Update API Objects in Place Using kubectl patch

Use kubectl patch to update Kubernetes API objects in place. Do a strategic merge patch or a JSON merge patch.

This task shows how to use `kubectl patch` to update an API object in place. The exercises in this task demonstrate a strategic merge patch and a JSON merge patch.

Before you begin

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

To check the version, enter `kubectl version`.

Use a strategic merge patch to update a Deployment

Here's the configuration file for a Deployment that has two replicas. Each replica is a Pod that has one container:

application/deployment-patch.yaml  Copy application/deployment-patch.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: patch-demospec:  replicas: 2  selector:    matchLabels:      app: nginx  template:    metadata:
```

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/application/deployment-patch.yaml
```

View the Pods associated with your Deployment:

```
kubectl get pods
```

The output shows that the Deployment has two Pods. The `1/1` indicates that each Pod has one container:

```
NAME                          READY     STATUS    RESTARTS   AGE
patch-demo-28633765-670qr   1/1       Running   0          23s
patch-demo-28633765-j5qs3   1/1       Running   0          23s
```

Make a note of the names of the running Pods. Later, you will see that these Pods get terminated and replaced by new ones.

At this point, each Pod has one Container that runs the nginx image. Now suppose you want each Pod to have two containers: one that runs nginx and one that runs redis.

Create a file named `patch-file.yaml` that has this content:

```
spec:
  template:
    spec:
      containers:
      - name: patch-demo-ctr-2
        image: redis
```

Patch your Deployment:

```
kubectl patch deployment patch-demo --patch-file patch-file.yaml
```

View the patched Deployment:

```
kubectl get deployment patch-demo --output yaml
```

The output shows that the PodSpec in the Deployment has two Containers:

```
containers:
- image: redis  imagePullPolicy: Always  name: patch-demo-ctr-2  ...- image: nginx  imagePullPolicy: Always  name: patch-demo-ctr
```

View the Pods associated with your patched Deployment:

```
kubectl get pods
```

The output shows that the running Pods have different names from the Pods that were running previously. The Deployment terminated the old Pods and created two new Pods that comply with the updated Deployment spec. The `2/2` indicates that each Pod has two Containers:

```
NAME                          READY     STATUS    RESTARTS   AGE
patch-demo-1081991389-2wrn5   2/2       Running   0          1m
patch-demo-1081991389-jmg7b   2/2       Running   0          1m
```

Take a closer look at one of the patch-demo Pods:

```
kubectl get pod <your-pod-name> --output yaml
```

The output shows that the Pod has two Containers: one running nginx and one running redis:

```
containers:
- image: redis
  ...
- image: nginx
  ...
```

Notes on the strategic merge patch

The patch you did in the preceding exercise is called a *strategic merge patch*. Notice that the patch did not replace the `containers` list. Instead it added a new Container to the list. In other words, the list in the patch was merged with the existing list. This is not always what happens when you use a strategic merge patch on a list. In some cases, the list is replaced, not merged.

With a strategic merge patch, a list is either replaced or merged depending on its patch strategy. The patch strategy is specified by the value of the `patchStrategy` key in a field tag in the Kubernetes source code. For example, the `Containers` field of `PodSpec` struct has a `patchStrategy` of `merge`:

```
type PodSpec struct {
  ...
  Containers []Container `json:"containers" patchStrategy:"merge" patchMergeKey:"name" ...`
  ...
}
```

You can also see the patch strategy in the OpenApi spec:

```
"io.k8s.api.core.v1.PodSpec": {
    ...,
    "containers": {
        "description": "List of containers belonging to the pod.  ...."
    },
    "x-kubernetes-patch-merge-key": "name",
    "x-kubernetes-patch-strategy": "merge"
}
```

And you can see the patch strategy in the Kubernetes API documentation.

Create a file named `patch-file-tolerations.yaml` that has this content:

```
spec:
  template:
    spec:
      tolerations:
      - effect: NoSchedule
        key: disktype
        value: ssd
```

Patch your Deployment:

```
kubectl patch deployment patch-demo --patch-file patch-file-tolerations.yaml
```

View the patched Deployment:

```
kubectl get deployment patch-demo --output yaml
```

The output shows that the PodSpec in the Deployment has only one Toleration:

```
tolerations:
- effect: NoSchedule  key: disktype  value: ssd
```

Notice that the `tolerations` list in the PodSpec was replaced, not merged. This is because the Tolerations field of PodSpec does not have a `patchStrategy` key in its field tag. So the strategic merge patch uses the default patch strategy, which is `replace`.

```
type PodSpec struct {
  ...
  Tolerations []Toleration `json:"tolerations,omitempty" protobuf:"bytes,22,opt,name=tolerations"`
  ...
}
```

Use a JSON merge patch to update a Deployment

A strategic merge patch is different from a JSON merge patch. With a JSON merge patch, if you want to update a list, you have to specify the entire new list. And the new list completely replaces the existing list.

The `kubectl patch` command has a `type` parameter that you can set to one of these values:

Parameter value	Merge type
json	JSON Patch, RFC 6902
merge	JSON Merge Patch, RFC 7386
strategic	Strategic merge patch

For a comparison of JSON patch and JSON merge patch, see JSON Patch and JSON Merge Patch.

The default value for the `type` parameter is `strategic`. So in the preceding exercise, you did a strategic merge patch.

Next, do a JSON merge patch on your same Deployment. Create a file named `patch-file-2.yaml` that has this content:

```
spec:
  template:
    spec:
      containers:
      - name: patch-demo-ctr-3
        image: gcr.io/google-samples/hello-app:2.0
```

In your patch command, set `type` to `merge`:

```
kubectl patch deployment patch-demo --type merge --patch-file patch-file-2.yaml
```

View the patched Deployment:

```
kubectl get deployment patch-demo --output yaml
```

The `containers` list that you specified in the patch has only one Container. The output shows that your list of one Container replaced the existing `containers` list.

```
spec:
  containers:
  - image: gcr.io/google-samples/hello-app:2.0
    ...
    name: patch-demo-ctr-3
```

List the running Pods:

```
kubectl get pods
```

In the output, you can see that the existing Pods were terminated, and new Pods were created. The `1/1` indicates that each new Pod is running only one Container.

```
NAME                             READY     STATUS    RESTARTS   AGE
patch-demo-1307768864-69308   1/1       Running   0          1m
patch-demo-1307768864-c86dc   1/1       Running   0          1m
```

Use strategic merge patch to update a Deployment using the retainKeys strategy

Here's the configuration file for a Deployment that uses the `RollingUpdate` strategy:

application/deployment-retainkeys.yaml Copy application/deployment-retainkeys.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: retainkeys-demospec:  selector:    matchLabels:      app: nginx  strategy:    rollingUpdate:
```

Create the deployment:

```
kubectl apply -f https://k8s.io/examples/application/deployment-retainkeys.yaml
```

At this point, the deployment is created and is using the `RollingUpdate` strategy.

Create a file named `patch-file-no-retainkeys.yaml` that has this content:

```
spec:
  strategy:
    type: Recreate
```

Patch your Deployment:

```
kubectl patch deployment retainkeys-demo --type strategic --patch-file patch-file-no-retainkeys.yaml
```

In the output, you can see that it is not possible to set `type` as `Recreate` when a value is defined for `spec.strategy.rollingUpdate`:

```
The Deployment "retainkeys-demo" is invalid: spec.strategy.rollingUpdate: Forbidden: may not be specified when strategy `type` is
```

The way to remove the value for `spec.strategy.rollingUpdate` when updating the value for `type` is to use the `retainKeys` strategy for the strategic merge.

Create another file named `patch-file-retainkeys.yaml` that has this content:

```
spec:
  strategy:
    $retainKeys:
    - type
    type: Recreate
```

With this patch, we indicate that we want to retain only the `type` key of the `strategy` object. Thus, the `rollingUpdate` will be removed during the patch operation.

Patch your Deployment again with this new patch:

```
kubectl patch deployment retainkeys-demo --type strategic --patch-file patch-file-retainkeys.yaml
```

Examine the content of the Deployment:

```
kubectl get deployment retainkeys-demo --output yaml
```

The output shows that the strategy object in the Deployment does not contain the `rollingUpdate` key anymore:

