

Troubleshooting Applications

Debugging common containerized application issues.

- 1: [Debug Pods](#)
- 2: [Debug Services](#)
- 3: [Debug a StatefulSet](#)
- 4: [Debug Init Containers](#)
- 5: [Debug Running Pods](#)
- 6: [Determine the Reason for Pod Failure](#)
- 7: [Get a Shell to a Running Container](#)

This doc contains a set of resources for fixing issues with containerized applications. It covers things like common issues with Kubernetes resources (like Pods, Services, or StatefulSets), advice on making sense of container termination messages, and ways to debug running containers.

1 - Debug Pods

This guide is to help users debug applications that are deployed into Kubernetes and not behaving correctly. This is *not* a guide for people who want to debug their cluster. For that you should check out [this guide](#).

Diagnosing the problem

The first step in troubleshooting is triage. What is the problem? Is it your Pods, your Replication Controller or your Service?

- [Debugging Pods](#)
- [Debugging Replication Controllers](#)
- [Debugging Services](#)

Debugging Pods

The first step in debugging a Pod is taking a look at it. Check the current state of the Pod and recent events with the following command:

```
kubectl describe pods ${POD_NAME}
```

Look at the state of the containers in the pod. Are they all `Running` ? Have there been recent restarts?

Continue debugging depending on the state of the pods.

My pod stays pending

If a Pod is stuck in `Pending` it means that it can not be scheduled onto a node. Generally this is because there are insufficient resources of one type or another that prevent scheduling. Look at the output of the `kubectl describe ...` command above. There should be messages from the scheduler about why it can not schedule your pod. Reasons include:

- **You don't have enough resources:** You may have exhausted the supply of CPU or Memory in your cluster, in this case you need to delete Pods, adjust resource requests, or add new nodes to your cluster. See [Compute Resources document](#) for more information.
- **You are using `hostPort`:** When you bind a Pod to a `hostPort` there are a limited number of places that pod can be scheduled. In most cases, `hostPort` is unnecessary, try using a Service object to expose your Pod. If you do require `hostPort` then you can only schedule as many Pods as there are nodes in your Kubernetes cluster.

My pod stays waiting

If a Pod is stuck in the `Waiting` state, then it has been scheduled to a worker node, but it can't run on that machine. Again, the information from `kubectl describe ...` should be informative. The most common cause of `Waiting` pods is a failure to pull the image. There are three things to check:

- Make sure that you have the name of the image correct.
- Have you pushed the image to the registry?
- Try to manually pull the image to see if the image can be pulled. For example, if you use Docker on your PC, run `docker pull <image>`.

My pod is crashing or otherwise unhealthy

Once your pod has been scheduled, the methods described in [Debug Running Pods](#) are available for debugging.

My pod is running but not doing what I told it to do

If your pod is not behaving as you expected, it may be that there was an error in your pod description (e.g. `mypod.yaml` file on your local machine), and that the error was silently ignored when you created the pod. Often a section of the pod description is nested incorrectly, or a key name is typed incorrectly, and so the key is ignored. For example, if you misspelled `command` as `commnd` then the pod will be created but will not use the command line you intended it to use.

The first thing to do is to delete your pod and try creating it again with the `--validate` option. For example, run `kubectl apply --validate -f mypod.yaml`. If you misspelled `command` as `commnd` then will give an error like this:

```
I0805 10:43:25.129850 46757 schema.go:126] unknown field: commnd
I0805 10:43:25.129973 46757 schema.go:129] this may be a false alarm, see https://github.com/kubernetes/kub
pods/mypod
```

The next thing to check is whether the pod on the apiserver matches the pod you meant to create (e.g. in a yaml file on your local machine). For example, run `kubectl get pods/mypod -o yaml > mypod-on-apiserver.yaml` and then manually compare the original pod description, `mypod.yaml` with the one you got back from apiserver, `mypod-on-apiserver.yaml`. There will typically be some lines on the "apiserver" version that are not on the original version. This is expected. However, if there are lines on the original that are not on the apiserver version, then this may indicate a problem with your pod spec.

Debugging Replication Controllers

Replication controllers are fairly straightforward. They can either create Pods or they can't. If they can't create pods, then please refer to the [instructions above](#) to debug your pods.

You can also use `kubectl describe rc ${CONTROLLER_NAME}` to introspect events related to the replication controller.

Debugging Services

Services provide load balancing across a set of pods. There are several common problems that can make Services not work properly. The following instructions should help debug Service problems.

First, verify that there are endpoints for the service. For every Service object, the apiserver makes an `endpoints` resource available.

You can view this resource with:

```
kubectl get endpoints ${SERVICE_NAME}
```

Make sure that the endpoints match up with the number of pods that you expect to be members of your service. For example, if your Service is for an nginx container with 3 replicas, you would expect to see three different IP addresses in the Service's endpoints.

My service is missing endpoints

If you are missing endpoints, try listing pods using the labels that Service uses. Imagine that you have a Service where the labels are:

```
...
spec:
  - selector:
      name: nginx
      type: frontend
```

You can use:

```
kubectl get pods --selector=name=nginx,type=frontend
```

to list pods that match this selector. Verify that the list matches the Pods that you expect to provide your Service. Verify that the pod's `containerPort` matches up with the Service's `targetPort`

Network traffic is not forwarded

Please see [debugging service](#) for more information.

What's next

If none of the above solves your problem, follow the instructions in [Debugging Service document](#) to make sure that your `Service` is running, has `Endpoints`, and your `Pods` are actually serving; you have DNS working, iptables rules installed, and kube-proxy does not seem to be misbehaving.

You may also visit [troubleshooting document](#) for more information.

2 - Debug Services

An issue that comes up rather frequently for new installations of Kubernetes is that a Service is not working properly. You've run your Pods through a Deployment (or other workload controller) and created a Service, but you get no response when you try to access it. This document will hopefully help you to figure out what's going wrong.

Running commands in a Pod

For many steps here you will want to see what a Pod running in the cluster sees. The simplest way to do this is to run an interactive busybox Pod:

```
kubectl run -it --rm --restart=Never busybox --image=gcr.io/google-containers/busybox sh
```

Note: If you don't see a command prompt, try pressing enter.

If you already have a running Pod that you prefer to use, you can run a command in it using:

```
kubectl exec <POD-NAME> -c <CONTAINER-NAME> -- <COMMAND>
```

Setup

For the purposes of this walk-through, let's run some Pods. Since you're probably debugging your own Service you can substitute your own details, or you can follow along and get a second data point.

```
kubectl create deployment hostnames --image=registry.k8s.io/serve_hostname
```

```
deployment.apps/hostnames created
```

`kubectl` commands will print the type and name of the resource created or mutated, which can then be used in subsequent commands.

