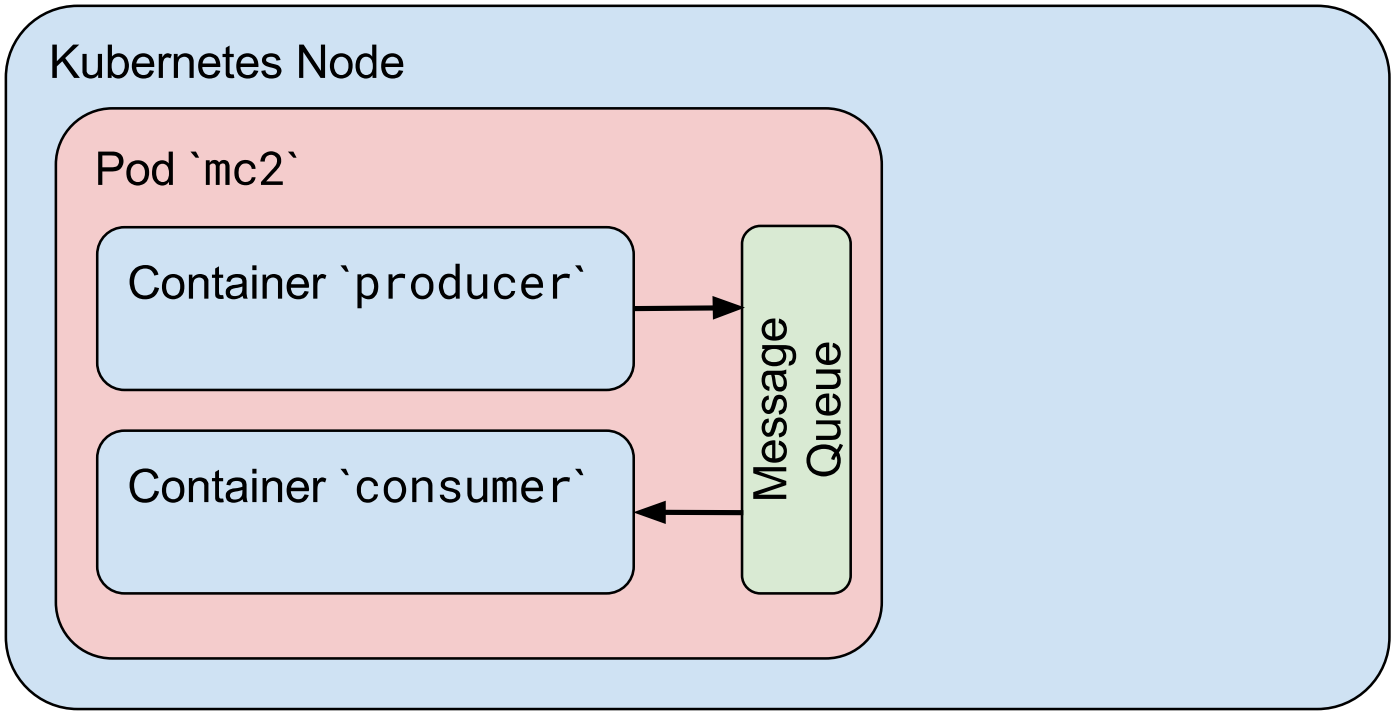
Create the pod using `kubectl create` and watch the Pod status:

```
$ 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         29s
```

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 1st
...
Produced: f4
Produced: 1d
Produced: 9e
Produced: 27
```

