Monitoring, Logging, and Debugging

Set up monitoring and logging to troubleshoot a cluster, or debug a containerized application.

1 - Application Introspection and Debugging

Once your application is running, you'll inevitably need to debug problems with it. Earlier we described how you can use `kubectl get pods` to retrieve simple status information about your pods. But there are a number of ways to get even more information about your application.

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

```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-1006230814-6winp   1/1       Running   0          11s
nginx-deployment-1006230814-fmgu3   1/1       Running   0          11s
```

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

```
kubectl describe pod nginx-deployment-1006230814-6winp
```

```
Name:           nginx-deployment-1006230814-6winp
Namespace:      default
Node:           kubernetes-node-wul5/10.240.0.9
Start Time:     Thu, 24 Mar 2016 01:39:49 +0000
Labels:         app=nginx,pod-template-hash=1006230814
Annotations:    kubernetes.io/created-by={"kind":"SerializedReference","apiVersion":
Status:         Running
IP:             10.244.0.6
Controllers:    ReplicaSet/nginx-deployment-1006230814
Containers:
  nginx:
    Container ID:       docker://90315cc9f513c724e9957a4788d3e625a078de84750f244a40f
    Image:              nginx
    Image ID:           docker://6f62f48c4e55d700cf3eb1b5e33fa051802986b77b874cc351c
    Port:               80/TCP
    QoS Tier:
      cpu:      Guaranteed
      memory:   Guaranteed
    Limits:
      cpu:      500m
      memory:   128Mi
    Requests:
      memory:           128Mi
      cpu:              500m
    State:              Running
      Started:          Thu, 24 Mar 2016 01:39:51 +0000
    Ready:              True
    Restart Count:      0
    Environment:        <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-5kdvl (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
Volumes:
  default-token-4bcbi:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-4bcbi
    Optional:   false
QoS Class:      Guaranteed
Node-Selectors: <none>
Tolerations:    <none>
Events:
  FirstSeen     LastSeen        Count   From                                    Subc
  ---------     --------        -----   ----                                    ----
  54s           54s             1       {default-scheduler }
  54s           54s             1       {kubelet kubernetes-node-wul5}  spec.contain
  53s           53s             1       {kubelet kubernetes-node-wul5}  spec.contain
  53s           53s             1       {kubelet kubernetes-node-wul5}  spec.contain
  53s           53s             1       {kubelet kubernetes-node-wul5}  spec.contain
```

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.)

```
kubectl 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 `kubectl describe pod` on the pending Pod and look at its events:

```
kubectl 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                            SubobjectPat
  ---------   --------        -----   ----                            -------------
  1m              48s             7       {default-scheduler }
fit failure on node (kubernetes-node-6ta5): Node didn't have enough resource: CPU,
fit failure on node (kubernetes-node-wul5): Node didn't have enough resource: CPU,
```

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
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/created-by: |
            {"kind":"SerializedReference","apiVersion":"v1","reference":{"kind":"Rep
  creationTimestamp: 2016-03-24T01:39:50Z
  generateName: nginx-deployment-1006230814-
  labels:
    app: nginx
    pod-template-hash: "1006230814"
  name: nginx-deployment-1006230814-6winp
  namespace: default
  resourceVersion: "133447"
  uid: 4c879808-f161-11e5-9a78-42010af00005
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
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-4bcbi
      readOnly: true
  dnsPolicy: ClusterFirst
  nodeName: kubernetes-node-wul5
  restartPolicy: Always
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  volumes:
  - name: default-token-4bcbi
    secret:
      secretName: default-token-4bcbi
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: 2016-03-24T01:39:51Z
    status: "True"
    type: Ready
  containerStatuses:
  - containerID: docker://90315cc9f513c724e9957a4788d3e625a078de84750f244a40f97ae355
    image: nginx
    imageID: docker://6f62f48c4e55d700cf3eb1b5e33fa051802986b77b874cc351cce539e51637
    lastState: {}
    name: nginx
    ready: true
    restartCount: 0
    state:
      running:
        startedAt: 2016-03-24T01:39:51Z
  hostIP: 10.240.0.9
  phase: Running
  podIP: 10.244.0.6
  startTime: 2016-03-24T01:39:49Z
```

Example: debugging a down/unreachable node

Sometimes when debugging it can be useful to look at the status of a node -- for example, because you've noticed strange behavior of a Pod that's running on the node, or to find out why a Pod won't schedule onto the node. As with Pods, you can use `kubectl describe node` and `kubectl get node -o yaml` to retrieve detailed information about nodes. For example, here's what you'll see if a node is down (disconnected from the network, or kubelet dies and won't restart, etc.). Notice the events that show the node is NotReady, and also notice that the pods are no longer running (they are evicted after five minutes of NotReady status).

```
kubectl get nodes
```

```
NAME                     STATUS     ROLES     AGE     VERSION
kubernetes-node-861h     NotReady   <none>    1h      v1.13.0
kubernetes-node-bols     Ready      <none>    1h      v1.13.0
kubernetes-node-st6x     Ready      <none>    1h      v1.13.0
kubernetes-node-unaj     Ready      <none>    1h      v1.13.0
```

```
kubectl describe node kubernetes-node-861h
```

```
Name:                   kubernetes-node-861h
Role
Labels:          kubernetes.io/arch=amd64
           kubernetes.io/os=linux
           kubernetes.io/hostname=kubernetes-node-861h
Annotations:        node.alpha.kubernetes.io/ttl=0
                    volumes.kubernetes.io/controller-managed-attach-detach=true
Taints:             <none>
CreationTimestamp:      Mon, 04 Sep 2017 17:13:23 +0800
Phase:
Conditions:
  Type          Status          LastHeartbeatTime                       LastTransiti
  ----    ------    -----------------     ------------------      ------          --
  OutOfDisk             Unknown         Fri, 08 Sep 2017 16:04:28 +0800         Fri,
  MemoryPressure        Unknown         Fri, 08 Sep 2017 16:04:28 +0800         Fri,
  DiskPressure          Unknown         Fri, 08 Sep 2017 16:04:28 +0800         Fri,
  Ready                 Unknown         Fri, 08 Sep 2017 16:04:28 +0800         Fri,
Addresses:      10.240.115.55,104.197.0.26
Capacity:
 cpu:           2
 hugePages:     0
 memory:        4046788Ki
 pods:          110
Allocatable:
 cpu:           1500m
 hugePages:     0
 memory:        1479263Ki
 pods:          110
System Info:
 Machine ID:                    8e025a21a4254e11b028584d9d8b12c4
 System UUID:                   349075D1-D169-4F25-9F2A-E886850C47E3
 Boot ID:                       5cd18b37-c5bd-4658-94e0-e436d3f110e0
 Kernel Version:                4.4.0-31-generic
 OS Image:                      Debian GNU/Linux 8 (jessie)
 Operating System:              linux
 Architecture:                  amd64
 Container Runtime Version:     docker://1.12.5
 Kubelet Version:               v1.6.9+a3d1dfa6f4335
 Kube-Proxy Version:            v1.6.9+a3d1dfa6f4335
ExternalID:                     15233045891481496305
Non-terminated Pods:            (9 in total)
  Namespace                     Name                                            CPU
  ---------                     ----                                            ----
......
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits      Memory Requests         Memory Limits
  ------------  ----------      ---------------         -------------
  900m (60%)    2200m (146%)    1009286400 (66%)        5681286400 (375%)
Events:         <none>
```

```
kubectl get node kubernetes-node-861h -o yaml
```

```yaml
apiVersion: v1
kind: Node
metadata:
  creationTimestamp: 2015-07-10T21:32:29Z
  labels:
    kubernetes.io/hostname: kubernetes-node-861h
  name: kubernetes-node-861h
  resourceVersion: "757"
  uid: 2a69374e-274b-11e5-a234-42010af0d969
spec:
  externalID: "15233045891481496305"
  podCIDR: 10.244.0.0/24
  providerID: gce://striped-torus-760/us-central1-b/kubernetes-node-861h
```

```
status:
  addresses:
  - address: 10.240.115.55
    type: InternalIP
  - address: 104.197.0.26
    type: ExternalIP
  capacity:
    cpu: "1"
    memory: 3800808Ki
    pods: "100"
  conditions:
  - lastHeartbeatTime: 2015-07-10T21:34:32Z
    lastTransitionTime: 2015-07-10T21:35:15Z
    reason: Kubelet stopped posting node status.
    status: Unknown
    type: Ready
  nodeInfo:
    bootID: 4e316776-b40d-4f78-a4ea-ab0d73390897
    containerRuntimeVersion: docker://Unknown
    kernelVersion: 3.16.0-0.bpo.4-amd64
    kubeProxyVersion: v0.21.1-185-gffc5a86098dc01
    kubeletVersion: v0.21.1-185-gffc5a86098dc01
    machineID: ""
    osImage: Debian GNU/Linux 7 (wheezy)
    systemUUID: ABE5F6B4-D44B-108B-C46A-24CCE16C8B6E
```