Let's scale the deployment to 3 replicas.

```
kubectl scale deployment hostnames --replicas=3
```

```
deployment.apps/hostnames scaled
```

Note that this is the same as if you had started the Deployment with the following YAML:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: hostnames
  name: hostnames
spec:
  selector:
    matchLabels:
      app: hostnames
  replicas: 3
  template:
    metadata:
      labels:
        app: hostnames
    spec:
      containers:
        - name: hostnames
          image: registry.k8s.io/serve_hostname

```

The label "app" is automatically set by `kubectl create deployment` to the name of the Deployment.

You can confirm your Pods are running:

```
kubectl get pods -l app=hostnames
```

NAME	READY	STATUS	RESTARTS	AGE
hostnames-632524106-bbpiw	1/1	Running	0	2m
hostnames-632524106-ly40y	1/1	Running	0	2m
hostnames-632524106-tlaok	1/1	Running	0	2m

You can also confirm that your Pods are serving. You can get the list of Pod IP addresses and test them directly.

```
kubectl get pods -l app=hostnames \
  -o go-template='{{range .items}}{{.status.podIP}}{{"\n"}}{{end}}'
```

```

10.244.0.5
10.244.0.6
10.244.0.7

```

The example container used for this walk-through serves its own hostname via HTTP on port 9376, but if you are debugging your own app, you'll want to use whatever port number your Pods are listening on.

From within a pod:

```

for ep in 10.244.0.5:9376 10.244.0.6:9376 10.244.0.7:9376; do
  wget -qO- $ep
done

```

This should produce something like:

```

hostnames-632524106-bbpiw
hostnames-632524106-ly40y
hostnames-632524106-tlaok

```

If you are not getting the responses you expect at this point, your Pods might not be healthy or might not be listening on the port you think they are. You might find `kubectl logs` to be useful for seeing what is happening, or perhaps you need to `kubectl exec` directly into your Pods and debug from there.

Assuming everything has gone to plan so far, you can start to investigate why your Service doesn't work.

Does the Service exist?

The astute reader will have noticed that you did not actually create a Service yet - that is intentional. This is a step that sometimes gets forgotten, and is the first thing to check.

What would happen if you tried to access a non-existent Service? If you have another Pod that consumes this Service by name you would get something like:

```
wget -O- hostnames
```

```
Resolving hostnames (hostnames)... failed: Name or service not known.  
wget: unable to resolve host address 'hostnames'
```

The first thing to check is whether that Service actually exists:

```
kubectl get svc hostnames
```

```
No resources found.  
Error from server (NotFound): services "hostnames" not found
```

Let's create the Service. As before, this is for the walk-through - you can use your own Service's details here.

```
kubectl expose deployment hostnames --port=80 --target-port=9376
```

```
service/hostnames exposed
```

And read it back:

```
kubectl get svc hostnames
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
hostnames	ClusterIP	10.0.1.175	<none>	80/TCP	5s

Now you know that the Service exists.

As before, this is the same as if you had started the Service with YAML:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: hostnames
  name: hostnames
spec:
  selector:
    app: hostnames
  ports:
  - name: default
    protocol: TCP
    port: 80
    targetPort: 9376
```

In order to highlight the full range of configuration, the Service you created here uses a different port number than the Pods. For many real-world Services, these values might be the same.

Any Network Policy Ingress rules affecting the target Pods?

If you have deployed any Network Policy Ingress rules which may affect incoming traffic to `hostnames-*` Pods, these need to be reviewed.

Please refer to [Network Policies](#) for more details.

Does the Service work by DNS name?

One of the most common ways that clients consume a Service is through a DNS name.

From a Pod in the same Namespace:

```
nslookup hostnames
```

```
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      hostnames
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

If this fails, perhaps your Pod and Service are in different Namespaces, try a namespace-qualified name (again, from within a Pod):

```
nslookup hostnames.default
```

```
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      hostnames.default
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

If this works, you'll need to adjust your app to use a cross-namespace name, or run your app and Service in the same Namespace. If this still fails, try a fully-qualified name:

```
nslookup hostnames.default.svc.cluster.local
```

```
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local
```

```
Name:      hostnames.default.svc.cluster.local
```

```
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

Note the suffix here: "default.svc.cluster.local". The "default" is the Namespace you're operating in. The "svc" denotes that this is a Service. The "cluster.local" is your cluster domain, which COULD be different in your own cluster.

You can also try this from a Node in the cluster:

Note: 10.0.0.10 is the cluster's DNS Service IP, yours might be different.

```
nslookup hostnames.default.svc.cluster.local 10.0.0.10
```

```
Server:      10.0.0.10
```

```
Address:     10.0.0.10#53
```

```
Name:  hostnames.default.svc.cluster.local
```

```
Address: 10.0.1.175
```

If you are able to do a fully-qualified name lookup but not a relative one, you need to check that your `/etc/resolv.conf` file in your Pod is correct. From within a Pod:

```
cat /etc/resolv.conf
```

You should see something like:

```
nameserver 10.0.0.10
```

```
search default.svc.cluster.local svc.cluster.local cluster.local example.com
```

```
options ndots:5
```

The `nameserver` line must indicate your cluster's DNS Service. This is passed into `kubelet` with the `--cluster-dns` flag.

The `search` line must include an appropriate suffix for you to find the Service name. In this case it is looking for Services in the local Namespace ("default.svc.cluster.local"), Services in all Namespaces ("svc.cluster.local"), and lastly for names in the cluster ("cluster.local"). Depending on your own install you might have additional records after that (up to 6 total). The cluster suffix is passed into `kubelet` with the `--cluster-domain` flag. Throughout this document, the cluster suffix is assumed to be "cluster.local". Your own clusters might be configured differently, in which case you should change that in all of the previous commands.

The `options` line must set `ndots` high enough that your DNS client library considers search paths at all. Kubernetes sets this to 5 by default, which is high enough to cover all of the DNS names it generates.

Does any Service work by DNS name?

If the above still fails, DNS lookups are not working for your Service. You can take a step back and see what else is not working. The Kubernetes master Service should always work. From within a Pod:

```
nslookup kubernetes.default
```



```
Server:      10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:        kubernetes.default
Address 1: 10.0.0.1 kubernetes.default.svc.cluster.local
```

If this fails, please see the [kube-proxy](#) section of this document, or even go back to the top of this document and start over, but instead of debugging your own Service, debug the DNS Service.

Does the Service work by IP?

Assuming you have confirmed that DNS works, the next thing to test is whether your Service works by its IP address. From a Pod in your cluster, access the Service's IP (from `kubectl get` above).

```
for i in $(seq 1 3); do
  wget -qO- 10.0.1.175:80
done
```

This should produce something like:

```
hostnames-632524106-bbpiw
hostnames-632524106-ly40y
hostnames-632524106-tlaok
```

If your Service is working, you should get correct responses. If not, there are a number of things that could be going wrong. Read on.

Is the Service defined correctly?

