Red Hat
OpenShift

Pods and Services

Pods

OpenShift Container Platform leverages the Kubernetes concept of a *pod*, which is one or more containers deployed together on one host, and the smallest compute unit that can be defined, deployed, and managed.

Pods are the rough equivalent of a machine instance (physical or virtual) to a container. Each pod is allocated its own internal IP address, therefore owning its entire port space, and containers within pods can share their local storage and networking.

Pods have a lifecycle; they are defined, then they are assigned to run on a node, then they run until their container(s) exit or they are removed for some other reason. Pods, depending on policy and exit code, may be removed after exiting, or may be retained in order to enable access to the logs of their containers.

OpenShift Container Platform treats pods as largely immutable; changes cannot be made to a pod definition while it is running. OpenShift Container Platform implements changes by terminating an existing pod and recreating it with modified configuration, base image(s), or both. Pods are also treated as expendable, and do not maintain state when recreated. Therefore pods should usually be managed by higher-level controllers, rather than directly by users.

For the maximum number of pods per OpenShift Container Platform node host, see the Cluster Maximums.

Bare pods that are not managed by a replication controller will be not rescheduled upon node disruption.

Below is an example definition of a pod that provides a long-running service, which is actually a part of the OpenShift Container Platform infrastructure: the integrated container image registry. It demonstrates many features of pods, most of which are discussed in other topics and thus only briefly mentioned here:

Pod Object Definition (YAML)

```
apiVersion: v1
kind: Pod
metadata:
  annotations: { ... }
  labels:                                    (1)
    deployment: docker-registry-1
    deploymentconfig: docker-registry
    docker-registry: default
  generateName: docker-registry-1-           (2)
spec:
  containers:                                (3)
  - env:                                     (4)
    - name: OPENSHIFT_CA_DATA
      value: ...
    - name: OPENSHIFT_CERT_DATA
      value: ...
    - name: OPENSHIFT_INSECURE
      value: "false"
    - name: OPENSHIFT_KEY_DATA
      value: ...
    - name: OPENSHIFT_MASTER
      value: https://master.example.com:8443
    image: openshift/origin-docker-registry:v0.6.2 (5)
    imagePullPolicy: IfNotPresent
    name: registry
    ports:                                  (6)
    - containerPort: 5000
      protocol: TCP
    resources: {}
    securityContext: { ... }                (7)
    volumeMounts:                           (8)
    - mountPath: /registry
      name: registry-storage
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-br6yz
      readOnly: true
  dnsPolicy: ClusterFirst
  imagePullSecrets:
  - name: default-dockercfg-at06w
  restartPolicy: Always                     (9)
  serviceAccount: default                   (10)
  volumes:                                  (11)
  - emptyDir: {}
```

```
    name: registry-storage
  - name: default-token-br6yz
    secret:
      secretName: default-token-br6yz
```

1 Pods can be "tagged" with one or more labels, which can then be used to select and manage groups of pods in a single operation. The labels are stored in key/value format in the `metadata` hash. One label in this example is **docker-registry=default**.

2 Pods must have a unique name within their namespace. A pod definition may specify the basis of a name with the `generateName` attribute, and random characters will be added automatically to generate a unique name.

3 `containers` specifies an array of container definitions; in this case (as with most), just one.

4 Environment variables can be specified to pass necessary values to each container.

5 Each container in the pod is instantiated from its own Docker-formatted container image.

6 The container can bind to ports which will be made available on the pod's IP.

7 OpenShift Container Platform defines a security context for containers which specifies whether they are allowed to run as privileged containers, run as a user of their choice, and more. The default context is very restrictive but administrators can modify this as needed.

8 The container specifies where external storage volumes should be mounted within the container. In this case, there is a volume for storing the registry's data, and one for access to credentials the registry needs for making requests against the OpenShift Container Platform API.

9 The pod restart policy with possible values `Always`, `OnFailure`, and `Never`. The default value is `Always`.

10 Pods making requests against the OpenShift Container Platform API is a common enough pattern that there is a `serviceAccount` field for specifying which service account user the pod should authenticate as when making the requests. This enables fine-grained access control for custom infrastructure components.