What's next

Learn about additional debugging tools, including:

- Logging
- Monitoring
- Getting into containers via exec
- Connecting to containers via proxies
- Connecting to containers via port forwarding
- Inspect Kubernetes node with crictl

2 - Auditing

Kubernetes *auditing* provides a security-relevant, chronological set of records documenting the sequence of actions in a cluster. The cluster audits the activities generated by users, by applications that use the Kubernetes API, and by the control plane itself.

Auditing allows cluster administrators to answer the following questions:

- what happened?
- when did it happen?
- who initiated it?
- on what did it happen?
- where was it observed?
- from where was it initiated?
- to where was it going?

Audit records begin their lifecycle inside the kube-apiserver component. Each request on each stage of its execution generates an audit event, which is then pre-processed according to a certain policy and written to a backend. The policy determines what's recorded and the backends persist the records. The current backend implementations include logs files and webhooks.

Each request can be recorded with an associated *stage*. The defined stages are:

- `RequestReceived` - The stage for events generated as soon as the audit handler receives the request, and before it is delegated down the handler chain.
- `ResponseStarted` - Once the response headers are sent, but before the response body is sent. This stage is only generated for long-running requests (e.g. watch).
- `ResponseComplete` - The response body has been completed and no more bytes will be sent.
- `Panic` - Events generated when a panic occurred.

> **Note:** Audit events are different from the Event API object.

The audit logging feature increases the memory consumption of the API server because some context required for auditing is stored for each request. Memory consumption depends on the audit logging configuration.

Audit policy

Audit policy defines rules about what events should be recorded and what data they should include. The audit policy object structure is defined in the `audit.k8s.io` API group. When an event is processed, it's compared against the list of rules in order. The first matching rule sets the *audit level* of the event. The defined audit levels are:

- `None` - don't log events that match this rule.
- `Metadata` - log request metadata (requesting user, timestamp, resource, verb, etc.) but not request or response body.
- `Request` - log event metadata and request body but not response body. This does not apply for non-resource requests.
- `RequestResponse` - log event metadata, request and response bodies. This does not apply for non-resource requests.

You can pass a file with the policy to `kube-apiserver` using the `--audit-policy-file` flag. If the flag is omitted, no events are logged. Note that the `rules` field **must** be provided in the audit policy file. A policy with no (0) rules is treated as illegal.

Below is an example audit policy file:

audit/audit-policy.yaml

```yaml
apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy
# Don't generate audit events for all requests in RequestReceived stage.
omitStages:
  - "RequestReceived"
rules:
  # Log pod changes at RequestResponse level
  - level: RequestResponse
    resources:
    - group: ""
      # Resource "pods" doesn't match requests to any subresource of pods,
      # which is consistent with the RBAC policy.
      resources: ["pods"]
  # Log "pods/log", "pods/status" at Metadata level
  - level: Metadata
    resources:
    - group: ""
      resources: ["pods/log", "pods/status"]

  # Don't log requests to a configmap called "controller-leader"
  - level: None
    resources:
    - group: ""
      resources: ["configmaps"]
      resourceNames: ["controller-leader"]

  # Don't log watch requests by the "system:kube-proxy" on endpoints or services
  - level: None
    users: ["system:kube-proxy"]
    verbs: ["watch"]
    resources:
    - group: "" # core API group
      resources: ["endpoints", "services"]

  # Don't log authenticated requests to certain non-resource URL paths.
  - level: None
    userGroups: ["system:authenticated"]
    nonResourceURLs:
    - "/api*" # Wildcard matching.
    - "/version"

  # Log the request body of configmap changes in kube-system.
  - level: Request
    resources:
    - group: "" # core API group
      resources: ["configmaps"]
    # This rule only applies to resources in the "kube-system" namespace.
    # The empty string "" can be used to select non-namespaced resources.
    namespaces: ["kube-system"]

  # Log configmap and secret changes in all other namespaces at the Metadata level.
  - level: Metadata
    resources:
    - group: "" # core API group
      resources: ["secrets", "configmaps"]

  # Log all other resources in core and extensions at the Request level.
  - level: Request
    resources:
    - group: "" # core API group
    - group: "extensions" # Version of group should NOT be included.

  # A catch-all rule to log all other requests at the Metadata level.
  - level: Metadata
    # Long-running requests like watches that fall under this rule will not
    # generate an audit event in RequestReceived.
    omitStages:
      - "RequestReceived"
```

You can use a minimal audit policy file to log all requests at the `Metadata` level:

```
# Log all requests at the Metadata level.
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: Metadata
```

If you're crafting your own audit profile, you can use the audit profile for Google Container-Optimized OS as a starting point. You can check the configure-helper.sh script, which generates an audit policy file. You can see most of the audit policy file by looking directly at the script.

Audit backends

Audit backends persist audit events to an external storage. Out of the box, the kube-apiserver provides two backends:

- Log backend, which writes events into the filesystem
- Webhook backend, which sends events to an external HTTP API

In all cases, audit events follow a structure defined by the Kubernetes API in the `audit.k8s.io` API group. For Kubernetes v1.20.0, that API is at version `v1`.

Note:
In case of patches, request body is a JSON array with patch operations, not a JSON object with an appropriate Kubernetes API object. For example, the following request body is a valid patch request to `/apis/batch/v1/namespaces/some-namespace/jobs/some-job-name`:

```
[
  {
    "op": "replace",
    "path": "/spec/parallelism",
    "value": 0
  },
  {
    "op": "remove",
    "path": "/spec/template/spec/containers/0/terminationMessagePolicy"
  }
]
```

Log backend

The log backend writes audit events to a file in JSONlines format. You can configure the log audit backend using the following `kube-apiserver` flags:

- `--audit-log-path` specifies the log file path that log backend uses to write audit events. Not specifying this flag disables log backend. `-` means standard out
- `--audit-log-maxage` defined the maximum number of days to retain old audit log files
- `--audit-log-maxbackup` defines the maximum number of audit log files to retain
- `--audit-log-maxsize` defines the maximum size in megabytes of the audit log file before it gets rotated

If your cluster's control plane runs the kube-apiserver as a Pod, remember to mount the `hostPath` to the location of the policy file and log file, so that audit records are persisted. For example:

```
    --audit-policy-file=/etc/kubernetes/audit-policy.yaml \
```

```
    --audit-log-path=/var/log/audit.log
```

then mount the volumes:

```
...
volumeMounts:
  - mountPath: /etc/kubernetes/audit-policy.yaml
    name: audit
    readOnly: true
  - mountPath: /var/log/audit.log
    name: audit-log
    readOnly: false
```

and finally configure the `hostPath`:

```
...
- name: audit
  hostPath:
    path: /etc/kubernetes/audit-policy.yaml
    type: File

- name: audit-log
  hostPath:
    path: /var/log/audit.log
    type: FileOrCreate
```

Webhook backend

The webhook audit backend sends audit events to a remote web API, which is assumed to be a form of the Kubernetes API, including means of authentication. You can configure a webhook audit backend using the following kube-apiserver flags:

- `--audit-webhook-config-file` specifies the path to a file with a webhook configuration. The webhook configuration is effectively a specialized kubeconfig.
- `--audit-webhook-initial-backoff` specifies the amount of time to wait after the first failed request before retrying. Subsequent requests are retried with exponential backoff.