It might sound silly, but you should really double and triple check that your Service is correct and matches your Pod's port. Read back your Service and verify it:

```
kubectl get service hostnames -o json
```

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "hostnames",
    "namespace": "default",
    "uid": "428c8b6c-24bc-11e5-936d-42010af0a9bc",
    "resourceVersion": "347189",
    "creationTimestamp": "2015-07-07T15:24:29Z",
    "labels": {
      "app": "hostnames"
    }
  },
  "spec": {
    "ports": [
      {
        "name": "default",
        "protocol": "TCP",
        "port": 80,
        "targetPort": 9376,
        "nodePort": 0
      }
    ],
    "selector": {
      "app": "hostnames"
    },
    "clusterIP": "10.0.1.175",
    "type": "ClusterIP",
    "sessionAffinity": "None"
  },
  "status": {
    "loadBalancer": {}
  }
}
```

- Is the Service port you are trying to access listed in `spec.ports`?
- Is the `targetPort` correct for your Pods (some Pods use a different port than the Service)?
- If you meant to use a numeric port, is it a number (9376) or a string "9376"?
- If you meant to use a named port, do your Pods expose a port with the same name?
- Is the port's `protocol` correct for your Pods?

Does the Service have any Endpoints?

If you got this far, you have confirmed that your Service is correctly defined and is resolved by DNS. Now let's check that the Pods you ran are actually being selected by the Service.

Earlier you saw that the Pods were running. You can re-check that:

```
kubectl get pods -l app=hostnames
```

NAME	READY	STATUS	RESTARTS	AGE
hostnames-632524106-bbpiw	1/1	Running	0	1h
hostnames-632524106-ly40y	1/1	Running	0	1h
hostnames-632524106-tlaok	1/1	Running	0	1h

The `-l app=hostnames` argument is a label selector configured on the Service.

The "AGE" column says that these Pods are about an hour old, which implies that they are running fine and not crashing.

The "RESTARTS" column says that these pods are not crashing frequently or being restarted. Frequent restarts could lead to intermittent connectivity issues. If the restart count is high, read more about how to [debug pods](#).

Inside the Kubernetes system is a control loop which evaluates the selector of every Service and saves the results into a corresponding Endpoints object.

```
kubectl get endpoints hostnames
```

NAME	ENDPOINTS
hostnames	10.244.0.5:9376,10.244.0.6:9376,10.244.0.7:9376

This confirms that the endpoints controller has found the correct Pods for your Service. If the `ENDPOINTS` column is `<none>`, you should check that the `spec.selector` field of your Service actually selects for `metadata.labels` values on your Pods. A common mistake is to have a typo or other error, such as the Service selecting for `app=hostnames`, but the Deployment specifying `run=hostnames`, as in versions previous to 1.18, where the `kubectl run` command could have been also used to create a Deployment.

Are the Pods working?

At this point, you know that your Service exists and has selected your Pods. At the beginning of this walk-through, you verified the Pods themselves. Let's check again that the Pods are actually working - you can bypass the Service mechanism and go straight to the Pods, as listed by the Endpoints above.

Note: These commands use the Pod port (9376), rather than the Service port (80).

From within a Pod:

```
for ep in 10.244.0.5:9376 10.244.0.6:9376 10.244.0.7:9376; do
  wget -qO- $ep
done
```

This should produce something like:

```
hostnames-632524106-bbpiw
hostnames-632524106-ly40y
hostnames-632524106-tlaok
```

You expect each Pod in the Endpoints list to return its own hostname. If this is not what happens (or whatever the correct behavior is for your own Pods), you should investigate what's happening there.

Is the kube-proxy working?

If you get here, your Service is running, has Endpoints, and your Pods are actually serving. At this point, the whole Service proxy mechanism is suspect. Let's confirm it, piece by piece.

The default implementation of Services, and the one used on most clusters, is kube-proxy. This is a program that runs on every node and configures one of a small set of mechanisms for providing the Service abstraction. If your cluster does not use kube-proxy, the following sections will not apply, and you will have to investigate whatever implementation of Services you are using.

Is kube-proxy running?

Confirm that `kube-proxy` is running on your Nodes. Running directly on a Node, you should get something like the below:

```
ps auxw | grep kube-proxy
```

```
root 4194 0.4 0.1 101864 17696 ? Sl Jul04 25:43 /usr/local/bin/kube-proxy --master=https://kubernetes-
```

Next, confirm that it is not failing something obvious, like contacting the master. To do this, you'll have to look at the logs. Accessing the logs depends on your Node OS. On some OSes it is a file, such as `/var/log/kube-proxy.log`, while other OSes use `journalctl` to access logs. You should see something like:

```
I1027 22:14:53.995134 5063 server.go:200] Running in resource-only container "/kube-proxy"
I1027 22:14:53.998163 5063 server.go:247] Using iptables Proxier.
I1027 22:14:53.999055 5063 server.go:255] Tearing down userspace rules. Errors here are acceptable.
I1027 22:14:54.038140 5063 proxier.go:352] Setting endpoints for "kube-system/kube-dns:dns-tcp" to [10.244.
I1027 22:14:54.038164 5063 proxier.go:352] Setting endpoints for "kube-system/kube-dns:dns" to [10.244.1.3
I1027 22:14:54.038209 5063 proxier.go:352] Setting endpoints for "default/kubernetes:https" to [10.240.0.2
I1027 22:14:54.038238 5063 proxier.go:429] Not syncing iptables until Services and Endpoints have been rec
I1027 22:14:54.040048 5063 proxier.go:294] Adding new service "default/kubernetes:https" at 10.0.0.1:443/T
I1027 22:14:54.040154 5063 proxier.go:294] Adding new service "kube-system/kube-dns:dns" at 10.0.0.10:53/U
I1027 22:14:54.040223 5063 proxier.go:294] Adding new service "kube-system/kube-dns:dns-tcp" at 10.0.0.10:
```

If you see error messages about not being able to contact the master, you should double-check your Node configuration and installation steps.

One of the possible reasons that `kube-proxy` cannot run correctly is that the required `conntrack` binary cannot be found. This may happen on some Linux systems, depending on how you are installing the cluster, for example, you are installing Kubernetes from scratch. If this is the case, you need to manually install the `conntrack` package (e.g. `sudo apt install conntrack` on Ubuntu) and then retry.

Kube-proxy can run in one of a few modes. In the log listed above, the line `Using iptables Proxier` indicates that kube-proxy is running in "iptables" mode. The most common other mode is "ipvs". The older "userspace" mode has largely been replaced by these.

Iptables mode

In "iptables" mode, you should see something like the following on a Node:

```
iptables-save | grep hostnames
```

```
-A KUBE-SEP-57KPRZ3JQVENLNBR -s 10.244.3.6/32 -m comment --comment "default/hostnames:" -j MARK --set-xmark 0
-A KUBE-SEP-57KPRZ3JQVENLNBR -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT --to-destination
-A KUBE-SEP-WNBA2IHDGP2B0BGZ -s 10.244.1.7/32 -m comment --comment "default/hostnames:" -j MARK --set-xmark 0
-A KUBE-SEP-WNBA2IHDGP2B0BGZ -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT --to-destination
-A KUBE-SEP-X3P2623AGDH6CDF3 -s 10.244.2.3/32 -m comment --comment "default/hostnames:" -j MARK --set-xmark 0
-A KUBE-SEP-X3P2623AGDH6CDF3 -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT --to-destination
-A KUBE-SERVICES -d 10.0.1.175/32 -p tcp -m comment --comment "default/hostnames: cluster IP" -m tcp --dport
-A KUBE-SVC-NWV5X2332I40T4T3 -m comment --comment "default/hostnames:" -m statistic --mode random --probabili
-A KUBE-SVC-NWV5X2332I40T4T3 -m comment --comment "default/hostnames:" -m statistic --mode random --probabili
-A KUBE-SVC-NWV5X2332I40T4T3 -m comment --comment "default/hostnames:" -j KUBE-SEP-57KPRZ3JQVENLNBR
```

For each port of each Service, there should be 1 rule in `KUBE-SERVICES` and one `KUBE-SVC-<hash>` chain. For each Pod endpoint, there should be a small number of rules in that `KUBE-SVC-<hash>` and one `KUBE-SEP-<hash>` chain with a small number of rules in it. The exact rules will vary based on your exact config (including node-ports and load-balancers).

IPVS mode

In "ipvs" mode, you should see something like the following on a Node:

```
ipvsadm -ln
```

```
Prot LocalAddress:Port Scheduler Flags
-> RemoteAddress:Port      Forward Weight ActiveConn InActConn
...
TCP 10.0.1.175:80 rr
-> 10.244.0.5:9376          Masq    1      0      0
-> 10.244.0.6:9376          Masq    1      0      0
-> 10.244.0.7:9376          Masq    1      0      0
...
```

For each port of each Service, plus any NodePorts, external IPs, and load-balancer IPs, kube-proxy will create a virtual server. For each Pod endpoint, it will create corresponding real servers. In this example, service hostnames(`10.0.1.175:80`) has 3 endpoints(`10.244.0.5:9376` , `10.244.0.6:9376` , `10.244.0.7:9376`).

Userspace mode

In rare cases, you may be using "userspace" mode. From your Node:

```
iptables-save | grep hostnames
```

```
-A KUBE-PORTALS-CONTAINER -d 10.0.1.175/32 -p tcp -m comment --comment "default/hostnames:default" -m tcp --d
-A KUBE-PORTALS-HOST -d 10.0.1.175/32 -p tcp -m comment --comment "default/hostnames:default" -m tcp --dport
```

There should be 2 rules for each port of your Service (only one in this example) - a "KUBE-PORTALS-CONTAINER" and a "KUBE-PORTALS-HOST".

Almost nobody should be using the "userspace" mode any more, so you won't spend more time on it here.

Is kube-proxy proxying?

Assuming you do see one the above cases, try again to access your Service by IP from one of your Nodes:

```
curl 10.0.1.175:80
```

```
hostnames-632524106-bbpiw
```

If this fails and you are using the userspace proxy, you can try accessing the proxy directly. If you are using the iptables proxy, skip this section.

Look back at the `iptables-save` output above, and extract the port number that `kube-proxy` is using for your Service. In the above examples it is "48577". Now connect to that:

```
curl localhost:48577
```

```
hostnames-632524106-tlaok
```

If this still fails, look at the `kube-proxy` logs for specific lines like:

```
Setting endpoints for default/hostnames:default to [10.244.0.5:9376 10.244.0.6:9376 10.244.0.7:9376]
```

If you don't see those, try restarting `kube-proxy` with the `-v` flag set to 4, and then look at the logs again.

Edge case: A Pod fails to reach itself via the Service IP

This might sound unlikely, but it does happen and it is supposed to work.

This can happen when the network is not properly configured for "hairpin" traffic, usually when `kube-proxy` is running in `iptables` mode and Pods are connected with bridge network. The `Kubelet` exposes a `hairpin-mode` flag that allows endpoints of a Service to loadbalance back to themselves if they try to access their own Service VIP. The `hairpin-mode` flag must either be set to `hairpin-veth` or `promiscuous-bridge`.

The common steps to trouble shoot this are as follows:

- Confirm `hairpin-mode` is set to `hairpin-veth` or `promiscuous-bridge`. You should see something like the below. `hairpin-mode` is set to `promiscuous-bridge` in the following example.

```
ps auxw | grep kubelet
```

```
root      3392  1.1  0.8 186804 65208 ?        Sl   00:51   11:11 /usr/local/bin/kubelet --enable-debugging-ha
```

- Confirm the effective `hairpin-mode`. To do this, you'll have to look at kubelet log. Accessing the logs depends on your Node OS. On some OSes it is a file, such as `/var/log/kubelet.log`, while other OSes use `journalctl` to access logs. Please be noted that the effective hairpin mode may not match `--hairpin-mode` flag due to compatibility. Check if there is any log lines with key word `hairpin` in `kubelet.log`. There should be log lines indicating the effective hairpin mode, like something below.

```
I0629 00:51:43.648698    3252 kubelet.go:380] Hairpin mode set to "promiscuous-bridge"
```

- If the effective hairpin mode is `hairpin-veth`, ensure the `Kubelet` has the permission to operate in `/sys` on node. If everything works properly, you should see something like:

```
for intf in /sys/devices/virtual/net/cbr0/brif/*; do cat ${intf}/hairpin_mode; done
```

```
1
1
1
1
```

- If the effective hairpin mode is `promiscuous-bridge`, ensure `Kubelet` has the permission to manipulate linux bridge on node. If `cbr0` bridge is used and configured properly, you should see:

```
ifconfig cbr0 |grep PROMISC
```

```
UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1460  Metric:1
```

- Seek help if none of above works out.

Seek help

If you get this far, something very strange is happening. Your Service is running, has Endpoints, and your Pods are actually serving. You have DNS working, and `kube-proxy` does not seem to be misbehaving. And yet your Service is not working. Please let us know what is going on, so we can help investigate!

Contact us on [Slack](#) or [Forum](#) or [GitHub](#).

What's next

Visit the [troubleshooting overview document](#) for more information.

3 - Debug a StatefulSet

This task shows you how to debug a StatefulSet.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster.
- You should have a StatefulSet running that you want to investigate.

Debugging a StatefulSet

In order to list all the pods which belong to a StatefulSet, which have a label `app.kubernetes.io/name=MyApp` set on them, you can use the following:

```
kubectl get pods -l app.kubernetes.io/name=MyApp
```

If you find that any Pods listed are in `Unknown` or `Terminating` state for an extended period of time, refer to the [Deleting StatefulSet Pods](#) task for instructions on how to deal with them. You can debug individual Pods in a StatefulSet using the [Debugging Pods](#) guide.

What's next

Learn more about [debugging an init-container](#).

4 - Debug Init Containers

This page shows how to investigate problems related to the execution of Init Containers. The example command lines below refer to the Pod as `<pod-name>` and the Init Containers as `<init-container-1>` and `<init-container-2>`.

Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Killercodea](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

- You should be familiar with the basics of [Init Containers](#).
- You should have [Configured an Init Container](#).

Checking the status of Init Containers

Display the status of your pod:

```
kubectl get pod <pod-name>
```

For example, a status of `Init:1/2` indicates that one of two Init Containers has completed successfully:

NAME	READY	STATUS	RESTARTS	AGE
<pod-name>	0/1	Init:1/2	0	7s

See [Understanding Pod status](#) for more examples of status values and their meanings.

Getting details about Init Containers

View more detailed information about Init Container execution:

```
kubectl describe pod <pod-name>
```

For example, a Pod with two Init Containers might show the following:

```
Init Containers:
  <init-container-1>:
    Container ID:   ...
    ...
    State:          Terminated
      Reason:        Completed
      Exit Code:     0
      Started:       ...
      Finished:      ...
    Ready:          True
    Restart Count:  0
    ...
  <init-container-2>:
    Container ID:   ...
    ...
    State:          Waiting
      Reason:        CrashLoopBackOff
    Last State:     Terminated
      Reason:        Error
      Exit Code:     1
      Started:       ...
      Finished:      ...
    Ready:          False
    Restart Count:  3
    ...
```

You can also access the Init Container statuses programmatically by reading the `status.initContainerStatuses` field on the Pod Spec:

```
kubectl get pod nginx --template '{{.status.initContainerStatuses}}'
```

This command will return the same information as above in raw JSON.

Accessing logs from Init Containers

Pass the Init Container name along with the Pod name to access its logs.

```
kubectl logs <pod-name> -c <init-container-2>
```

Init Containers that run a shell script print commands as they're executed. For example, you can do this in Bash by running `set -x` at the beginning of the script.

Understanding Pod status

A Pod status beginning with `Init:` summarizes the status of Init Container execution. The table below describes some example status values that you might see while debugging Init Containers.

Status	Meaning
Init:N/M	The Pod has <code>M</code> Init Containers, and <code>N</code> have completed so far.
Init:Error	An Init Container has failed to execute.
Init:CrashLoopBackOff	An Init Container has failed repeatedly.

Status	Meaning
Pending	The Pod has not yet begun executing Init Containers.
PodInitializing or Running	The Pod has already finished executing Init Containers.

5 - Debug Running Pods

This page explains how to debug Pods running (or crashing) on a Node.

Before you begin

- Your Pod should already be scheduled and running. If your Pod is not yet running, start with [Debugging Pods](#).
- For some of the advanced debugging steps you need to know on which Node the Pod is running and have shell access to run commands on that Node. You don't need that access to run the standard debug steps that use `kubectl`.

Using `kubectl describe pod` to fetch details about pods

For this example we'll use a Deployment to create two pods, similar to the earlier example.

[application/nginx-with-request.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 80
```

Create deployment by running following command:

```
kubectl apply -f https://k8s.io/examples/application/nginx-with-request.yaml
```

```
deployment.apps/nginx-deployment created
```

Check pod status by following command:

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-67d4bdd6f5-cx2nz	1/1	Running	0	13s
nginx-deployment-67d4bdd6f5-w6kd7	1/1	Running	0	13s

We can retrieve a lot more information about each of these pods using `kubectl describe pod`. For example:

```
kubectl describe pod nginx-deployment-67d4bdd6f5-w6kd7
```

```

Name:      nginx-deployment-67d4bdd6f5-w6kd7
Namespace: default
Priority:   0
Node:      kube-worker-1/192.168.0.113
Start Time: Thu, 17 Feb 2022 16:51:01 -0500
Labels:    app=nginx
           pod-template-hash=67d4bdd6f5
Annotations: <none>
Status:     Running
IP:         10.88.0.3
IPs:
  IP:       10.88.0.3
  IP:       2001:db8::1
Controlled By: ReplicaSet/nginx-deployment-67d4bdd6f5
Containers:
  nginx:
    Container ID:   containerd://5403af59a2b46ee5a23fb0ae4b1e077f7ca5c5fb7af16e1ab21c00e0e616462a
    Image:          nginx
    Image ID:       docker.io/library/nginx@sha256:2834dc507516af02784808c5f48b7cbe38b8ed5d0f4837f16e78d00deb
    Port:          80/TCP
    Host Port:     0/TCP
    State:          Running
      Started:      Thu, 17 Feb 2022 16:51:05 -0500
    Ready:          True
    Restart Count:  0
    Limits:
      cpu:          500m
      memory:       128Mi
    Requests:
      cpu:          500m
      memory:       128Mi
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-bgsgp (ro)
Conditions:
  Type            Status
  Initialized      True
  Ready            True
  ContainersReady  True
  PodScheduled     True
Volumes:
  kube-api-access-bgsgp:
    Type:          Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds: 3607
    ConfigMapName:    kube-root-ca.crt
    ConfigMapOptional: <nil>
    DownwardAPI:      true
QoS Class:         Guaranteed
Node-Selectors:    <none>
Tolerations:       node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                   node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason      Age   From          Message
  ----    -
  Normal  Scheduled   34s   default-scheduler  Successfully assigned default/nginx-deployment-67d4bdd6f5-w6kd7
  Normal  Pulling     31s   kubelet        Pulling image "nginx"
  Normal  Pulled      30s   kubelet        Successfully pulled image "nginx" in 1.146417389s
  Normal  Created     30s   kubelet        Created container nginx
  Normal  Started     30s   kubelet        Started container nginx

```

Here you can see configuration information about the container(s) and Pod (labels, resource requirements, etc.), as well as status information about the container(s) and Pod (state, readiness, restart count, events, etc.).

The container state is one of Waiting, Running, or Terminated. Depending on the state, additional information will be provided -- here you can see that for a container in Running state, the system tells you when the container started.

Ready tells you whether the container passed its last readiness probe. (In this case, the container does not have a readiness

probe configured; the container is assumed to be ready if no readiness probe is configured.)

Restart Count tells you how many times the container has been restarted; this information can be useful for detecting crash loops in containers that are configured with a restart policy of 'always.'

Currently the only Condition associated with a Pod is the binary Ready condition, which indicates that the pod is able to service requests and should be added to the load balancing pools of all matching services.

Lastly, you see a log of recent events related to your Pod. The system compresses multiple identical events by indicating the first and last time it was seen and the number of times it was seen. "From" indicates the component that is logging the event, "SubobjectPath" tells you which object (e.g. container within the pod) is being referred to, and "Reason" and "Message" tell you what happened.