```
spec:
```

```
  strategy:
    type: Recreate
  template:
```

Notes on the strategic merge patch using the retainKeys strategy

The patch you did in the preceding exercise is called a *strategic merge patch with retainKeys strategy*. This method introduces a new directive `$retainKeys` that has the following strategies:

- It contains a list of strings.
- All fields needing to be preserved must be present in the `$retainKeys` list.
- The fields that are present will be merged with live object.
- All of the missing fields will be cleared when patching.
- All fields in the `$retainKeys` list must be a superset or the same as the fields present in the patch.

The `retainKeys` strategy does not work for all objects. It only works when the value of the `patchStrategy` key in a field tag in the Kubernetes source code contains `retainKeys`. For example, the `Strategy` field of the `DeploymentSpec` struct has a `patchStrategy` of `retainKeys`:

```
type DeploymentSpec struct {
  ...
  // +patchStrategy=retainKeys
  Strategy DeploymentStrategy `json:"strategy,omitempty" patchStrategy:"retainKeys" ...`  ...}
```

You can also see the `retainKeys` strategy in the OpenApi spec:

```
"io.k8s.api.apps.v1.DeploymentSpec": {
    ...,
    "strategy": {
        "$ref": "#/definitions/io.k8s.api.apps.v1.DeploymentStrategy",
        "description": "The deployment strategy to use to replace existing pods with new ones.",
        "x-kubernetes-patch-strategy": "retainKeys"
    },
    ....
}
```

And you can see the `retainKeys` strategy in the Kubernetes API documentation.

Alternate forms of the kubectl patch command

The `kubectl patch` command takes YAML or JSON. It can take the patch as a file or directly on the command line.

Create a file named `patch-file.json` that has this content:

```
{
   "spec": {
      "template": {
         "spec": {
            "containers": [
               {
                  "name": "patch-demo-ctr-2",
                  "image": "redis"
               }
            ]
         }
      }
   }
}
```

The following commands are equivalent:

```
kubectl patch deployment patch-demo --patch-file patch-file.yaml
kubectl patch deployment patch-demo --patch 'spec:\n template:\n  spec:\n   containers:\n   - name: patch-demo-ctr-2\n     image: 

kubectl patch deployment patch-demo --patch-file patch-file.json
kubectl patch deployment patch-demo --patch '{"spec": {"template": {"spec": {"containers": [{"name": "patch-demo-ctr-2","image": "
```

Update an object's replica count using `kubectl patch` with `--subresource`

The flag `--subresource=[subresource-name]` is used with kubectl commands like get, patch, edit, apply and replace to fetch and update `status`, `scale` and `resize` subresource of the resources you specify. You can specify a subresource for any of the Kubernetes API resources (built-in and CRs) that have `status`, `scale` or `resize` subresource.

For example, a Deployment has a `status` subresource and a `scale` subresource, so you can use `kubectl` to get or modify just the `status` subresource of a Deployment.

Here's a manifest for a Deployment that has two replicas:

application/deployment.yaml  Copy application/deployment.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deploymentspec:  selector:    matchLabels:      app: nginx  replicas: 2 # tells deployment
```

Create the Deployment:

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

View the Pods associated with your Deployment:

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

In the output, you can see that Deployment has two Pods. For example:

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-7fb96c846b-22567   1/1     Running   0          47s
nginx-deployment-7fb96c846b-mlgns   1/1     Running   0          47s
```

Now, patch that Deployment with `--subresource=[subresource-name]` flag:

```
kubectl patch deployment nginx-deployment --subresource='scale' --type='merge' -p '{"spec":{"replicas":3}}'
```

The output is:

```
scale.autoscaling/nginx-deployment patched
```

View the Pods associated with your patched Deployment:

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

In the output, you can see one new pod is created, so now you have 3 running pods.

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-7fb96c846b-22567   1/1     Running   0          107s
nginx-deployment-7fb96c846b-lxfr2   1/1     Running   0          14s
nginx-deployment-7fb96c846b-mlgns   1/1     Running   0          107s
```

View the patched Deployment:

```
kubectl get deployment nginx-deployment -o yaml
```

```
...
spec:  replicas: 3  ...status:  ...  availableReplicas: 3  readyReplicas: 3  replicas: 3
```

Note:

If you run `kubectl patch` and specify `--subresource` flag for resource that doesn't support that particular subresource, the API server returns a 404 Not Found error.

Summary

In this exercise, you used `kubectl patch` to change the live configuration of a Deployment object. You did not change the configuration file that you originally used to create the Deployment object. Other commands for updating API objects include kubectl annotate, kubectl edit, kubectl replace, kubectl scale, and kubectl apply.

Note:

Strategic merge patch is not supported for custom resources.

What's next

- Kubernetes Object Management
- Managing Kubernetes Objects Using Imperative Commands
- Imperative Management of Kubernetes Objects Using Configuration Files
- Declarative Management of Kubernetes Objects Using Configuration Files

---

Manage Cluster Daemons

Perform common tasks for managing a DaemonSet, such as performing a rolling update.

Building a Basic DaemonSet

Perform a Rolling Update on a DaemonSet

Perform a Rollback on a DaemonSet

Running Pods on Only Some Nodes

Running Pods on Only Some Nodes

This page demonstrates how can you run Pods on only some Nodes as part of a DaemonSet

Before you begin

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

minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Running Pods on only some Nodes

Imagine that you want to run a DaemonSet, but you only need to run those daemon pods on nodes that have local solid state (SSD) storage. For example, the Pod might provide cache service to the node, and the cache is only useful when low-latency local storage is available.

Step 1: Add labels to your nodes

Add the label `ssd=true` to the nodes which have SSDs.

```
kubectl label nodes example-node-1 example-node-2 ssd=true
```

Step 2: Create the manifest

Let's create a DaemonSet which will provision the daemon pods on the SSD labeled nodes only.

Next, use a `nodeSelector` to ensure that the DaemonSet only runs Pods on nodes with the `ssd` label set to `"true"`.

controllers/daemonset-label-selector.yaml Copy controllers/daemonset-label-selector.yaml to clipboard

```
apiVersion: apps/v1
kind: DaemonSetmetadata:  name: ssd-driver  labels:    app: nginxspec:  selector:    matchLabels:      app: ssd-driver-pod  templa
```

Step 3: Create the DaemonSet

Create the DaemonSet from the manifest by using `kubectl create` or `kubectl apply`

Let's label another node as `ssd=true`.

```
kubectl label nodes example-node-3 ssd=true
```

Labelling the node automatically triggers the control plane (specifically, the DaemonSet controller) to run a new daemon pod on that node.

```
kubectl get pods -o wide
```

The output is similar to:

```
NAME                                READY     STATUS    RESTARTS   AGE     IP      NODE
<daemonset-name><some-hash-01>      1/1       Running   0          13s     .....   example-node-1
<daemonset-name><some-hash-02>      1/1       Running   0          13s     .....   example-node-2
<daemonset-name><some-hash-03>      1/1       Running   0          5s      .....   example-node-3
```

Perform a Rolling Update on a DaemonSet

This page shows how to perform a rolling update on a DaemonSet.

Before you begin

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

DaemonSet Update Strategy

DaemonSet has two update strategy types:

- `OnDelete`: With `OnDelete` update strategy, after you update a DaemonSet template, new DaemonSet pods will *only* be created when you manually delete old DaemonSet pods. This is the same behavior of DaemonSet in Kubernetes version 1.5 or before.
- `RollingUpdate`: This is the default update strategy.
  With `RollingUpdate` update strategy, after you update a DaemonSet template, old DaemonSet pods will be killed, and new DaemonSet pods will be created automatically, in a controlled fashion. At most one pod of the DaemonSet will be running on each node during the whole update process.

Performing a Rolling Update

To enable the rolling update feature of a DaemonSet, you must set its `.spec.updateStrategy.type` to `RollingUpdate`.

You may want to set `.spec.updateStrategy.rollingUpdate.maxUnavailable` (default to 1), `.spec.minReadySeconds` (default to 0) and `.spec.updateStrategy.rollingUpdate.maxSurge` (defaults to 0) as well.

Creating a DaemonSet with `RollingUpdate` update strategy

This YAML file specifies a DaemonSet with an update strategy as 'RollingUpdate'

controllers/fluentd-daemonset.yaml 

```
apiVersion: apps/v1
kind: DaemonSetmetadata:  name: fluentd-elasticsearch  namespace: kube-system  labels:    k8s-app: fluentd-loggingspec:  selector:
```

After verifying the update strategy of the DaemonSet manifest, create the DaemonSet:

```
kubectl create -f https://k8s.io/examples/controllers/fluentd-daemonset.yaml
```

Alternatively, use `kubectl apply` to create the same DaemonSet if you plan to update the DaemonSet with `kubectl apply`.

```
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset.yaml
```

Checking DaemonSet `RollingUpdate` update strategy

Check the update strategy of your DaemonSet, and make sure it's set to `RollingUpdate`:

```
kubectl get ds/fluentd-elasticsearch -o go-template='{{.spec.updateStrategy.type}}{{"\n"}}' -n kube-system
```

If you haven't created the DaemonSet in the system, check your DaemonSet manifest with the following command instead:

```
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset.yaml --dry-run=client -o go-template='{{.spec.updateStrateg
```

The output from both commands should be:

```
RollingUpdate
```

If the output isn't `RollingUpdate`, go back and modify the DaemonSet object or manifest accordingly.