The webhook config file uses the kubeconfig format to specify the remote address of the service and credentials used to connect to it.

Event batching

Both log and webhook backends support batching. Using webhook as an example, here's the list of available flags. To get the same flag for log backend, replace `webhook` with `log` in the flag name. By default, batching is enabled in `webhook` and disabled in `log`. Similarly, by default throttling is enabled in `webhook` and disabled in `log`.

- `--audit-webhook-mode` defines the buffering strategy. One of the following:
  - `batch` - buffer events and asynchronously process them in batches. This is the default.
  - `blocking` - block API server responses on processing each individual event.
  - `blocking-strict` - Same as blocking, but when there is a failure during audit logging at the RequestReceived stage, the whole request to the kube-apiserver fails.

The following flags are used only in the `batch` mode:

- `--audit-webhook-batch-buffer-size` defines the number of events to buffer before batching. If the rate of incoming events overflows the buffer, events are dropped.

- `--audit-webhook-batch-max-size` defines the maximum number of events in one batch.
- `--audit-webhook-batch-max-wait` defines the maximum amount of time to wait before unconditionally batching events in the queue.
- `--audit-webhook-batch-throttle-qps` defines the maximum average number of batches generated per second.
- `--audit-webhook-batch-throttle-burst` defines the maximum number of batches generated at the same moment if the allowed QPS was underutilized previously.

Parameter tuning

Parameters should be set to accommodate the load on the API server.

For example, if kube-apiserver receives 100 requests each second, and each request is audited only on `ResponseStarted` and `ResponseComplete` stages, you should account for ≈200 audit events being generated each second. Assuming that there are up to 100 events in a batch, you should set throttling level at least 2 queries per second. Assuming that the backend can take up to 5 seconds to write events, you should set the buffer size to hold up to 5 seconds of events; that is: 10 batches, or 1000 events.

In most cases however, the default parameters should be sufficient and you don't have to worry about setting them manually. You can look at the following Prometheus metrics exposed by kube-apiserver and in the logs to monitor the state of the auditing subsystem.

- `apiserver_audit_event_total` metric contains the total number of audit events exported.
- `apiserver_audit_error_total` metric contains the total number of events dropped due to an error during exporting.

Log entry truncation

Both log and webhook backends support limiting the size of events that are logged. As an example, the following is the list of flags available for the log backend:

- `audit-log-truncate-enabled` whether event and batch truncating is enabled.
- `audit-log-truncate-max-batch-size` maximum size in bytes of the batch sent to the underlying backend.
- `audit-log-truncate-max-event-size` maximum size in bytes of the audit event sent to the underlying backend.

By default truncate is disabled in both `webhook` and `log`, a cluster administrator should set `audit-log-truncate-enabled` or `audit-webhook-truncate-enabled` to enable the feature.

What's next

- Learn about Mutating webhook auditing annotations.

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=myapp` set on them, you can use the following:

```
kubectl get pods -l app=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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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 `M` Init Containers, and `N` have completed so far.
`Init:Error`	An Init Container has failed to execute.
`Init:CrashLoopBackOff`	An Init Container has failed repeatedly.
`Pending`	The Pod has not yet begun executing Init Containers.

Status	Meaning
`PodInitializing` or `Running`	The Pod has already finished executing Init Containers.

5 - Debug Pods and ReplicationControllers

This page shows how to debug Pods and ReplicationControllers.

Before you begin

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

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

- You should be familiar with the basics of Pods and with Pods' lifecycles.

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:

Insufficient resources

You may have exhausted the supply of CPU or Memory in your cluster. In this case you can try several things:

- Add more nodes to the cluster.
- Terminate unneeded pods to make room for pending pods.
- Check that the pod is not larger than your nodes. For example, if all nodes have a capacity of `cpu:1`, then a pod with a request of `cpu: 1.1` will never be scheduled.

  You can check node capacities with the `kubectl get nodes -o <format>` command. Here are some example command lines that extract the necessary information:

```
kubectl get nodes -o yaml | egrep '\sname:|cpu:|memory:'
kubectl get nodes -o json | jq '.items[] | {name: .metadata.name, cap: .status.
```

The resource quota feature can be configured to limit the total amount of resources that can be consumed. If used in conjunction with namespaces, it can prevent one team from hogging all the resources.

Using hostPort

When you bind a pod to a `hostPort` there are a limited number of places that the 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 container 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 repository?
- Run a manual `docker pull <image>` on your machine to see if the image can be pulled.

My pod is crashing or otherwise unhealthy

Once your pod has been scheduled, the methods described in Debug Running Pods are available for debugging.

Debugging ReplicationControllers

ReplicationControllers 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 inspect events related to the replication controller.

6 - 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 Troubleshoot Applications.
- 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`.

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 v1.18 [alpha]`

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. `kubectl` has an alpha command that can create ephemeral containers for debugging beginning with version `v1.18`.

Example debugging using ephemeral containers

> **Note:** The examples in this section require the `EphemeralContainers` feature gate enabled in your cluster and `kubectl` version v1.18 or later.

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=k8s.gcr.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 pro
```

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 --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.

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://b888f9adfd15bd5739fefaa39e1df4dd3c617b9902082b1cfdc29c4
    Image:          busybox
    Image ID:       docker-pullable://busybox@sha256:1828edd60c5efd34b2bf5dd3282ec0c
    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 --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 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 --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 privileged Pod running in the host namespaces. 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 my
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 container runs in the host IPC, Network, and PID namespaces.
- The root filesystem of the Node will be mounted at `/host`.

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

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

7 - 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/busybo
```

> **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=k8s.gcr.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:

```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: k8s.gcr.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.

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 --m
```

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 "/ku
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 he
I1027 22:14:54.038140    5063 proxier.go:352] Setting endpoints for "kube-system/kub
I1027 22:14:54.038164    5063 proxier.go:352] Setting endpoints for "kube-system/kub
I1027 22:14:54.038209    5063 proxier.go:352] Setting endpoints for "default/kuberne
I1027 22:14:54.038238    5063 proxier.go:429] Not syncing iptables until Services ar
I1027 22:14:54.040048    5063 proxier.go:294] Adding new service "default/kubernetes
I1027 22:14:54.040154    5063 proxier.go:294] Adding new service "kube-system/kube-d
I1027 22:14:54.040223    5063 proxier.go:294] Adding new service "kube-system/kube-d
```

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/hostname
-A KUBE-SEP-57KPRZ3JQVENLNBR -p tcp -m comment --comment "default/hostnames:" -m tcp
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -s 10.244.1.7/32 -m comment --comment "default/hostname
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -p tcp -m comment --comment "default/hostnames:" -m tcp
-A KUBE-SEP-X3P2623AGDH6CDF3 -s 10.244.2.3/32 -m comment --comment "default/hostname
-A KUBE-SEP-X3P2623AGDH6CDF3 -p tcp -m comment --comment "default/hostnames:" -m tcp
-A KUBE-SERVICES -d 10.0.1.175/32 -p tcp -m comment --comment "default/hostnames: cl
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:" -m statistic
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:" -m statistic
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:" -j KUBE-SEP-5
```

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

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
```

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/kube
```

- 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-bridg
```

- 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 troubleshooting document for more information.

8 - Debugging Kubernetes nodes with crictl

FEATURE STATE: `Kubernetes v1.11 [stable]`

`crictl` is a command-line interface for CRI-compatible container runtimes. You can use it to inspect and debug container runtimes and applications on a Kubernetes node. `crictl` and its source are hosted in the cri-tools repository.

Before you begin

`crictl` requires a Linux operating system with a CRI runtime.

Installing crictl

You can download a compressed archive `crictl` from the cri-tools release page, for several different architectures. Download the version that corresponds to your version of Kubernetes. Extract it and move it to a location on your system path, such as `/usr/local/bin/`.