11 The pod defines storage volumes that are available to its container(s) to use. In this case, it provides an ephemeral volume for the registry storage and a `secret` volume containing the service account credentials.

This pod definition does not include attributes that are filled by OpenShift Container Platform automatically after the pod is created and its lifecycle begins. The Kubernetes pod documentation has details about the functionality and purpose of pods.

Pod Restart Policy

A pod restart policy determines how OpenShift Container Platform responds when containers in that pod exit. The policy applies to all containers in that pod.

The possible values are:

- `Always` - Tries restarting a successfully exited container on the pod continuously, with an exponential back-off delay (10s, 20s, 40s) until the pod is restarted. The default is `Always`.
- `OnFailure` - Tries restarting a failed container on the pod with an exponential back-off delay (10s, 20s, 40s) capped at 5 minutes.
- `Never` - Does not try to restart exited or failed containers on the pod. Pods immediately fail and exit.

Once bound to a node, a pod will never be bound to another node. This means that a controller is necessary in order for a pod to survive node failure:

Condition	Controller Type	Restart Policy
Pods that are expected to terminate (such as batch computations)	Job	`OnFailure` or `Never`
Pods that are expected to not terminate (such as web servers)	Replication Controller	`Always`.
Pods that need to run one-per-machine	Daemonset	Any

If a container on a pod fails and the restart policy is set to `OnFailure`, the pod stays on the node and the container is restarted. If you do not want the container to restart, use a restart policy of `Never`.

If an entire pod fails, OpenShift Container Platform starts a new pod. Developers need to address the possibility that applications might be restarted in a new pod. In particular, applications need to handle temporary files, locks, incomplete output, and so forth caused by previous runs.

> Kubernetes architecture expects reliable endpoints from cloud providers. When a cloud provider is down, the kubelet prevents OpenShift Container Platform from restarting.
>
> If the underlying cloud provider endpoints are not reliable, do not install a cluster using cloud provider integration. Install the cluster as if it was in a no-cloud environment. It is not recommended to toggle cloud provider integration on or off in an installed cluster.

For details on how OpenShift Container Platform uses restart policy with failed containers, see the Example States in the Kubernetes documentation.

Init Containers

An init container is a container in a pod that is started before the pod app containers are started. Init containers can share volumes, perform network operations, and perform computations before the remaining containers start. Init containers can also block or delay the startup of application containers until some precondition is met.

When a pod starts, after the network and volumes are initialized, the init containers are started in order. Each init container must exit successfully before the next is invoked. If an init container fails to start (due to the runtime) or exits with failure, it is retried according to the pod restart policy.

A pod cannot be ready until all init containers have succeeded.

See the Kubernetes documentation for some init container usage examples.

The following example outlines a simple pod which has two init containers. The first init container waits for `myservice` and the second waits for `mydb`. Once both containers succeed, the Pod starts.

Sample Init Container Pod Object Definition (YAML)

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice (1)
    image: busybox
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting
for myservice; sleep 2; done;']
  - name: init-mydb (2)
    image: busybox
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for
mydb; sleep 2; done;']
```

1 Specifies the `myservice` container.

2 Specifies the `mydb` container.

Each init container has all of the fields of an app container except for `readinessProbe`. Init containers must exit for pod startup to continue and cannot define readiness other than completion.

Init containers can include `activeDeadlineSeconds` on the pod and `livenessProbe` on the container to prevent init containers from failing forever. The active deadline includes init containers.

Services

A Kubernetes service serves as an internal load balancer. It identifies a set of replicated pods in order to proxy the connections it receives to them. Backing pods can be added to or removed from a service arbitrarily while the service remains consistently available, enabling anything that depends on the service to refer to it at a consistent address. The default service clusterIP addresses are from the OpenShift Container Platform internal network and they are used to permit pods to access each other.

To permit external access to the service, additional `externalIP` and `ingressIP` addresses that are external to the cluster can be assigned to the service. These `externalIP` addresses can also be virtual IP addresses that provide highly available access to the service.

Services are assigned an IP address and port pair that, when accessed, proxy to an appropriate backing pod. A service uses a label selector to find all the containers running that provide a certain network service on a certain port.

Like pods, services are REST objects. The following example shows the definition of a service for the pod defined above:

Service Object Definition (YAML)