Updating a DaemonSet template

Any updates to a `RollingUpdate` DaemonSet `.spec.template` will trigger a rolling update. Let's update the DaemonSet by applying a new YAML file. This can be done with several different `kubectl` commands.

controllers/fluentd-daemonset-update.yaml 

```
apiVersion: apps/v1
kind: DaemonSetmetadata:  name: fluentd-elasticsearch  namespace: kube-system  labels:    k8s-app: fluentd-loggingspec:  selector:
```

Declarative commands

If you update DaemonSets using configuration files, use `kubectl apply`:

```
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset-update.yaml
```

Imperative commands

If you update DaemonSets using imperative commands, use `kubectl edit` :

```
kubectl edit ds/fluentd-elasticsearch -n kube-system
```

Updating only the container image

If you only need to update the container image in the DaemonSet template, i.e. `.spec.template.spec.containers[*].image`, use `kubectl set image`:

```
kubectl set image ds/fluentd-elasticsearch fluentd-elasticsearch=quay.io/fluentd_elasticsearch/fluentd:v2.6.0 -n kube-system
```

Watching the rolling update status

Finally, watch the rollout status of the latest DaemonSet rolling update:

```
kubectl rollout status ds/fluentd-elasticsearch -n kube-system
```

When the rollout is complete, the output is similar to this:

```
daemonset "fluentd-elasticsearch" successfully rolled out
```

Troubleshooting

DaemonSet rolling update is stuck

Sometimes, a DaemonSet rolling update may be stuck. Here are some possible causes:

Some nodes run out of resources

The rollout is stuck because new DaemonSet pods can't be scheduled on at least one node. This is possible when the node is running out of resources.

When this happens, find the nodes that don't have the DaemonSet pods scheduled on by comparing the output of `kubectl get nodes` and the output of:

```
kubectl get pods -l name=fluentd-elasticsearch -o wide -n kube-system
```

Once you've found those nodes, delete some non-DaemonSet pods from the node to make room for new DaemonSet pods.

Note:

This will cause service disruption when deleted pods are not controlled by any controllers or pods are not replicated. This does not respect PodDisruptionBudget either.

Broken rollout

If the recent DaemonSet template update is broken, for example, the container is crash looping, or the container image doesn't exist (often due to a typo), DaemonSet rollout won't progress.

To fix this, update the DaemonSet template again. New rollout won't be blocked by previous unhealthy rollouts.

Clock skew

If `.spec.minReadySeconds` is specified in the DaemonSet, clock skew between master and nodes will make DaemonSet unable to detect the right rollout progress.

Clean up

Delete DaemonSet from a namespace :

```
kubectl delete ds fluentd-elasticsearch -n kube-system
```

What's next

- See Performing a rollback on a DaemonSet
- See Creating a DaemonSet to adopt existing DaemonSet pods

Building a Basic DaemonSet

This page demonstrates how to build a basic DaemonSet that runs a Pod on every node in a Kubernetes cluster. It covers a simple use case of mounting a file from the host, logging its contents using an init container, and utilizing a pause container.

Before you begin

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

A Kubernetes cluster with at least two nodes (one control plane node and one worker node) to demonstrate the behavior of DaemonSets.

Define the DaemonSet

In this task, a basic DaemonSet is created which ensures that the copy of a Pod is scheduled on every node. The Pod will use an init container to read and log the contents of `/etc/machine-id` from the host, while the main container will be a `pause` container, which keeps the Pod running.

application/basic-daemonset.yaml Copy application/basic-daemonset.yaml to clipboard

```
apiVersion: apps/v1
kind: DaemonSetmetadata:  name: example-daemonsetspec:  selector:    matchLabels:      app.kubernetes.io/name: example  template:
```

1. Create a DaemonSet based on the (YAML) manifest:

   ```
   kubectl apply -f https://k8s.io/examples/application/basic-daemonset.yaml
   ```

2. Once applied, you can verify that the DaemonSet is running a Pod on every node in the cluster:

   ```
   kubectl get pods -o wide
   ```

   The output will list one Pod per node, similar to:

   ```
   NAME                              READY   STATUS    RESTARTS   AGE     IP        NODE
   example-daemonset-xxxxx           1/1     Running   0          5m      x.x.x.x   node-1
   example-daemonset-yyyyy           1/1     Running   0          5m      x.x.x.x   node-2
   ```

3. You can inspect the contents of the logged `/etc/machine-id` file by checking the log directory mounted from the host:

   ```
   kubectl exec <pod-name> -- cat /var/log/machine-id.log
   ```

   Where `<pod-name>` is the name of one of your Pods.

Cleaning up

To delete the DaemonSet, run this command:

```
kubectl delete --cascade=foreground --ignore-not-found --now daemonsets/example-daemonset
```

This simple DaemonSet example introduces key components like init containers and host path volumes, which can be expanded upon for more advanced use cases. For more details refer to DaemonSet.

What's next

- See Performing a rolling update on a DaemonSet
- See Creating a DaemonSet to adopt existing DaemonSet pods

Schedule GPUs

Configure and schedule GPUs for use as a resource by nodes in a cluster.
FEATURE STATE: `Kubernetes v1.26 [stable]`

Kubernetes includes **stable** support for managing AMD and NVIDIA GPUs (graphical processing units) across different nodes in your cluster, using device plugins.

This page describes how users can consume GPUs, and outlines some of the limitations in the implementation.

Using device plugins

Kubernetes implements device plugins to let Pods access specialized hardware features such as GPUs.

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

As an administrator, you have to install GPU drivers from the corresponding hardware vendor on the nodes and run the corresponding device plugin from the GPU vendor. Here are some links to vendors' instructions:

- AMD
- Intel
- NVIDIA

Once you have installed the plugin, your cluster exposes a custom schedulable resource such as `amd.com/gpu` or `nvidia.com/gpu`.

You can consume these GPUs from your containers by requesting the custom GPU resource, the same way you request `cpu` or `memory`. However, there are some limitations in how you specify the resource requirements for custom devices.

GPUs are only supposed to be specified in the `limits` section, which means:

- You can specify GPU `limits` without specifying `requests`, because Kubernetes will use the limit as the request value by default.
- You can specify GPU in both `limits` and `requests` but these two values must be equal.
- You cannot specify GPU `requests` without specifying `limits`.

Here's an example manifest for a Pod that requests a GPU:

```
apiVersion: v1
kind: Podmetadata:  name: example-vector-addspec:  restartPolicy: OnFailure  containers:    - name: example-vector-add      image:
```

Manage clusters with different types of GPUs

If different nodes in your cluster have different types of GPUs, then you can use Node Labels and Node Selectors to schedule pods to appropriate nodes.

For example:

```
# Label your nodes with the accelerator type they have.
kubectl label nodes node1 accelerator=example-gpu-x100
kubectl label nodes node2 accelerator=other-gpu-k915
```

That label key `accelerator` is just an example; you can use a different label key if you prefer.

Automatic node labelling

As an administrator, you can automatically discover and label all your GPU enabled nodes by deploying Kubernetes Node Feature Discovery (NFD). NFD detects the hardware features that are available on each node in a Kubernetes cluster. Typically, NFD is configured to advertise those features as node labels, but NFD can also add extended resources, annotations, and node taints. NFD is compatible with all supported versions of Kubernetes. By default NFD create

the feature labels for the detected features. Administrators can leverage NFD to also taint nodes with specific features, so that only pods that request those features can be scheduled on those nodes.

You also need a plugin for NFD that adds appropriate labels to your nodes; these might be generic labels or they could be vendor specific. Your GPU vendor may provide a third party plugin for NFD; check their documentation for more details.

```
apiVersion: v1
kind: Podmetadata:  name: example-vector-addspec:  restartPolicy: OnFailure  # You can use Kubernetes node affinity to schedule th
```

GPU vendor implementations

- Intel
- NVIDIA

Imperative Management of Kubernetes Objects Using Configuration Files

Kubernetes objects can be created, updated, and deleted by using the `kubectl` command-line tool along with an object configuration file written in YAML or JSON. This document explains how to define and manage objects using configuration files.