General usage

The `crictl` command has several subcommands and runtime flags. Use `crictl help` or `crictl <subcommand> help` for more details.

`crictl` connects to `unix:///var/run/dockershim.sock` by default. For other runtimes, you can set the endpoint in multiple different ways:

- By setting flags `--runtime-endpoint` and `--image-endpoint`
- By setting environment variables `CONTAINER_RUNTIME_ENDPOINT` and `IMAGE_SERVICE_ENDPOINT`
- By setting the endpoint in the config file `--config=/etc/crictl.yaml`

You can also specify timeout values when connecting to the server and enable or disable debugging, by specifying `timeout` or `debug` values in the configuration file or using the `--timeout` and `--debug` command-line flags.

To view or edit the current configuration, view or edit the contents of `/etc/crictl.yaml`.

```
cat /etc/crictl.yaml
runtime-endpoint: unix:///var/run/dockershim.sock
image-endpoint: unix:///var/run/dockershim.sock
timeout: 10
debug: true
```

Example crictl commands

The following examples show some `crictl` commands and example output.

> **Warning:** If you use `crictl` to create pod sandboxes or containers on a running Kubernetes cluster, the Kubelet will eventually delete them. `crictl` is not a general purpose workflow tool, but a tool that is useful for debugging.

List pods

List all pods:

```
crictl pods
```

The output is similar to this:

```
POD ID              CREATED              STATE               NAME
926f1b5a1d33a       About a minute ago   Ready               sh-84d7dcf559-4r2gq
4dccb216c4adb       About a minute ago   Ready               nginx-65899c769f-wv2gp
a86316e96fa89       17 hours ago         Ready               kube-proxy-gblk4
919630b8f81f1       17 hours ago         Ready               nvidia-device-plugin-zg
```

List pods by name:

```
crictl pods --name nginx-65899c769f-wv2gp
```

The output is similar to this:

```
POD ID              CREATED             STATE               NAME
4dccb216c4adb       2 minutes ago       Ready               nginx-65899c769f-wv2gp
```

List pods by label:

```
crictl pods --label run=nginx
```

The output is similar to this:

```
POD ID              CREATED             STATE               NAME
4dccb216c4adb       2 minutes ago       Ready               nginx-65899c769f-wv2gp
```

List images

List all images:

```
crictl images
```

The output is similar to this:

```
IMAGE                                TAG                 IMAGE ID            SI
busybox                              latest              8c811b4aec35f       1.
k8s-gcrio.azureedge.net/hyperkube-amd64   v1.10.3             e179bbfe5d238       66
k8s-gcrio.azureedge.net/pause-amd64       3.1                 da86e6ba6ca19       74
nginx                                latest              cd5239a0906a6       10
```

List images by repository:

```
crictl images nginx
```

The output is similar to this:

```
IMAGE              TAG                IMAGE ID            SIZE
nginx              latest             cd5239a0906a6       109MB
```

Only list image IDs:

```
crictl images -q
```

The output is similar to this:

```
sha256:8c811b4aec35f259572d0f79207bc0678df4c736eeec50bc9fec37ed936a472a
sha256:e179bbfe5d238de6069f3b03fccbecc3fb4f2019af741bfff1233c4d7b2970c5
sha256:da86e6ba6ca197bf6bc5e9d900febd906b133eaa4750e6bed647b0fbe50ed43e
sha256:cd5239a0906a6ccf0562354852fae04bc5b52d72a2aff9a871ddb6bd57553569
```

List containers

List all containers:

```
crictl ps -a
```

The output is similar to this:

```
CONTAINER ID        IMAGE
1f73f2d81bf98       busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea
9c5951df22c78       busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea
87d3992f84f74       nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d8aa315263f1e7806bf8
1941fb4da154f       k8s-gcrio.azureedge.net/hyperkube-amd64@sha256:00d814b1f7763f4ab
```

List running containers:

```
crictl ps
```

The output is similar to this:

```
CONTAINER ID        IMAGE
1f73f2d81bf98       busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea
87d3992f84f74       nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d8aa315263f1e7806bf8
1941fb4da154f       k8s-gcrio.azureedge.net/hyperkube-amd64@sha256:00d814b1f7763f4ab
```

Execute a command in a running container

```
crictl exec -i -t 1f73f2d81bf98 ls
```

The output is similar to this:

```
bin   dev   etc   home  proc  root  sys   tmp   usr   var
```

Get a container's logs

Get all container logs:

```
crictl logs 87d3992f84f74
```

The output is similar to this:

```
10.240.0.96 - - [06/Jun/2018:02:45:49 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47
10.240.0.96 - - [06/Jun/2018:02:45:50 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47
10.240.0.96 - - [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47
```

Get only the latest `N` lines of logs:

```
crictl logs --tail=1 87d3992f84f74
```

The output is similar to this:

```
10.240.0.96 - - [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47
```

Run a pod sandbox

Using `crictl` to run a pod sandbox is useful for debugging container runtimes. On a running Kubernetes cluster, the sandbox will eventually be stopped and deleted by the Kubelet.

1. Create a JSON file like the following:

   ```
   {
       "metadata": {
           "name": "nginx-sandbox",
           "namespace": "default",
           "attempt": 1,
           "uid": "hdishd83djaidwnduwk28bcsb"
       },
       "logDirectory": "/tmp",
       "linux": {
       }
   }
   ```

2. Use the `crictl runp` command to apply the JSON and run the sandbox.

   ```
   crictl runp pod-config.json
   ```

   The ID of the sandbox is returned.

Create a container

Using `crictl` to create a container is useful for debugging container runtimes. On a running Kubernetes cluster, the sandbox will eventually be stopped and deleted by the Kubelet.

1. Pull a busybox image

   ```
   crictl pull busybox
   Image is up to date for busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b
   ```

2. Create configs for the pod and the container:

   Pod config:

   ```
   {
       "metadata": {
           "name": "nginx-sandbox",
           "namespace": "default",
           "attempt": 1,
           "uid": "hdishd83djaidwnduwk28bcsb"
       },
       "log_directory": "/tmp",
       "linux": {
       }
   }
   ```

   Container config:

   ```
   {
     "metadata": {
         "name": "busybox"
     },
     "image":{
         "image": "busybox"
     },
     "command": [
         "top"
     ],
     "log_path":"busybox.log",
     "linux": {
     }
   }
   ```

3. Create the container, passing the ID of the previously-created pod, the container config file, and the pod config file. The ID of the container is returned.

   ```
   crictl create f84dd361f8dc51518ed291fbadd6db537b0496536c1d2d6c05ff943ce8c9a54f
   ```

4. List all containers and verify that the newly-created container has its state set to `Created`.

   ```
   crictl ps -a
   ```

   The output is similar to this:

   ```
   CONTAINER ID        IMAGE               CREATED             STATE
   3e025dd50a72d       busybox             32 seconds ago      Created
   ```

Start a container

To start a container, pass its ID to `crictl start`:

```
crictl start 3e025dd50a72d956c4f14881fbb5b1080c9275674e95fb67f965f6478a957d60
```

The output is similar to this:

```
3e025dd50a72d956c4f14881fbb5b1080c9275674e95fb67f965f6478a957d60
```

Check the container has its state set to `Running`.

```
crictl ps
```

The output is similar to this:

```
CONTAINER ID        IMAGE               CREATED             STATE               NAM
3e025dd50a72d       busybox             About a minute ago  Running             bus
```

See kubernetes-sigs/cri-tools for more information.

Mapping from docker cli to crictl

The exact versions for below mapping table are for docker cli v1.40 and crictl v1.19.0. Please note that the list is not exhaustive. For example, it doesn't include experimental commands of docker cli.

> **Note:** The output format of CRICTL is similar to Docker CLI, despite some missing columns for some CLI. Make sure to check output for the specific command if your script output parsing.