```
$ kubectl logs mc2 -c 2nd
...
Consumed: f4
Consumed: 1d
Consumed: 9e
Consumed: 27
Consumed: done
```

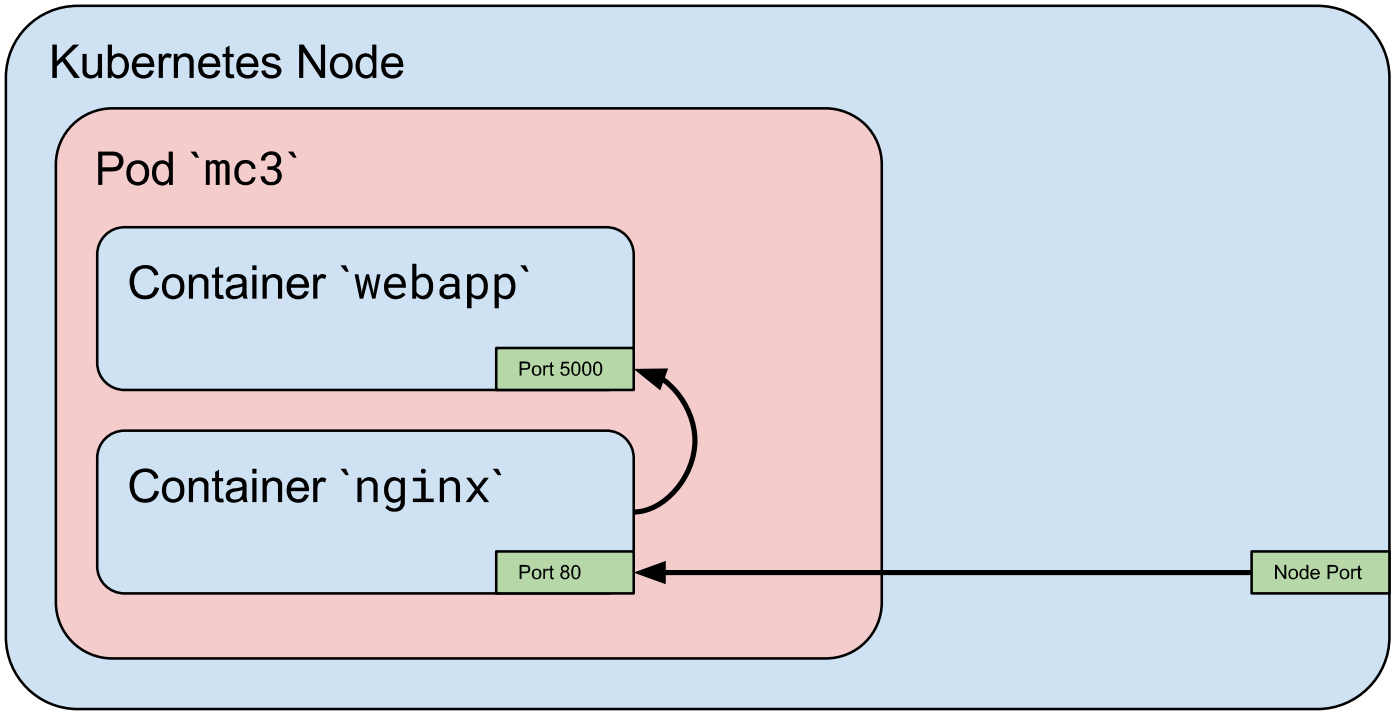


Note: there is one major problem with this Pod. Can you see it? Check "Additional Details about Multi-Containers Pods" for the explanation.

Network

Containers in a Pod are accessible via "localhost", they use the same network namespace. For containers, the observable host name is a Pod's name. Since containers share the same IP address and port space, you should use different ports in containers for incoming connections. Because of this, 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 nginx configuration file. Incoming HTTP requests to the port 80 will be forwarded to the port 5000 on localhost:

YAML

```yaml
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;

        log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                          '$status $body_bytes_sent "$http_referer" '
                          '"$http_user_agent" "$http_x_forwarded_for"';

        access_log  /var/log/nginx/access.log  main;

        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 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

```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.

Additional Details about Multi-Containers Pods

How to expose different containers in a Pod?

It's quite common case when several containers in a Pod listen on different ports and you need to expose all this ports. You can use two services or one service with two exposed ports.

In what order containers are being started in a Pod?

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 (however, there are Kubernetes Init Containers (https://kubernetes.io/docs/concepts/workloads/pods/init-containers/)). Therefore, in your IPC example there is a chance that the second container starts before the first one. In this case, the second container will fail, because in the `consumer` mode it expects that the message queue exists. To fix this issue we, for example, can change the application to wait for the message queue to be created.