Before you begin

Install `kubectl`.

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

To check the version, enter `kubectl version`.

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See Kubernetes Object Management for a discussion of the advantages and disadvantage of each kind of object management.

How to create objects

You can use `kubectl create -f` to create an object from a configuration file. Refer to the kubernetes API reference for details.

- `kubectl create -f <filename|url>`

How to update objects

Warning:

Updating objects with the `replace` command drops all parts of the spec not specified in the configuration file. This should not be used with objects whose specs are partially managed by the cluster, such as Services of type `LoadBalancer`, where the `externalIPs` field is managed independently from the configuration file. Independently managed fields must be copied to the configuration file to prevent `replace` from dropping them.

You can use `kubectl replace -f` to update a live object according to a configuration file.

- `kubectl replace -f <filename|url>`

How to delete objects

You can use `kubectl delete -f` to delete an object that is described in a configuration file.

- `kubectl delete -f <filename|url>`

Note:

If configuration file has specified the `generateName` field in the `metadata` section instead of the `name` field, you cannot delete the object using `kubectl delete -f <filename|url>`. You will have to use other flags for deleting the object. For example:

```
kubectl delete <type> <name>
```

```
kubectl delete <type> -l <label>
```

How to view an object

You can use `kubectl get -f` to view information about an object that is described in a configuration file.

- `kubectl get -f <filename|url> -o yaml`

The `-o yaml` flag specifies that the full object configuration is printed. Use `kubectl get -h` to see a list of options.

Limitations

The `create`, `replace`, and `delete` commands work well when each object's configuration is fully defined and recorded in its configuration file. However when a live object is updated, and the updates are not merged into its configuration file, the updates will be lost the next time a `replace` is executed. This can happen if a controller, such as a HorizontalPodAutoscaler, makes updates directly to a live object. Here's an example:

1. You create an object from a configuration file.
2. Another source updates the object by changing some field.
3. You replace the object from the configuration file. Changes made by the other source in step 2 are lost.

If you need to support multiple writers to the same object, you can use `kubectl apply` to manage the object.

Creating and editing an object from a URL without saving the configuration

Suppose you have the URL of an object configuration file. You can use `kubectl create --edit` to make changes to the configuration before the object is created. This is particularly useful for tutorials and tasks that point to a configuration file that could be modified by the reader.

```
kubectl create -f <url> --edit
```

Migrating from imperative commands to imperative object configuration

Migrating from imperative commands to imperative object configuration involves several manual steps.

1. Export the live object to a local object configuration file:

   ```
   kubectl get <kind>/<name> -o yaml > <kind>_<name>.yaml
   ```

2. Manually remove the status field from the object configuration file.

3. For subsequent object management, use `replace` exclusively.

   ```
   kubectl replace -f <kind>_<name>.yaml
   ```

Defining controller selectors and PodTemplate labels

Warning:

Updating selectors on controllers is strongly discouraged.

The recommended approach is to define a single, immutable PodTemplate label used only by the controller selector with no other semantic meaning.

Example label:

```
selector:
  matchLabels:
      controller-selector: "apps/v1/deployment/nginx"
template:  metadata:    labels:      controller-selector: "apps/v1/deployment/nginx"
```

What's next

- Managing Kubernetes Objects Using Imperative Commands
- Declarative Management of Kubernetes Objects Using Configuration Files
- Kubectl Command Reference
- Kubernetes API Reference

Manage Kubernetes Objects

Declarative and imperative paradigms for interacting with the Kubernetes API.

Declarative Management of Kubernetes Objects Using Configuration Files

Declarative Management of Kubernetes Objects Using Kustomize

Managing Kubernetes Objects Using Imperative Commands

Imperative Management of Kubernetes Objects Using Configuration Files

[Update API Objects in Place Using kubectl patch]

Use kubectl patch to update Kubernetes API objects in place. Do a strategic merge patch or a JSON merge patch.

[Migrate Kubernetes Objects Using Storage Version Migration]

Manage HugePages

Configure and manage huge pages as a schedulable resource in a cluster.
FEATURE STATE: `Kubernetes v1.14 [stable]` (enabled by default: true)

Kubernetes supports the allocation and consumption of pre-allocated huge pages by applications in a Pod. This page describes how users can consume huge pages.

Before you begin

Kubernetes nodes must pre-allocate huge pages in order for the node to report its huge page capacity.

A node can pre-allocate huge pages for multiple sizes, for instance, the following line in `/etc/default/grub` allocates `2*1GiB` of 1 GiB and `512*2 MiB` of 2 MiB pages:

```
GRUB_CMDLINE_LINUX="hugepagesz=1G hugepages=2 hugepagesz=2M hugepages=512"
```

The nodes will automatically discover and report all huge page resources as schedulable resources.

When you describe the Node, you should see something similar to the following in the following in the `Capacity` and `Allocatable` sections:

```
Capacity:
  cpu:                ...
  ephemeral-storage:  ...
  hugepages-1Gi:      2Gi
  hugepages-2Mi:      1Gi
  memory:             ...
  pods:               ...
Allocatable:
  cpu:                ...
  ephemeral-storage:  ...
  hugepages-1Gi:      2Gi
  hugepages-2Mi:      1Gi
  memory:             ...
  pods:               ...
```

Note:

For dynamically allocated pages (after boot), the Kubelet needs to be restarted for the new allocations to be refrelected.

API

Huge pages can be consumed via container level resource requirements using the resource name `hugepages-<size>`, where `<size>` is the most compact binary notation using integer values supported on a particular node. For example, if a node supports 2048KiB and 1048576KiB page sizes, it will expose a schedulable resources `hugepages-2Mi` and `hugepages-1Gi`. Unlike CPU or memory, huge pages do not support overcommit. Note that when requesting hugepage resources, either memory or CPU resources must be requested as well.

A pod may consume multiple huge page sizes in a single pod spec. In this case it must use `medium: HugePages-<hugepagesize>` notation for all volume mounts.

```
apiVersion: v1
kind: Podmetadata:  name: huge-pages-examplespec:  containers:  - name: example    image: fedora:latest    command:    - sleep    
```

A pod may use `medium: HugePages` only if it requests huge pages of one size.

```
apiVersion: v1
kind: Podmetadata:  name: huge-pages-examplespec:  containers:  - name: example    image: fedora:latest    command:    - sleep    
```

- Huge page requests must equal the limits. This is the default if limits are specified, but requests are not.
- Huge pages are isolated at a container scope, so each container has own limit on their cgroup sandbox as requested in a container spec.
- EmptyDir volumes backed by huge pages may not consume more huge page memory than the pod request.
- Applications that consume huge pages via `shmget()` with `SHM_HUGETLB` must run with a supplemental group that matches `proc/sys/vm/hugetlb_shm_group`.
- Huge page usage in a namespace is controllable via ResourceQuota similar to other compute resources like `cpu` or `memory` using the `hugepages-<size>` token.

Migrate Kubernetes Objects Using Storage Version Migration

FEATURE STATE: `Kubernetes v1.30 [alpha]` (enabled by default: false)

Kubernetes relies on API data being actively re-written, to support some maintenance activities related to at rest storage. Two prominent examples are the versioned schema of stored resources (that is, the preferred storage schema changing from v1 to v2 for a given resource) and encryption at rest (that is,

rewriting stale data based on a change in how the data should be encrypted).

Before you begin

Install `kubectl`.

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.30.

To check the version, enter `kubectl version`.

Ensure that your cluster has the `StorageVersionMigrator` and `InformerResourceVersion` feature gates enabled. You will need control plane administrator access to make that change.

Enable storage version migration REST api by setting runtime config `storagemigration.k8s.io/v1alpha1` to `true` for the API server. For more information on how to do that, read enable or disable a Kubernetes API.

Re-encrypt Kubernetes secrets using storage version migration

- To begin with, configure KMS provider to encrypt data at rest in etcd using following encryption configuration.

  ```
  kind: EncryptionConfiguration
  apiVersion: apiserver.config.k8s.io/v1resources:- resources:  - secrets  providers:  - aescbc:      keys:      - name: key1
  ```

  Make sure to enable automatic reload of encryption configuration file by setting `--encryption-provider-config-automatic-reload` to true.