Retrieve Debugging Information

docker cli	crictl	Description	Unsupported Features
`attach`	`attach`	Attach to a running container	`--detach-keys`, `--sig-proxy`
`exec`	`exec`	Run a command in a running container	`--privileged`, `--user`, `--detach-keys`
`images`	`images`	List images	
`info`	`info`	Display system-wide information	
`inspect`	`inspect`, `inspecti`	Return low-level information on a container, image or task	
`logs`	`logs`	Fetch the logs of a container	`--details`
`ps`	`ps`	List containers	
`stats`	`stats`	Display a live stream of container(s) resource usage statistics	Column: NET/BLOCK I/O, PIDs
`version`	`version`	Show the runtime (Docker, ContainerD, or others) version information	

Perform Changes

docker cli	crictl	Description	Unsupported Features
`create`	`create`	Create a new container	
`kill`	`stop` (timeout = 0)	Kill one or more running container	`--signal`
`pull`	`pull`	Pull an image or a repository from a registry	`--all-tags`, `--disable-content-trust`
`rm`	`rm`	Remove one or more containers	
`rmi`	`rmi`	Remove one or more images	
`run`	`run`	Run a command in a new container	
`start`	`start`	Start one or more stopped containers	`--detach-keys`
`stop`	`stop`	Stop one or more running containers	
`update`	`update`	Update configuration of one or more containers	`--restart`, `--blkio-weight` and some other resource limit not supported by CRI.

Supported only in crictl

crictl	Description
`imagefsinfo`	Return image filesystem info
`inspectp`	Display the status of one or more pods
`port-forward`	Forward local port to a pod
`pods`	List pods
`runp`	Run a new pod
`rmp`	Remove one or more pods
`stopp`	Stop one or more running pods

9 - 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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

```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 `cmd` 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.containerStat
```

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 will be limited to 12KiB. 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.

10 - Developing and debugging services locally

Kubernetes applications usually consist of multiple, separate services, each running in its own container. Developing and debugging these services on a remote Kubernetes cluster can be cumbersome, requiring you to get a shell on a running container and running your tools inside the remote shell.

`telepresence` is a tool to ease the process of developing and debugging services locally, while proxying the service to a remote Kubernetes cluster. Using `telepresence` allows you to use custom tools, such as a debugger and IDE, for a local service and provides the service full access to ConfigMap, secrets, and the services running on the remote cluster.

This document describes using `telepresence` to develop and debug services running on a remote cluster locally.

Before you begin

- Kubernetes cluster is installed
- `kubectl` is configured to communicate with the cluster
- Telepresence is installed

Getting a shell on a remote cluster

Open a terminal and run `telepresence` with no arguments to get a `telepresence` shell. This shell runs locally, giving you full access to your local filesystem.

The `telepresence` shell can be used in a variety of ways. For example, write a shell script on your laptop, and run it directly from the shell in real time. You can do this on a remote shell as well, but you might not be able to use your preferred code editor, and the script is deleted when the container is terminated.

Enter `exit` to quit and close the shell.

Developing or debugging an existing service

When developing an application on Kubernetes, you typically program or debug a single service. The service might require access to other services for testing and debugging. One option is to use the continuous deployment pipeline, but even the fastest deployment pipeline introduces a delay in the program or debug cycle.

Use the `--swap-deployment` option to swap an existing deployment with the Telepresence proxy. Swapping allows you to run a service locally and connect to the remote Kubernetes cluster. The services in the remote cluster can now access the locally running instance.

To run telepresence with `--swap-deployment`, enter:

```
telepresence --swap-deployment $DEPLOYMENT_NAME
```

where $DEPLOYMENT_NAME is the name of your existing deployment.

Running this command spawns a shell. In the shell, start your service. You can then make edits to the source code locally, save, and see the changes take effect immediately. You can also run your service in a debugger, or any other local development tool.

What's next

If you're interested in a hands-on tutorial, check out this tutorial that walks through locally developing the Guestbook application on Google Kubernetes Engine.

Telepresence has numerous proxying options, depending on your situation.

For further reading, visit the Telepresence website.

11 - 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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

```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

12 - Logging Using Stackdriver

Before reading this page, it's highly recommended to familiarize yourself with the overview of logging in Kubernetes.

> **Note:** By default, Stackdriver logging collects only your container's standard output and standard error streams. To collect any logs your application writes to a file (for example), see the sidecar approach in the Kubernetes logging overview.

Deploying

To ingest logs, you must deploy the Stackdriver Logging agent to each node in your cluster. The agent is a configured `fluentd` instance, where the configuration is stored in a `ConfigMap` and the instances are managed using a Kubernetes `DaemonSet`. The actual deployment of the `ConfigMap` and `DaemonSet` for your cluster depends on your individual cluster setup.

Deploying to a new cluster

Google Kubernetes Engine

Stackdriver is the default logging solution for clusters deployed on Google Kubernetes Engine. Stackdriver Logging is deployed to a new cluster by default unless you explicitly opt-out.

Other platforms

To deploy Stackdriver Logging on a *new* cluster that you're creating using `kube-up.sh`, do the following:

1. Set the `KUBE_LOGGING_DESTINATION` environment variable to `gcp`.
2. **If not running on GCE**, include the `beta.kubernetes.io/fluentd-ds-ready=true` in the `KUBE_NODE_LABELS` variable.

Once your cluster has started, each node should be running the Stackdriver Logging agent. The `DaemonSet` and `ConfigMap` are configured as addons. If you're not using `kube-up.sh`, consider starting a cluster without a pre-configured logging solution and then deploying Stackdriver Logging agents to the running cluster.

> **Warning:** The Stackdriver logging daemon has known issues on platforms other than Google Kubernetes Engine. Proceed at your own risk.

Deploying to an existing cluster

1. Apply a label on each node, if not already present.

   The Stackdriver Logging agent deployment uses node labels to determine to which nodes it should be allocated. These labels were introduced to distinguish nodes with the Kubernetes version 1.6 or higher. If the cluster was created with Stackdriver Logging configured and node has version 1.5.X or lower, it will have fluentd as static pod. Node cannot have more than one instance of fluentd, therefore only apply labels to the nodes that don't have fluentd pod allocated already. You can ensure that your node is labelled properly by running `kubectl describe` as follows:

   ```
   kubectl describe node $NODE_NAME
   ```

   The output should be similar to this:

```
Name:           NODE_NAME
Role:
Labels:         beta.kubernetes.io/fluentd-ds-ready=true
...
```

Ensure that the output contains the label `beta.kubernetes.io/fluentd-ds-ready=true`. If it is not present, you can add it using the `kubectl label` command as follows:

```
kubectl label node $NODE_NAME beta.kubernetes.io/fluentd-ds-ready=true
```

> **Note:** If a node fails and has to be recreated, you must re-apply the label to the recreated node. To make this easier, you can use Kubelet's command-line parameter for applying node labels in your node startup script.

2. Deploy a `ConfigMap` with the logging agent configuration by running the following command:

```
kubectl apply -f https://k8s.io/examples/debug/fluentd-gcp-configmap.yaml
```

The command creates the `ConfigMap` in the `default` namespace. You can download the file manually and change it before creating the `ConfigMap` object.

3. Deploy the logging agent `DaemonSet` by running the following command:

```
kubectl apply -f https://k8s.io/examples/debug/fluentd-gcp-ds.yaml
```

You can download and edit this file before using it as well.

Verifying your Logging Agent Deployment

After Stackdriver `DaemonSet` is deployed, you can discover logging agent deployment status by running the following command:

```
kubectl get ds --all-namespaces
```

If you have 3 nodes in the cluster, the output should looks similar to this:

```
NAMESPACE     NAME              DESIRED   CURRENT   READY     NODE-SELECTOR
...
default       fluentd-gcp-v2.0  3         3         3         beta.kubernetes.io/fl
...
```

To understand how logging with Stackdriver works, consider the following synthetic log generator pod specification counter-pod.yaml:

debug/counter-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
```

```
    args: [/bin/sh, -c,
            'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']
```

This pod specification has one container that runs a bash script that writes out the value of a counter and the datetime once per second, and runs indefinitely. Let's create this pod in the default namespace.