```
apiVersion: v1
kind: Service
metadata:
  name: docker-registry      (1)
spec:
  selector:                  (2)
    docker-registry: default
  clusterIP: 172.30.136.123   (3)
  ports:
  - nodePort: 0
    port: 5000               (4)
    protocol: TCP
    targetPort: 5000         (5)
```

1 The service name **docker-registry** is also used to construct an environment variable with the service IP that is inserted into other pods in the same namespace. The maximum name length is 63 characters.

2 The label selector identifies all pods with the **docker-registry=default** label attached as its backing pods.

3 Virtual IP of the service, allocated automatically at creation from a pool of internal IPs.

4 Port the service listens on.

5 Port on the backing pods to which the service forwards connections.

The Kubernetes documentation has more information on services.

Service externalIPs

In addition to the cluster's internal IP addresses, the user can configure IP addresses that are external to the cluster. The administrator is responsible for ensuring that traffic arrives at a node with this IP.

The externalIPs must be selected by the cluster administrators from the **externalIPNetworkCIDRs** range configured in ***master-config.yaml*** file. When ***master-config.yaml*** is changed, the master services must be restarted.

Sample externalIPNetworkCIDR /etc/origin/master/master-config.yaml

```
networkConfig:
  externalIPNetworkCIDRs:
  - 192.0.1.0.0/24
```

Service externalIPs Definition (JSON)

```
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "my-service"
    },
    "spec": {
        "selector": {
            "app": "MyApp"
        },
        "ports": [
            {
                "name": "http",
                "protocol": "TCP",
                "port": 80,
                "targetPort": 9376
            }
        ],
        "externalIPs" : [
            "192.0.1.1"         (1)
        ]
    }
}
```

1 List of external IP addresses on which the **port** is exposed. This list is in addition to the internal IP address list.

Service ingressIPs

In non-cloud clusters, externalIP addresses can be automatically assigned from a pool of addresses. This eliminates the need for the administrator manually assigning them.

The pool is configured in ***/etc/origin/master/master-config.yaml*** file. After changing this file, restart the master service.

The `ingressIPNetworkCIDR` is set to `172.29.0.0/16` by default. If the cluster environment is not already using this private range, use the default range or set a custom range.

If you are using high availability, then this range must be less than 256 addresses.

Sample ingressIPNetworkCIDR /etc/origin/master/master-config.yaml

```
networkConfig:
  ingressIPNetworkCIDR: 172.29.0.0/16
```

Service NodePort

Setting the service `type=NodePort` will allocate a port from a flag-configured range (default: 30000-32767), and each node will proxy that port (the same port number on every node) into your service.

The selected port will be reported in the service configuration, under `spec.ports[*].nodePort`.

To specify a custom port just place the port number in the nodePort field. The custom port number must be in the configured range for nodePorts. When '**master-config.yaml**' is changed the master services must be restarted.

Sample servicesNodePortRange /etc/origin/master/master-config.yaml

```
kubernetesMasterConfig:
  servicesNodePortRange: ""
```

The service will be visible as both the `<NodeIP>:spec.ports[].nodePort` and `spec.clusterIp:spec.ports[].port`

Setting a nodePort is a privileged operation.

Service Proxy Mode

OpenShift Container Platform has two different implementations of the service-routing infrastructure. The default implementation is entirely **iptables**-based, and uses probabilistic **iptables** rewriting rules to distribute incoming service connections between the endpoint pods. The older implementation uses a user space process to accept incoming connections and then proxy traffic between the client and one of the endpoint pods.

The **iptables**-based implementation is much more efficient, but it requires that all endpoints are always able to accept connections; the user space implementation is slower, but can try multiple endpoints in turn until it finds one that works. If you have good readiness checks (or generally reliable nodes and pods), then the **iptables**-based service proxy is the best choice. Otherwise, you can enable the user space-based proxy when installing, or after deploying the cluster by editing the node configuration file.

Headless services

If your application does not need load balancing or single-service IP addresses, you can create a headless service. When you create a headless service, no load-balancing or proxying is done and no cluster IP is allocated for this service. For such services, DNS is automatically configured depending on whether the service has selectors defined or not.

Services with selectors: For headless services that define selectors, the endpoints controller creates `Endpoints` records in the API and modifies the DNS configuration to return `A` records (addresses) that point directly to the pods backing the service.

Services without selectors: For headless services that do not define selectors, the endpoints controller does not create `Endpoints` records. However, the DNS system looks for and configures the following records:

- For `ExternalName` type services, `CNAME` records.
- For all other service types, `A` records for any endpoints that share a name with the service.

Creating a headless service

Creating a headless service is similar to creating a standard service, but you do not declare the `ClusterIP` address. To create a headless service, add the `clusterIP: None` parameter value to the service YAML definition.

For example, for a group of pods that you want to be a part of the same cluster or service.

List of Pods