- Create a Secret using kubectl.

  ```
  kubectl create secret generic my-secret --from-literal=key1=supersecret
  ```

- Verify the serialized data for that Secret object is prefixed with `k8s:enc:aescbc:v1:key1`.

- Update the encryption configuration file as follows to rotate the encryption key.

  ```
  kind: EncryptionConfiguration
  apiVersion: apiserver.config.k8s.io/v1resources:- resources:  - secrets  providers:  - aescbc:      keys:      - name: key2
  ```

- To ensure that previously created secret `my-secret` is re-encrypted with new key `key2`, you will use *Storage Version Migration*.

- Create a StorageVersionMigration manifest named `migrate-secret.yaml` as follows:

  ```
  kind: StorageVersionMigration
  apiVersion: storagemigration.k8s.io/v1alpha1metadata:  name: secrets-migrationspec:  resource:    group: ""    version: v1
  ```

  Create the object using *kubectl* as follows:

  ```
  kubectl apply -f migrate-secret.yaml
  ```

- Monitor migration of Secrets by checking the `.status` of the StorageVersionMigration. A successful migration should have its `Succeeded` condition set to true. Get the StorageVersionMigration object as follows:

  ```
  kubectl get storageversionmigration.storagemigration.k8s.io/secrets-migration -o yaml
  ```

  The output is similar to:

  ```
  kind: StorageVersionMigration
  apiVersion: storagemigration.k8s.io/v1alpha1metadata:  name: secrets-migration  uid: 628f6922-a9cb-4514-b076-12d3c178967c  re
  ```

- Verify the stored secret is now prefixed with `k8s:enc:aescbc:v1:key2`.

Update the preferred storage schema of a CRD

Consider a scenario where a CustomResourceDefinition (CRD) is created to serve custom resources (CRs) and is set as the preferred storage schema. When it's time to introduce v2 of the CRD, it can be added for serving only with a conversion webhook. This enables a smoother transition where users can create CRs using either the v1 or v2 schema, with the webhook in place to perform the necessary schema conversion between them. Before setting v2 as the preferred storage schema version, it's important to ensure that all existing CRs stored as v1 are migrated to v2. This migration can be achieved through *Storage Version Migration* to migrate all CRs from v1 to v2.

- Create a manifest for the CRD, named `test-crd.yaml`, as follows:

  ```
  apiVersion: apiextensions.k8s.io/v1
  kind: CustomResourceDefinitionmetadata:  name: selfierequests.stable.example.comspec:  group: stable.example.com  names:    p
  ```

  Create CRD using kubectl:

  ```
  kubectl apply -f test-crd.yaml
  ```

- Create a manifest for an example testcrd. Name the manifest `cr1.yaml` and use these contents:

  ```
  apiVersion: stable.example.com/v1
  kind: SelfieRequestmetadata:  name: cr1  namespace: default
  ```

  Create CR using kubectl:

  ```
  kubectl apply -f cr1.yaml
  ```

- Verify that CR is written and stored as v1 by getting the object from etcd.

  ```
  ETCDCTL_API=3 etcdctl get /kubernetes.io/stable.example.com/testcrds/default/cr1 [...] | hexdump -C
  ```

  where `[...]` contains the additional arguments for connecting to the etcd server.

- Update the CRD `test-crd.yaml` to include v2 version for serving and storage and v1 as serving only, as follows:

  ```
  apiVersion: apiextensions.k8s.io/v1
  kind: CustomResourceDefinitionmetadata:name: selfierequests.stable.example.comspec:  group: stable.example.com  names:    plu
  ```

  Update CRD using kubectl:

  ```
  kubectl apply -f test-crd.yaml
  ```

- Create CR resource file with name `cr2.yaml` as follows:

  ```
  apiVersion: stable.example.com/v2
  kind: SelfieRequestmetadata:  name: cr2  namespace: default
  ```

- Create CR using kubectl:

  ```
  kubectl apply -f cr2.yaml
  ```

- Verify that CR is written and stored as v2 by getting the object from etcd.

  ```
  ETCDCTL_API=3 etcdctl get /kubernetes.io/stable.example.com/testcrds/default/cr2 [...] | hexdump -C
  ```

  where `[...]` contains the additional arguments for connecting to the etcd server.

- Create a StorageVersionMigration manifest named `migrate-crd.yaml`, with the contents as follows:

  ```
  kind: StorageVersionMigration
  apiVersion: storagemigration.k8s.io/v1alpha1metadata:  name: crdsvmspec:  resource:    group: stable.example.com    version:
  ```

  Create the object using *kubectl* as follows:

  ```
  kubectl apply -f migrate-crd.yaml
  ```

- Monitor migration of secrets using status. Successful migration should have `Succeeded` condition set to "True" in the status field. Get the migration resource as follows:

  ```
  kubectl get storageversionmigration.storagemigration.k8s.io/crdsvm -o yaml
  ```

  The output is similar to:

  ```
  kind: StorageVersionMigration
  apiVersion: storagemigration.k8s.io/v1alpha1metadata:  name: crdsvm  uid: 13062fe4-32d7-47cc-9528-5067fa0c6ac8  resourceVersi
  ```

- Verify that previously created cr1 is now written and stored as v2 by getting the object from etcd.

  ```
  ETCDCTL_API=3 etcdctl get /kubernetes.io/stable.example.com/testcrds/default/cr1 [...] | hexdump -C
  ```

  where `[...]` contains the additional arguments for connecting to the etcd server.

Managing Kubernetes Objects Using Imperative Commands

Kubernetes objects can quickly be created, updated, and deleted directly using imperative commands built into the `kubectl` command-line tool. This document explains how those commands are organized and how to use them to manage live objects.

Before you begin

Install `kubectl`.

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

To check the version, enter `kubectl version`.

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See Kubernetes Object Management for a discussion of the advantages and disadvantage of each kind of object management.

How to create objects

The `kubectl` tool supports verb-driven commands for creating some of the most common object types. The commands are named to be recognizable to users unfamiliar with the Kubernetes object types.

- `run`: Create a new Pod to run a Container.
- `expose`: Create a new Service object to load balance traffic across Pods.
- `autoscale`: Create a new Autoscaler object to automatically horizontally scale a controller, such as a Deployment.

The `kubectl` tool also supports creation commands driven by object type. These commands support more object types and are more explicit about their intent, but require users to know the type of objects they intend to create.

- `create <objecttype> [<subtype>] <instancename>`

Some objects types have subtypes that you can specify in the `create` command. For example, the Service object has several subtypes including ClusterIP, LoadBalancer, and NodePort. Here's an example that creates a Service with subtype NodePort:

```
kubectl create service nodeport <myservicename>
```

In the preceding example, the `create service nodeport` command is called a subcommand of the `create service` command.

You can use the `-h` flag to find the arguments and flags supported by a subcommand:

```
kubectl create service nodeport -h
```

How to update objects

The `kubectl` command supports verb-driven commands for some common update operations. These commands are named to enable users unfamiliar with Kubernetes objects to perform updates without knowing the specific fields that must be set:

- `scale`: Horizontally scale a controller to add or remove Pods by updating the replica count of the controller.
- `annotate`: Add or remove an annotation from an object.
- `label`: Add or remove a label from an object.

The `kubectl` command also supports update commands driven by an aspect of the object. Setting this aspect may set different fields for different object types:

- `set <field>`: Set an aspect of an object.

Note:

In Kubernetes version 1.5, not every verb-driven command has an associated aspect-driven command.

The `kubectl` tool supports these additional ways to update a live object directly, however they require a better understanding of the Kubernetes object schema.

- `edit`: Directly edit the raw configuration of a live object by opening its configuration in an editor.
- `patch`: Directly modify specific fields of a live object by using a patch string. For more details on patch strings, see the patch section in API Conventions.

How to delete objects

You can use the `delete` command to delete an object from a cluster:

- `delete <type>/<name>`

Note:

You can use `kubectl delete` for both imperative commands and imperative object configuration. The difference is in the arguments passed to the command. To use `kubectl delete` as an imperative command, pass the object to be deleted as an argument. Here's an example that passes a Deployment object named nginx:

```
kubectl delete deployment/nginx
```

How to view an object

There are several commands for printing information about an object:

- `get`: Prints basic information about matching objects. Use `get -h` to see a list of options.

- `describe`: Prints aggregated detailed information about matching objects.
- `logs`: Prints the stdout and stderr for a container running in a Pod.