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

You can observe the running pod:

```
kubectl get pods
```

```
NAME                                     READY     STATUS    RESTARTS   AGE
counter                                  1/1       Running   0          5m
```

For a short period of time you can observe the 'Pending' pod status, because the kubelet has to download the container image first. When the pod status changes to `Running` you can use the `kubectl logs` command to view the output of this counter pod.

```
kubectl logs counter
```

```
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

As described in the logging overview, this command fetches log entries from the container log file. If the container is killed and then restarted by Kubernetes, you can still access logs from the previous container. However, if the pod is evicted from the node, log files are lost. Let's demonstrate this by deleting the currently running counter container:

```
kubectl delete pod counter
```

```
pod "counter" deleted
```

and then recreating it:

```
kubectl create -f https://k8s.io/examples/debug/counter-pod.yaml
```

```
pod/counter created
```

After some time, you can access logs from the counter pod again:

```
kubectl logs counter
```

```
0: Mon Jan  1 00:01:00 UTC 2001
1: Mon Jan  1 00:01:01 UTC 2001
2: Mon Jan  1 00:01:02 UTC 2001
...
```

As expected, only recent log lines are present. However, for a real-world application you will likely want to be able to access logs from all containers, especially for the debug purposes. This is exactly when the previously enabled Stackdriver Logging can help.

Viewing logs

Stackdriver Logging agent attaches metadata to each log entry, for you to use later in queries to select only the messages you're interested in: for example, the messages from a particular pod.

The most important pieces of metadata are the resource type and log name. The resource type of a container log is `container`, which is named `GKE Containers` in the UI (even if the Kubernetes cluster is not on Google Kubernetes Engine). The log name is the name of the container, so that if you have a pod with two containers, named `container_1` and `container_2` in the spec, their logs will have log names `container_1` and `container_2` respectively.

System components have resource type `compute`, which is named `GCE VM Instance` in the interface. Log names for system components are fixed. For a Google Kubernetes Engine node, every log entry from a system component has one of the following log names:

- docker
- kubelet
- kube-proxy

You can learn more about viewing logs on the dedicated Stackdriver page.

One of the possible ways to view logs is using the `gcloud logging` command line interface from the Google Cloud SDK. It uses Stackdriver Logging filtering syntax to query specific logs. For example, you can run the following command:

```
gcloud beta logging read 'logName="projects/$YOUR_PROJECT_ID/logs/count"' --format j
```

```
...
"2: Mon Jan  1 00:01:02 UTC 2001\n"
"1: Mon Jan  1 00:01:01 UTC 2001\n"
"0: Mon Jan  1 00:01:00 UTC 2001\n"
...
"2: Mon Jan  1 00:00:02 UTC 2001\n"
"1: Mon Jan  1 00:00:01 UTC 2001\n"
"0: Mon Jan  1 00:00:00 UTC 2001\n"
```

As you can see, it outputs messages for the count container from both the first and second runs, despite the fact that the kubelet already deleted the logs for the first container.

Exporting logs

You can export logs to Google Cloud Storage or to BigQuery to run further analysis. Stackdriver Logging offers the concept of sinks, where you can specify the destination of log entries. More information is available on the Stackdriver Exporting Logs page.

Configuring Stackdriver Logging Agents

Sometimes the default installation of Stackdriver Logging may not suit your needs, for example:

- You may want to add more resources because default performance doesn't suit your needs.

- You may want to introduce additional parsing to extract more metadata from your log messages, like severity or source code reference.
- You may want to send logs not only to Stackdriver or send it to Stackdriver only partially.

In this case you need to be able to change the parameters of `DaemonSet` and `ConfigMap`.

Prerequisites

If you're using GKE and Stackdriver Logging is enabled in your cluster, you cannot change its configuration, because it's managed and supported by GKE. However, you can disable the default integration and deploy your own.

> **Note:** You will have to support and maintain a newly deployed configuration yourself: update the image and configuration, adjust the resources and so on.

To disable the default logging integration, use the following command:

```
gcloud beta container clusters update --logging-service=none CLUSTER
```

You can find notes on how to then install Stackdriver Logging agents into a running cluster in the Deploying section.

Changing `DaemonSet` parameters

When you have the Stackdriver Logging `DaemonSet` in your cluster, you can modify the `template` field in its spec. The DaemonSet controller manages the pods for you. For example, assume you've installed the Stackdriver Logging as described above. Now you want to change the memory limit to give fluentd more memory to safely process more logs.

Get the spec of `DaemonSet` running in your cluster:

```
kubectl get ds fluentd-gcp-v2.0 --namespace kube-system -o yaml > fluentd-gcp-ds.yam
```

Then edit resource requirements in the spec file and update the `DaemonSet` object in the apiserver using the following command:

```
kubectl replace -f fluentd-gcp-ds.yaml
```

After some time, Stackdriver Logging agent pods will be restarted with the new configuration.

Changing fluentd parameters

Fluentd configuration is stored in the `ConfigMap` object. It is effectively a set of configuration files that are merged together. You can learn about fluentd configuration on the official site.

Imagine you want to add a new parsing logic to the configuration, so that fluentd can understand default Python logging format. An appropriate fluentd filter looks similar to this:

```
<filter reform.**>
  type parser
  format /^(?<severity>\w):(?<logger_name>\w):(?<log>.*)/
  reserve_data true
  suppress_parse_error_log true
  key_name log
</filter>
```

Now you have to put it in the configuration and make Stackdriver Logging agents pick it up. Get the current version of the Stackdriver Logging `ConfigMap` in your cluster by running the following command:

```
kubectl get cm fluentd-gcp-config --namespace kube-system -o yaml > fluentd-gcp-conf
```

Then in the value of the key `containers.input.conf` insert a new filter right after the `source` section.

> **Note:** Order is important.

Updating `ConfigMap` in the apiserver is more complicated than updating `DaemonSet`. It's better to consider `ConfigMap` to be immutable. Then, in order to update the configuration, you should create `ConfigMap` with a new name and then change `DaemonSet` to point to it using guide above.

Adding fluentd plugins

Fluentd is written in Ruby and allows to extend its capabilities using plugins. If you want to use a plugin, which is not included in the default Stackdriver Logging container image, you have to build a custom image. Imagine you want to add Kafka sink for messages from a particular container for additional processing. You can re-use the default container image sources with minor changes:

- Change Makefile to point to your container repository, for example `PREFIX=gcr.io/<your-project-id>`.
- Add your dependency to the Gemfile, for example `gem 'fluent-plugin-kafka'`.

Then run `make build push` from this directory. After updating `DaemonSet` to pick up the new image, you can use the plugin you installed in the fluentd configuration.

13 - Monitor Node Health

Node Problem Detector is a daemon for monitoring and reporting about a node's health. You can run Node Problem Detector as a `DaemonSet` or as a standalone daemon. Node Problem Detector collects information about node problems from various daemons and reports these conditions to the API server as NodeCondition and Event.

To learn how to install and use Node Problem Detector, see Node Problem Detector project documentation.

Before you begin

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

- Katacoda
- Play with Kubernetes

Limitations

- Node Problem Detector only supports file based kernel log. Log tools such as `journald` are not supported.
- Node Problem Detector uses the kernel log format for reporting kernel issues. To learn how to extend the kernel log format, see Add support for another log format.

Enabling Node Problem Detector

Some cloud providers enable Node Problem Detector as an Addon. You can also enable Node Problem Detector with `kubectl` or by creating an Addon pod.

Using kubectl to enable Node Problem Detector

`kubectl` provides the most flexible management of Node Problem Detector. You can overwrite the default configuration to fit it into your environment or to detect customized node problems. For example:

1. Create a Node Problem Detector configuration similar to `node-problem-detector.yaml`:

debug/node-problem-detector.yaml

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: node-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
```

```
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
      - name: node-problem-detector
        image: k8s.gcr.io/node-problem-detector:v0.1
        securityContext:
          privileged: true
        resources:
          limits:
            cpu: "200m"
            memory: "100Mi"
          requests:
            cpu: "20m"
            memory: "20Mi"
        volumeMounts:
        - name: log
          mountPath: /log
          readOnly: true
      volumes:
      - name: log
        hostPath:
          path: /var/log/
```

> **Note:** You should verify that the system log directory is right for your operating system distribution.