```
$ oc get pods -o wide
```

Example Output

```
NAME                READY   STATUS    RESTARTS   AGE    IP
NODE
frontend-1-287hw    1/1     Running   0          7m     172.17.0.3
node_1
frontend-1-68km5    1/1     Running   0          7m     172.17.0.6
node_1
```

You can define the headless service as:

Headless Service Definition

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: ruby-helloworld-sample
    template: application-template-stibuild
  name: frontend-headless  (1)
spec:
  clusterIP: None  (2)
  ports:
  - name: web
    port: 5432
    protocol: TCP
    targetPort: 8080
  selector:
    name: frontend  (3)
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

1 Name of the headless service.
2 Setting `clusterIP` variable to `None` declares a headless service.
3 Selects all pods that have `frontend` label.

Also, headless service does not have any IP address of its own.

```
$ oc get svc
```

Example Output

```
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP
PORT(S)    AGE
frontend            ClusterIP   172.30.232.77    <none>
5432/TCP   12m
frontend-headless   ClusterIP   None             <none>
5432/TCP   10m
```

Endpoint discovery by using a headless service

The benefit of using a headless service is that you can discover a pod's IP address directly. Standard services act as load balancer or proxy and give access to the workload object by using the service name. With headless services, the service name resolves to the set of IP addresses of the pods that are grouped by the service.

When you look up the DNS `A` record for a standard service, you get the loadbalanced IP of the service.

```
$ dig frontend.test A +search +short
```

Example Output

```
172.30.232.77
```

But for a headless service, you get the list of IPs of individual pods.

```
$ dig frontend-headless.test A +search +short
```

Example Output

```
172.17.0.3
172.17.0.6
```

For using a headless service with a StatefulSet and related use cases where you need to resolve DNS for the pod during initialization and termination, set `publishNotReadyAddresses` to `true` (the default value is `false`). When `publishNotReadyAddresses` is set to `true`, it indicates that DNS implementations must publish the `notReadyAddresses` of subsets for the Endpoints associated with the Service.

Labels

Labels are used to organize, group, or select API objects. For example, pods are "tagged" with labels, and then services use label selectors to identify the pods they proxy to. This makes it possible for services to reference groups of pods, even treating pods with potentially different containers as related entities.

Most objects can include labels in their metadata. So labels can be used to group arbitrarily-related objects; for example, all of the pods, services, replication controllers, and deployment configurations of a particular application can be grouped.

Labels are simple key/value pairs, as in the following example:

```
labels:
  key1: value1
  key2: value2
```

Consider:

- A pod consisting of an **nginx** container, with the label **role=webserver**.
- A pod consisting of an **Apache httpd** container, with the same label **role=webserver**.

A service or replication controller that is defined to use pods with the **role=webserver** label treats both of these pods as part of the same group.

The Kubernetes documentation has more information on labels.

Endpoints

The servers that back a service are called its endpoints, and are specified by an object of type **Endpoints** with the same name as the service. When a service is backed by pods, those pods are normally specified by a label selector in the service specification, and OpenShift Container Platform automatically creates the Endpoints object pointing to those pods.

In some cases, you may want to create a service but have it be backed by external hosts rather than by pods in the OpenShift Container Platform cluster. In this case, you can leave out the **`selector`** field in the service, and create the Endpoints object manually.

Note that OpenShift Container Platform will not let most users manually create an Endpoints object that points to an IP address in the network blocks reserved for pod and service IPs. Only cluster admins or other users with permission to `create` resources under `endpoints/restricted` can create such Endpoint objects.