Using `set` commands to modify objects before creation

There are some object fields that don't have a flag you can use in a `create` command. In some of those cases, you can use a combination of `set` and `create` to specify a value for the field before object creation. This is done by piping the output of the `create` command to the `set` command, and then back to the `create` command. Here's an example:

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --dry-run=client | kubectl set selector --local -f - 'environme
```

1. The `kubectl create service -o yaml --dry-run=client` command creates the configuration for the Service, but prints it to stdout as YAML instead of sending it to the Kubernetes API server.
2. The `kubectl set selector --local -f - -o yaml` command reads the configuration from stdin, and writes the updated configuration to stdout as YAML.
3. The `kubectl create -f -` command creates the object using the configuration provided via stdin.

Using `--edit` to modify objects before creation

You can use `kubectl create --edit` to make arbitrary changes to an object before it is created. Here's an example:

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --dry-run=client > /tmp/srv.yaml
kubectl create --edit -f /tmp/srv.yaml
```

1. The `kubectl create service` command creates the configuration for the Service and saves it to `/tmp/srv.yaml`.
2. The `kubectl create --edit` command opens the configuration file for editing before it creates the object.

What's next

- Imperative Management of Kubernetes Objects Using Configuration Files
- Declarative Management of Kubernetes Objects Using Configuration Files
- Kubectl Command Reference
- Kubernetes API Reference

Declarative Management of Kubernetes Objects Using Kustomize

Kustomize is a standalone tool to customize Kubernetes objects through a kustomization file.

Since 1.14, kubectl also supports the management of Kubernetes objects using a kustomization file. To view resources found in a directory containing a kustomization file, run the following command:

```
kubectl kustomize <kustomization_directory>
```

To apply those resources, run `kubectl apply` with `--kustomize` or `-k` flag:

```
kubectl apply -k <kustomization_directory>
```

Before you begin

Install `kubectl`.

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

To check the version, enter `kubectl version`.

Overview of Kustomize

Kustomize is a tool for customizing Kubernetes configurations. It has the following features to manage application configuration files:

- generating resources from other sources
- setting cross-cutting fields for resources
- composing and customizing collections of resources

Generating Resources

ConfigMaps and Secrets hold configuration or sensitive data that are used by other Kubernetes objects, such as Pods. The source of truth of ConfigMaps or Secrets are usually external to a cluster, such as a `.properties` file or an SSH keyfile. Kustomize has `secretGenerator` and `configMapGenerator`, which generate Secret and ConfigMap from files or literals.

configMapGenerator

To generate a ConfigMap from a file, add an entry to the `files` list in `configMapGenerator`. Here is an example of generating a ConfigMap with a data item from a `.properties` file:

```
# Create a application.properties file
cat <<EOF >application.properties
FOO=Bar
EOF

cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-1
  files:
  - application.properties
EOF
```

The generated ConfigMap can be examined with the following command:

```
kubectl kustomize ./
```

The generated ConfigMap is:

```
apiVersion: v1
data:  application.properties: |    FOO=Bar    kind: ConfigMapmetadata:  name: example-configmap-1-8mbdf7882g
```

To generate a ConfigMap from an env file, add an entry to the `envs` list in `configMapGenerator`. Here is an example of generating a ConfigMap with a data item from a `.env` file:

```
# Create a .env file
cat <<EOF >.env
FOO=Bar
EOF

cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-1
  envs:
  - .env
EOF
```

The generated ConfigMap can be examined with the following command:

```
kubectl kustomize ./
```

The generated ConfigMap is:

```
apiVersion: v1
data:  FOO: Barkind: ConfigMapmetadata:  name: example-configmap-1-42cfbf598f
```

Note:

Each variable in the `.env` file becomes a separate key in the ConfigMap that you generate. This is different from the previous example which embeds a file named `application.properties` (and all its entries) as the value for a single key.

ConfigMaps can also be generated from literal key-value pairs. To generate a ConfigMap from a literal key-value pair, add an entry to the `literals` list in configMapGenerator. Here is an example of generating a ConfigMap with a data item from a key-value pair:

```
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-2
  literals:
  - FOO=Bar
EOF
```

The generated ConfigMap can be checked by the following command:

```
kubectl kustomize ./
```

The generated ConfigMap is:

```
apiVersion: v1
data:  FOO: Barkind: ConfigMapmetadata:  name: example-configmap-2-g2hdhfc6tk
```

To use a generated ConfigMap in a Deployment, reference it by the name of the configMapGenerator. Kustomize will automatically replace this name with the generated name.

This is an example deployment that uses a generated ConfigMap:

```
# Create an application.properties file
cat <<EOF >application.propertiesFOO=BarEOFcat <<EOF >deployment.yamlapiVersion: apps/v1kind: Deploymentmetadata:  name: my-app  l
```

Generate the ConfigMap and Deployment:

```
kubectl kustomize ./
```

The generated Deployment will refer to the generated ConfigMap by name:

```
apiVersion: v1
data:  application.properties: |    FOO=Bar    kind: ConfigMapmetadata:  name: example-configmap-1-g4hk9g2ff8---apiVersion: apps/v
```

secretGenerator

You can generate Secrets from files or literal key-value pairs. To generate a Secret from a file, add an entry to the `files` list in `secretGenerator`. Here is an example of generating a Secret with a data item from a file:

```
# Create a password.txt file
cat <<EOF >./password.txt
username=admin
password=secret
EOF

cat <<EOF >./kustomization.yaml
secretGenerator:
- name: example-secret-1
  files:
  - password.txt
EOF
```

The generated Secret is as follows:

```
apiVersion: v1
data:  password.txt: dXNlcm5hbWU9YWRtaW4KcGFzc3dvcmQ9c2VjcmV0Cg==kind: Secretmetadata:  name: example-secret-1-t2kt65hgtbtype: Opa
```

To generate a Secret from a literal key-value pair, add an entry to `literals` list in `secretGenerator`. Here is an example of generating a Secret with a data item from a key-value pair:

```
cat <<EOF >./kustomization.yaml
secretGenerator:
- name: example-secret-2
  literals:
  - username=admin
  - password=secret
EOF
```

The generated Secret is as follows:

```
apiVersion: v1
data:  password: c2VjcmV0  username: YWRtaW4=kind: Secretmetadata:  name: example-secret-2-t52t6g96d8type: Opaque
```

Like ConfigMaps, generated Secrets can be used in Deployments by referring to the name of the secretGenerator:

```
# Create a password.txt file
cat <<EOF >./password.txt
username=admin
password=secret
EOF

cat <<EOF >deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: app
        image: my-app
        volumeMounts:
        - name: password
          mountPath: /secrets
      volumes:
      - name: password
        secret:
          secretName: example-secret-1
EOF

cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
secretGenerator:
- name: example-secret-1
  files:
  - password.txt
EOF
```

generatorOptions

The generated ConfigMaps and Secrets have a content hash suffix appended. This ensures that a new ConfigMap or Secret is generated when the contents are changed. To disable the behavior of appending a suffix, one can use `generatorOptions`. Besides that, it is also possible to specify cross-cutting options for generated ConfigMaps and Secrets.

```
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-3
  literals:
  - FOO=Bar
```

```
generatorOptions:
  disableNameSuffixHash: true
  labels:
    type: generated
  annotations:
    note: generated
EOF
```

Run `kubectl kustomize ./` to view the generated ConfigMap:

```
apiVersion: v1
data:  FOO: Barkind: ConfigMapmetadata:  annotations:    note: generated  labels:    type: generated  name: example-configmap-3
```

Setting cross-cutting fields

It is quite common to set cross-cutting fields for all Kubernetes resources in a project. Some use cases for setting cross-cutting fields:

- setting the same namespace for all resources
- adding the same name prefix or suffix
- adding the same set of labels
- adding the same set of annotations

Here is an example:

```
# Create a deployment.yaml
cat <<EOF >./deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
EOF

cat <<EOF >./kustomization.yaml
namespace: my-namespace
namePrefix: dev-
nameSuffix: "-001"
labels:
  - pairs:
      app: bingo
    includeSelectors: true
commonAnnotations:
  oncallPager: 800-555-1212
resources:
- deployment.yaml
EOF
```

Run `kubectl kustomize ./` to view those fields are all set in the Deployment Resource:

```
apiVersion: apps/v1
kind: Deploymentmetadata:  annotations:    oncallPager: 800-555-1212  labels:    app: bingo  name: dev-nginx-deployment-001  namesp
```

Composing and Customizing Resources

It is common to compose a set of resources in a project and manage them inside the same file or directory. Kustomize offers composing resources from different files and applying patches or other customization to them.