2. Start node problem detector with `kubectl`:

```
kubectl apply -f https://k8s.io/examples/debug/node-problem-detector.yaml
```

Using an Addon pod to enable Node Problem Detector

If you are using a custom cluster bootstrap solution and don't need to overwrite the default configuration, you can leverage the Addon pod to further automate the deployment.

Create `node-problem-detector.yaml`, and save the configuration in the Addon pod's directory `/etc/kubernetes/addons/node-problem-detector` on a control plane node.

Overwrite the configuration

The default configuration is embedded when building the Docker image of Node Problem Detector.

However, you can use a `ConfigMap` to overwrite the configuration:

1. Change the configuration files in `config/`
2. Create the `ConfigMap` `node-problem-detector-config`:

```
kubectl create configmap node-problem-detector-config --from-file=config/
```

3. Change the `node-problem-detector.yaml` to use the `ConfigMap`:

debug/node-problem-detector-configmap.yaml

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: node-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
      - name: node-problem-detector
        image: k8s.gcr.io/node-problem-detector:v0.1
        securityContext:
          privileged: true
        resources:
          limits:
            cpu: "200m"
            memory: "100Mi"
          requests:
            cpu: "20m"
            memory: "20Mi"
        volumeMounts:
        - name: log
          mountPath: /log
          readOnly: true
        - name: config # Overwrite the config/ directory with ConfigMap volume
          mountPath: /config
          readOnly: true
      volumes:
      - name: log
        hostPath:
          path: /var/log/
      - name: config # Define ConfigMap volume
        configMap:
          name: node-problem-detector-config
```

4. Recreate the Node Problem Detector with the new configuration file:

```
# If you have a node-problem-detector running, delete before recreating
kubectl delete -f https://k8s.io/examples/debug/node-problem-detector.yaml
kubectl apply -f https://k8s.io/examples/debug/node-problem-detector-configmap.
```

> **Note:** This approach only applies to a Node Problem Detector started with `kubectl`.

Overwriting a configuration is not supported if a Node Problem Detector runs as a cluster Addon. The Addon manager does not support `ConfigMap`.

Kernel Monitor

Kernel Monitor is a system log monitor daemon supported in the Node Problem Detector. Kernel monitor watches the kernel log and detects known kernel issues following predefined rules.

The Kernel Monitor matches kernel issues according to a set of predefined rule list in `config/kernel-monitor.json`. The rule list is extensible. You can expand the rule list by overwriting the configuration.

Add new NodeConditions

To support a new `NodeCondition`, create a condition definition within the `conditions` field in `config/kernel-monitor.json`, for example:

```
{
  "type": "NodeConditionType",
  "reason": "CamelCaseDefaultNodeConditionReason",
  "message": "arbitrary default node condition message"
}
```

Detect new problems

To detect new problems, you can extend the `rules` field in `config/kernel-monitor.json` with a new rule definition:

```
{
  "type": "temporary/permanent",
  "condition": "NodeConditionOfPermanentIssue",
  "reason": "CamelCaseShortReason",
  "message": "regexp matching the issue in the kernel log"
}
```

Configure path for the kernel log device

Check your kernel log path location in your operating system (OS) distribution. The Linux kernel log device is usually presented as `/dev/kmsg`. However, the log path location varies by OS distribution. The `log` field in `config/kernel-monitor.json` represents the log path inside the container. You can configure the `log` field to match the device path as seen by the Node Problem Detector.

Add support for another log format

Kernel monitor uses the `Translator` plugin to translate the internal data structure of the kernel log. You can implement a new translator for a new log format.

Recommendations and restrictions

It is recommended to run the Node Problem Detector in your cluster to monitor node health. When running the Node Problem Detector, you can expect extra resource overhead on each node. Usually this is fine, because:

- The kernel log grows relatively slowly.
- A resource limit is set for the Node Problem Detector.
- Even under high load, the resource usage is acceptable. For more information, see the Node Problem Detector benchmark result.

14 - Resource metrics pipeline

Resource usage metrics, such as container CPU and memory usage, are available in Kubernetes through the Metrics API. These metrics can be accessed either directly by the user with the `kubectl top` command, or by a controller in the cluster, for example Horizontal Pod Autoscaler, to make decisions.

The Metrics API

Through the Metrics API, you can get the amount of resource currently used by a given node or a given pod. This API doesn't store the metric values, so it's not possible, for example, to get the amount of resources used by a given node 10 minutes ago.

The API is no different from any other API:

- it is discoverable through the same endpoint as the other Kubernetes APIs under the path: `/apis/metrics.k8s.io/`
- it offers the same security, scalability, and reliability guarantees

The API is defined in k8s.io/metrics repository. You can find more information about the API there.

> **Note:** The API requires the metrics server to be deployed in the cluster. Otherwise it will be not available.

Measuring Resource Usage

CPU

CPU is reported as the average usage, in CPU cores, over a period of time. This value is derived by taking a rate over a cumulative CPU counter provided by the kernel (in both Linux and Windows kernels). The kubelet chooses the window for the rate calculation.

Memory

Memory is reported as the working set, in bytes, at the instant the metric was collected. In an ideal world, the "working set" is the amount of memory in-use that cannot be freed under memory pressure. However, calculation of the working set varies by host OS, and generally makes heavy use of heuristics to produce an estimate. It includes all anonymous (non-file-backed) memory since Kubernetes does not support swap. The metric typically also includes some cached (file-backed) memory, because the host OS cannot always reclaim such pages.

Metrics Server

Metrics Server is a cluster-wide aggregator of resource usage data. By default, it is deployed in clusters created by `kube-up.sh` script as a Deployment object. If you use a different Kubernetes setup mechanism, you can deploy it using the provided deployment components.yaml file.

Metrics Server collects metrics from the Summary API, exposed by Kubelet on each node, and is registered with the main API server via Kubernetes aggregator.

Learn more about the metrics server in the design doc.

15 - Tools for Monitoring Resources

To scale an application and provide a reliable service, you need to understand how the application behaves when it is deployed. You can examine application performance in a Kubernetes cluster by examining the containers, pods, services, and the characteristics of the overall cluster. Kubernetes provides detailed information about an application's resource usage at each of these levels. This information allows you to evaluate your application's performance and where bottlenecks can be removed to improve overall performance.

In Kubernetes, application monitoring does not depend on a single monitoring solution. On new clusters, you can use resource metrics or full metrics pipelines to collect monitoring statistics.

Resource metrics pipeline

The resource metrics pipeline provides a limited set of metrics related to cluster components such as the Horizontal Pod Autoscaler controller, as well as the `kubectl top` utility. These metrics are collected by the lightweight, short-term, in-memory metrics-server and are exposed via the `metrics.k8s.io` API.

metrics-server discovers all nodes on the cluster and queries each node's kubelet for CPU and memory usage. The kubelet acts as a bridge between the Kubernetes master and the nodes, managing the pods and containers running on a machine. The kubelet translates each pod into its constituent containers and fetches individual container usage statistics from the container runtime through the container runtime interface. The kubelet fetches this information from the integrated cAdvisor for the legacy Docker integration. It then exposes the aggregated pod resource usage statistics through the metrics-server Resource Metrics API. This API is served at `/metrics/resource/v1beta1` on the kubelet's authenticated and read-only ports.

Full metrics pipeline

A full metrics pipeline gives you access to richer metrics. Kubernetes can respond to these metrics by automatically scaling or adapting the cluster based on its current state, using mechanisms such as the Horizontal Pod Autoscaler. The monitoring pipeline fetches metrics from the kubelet and then exposes them to Kubernetes via an adapter by implementing either the `custom.metrics.k8s.io` or `external.metrics.k8s.io` API.

Prometheus, a CNCF project, can natively monitor Kubernetes, nodes, and Prometheus itself. Full metrics pipeline projects that are not part of the CNCF are outside the scope of Kubernetes documentation.

16 - Troubleshoot Applications

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 repository?
- Run a manual `docker pull <image>` on your machine to see if the image can be pulled.

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://
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.

If the list of pods matches expectations, but your endpoints are still empty, it's possible that you don't have the right ports exposed. If your service has a `containerPort` specified, but the Pods that are selected don't have that port listed, then they won't be added to the endpoints list.

Verify that the pod's `containerPort` matches up with the Service's `targetPort`

Network traffic is not forwarded

If you can connect to the service, but the connection is immediately dropped, and there are endpoints in the endpoints list, it's likely that the proxy can't contact your pods.

There are three things to check:

- Are your pods working correctly? Look for restart count, and debug pods.
- Can you connect to your pods directly? Get the IP address for the Pod, and try to connect directly to that IP.
- Is your application serving on the port that you configured? Kubernetes doesn't do port remapping, so if your application serves on 8080, the `containerPort` field needs to be 8080.

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.

17 - Troubleshoot Clusters

This doc is about cluster troubleshooting; we assume you have already ruled out your application as the root cause of the problem you are experiencing. See the application troubleshooting guide for tips on application debugging. You may also visit troubleshooting document for more information.

Listing your cluster

The first thing to debug in your cluster is if your nodes are all registered correctly.

Run