Example: debugging Pending Pods

A common scenario that you can detect using events is when you've created a Pod that won't fit on any node. For example, the Pod might request more resources than are free on any node, or it might specify a label selector that doesn't match any nodes. Let's say we created the previous Deployment with 5 replicas (instead of 2) and requesting 600 millicores instead of 500, on a four-node cluster where each (virtual) machine has 1 CPU. In that case one of the Pods will not be able to schedule. (Note that because of the cluster addon pods such as fluentd, skydns, etc., that run on each node, if we requested 1000 millicores then none of the Pods would be able to schedule.)

```
kubect1 get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-1006230814-6winp	1/1	Running	0	7m
nginx-deployment-1006230814-fmgu3	1/1	Running	0	7m
nginx-deployment-1370807587-6ekbw	1/1	Running	0	1m
nginx-deployment-1370807587-fg172	0/1	Pending	0	1m
nginx-deployment-1370807587-fz9sd	0/1	Pending	0	1m

To find out why the nginx-deployment-1370807587-fz9sd pod is not running, we can use `kubect1 describe pod` on the pending Pod and look at its events:

```
kubect1 describe pod nginx-deployment-1370807587-fz9sd
```

```

Name:      nginx-deployment-1370807587-fz9sd
Namespace: default
Node:      /
Labels:    app=nginx,pod-template-hash=1370807587
Status:    Pending
IP:
Controllers: ReplicaSet/nginx-deployment-1370807587
Containers:
  nginx:
    Image:      nginx
    Port:      80/TCP
    QoS Tier:
      memory: Guaranteed
      cpu:     Guaranteed
    Limits:
      cpu:      1
      memory: 128Mi
    Requests:
      cpu:      1
      memory: 128Mi
    Environment Variables:
Volumes:
  default-token-4bcbi:
    Type:      Secret (a volume populated by a Secret)
    SecretName: default-token-4bcbi
Events:
  FirstSeen    LastSeen    Count   From              SubobjectPath  Type            Reason
  -----
  1m           48s         7       {default-scheduler }           Warning
  fit failure on node (kubernetes-node-6ta5): Node didn't have enough resource: CPU, requested: 1000, used: 1
  fit failure on node (kubernetes-node-wul5): Node didn't have enough resource: CPU, requested: 1000, used: 1

```

Here you can see the event generated by the scheduler saying that the Pod failed to schedule for reason `FailedScheduling` (and possibly others). The message tells us that there were not enough resources for the Pod on any of the nodes.

To correct this situation, you can use `kubectl scale` to update your Deployment to specify four or fewer replicas. (Or you could leave the one Pod pending, which is harmless.)

Events such as the ones you saw at the end of `kubectl describe pod` are persisted in etcd and provide high-level information on what is happening in the cluster. To list all events you can use

```
kubectl get events
```

but you have to remember that events are namespaced. This means that if you're interested in events for some namespaced object (e.g. what happened with Pods in namespace `my-namespace`) you need to explicitly provide a namespace to the command:

```
kubectl get events --namespace=my-namespace
```

To see events from all namespaces, you can use the `--all-namespaces` argument.

In addition to `kubectl describe pod`, another way to get extra information about a pod (beyond what is provided by `kubectl get pod`) is to pass the `-o yaml` output format flag to `kubectl get pod`. This will give you, in YAML format, even more information than `kubectl describe pod` --essentially all of the information the system has about the Pod. Here you will see things like annotations (which are key-value metadata without the label restrictions, that is used internally by Kubernetes system components), restart policy, ports, and volumes.

```
kubectl get pod nginx-deployment-1006230814-6winp -o yaml
```



```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2022-02-17T21:51:01Z"
  generateName: nginx-deployment-67d4bdd6f5-
  labels:
    app: nginx
    pod-template-hash: 67d4bdd6f5
  name: nginx-deployment-67d4bdd6f5-w6kd7
  namespace: default
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicaSet
    name: nginx-deployment-67d4bdd6f5
    uid: 7d41dfd4-84c0-4be4-88ab-cedbe626ad82
  resourceVersion: "1364"
  uid: a6501da1-0447-4262-98eb-c03d4002222e
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    ports:
    - containerPort: 80
      protocol: TCP
    resources:
      limits:
        cpu: 500m
        memory: 128Mi
      requests:
        cpu: 500m
        memory: 128Mi
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: kube-api-access-bgsgp
      readOnly: true
  dnsPolicy: ClusterFirst
  enableServiceLinks: true
  nodeName: kube-worker-1
  preemptionPolicy: PreemptLowerPriority
  priority: 0
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  tolerations:
  - effect: NoExecute
    key: node.kubernetes.io/not-ready
    operator: Exists
    tolerationSeconds: 300
  - effect: NoExecute
    key: node.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 300
  volumes:
  - name: kube-api-access-bgsgp
    projected:
      defaultMode: 420
      sources:
      - serviceAccountToken:
          expirationSeconds: 3607
          path: token
```

```

- configMap:
  items:
    - key: ca.crt
      path: ca.crt
      name: kube-root-ca.crt
- downwardAPI:
  items:
    - fieldRef:
        apiVersion: v1
        fieldPath: metadata.namespace
      path: namespace
status:
  conditions:
    - lastProbeTime: null
      lastTransitionTime: "2022-02-17T21:51:01Z"
      status: "True"
      type: Initialized
    - lastProbeTime: null
      lastTransitionTime: "2022-02-17T21:51:06Z"
      status: "True"
      type: Ready
    - lastProbeTime: null
      lastTransitionTime: "2022-02-17T21:51:06Z"
      status: "True"
      type: ContainersReady
    - lastProbeTime: null
      lastTransitionTime: "2022-02-17T21:51:01Z"
      status: "True"
      type: PodScheduled
  containerStatuses:
    - containerID: containerd://5403af59a2b46ee5a23fb0ae4b1e077f7ca5c5fb7af16e1ab21c00e0e616462a
      image: docker.io/library/nginx:latest
      imageID: docker.io/library/nginx@sha256:2834dc507516af02784808c5f48b7cbe38b8ed5d0f4837f16e78d00deb7e7767
      lastState: {}
      name: nginx
      ready: true
      restartCount: 0
      started: true
      state:
        running:
          startedAt: "2022-02-17T21:51:05Z"
  hostIP: 192.168.0.113
  phase: Running
  podIP: 10.88.0.3
  podIPs:
    - ip: 10.88.0.3
    - ip: 2001:db8::1
  qosClass: Guaranteed
  startTime: "2022-02-17T21:51:01Z"

```

Examining pod logs

First, look at the logs of the affected container:

```
kubectl logs ${POD_NAME} ${CONTAINER_NAME}
```

If your container has previously crashed, you can access the previous container's crash log with:

```
kubectl logs --previous ${POD_NAME} ${CONTAINER_NAME}
```

Debugging with container exec

If the `container image` includes debugging utilities, as is the case with images built from Linux and Windows OS base images, you can run commands inside a specific container with `kubectl exec` :

```
kubectl exec ${POD_NAME} -c ${CONTAINER_NAME} -- ${CMD} ${ARG1} ${ARG2} ... ${ARGN}
```

Note: `-c ${CONTAINER_NAME}` is optional. You can omit it for Pods that only contain a single container.

As an example, to look at the logs from a running Cassandra pod, you might run

```
kubectl exec cassandra -- cat /var/log/cassandra/system.log
```

You can run a shell that's connected to your terminal using the `-i` and `-t` arguments to `kubectl exec` , for example:

```
kubectl exec -it cassandra -- sh
```

For more details, see [Get a Shell to a Running Container](#).

Debugging with an ephemeral debug container

FEATURE STATE: [Kubernetes v1.25](#) [\[stable\]](#)

Ephemeral containers are useful for interactive troubleshooting when `kubectl exec` is insufficient because a container has crashed or a container image doesn't include debugging utilities, such as with [distroless images](#).

Example debugging using ephemeral containers

You can use the `kubectl debug` command to add ephemeral containers to a running Pod. First, create a pod for the example:

```
kubectl run ephemeral-demo --image=registry.k8s.io/pause:3.1 --restart=Never
```

The examples in this section use the `pause` container image because it does not contain debugging utilities, but this method works with all container images.

If you attempt to use `kubectl exec` to create a shell you will see an error because there is no shell in this container image.

```
kubectl exec -it ephemeral-demo -- sh
```

```
OCI runtime exec failed: exec failed: container_linux.go:346: starting container process caused "exec: \"sh\":
```

You can instead add a debugging container using `kubectl debug` . If you specify the `-i / --interactive` argument, `kubectl` will automatically attach to the console of the Ephemeral Container.

```
kubectl debug -it ephemeral-demo --image=busybox:1.28 --target=ephemeral-demo
```

```
Defaulting debug container name to debugger-8xzrl.  
If you don't see a command prompt, try pressing enter.  
/ #
```

This command adds a new busybox container and attaches to it. The `--target` parameter targets the process namespace of another container. It's necessary here because `kubectl run` does not enable [process namespace sharing](#) in the pod it creates.

Note: The `--target` parameter must be supported by the Container Runtime. When not supported, the Ephemeral Container may not be started, or it may be started with an isolated process namespace so that `ps` does not reveal processes in other containers.

You can view the state of the newly created ephemeral container using `kubectl describe`:

```
kubectl describe pod ephemeral-demo
```

```
...  
Ephemeral Containers:  
  debugger-8xzrl:  
    Container ID:   docker://b888f9adfd15bd5739fefaa39e1df4dd3c617b9902082b1cfdc29c4028ffb2eb  
    Image:          busybox  
    Image ID:       docker-pullable://busybox@sha256:1828edd60c5efd34b2bf5dd3282ec0cc04d47b2ff9caa0b6d4f07a21  
    Port:           <none>  
    Host Port:      <none>  
    State:          Running  
      Started:      Wed, 12 Feb 2020 14:25:42 +0100  
    Ready:          False  
    Restart Count:  0  
    Environment:    <none>  
    Mounts:         <none>  
...
```

Use `kubectl delete` to remove the Pod when you're finished:

```
kubectl delete pod ephemeral-demo
```

Debugging using a copy of the Pod

Sometimes Pod configuration options make it difficult to troubleshoot in certain situations. For example, you can't run `kubectl exec` to troubleshoot your container if your container image does not include a shell or if your application crashes on startup. In these situations you can use `kubectl debug` to create a copy of the Pod with configuration values changed to aid debugging.

Copying a Pod while adding a new container

Adding a new container can be useful when your application is running but not behaving as you expect and you'd like to add additional troubleshooting utilities to the Pod.

For example, maybe your application's container images are built on `busybox` but you need debugging utilities not included in `busybox`. You can simulate this scenario using `kubectl run`:

```
kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d
```

Run this command to create a copy of `myapp` named `myapp-debug` that adds a new Ubuntu container for debugging:

```
kubectl debug myapp -it --image=ubuntu --share-processes --copy-to=myapp-debug
```

Defaulting debug container name to debugger-w7xmf.
If you don't see a command prompt, try pressing enter.
root@myapp-debug:/#

Note:

- `kubectl debug` automatically generates a container name if you don't choose one using the `--container` flag.
- The `-i` flag causes `kubectl debug` to attach to the new container by default. You can prevent this by specifying `--attach=false`. If your session becomes disconnected you can reattach using `kubectl attach`.
- The `--share-processes` allows the containers in this Pod to see processes from the other containers in the Pod. For more information about how this works, see [Share Process Namespace between Containers in a Pod](#).

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod myapp myapp-debug
```

Copying a Pod while changing its command

Sometimes it's useful to change the command for a container, for example to add a debugging flag or because the application is crashing.

To simulate a crashing application, use `kubectl run` to create a container that immediately exits:

```
kubectl run --image=busybox:1.28 myapp -- false
```

You can see using `kubectl describe pod myapp` that this container is crashing:

```
Containers:
  myapp:
    Image:          busybox
    ...
    Args:
      false
    State:          Waiting
      Reason:       CrashLoopBackOff
    Last State:     Terminated
      Reason:       Error
      Exit Code:    1
```

You can use `kubectl debug` to create a copy of this Pod with the command changed to an interactive shell:

```
kubectl debug myapp -it --copy-to=myapp-debug --container=myapp -- sh
```

If you don't see a command prompt, try pressing enter.
/ #

Now you have an interactive shell that you can use to perform tasks like checking filesystem paths or running the container command manually.

Note:

- To change the command of a specific container you must specify its name using `--container` or `kubectl debug` will instead create a new container to run the command you specified.
- The `-i` flag causes `kubectl debug` to attach to the container by default. You can prevent this by specifying `--attach=false`. If your session becomes disconnected you can reattach using `kubectl attach`.

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod myapp myapp-debug
```

Copying a Pod while changing container images

In some situations you may want to change a misbehaving Pod from its normal production container images to an image containing a debugging build or additional utilities.

As an example, create a Pod using `kubectl run`:

```
kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d
```

Now use `kubectl debug` to make a copy and change its container image to `ubuntu`:

```
kubectl debug myapp --copy-to=myapp-debug --set-image=*=ubuntu
```

The syntax of `--set-image` uses the same `container_name=image` syntax as `kubectl set image`. `*=ubuntu` means change the image of all containers to `ubuntu`.

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod myapp myapp-debug
```

Debugging via a shell on the node

If none of these approaches work, you can find the Node on which the Pod is running and create a Pod running on the Node. To create an interactive shell on a Node using `kubectl debug`, run:

```
kubectl debug node/mynode -it --image=ubuntu
```

```
Creating debugging pod node-debugger-mynode-pdx84 with container debugger on node mynode.  
If you don't see a command prompt, try pressing enter.  
root@ek8s:/#
```

When creating a debugging session on a node, keep in mind that:

- `kubectl debug` automatically generates the name of the new Pod based on the name of the Node.
- The root filesystem of the Node will be mounted at `/host`.
- The container runs in the host IPC, Network, and PID namespaces, although the pod isn't privileged, so reading some process information may fail, and `chroot /host` will fail.
- If you need a privileged pod, create it manually.