Composing

Kustomize supports composition of different resources. The `resources` field, in the `kustomization.yaml` file, defines the list of resources to include in a configuration. Set the path to a resource's configuration file in the `resources` list. Here is an example of an NGINX application comprised of a Deployment and a Service:

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
```

```
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
EOF

# Create a service.yaml file
cat <<EOF > service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
EOF

# Create a kustomization.yaml composing them
cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
- service.yaml
EOF
```

The resources from `kubectl kustomize ./` contain both the Deployment and the Service objects.

Customizing

Patches can be used to apply different customizations to resources. Kustomize supports different patching mechanisms through `StrategicMerge` and `Json6902` using the `patches` field. `patches` may be a file or an inline string, targeting a single or multiple resources.

The `patches` field contains a list of patches applied in the order they are specified. The patch target selects resources by `group`, `version`, `kind`, `name`, `namespace`, `labelSelector` and `annotationSelector`.

Small patches that do one thing are recommended. For example, create one patch for increasing the deployment replica number and another patch for setting the memory limit. The target resource is matched using `group`, `version`, `kind`, and `name` fields from the patch file.

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
EOF

# Create a patch increase_replicas.yaml
cat <<EOF > increase_replicas.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
EOF

# Create another patch set_memory.yaml
cat <<EOF > set_memory.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  template:
    spec:
      containers:
      - name: my-nginx
        resources:
          limits:
            memory: 512Mi
EOF

cat <<EOF >./kustomization.yaml
```

```
resources:
- deployment.yaml
patches:
  - path: increase_replicas.yaml
  - path: set_memory.yaml
EOF
```

Run `kubectl kustomize ./` to view the Deployment:

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: my-nginxspec:  replicas: 3  selector:    matchLabels:      run: my-nginx  template:    metadata:
```

Not all resources or fields support `strategicMerge` patches. To support modifying arbitrary fields in arbitrary resources, Kustomize offers applying JSON patch through `Json6902`. To find the correct Resource for a `Json6902` patch, it is mandatory to specify the `target` field in `kustomization.yaml`.

For example, increasing the replica number of a Deployment object can also be done through `Json6902` patch. The target resource is matched using `group`, `version`, `kind`, and `name` from the `target` field.

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
EOF

# Create a json patch
cat <<EOF > patch.yaml
- op: replace
  path: /spec/replicas
  value: 3
EOF

# Create a kustomization.yaml
cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml

patches:
- target:
    group: apps
    version: v1
    kind: Deployment
    name: my-nginx
  path: patch.yaml
EOF
```

Run `kubectl kustomize ./` to see the `replicas` field is updated:

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: my-nginxspec:  replicas: 3  selector:    matchLabels:      run: my-nginx  template:    metadata:
```

In addition to patches, Kustomize also offers customizing container images or injecting field values from other objects into containers without creating patches. For example, you can change the image used inside containers by specifying the new image in the `images` field in `kustomization.yaml`.

```
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
EOF

cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
```

```
images:
- name: nginx
  newName: my.image.registry/nginx
  newTag: "1.4.0"
EOF
```

Run `kubectl kustomize ./` to see that the image being used is updated:

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: my-nginxspec:  replicas: 2  selector:    matchLabels:      run: my-nginx  template:    metadata:
```

Sometimes, the application running in a Pod may need to use configuration values from other objects. For example, a Pod from a Deployment object need to read the corresponding Service name from Env or as a command argument. Since the Service name may change as `namePrefix` or `nameSuffix` is added in the `kustomization.yaml` file. It is not recommended to hard code the Service name in the command argument. For this usage, Kustomize can inject the Service name into containers through `replacements`.

```
# Create a deployment.yaml file (quoting the here doc delimiter)
cat <<'EOF' > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        command: ["start", "--host", "MY_SERVICE_NAME_PLACEHOLDER"]
EOF

# Create a service.yaml file
cat <<EOF > service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
EOF

cat <<EOF >./kustomization.yaml
namePrefix: dev-
nameSuffix: "-001"

resources:
- deployment.yaml
- service.yaml

replacements:
- source:
    kind: Service
    name: my-nginx
    fieldPath: metadata.name
  targets:
  - select:
      kind: Deployment
      name: my-nginx
    fieldPaths:
    - spec.template.spec.containers.0.command.2
EOF
```

Run `kubectl kustomize ./` to see that the Service name injected into containers is `dev-my-nginx-001`:

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: dev-my-nginx-001spec:  replicas: 2  selector:    matchLabels:      run: my-nginx  template:    me
```

Bases and Overlays

Kustomize has the concepts of **bases** and **overlays**. A **base** is a directory with a `kustomization.yaml`, which contains a set of resources and associated customization. A base could be either a local directory or a directory from a remote repo, as long as a `kustomization.yaml` is present inside. An **overlay** is a directory with a `kustomization.yaml` that refers to other kustomization directories as its `bases`. A **base** has no knowledge of an overlay and can be used in multiple overlays.

The `kustomization.yaml` in an **overlay** directory may refer to multiple `bases`, combining all the resources defined in these bases into a unified configuration. Additionally, it can apply customizations on top of these resources to meet specific requirements.

Here is an example of a base:

```
# Create a directory to hold the base
mkdir base
# Create a base/deployment.yaml
cat <<EOF > base/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
EOF

# Create a base/service.yaml file
cat <<EOF > base/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
EOF
# Create a base/kustomization.yaml
cat <<EOF > base/kustomization.yaml
resources:
- deployment.yaml
- service.yaml
EOF
```

This base can be used in multiple overlays. You can add different `namePrefix` or other cross-cutting fields in different overlays. Here are two overlays using the same base.

```
mkdir dev
cat <<EOF > dev/kustomization.yaml
resources:
- ../base
namePrefix: dev-
EOF

mkdir prod
cat <<EOF > prod/kustomization.yaml
resources:
- ../base
namePrefix: prod-
EOF
```

How to apply/view/delete objects using Kustomize

Use `--kustomize` or `-k` in `kubectl` commands to recognize resources managed by `kustomization.yaml`. Note that `-k` should point to a kustomization directory, such as

```
kubectl apply -k <kustomization directory>/
```

Given the following `kustomization.yaml`,

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
EOF
```

```
# Create a kustomization.yaml
cat <<EOF >./kustomization.yaml
namePrefix: dev-
labels:
  - pairs:
      app: my-nginx
    includeSelectors: true
resources:
- deployment.yaml
EOF
```

Run the following command to apply the Deployment object `dev-my-nginx`:

```
> kubectl apply -k ./
deployment.apps/dev-my-nginx created
```

Run one of the following commands to view the Deployment object `dev-my-nginx`:

```
kubectl get -k ./
```

```
kubectl describe -k ./
```

Run the following command to compare the Deployment object `dev-my-nginx` against the state that the cluster would be in if the manifest was applied:

```
kubectl diff -k ./
```

Run the following command to delete the Deployment object `dev-my-nginx`:

```
> kubectl delete -k ./
deployment.apps "dev-my-nginx" deleted
```

Kustomize Feature List

Field	Type	Explanation
bases	[]string	Each entry in this list should resolve to a directory containing a kustomization.yaml file
commonAnnotations	map[string]string	annotations to add to all resources
commonLabels	map[string]string	labels to add to all resources and selectors
configMapGenerator	[]ConfigMapArgs	Each entry in this list generates a ConfigMap
configurations	[]string	Each entry in this list should resolve to a file containing Kustomize transformer configurations
crds	[]string	Each entry in this list should resolve to an OpenAPI definition file for Kubernetes types
generatorOptions	GeneratorOptions	Modify behaviors of all ConfigMap and Secret generator
images	[]Image	Each entry is to modify the name, tags and/or digest for one image without creating patches
labels	map[string]string	Add labels without automatically injecting corresponding selectors
namePrefix	string	value of this field is prepended to the names of all resources
nameSuffix	string	value of this field is appended to the names of all resources
patchesJson6902	[]Patch	Each entry in this list should resolve to a Kubernetes object and a Json Patch
patchesStrategicMerge	[]string	Each entry in this list should resolve a strategic merge patch of a Kubernetes object
replacements	[]Replacements	copy the value from a resource's field into any number of specified targets.
resources	[]string	Each entry in this list must resolve to an existing resource configuration file
secretGenerator	[]SecretArgs	Each entry in this list generates a Secret
vars	[]Var	Each entry is to capture text from one resource's field

What's next

- Kustomize
- Kubectl Book
- Kubectl Command Reference
- Kubernetes API Reference

Declarative Management of Kubernetes Objects Using Configuration Files

Kubernetes objects can be created, updated, and deleted by storing multiple object configuration files in a directory and using `kubectl apply` to recursively create and update those objects as needed. This method retains writes made to live objects without merging the changes back into the object configuration files. `kubectl diff` also gives you a preview of what changes `apply` will make.