```
kubectl get nodes
```

And verify that all of the nodes you expect to see are present and that they are all in the `Ready` state.

To get detailed information about the overall health of your cluster, you can run:

```
kubectl cluster-info dump
```

Looking at logs

For now, digging deeper into the cluster requires logging into the relevant machines. Here are the locations of the relevant log files. (note that on systemd-based systems, you may need to use `journalctl` instead)

Master

- `/var/log/kube-apiserver.log` - API Server, responsible for serving the API
- `/var/log/kube-scheduler.log` - Scheduler, responsible for making scheduling decisions
- `/var/log/kube-controller-manager.log` - Controller that manages replication controllers

Worker Nodes

- `/var/log/kubelet.log` - Kubelet, responsible for running containers on the node
- `/var/log/kube-proxy.log` - Kube Proxy, responsible for service load balancing

A general overview of cluster failure modes

This is an incomplete list of things that could go wrong, and how to adjust your cluster setup to mitigate the problems.

Root causes:

- VM(s) shutdown
- Network partition within cluster, or between cluster and users
- Crashes in Kubernetes software
- Data loss or unavailability of persistent storage (e.g. GCE PD or AWS EBS volume)
- Operator error, for example misconfigured Kubernetes software or application software

Specific scenarios:

- Apiserver VM shutdown or apiserver crashing
  - Results
    - unable to stop, update, or start new pods, services, replication controller
    - existing pods and services should continue to work normally, unless they depend on the Kubernetes API
- Apiserver backing storage lost
  - Results
    - apiserver should fail to come up
    - kubelets will not be able to reach it but will continue to run the same pods and provide the same service proxying
    - manual recovery or recreation of apiserver state necessary before apiserver is restarted
- Supporting services (node controller, replication controller manager, scheduler, etc) VM shutdown or crashes
  - currently those are colocated with the apiserver, and their unavailability has similar consequences as apiserver
  - in future, these will be replicated as well and may not be co-located
  - they do not have their own persistent state
- Individual node (VM or physical machine) shuts down
  - Results
    - pods on that Node stop running
- Network partition
  - Results
    - partition A thinks the nodes in partition B are down; partition B thinks the apiserver is down. (Assuming the master VM ends up in partition A.)
- Kubelet software fault
  - Results
    - crashing kubelet cannot start new pods on the node
    - kubelet might delete the pods or not
    - node marked unhealthy
    - replication controllers start new pods elsewhere
- Cluster operator error
  - Results
    - loss of pods, services, etc
    - lost of apiserver backing store
    - users unable to read API
    - etc.

Mitigations:

- Action: Use IaaS provider's automatic VM restarting feature for IaaS VMs
  - Mitigates: Apiserver VM shutdown or apiserver crashing
  - Mitigates: Supporting services VM shutdown or crashes
- Action: Use IaaS providers reliable storage (e.g. GCE PD or AWS EBS volume) for VMs with apiserver+etcd
  - Mitigates: Apiserver backing storage lost
- Action: Use high-availability configuration
  - Mitigates: Control plane node shutdown or control plane components (scheduler, API server, controller-manager) crashing
    - Will tolerate one or more simultaneous node or component failures
  - Mitigates: API server backing storage (i.e., etcd's data directory) lost
    - Assumes HA (highly-available) etcd configuration
- Action: Snapshot apiserver PDs/EBS-volumes periodically
  - Mitigates: Apiserver backing storage lost
  - Mitigates: Some cases of operator error
  - Mitigates: Some cases of Kubernetes software fault

- Action: use replication controller and services in front of pods
    - Mitigates: Node shutdown
    - Mitigates: Kubelet software fault
- Action: applications (containers) designed to tolerate unexpected restarts
    - Mitigates: Node shutdown
    - Mitigates: Kubelet software fault

18 - Troubleshooting

Sometimes things go wrong. This guide is aimed at making them right. It has two sections:

- Troubleshooting your application - Useful for users who are deploying code into Kubernetes and wondering why it is not working.
- Troubleshooting your cluster - Useful for cluster administrators and people whose Kubernetes cluster is unhappy.

You should also check the known issues for the release you're using.

Getting help

If your problem isn't answered by any of the guides above, there are variety of ways for you to get help from the Kubernetes community.

Questions

The documentation on this site has been structured to provide answers to a wide range of questions. Concepts explain the Kubernetes architecture and how each component works, while Setup provides practical instructions for getting started. Tasks show how to accomplish commonly used tasks, and Tutorials are more comprehensive walkthroughs of real-world, industry-specific, or end-to-end development scenarios. The Reference section provides detailed documentation on the Kubernetes API and command-line interfaces (CLIs), such as `kubectl`.

Help! My question isn't covered! I need help now!

Stack Overflow

Someone else from the community may have already asked a similar question or may be able to help with your problem. The Kubernetes team will also monitor posts tagged Kubernetes. If there aren't any existing questions that help, please ask a new one!

Slack

Many people from the Kubernetes community hang out on Kubernetes Slack in the `#kubernetes-users` channel. Slack requires registration; you can request an invitation, and registration is open to everyone). Feel free to come and ask any and all questions. Once registered, access the Kubernetes organisation in Slack via your web browser or via Slack's own dedicated app.

Once you are registered, browse the growing list of channels for various subjects of interest. For example, people new to Kubernetes may also want to join the `#kubernetes-novice` channel. As another example, developers should join the `#kubernetes-dev` channel.

There are also many country specific / local language channels. Feel free to join these channels for localized support and info:

Country	Channels
China	`#cn-users`, `#cn-events`
Finland	`#fi-users`
France	`#fr-users`, `#fr-events`
Germany	`#de-users`, `#de-events`

Country	Channels
India	`#in-users`, `#in-events`
Italy	`#it-users`, `#it-events`
Japan	`#jp-users`, `#jp-events`
Korea	`#kr-users`
Netherlands	`#nl-users`
Norway	`#norw-users`
Poland	`#pl-users`
Russia	`#ru-users`
Spain	`#es-users`
Sweden	`#se-users`
Turkey	`#tr-users`, `#tr-events`

Forum

You're welcome to join the official Kubernetes Forum: discuss.kubernetes.io.

Bugs and feature requests

If you have what looks like a bug, or you would like to make a feature request, please use the GitHub issue tracking system.

Before you file an issue, please search existing issues to see if your issue is already covered.

If filing a bug, please include detailed information about how to reproduce the problem, such as:

- Kubernetes version: `kubectl version`
- Cloud provider, OS distro, network configuration, and Docker version
- Steps to reproduce the problem