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod node-debugger-mynode-pdx84
```

6 - Determine the Reason for Pod Failure

This page shows how to write and read a Container termination message.

Termination messages provide a way for containers to write information about fatal events to a location where it can be easily retrieved and surfaced by tools like dashboards and monitoring software. In most cases, information that you put in a termination message should also be written to the general [Kubernetes logs](#).

Before you begin


You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Killercode](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Writing and reading a termination message

In this exercise, you create a Pod that runs one container. The configuration file specifies a command that runs when the container starts.

[debug/termination.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: termination-demo
spec:
  containers:
  - name: termination-demo-container
    image: debian
    command: ["/bin/sh"]
    args: ["-c", "sleep 10 && echo Sleep expired > /dev/termination-log"]
```

1. Create a Pod based on the YAML configuration file:

```
kubectl apply -f https://k8s.io/examples/debug/termination.yaml
```

In the YAML file, in the `command` and `args` fields, you can see that the container sleeps for 10 seconds and then writes "Sleep expired" to the `/dev/termination-log` file. After the container writes the "Sleep expired" message, it terminates.

2. Display information about the Pod:

```
kubectl get pod termination-demo
```

Repeat the preceding command until the Pod is no longer running.

3. Display detailed information about the Pod:

```
kubectl get pod termination-demo --output=yaml
```


The output includes the "Sleep expired" message:

```
apiVersion: v1
kind: Pod
...
  lastState:
    terminated:
      containerID: ...
      exitCode: 0
      finishedAt: ...
      message: |
        Sleep expired
      ...
```

4. Use a Go template to filter the output so that it includes only the termination message:

```
kubectl get pod termination-demo -o go-template="{{range .status.containerStatuses}}{{.lastState.termin
```

If you are running a multi-container pod, you can use a Go template to include the container's name. By doing so, you can discover which of the containers is failing:

```
kubectl get pod multi-container-pod -o go-template='{{range .status.containerStatuses}}{{printf "%s:\n%s\n\n"
```

Customizing the termination message

Kubernetes retrieves termination messages from the termination message file specified in the `terminationMessagePath` field of a Container, which has a default value of `/dev/termination-log`. By customizing this field, you can tell Kubernetes to use a different file. Kubernetes use the contents from the specified file to populate the Container's status message on both success and failure.

The termination message is intended to be brief final status, such as an assertion failure message. The kubelet truncates messages that are longer than 4096 bytes.

The total message length across all containers is limited to 12KiB, divided equally among each container. For example, if there are 12 containers (`initContainers` or `containers`), each has 1024 bytes of available termination message space.

The default termination message path is `/dev/termination-log`. You cannot set the termination message path after a Pod is launched

In the following example, the container writes termination messages to `/tmp/my-log` for Kubernetes to retrieve:

```
apiVersion: v1
kind: Pod
metadata:
  name: msg-path-demo
spec:
  containers:
  - name: msg-path-demo-container
    image: debian
    terminationMessagePath: "/tmp/my-log"
```

Moreover, users can set the `terminationMessagePolicy` field of a Container for further customization. This field defaults to "File" which means the termination messages are retrieved only from the termination message file. By setting the `terminationMessagePolicy` to "FallbackToLogsOnError", you can tell Kubernetes to use the last chunk of container log output if the termination message file is empty and the container exited with an error. The log output is limited to 2048 bytes or 80 lines, whichever is smaller.

What's next

- See the `terminationMessagePath` field in [Container](#).
- Learn about [retrieving logs](#).
- Learn about [Go templates](#).

7 - Get a Shell to a Running Container

This page shows how to use `kubectl exec` to get a shell to a running container.

Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Killercoda](#)
- [Play with Kubernetes](#)

Getting a shell to a container

In this exercise, you create a Pod that has one container. The container runs the `nginx` image. Here is the configuration file for the Pod:

[application/shell-demo.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: shell-demo
spec:
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
  hostNetwork: true
  dnsPolicy: Default
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/application/shell-demo.yaml
```

Verify that the container is running:

```
kubectl get pod shell-demo
```

Get a shell to the running container:

```
kubectl exec --stdin --tty shell-demo -- /bin/bash
```

Note: The double dash (`--`) separates the arguments you want to pass to the command from the `kubectl` arguments.

In your shell, list the root directory:

```
# Run this inside the container  
ls /
```

In your shell, experiment with other commands. Here are some examples:

```
# You can run these example commands inside the container  
ls /  
cat /proc/mounts  
cat /proc/1/maps  
apt-get update  
apt-get install -y tcpdump  
tcpdump  
apt-get install -y lsof  
lsof  
apt-get install -y procps  
ps aux  
ps aux | grep nginx
```

Writing the root page for nginx

Look again at the configuration file for your Pod. The Pod has an `emptyDir` volume, and the container mounts the volume at `/usr/share/nginx/html`.

In your shell, create an `index.html` file in the `/usr/share/nginx/html` directory:

```
# Run this inside the container  
echo 'Hello shell demo' > /usr/share/nginx/html/index.html
```

In your shell, send a GET request to the nginx server:

```
# Run this in the shell inside your container  
apt-get update  
apt-get install curl  
curl http://localhost/
```

The output shows the text that you wrote to the `index.html` file:

```
Hello shell demo
```

When you are finished with your shell, enter `exit`.

```
exit # To quit the shell in the container
```

Running individual commands in a container

In an ordinary command window, not your shell, list the environment variables in the running container:

```
kubectl exec shell-demo env
```

Experiment with running other commands. Here are some examples:

```
kubectl exec shell-demo -- ps aux
kubectl exec shell-demo -- ls /
kubectl exec shell-demo -- cat /proc/1/mounts
```

Opening a shell when a Pod has more than one container

If a Pod has more than one container, use `--container` or `-c` to specify a container in the `kubectl exec` command. For example, suppose you have a Pod named `my-pod`, and the Pod has two containers named *main-app* and *helper-app*. The following command would open a shell to the *main-app* container.

```
kubectl exec -i -t my-pod --container main-app -- /bin/bash
```

Note: The short options `-i` and `-t` are the same as the long options `--stdin` and `--tty`

What's next

- Read about [kubectl exec](#)