Before you begin

Install `kubectl`.

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

- iximiuz Labs

- Killercoda
- KodeKloud
- Play with Kubernetes

To check the version, enter `kubectl version`.

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See Kubernetes Object Management for a discussion of the advantages and disadvantage of each kind of object management.

Overview

Declarative object configuration requires a firm understanding of the Kubernetes object definitions and configuration. Read and complete the following documents if you have not already:

- Managing Kubernetes Objects Using Imperative Commands
- Imperative Management of Kubernetes Objects Using Configuration Files

Following are definitions for terms used in this document:

- *object configuration file* / *configuration file*: A file that defines the configuration for a Kubernetes object. This topic shows how to pass configuration files to `kubectl apply`. Configuration files are typically stored in source control, such as Git.
- *live object configuration* / *live configuration*: The live configuration values of an object, as observed by the Kubernetes cluster. These are kept in the Kubernetes cluster storage, typically etcd.
- *declarative configuration writer* / *declarative writer*: A person or software component that makes updates to a live object. The live writers referred to in this topic make changes to object configuration files and run `kubectl apply` to write the changes.

How to create objects

Use `kubectl apply` to create all objects, except those that already exist, defined by configuration files in a specified directory:

```
kubectl apply -f <directory>
```

This sets the `kubectl.kubernetes.io/last-applied-configuration: '{...}'` annotation on each object. The annotation contains the contents of the object configuration file that was used to create the object.

Note:

Add the `-R` flag to recursively process directories.

Here's an example of an object configuration file:

application/simple_deployment.yaml  Copy application/simple_deployment.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deploymentspec:  selector:    matchLabels:      app: nginx  minReadySeconds: 5  template:
```

Run `kubectl diff` to print the object that will be created:

```
kubectl diff -f https://k8s.io/examples/application/simple_deployment.yaml
```

Note:

`diff` uses server-side dry-run, which needs to be enabled on `kube-apiserver`.

Since `diff` performs a server-side apply request in dry-run mode, it requires granting `PATCH`, `CREATE`, and `UPDATE` permissions. See Dry-Run Authorization for details.

Create the object using `kubectl apply`:

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

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration` annotation was written to the live configuration, and it matches the configuration file:

```
kind: Deployment
metadata:  annotations:    # ...    # This is the json representation of simple_deployment.yaml    # It was written by kubectl app
```

How to update objects

You can also use `kubectl apply` to update all objects defined in a directory, even if those objects already exist. This approach accomplishes the following:

1. Sets fields that appear in the configuration file in the live configuration.
2. Clears fields removed from the configuration file in the live configuration.

```
kubectl diff -f <directory>
kubectl apply -f <directory>
```

Note:

Add the `-R` flag to recursively process directories.

Here's an example configuration file:

application/simple_deployment.yaml  Copy application/simple_deployment.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deploymentspec:  selector:    matchLabels:      app: nginx  minReadySeconds: 5  template:
```

Create the object using `kubectl apply`:

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

Note:

For purposes of illustration, the preceding command refers to a single configuration file instead of a directory.

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration` annotation was written to the live configuration, and it matches the configuration file:

```
kind: Deployment
metadata:  annotations:    # ...    # This is the json representation of simple_deployment.yaml    # It was written by kubectl app
```

Directly update the `replicas` field in the live configuration by using `kubectl scale`. This does not use `kubectl apply`:

```
kubectl scale deployment/nginx-deployment --replicas=2
```

Print the live configuration using `kubectl get`:

```
kubectl get deployment nginx-deployment -o yaml
```

The output shows that the `replicas` field has been set to 2, and the `last-applied-configuration` annotation does not contain a `replicas` field:

```
apiVersion: apps/v1
kind: Deploymentmetadata:  annotations:    # ...    # note that the annotation does not contain replicas    # because it was not u
```

Update the `simple_deployment.yaml` configuration file to change the image from `nginx:1.14.2` to `nginx:1.16.1`, and delete the `minReadySeconds` field:

application/update_deployment.yaml  Copy application/update_deployment.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deploymentspec:  selector:    matchLabels:      app: nginx  template:    metadata:      lab
```

Apply the changes made to the configuration file:

```
kubectl diff -f https://k8s.io/examples/application/update_deployment.yaml
kubectl apply -f https://k8s.io/examples/application/update_deployment.yaml
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/update_deployment.yaml -o yaml
```

The output shows the following changes to the live configuration:

- The `replicas` field retains the value of 2 set by `kubectl scale`. This is possible because it is omitted from the configuration file.
- The `image` field has been updated to `nginx:1.16.1` from `nginx:1.14.2`.
- The `last-applied-configuration` annotation has been updated with the new image.
- The `minReadySeconds` field has been cleared.
- The `last-applied-configuration` annotation no longer contains the `minReadySeconds` field.

```
apiVersion: apps/v1
kind: Deploymentmetadata:  annotations:    # ...    # The annotation contains the updated image to nginx 1.16.1,    # but does not
```

Warning:

Mixing `kubectl apply` with the imperative object configuration commands `create` and `replace` is not supported. This is because `create` and `replace` do not retain the `kubectl.kubernetes.io/last-applied-configuration` that `kubectl apply` uses to compute updates.

How to delete objects

There are two approaches to delete objects managed by `kubectl apply`.

Recommended: `kubectl delete -f <filename>`

Manually deleting objects using the imperative command is the recommended approach, as it is more explicit about what is being deleted, and less likely to result in the user deleting something unintentionally:

```
kubectl delete -f <filename>
```

Alternative: `kubectl apply -f <directory> --prune`

As an alternative to `kubectl delete`, you can use `kubectl apply` to identify objects to be deleted after their manifests have been removed from a directory in the local filesystem.

In Kubernetes 1.34, there are two pruning modes available in kubectl apply:

- Allowlist-based pruning: This mode has existed since kubectl v1.5 but is still in alpha due to usability, correctness and performance issues with its design. The ApplySet-based mode is designed to replace it.
- ApplySet-based pruning: An *apply set* is a server-side object (by default, a Secret) that kubectl can use to accurately and efficiently track set membership across **apply** operations. This mode was introduced in alpha in kubectl v1.27 as a replacement for allowlist-based pruning.

- Allow list
- Apply set

FEATURE STATE: `Kubernetes v1.5 [alpha]`

Warning:

Take care when using `--prune` with `kubectl apply` in allow list mode. Which objects are pruned depends on the values of the `--prune-allowlist`, `--selector` and `--namespace` flags, and relies on dynamic discovery of the objects in scope. Especially if flag values are changed between invocations, this can lead to objects being unexpectedly deleted or retained.

To use allowlist-based pruning, add the following flags to your `kubectl apply` invocation:

- `--prune`: Delete previously applied objects that are not in the set passed to the current invocation.
- `--prune-allowlist`: A list of group-version-kinds (GVKs) to consider for pruning. This flag is optional but strongly encouraged, as its default value is a partial list of both namespaced and cluster-scoped types, which can lead to surprising results.
- `--selector/-l`: Use a label selector to constrain the set of objects selected for pruning. This flag is optional but strongly encouraged.
- `--all`: use instead of `--selector/-l` to explicitly select all previously applied objects of the allowlisted types.

Allowlist-based pruning queries the API server for all objects of the allowlisted GVKs that match the given labels (if any), and attempts to match the returned live object configurations against the object manifest files. If an object matches the query, and it does not have a manifest in the directory, and it has a `kubectl.kubernetes.io/last-applied-configuration` annotation, it is deleted.

```
kubectl apply -f <directory> --prune -l <labels> --prune-allowlist=<gvk-list>
```

Warning:

Apply with prune should only be run against the root directory containing the object manifests. Running against sub-directories can cause objects to be unintentionally deleted if they were previously applied, have the labels given (if any), and do not appear in the subdirectory.

FEATURE STATE: `Kubernetes v1.27 [alpha]`

Caution:

`kubectl apply --prune --applyset` is in alpha, and backwards incompatible changes might be introduced in subsequent releases.

To use ApplySet-based pruning, set the `KUBECTL_APPLYSET=true` environment variable, and add the following flags to your `kubectl apply` invocation:

- `--prune`: Delete previously applied objects that are not in the set passed to the current invocation.
- `--applyset`: The name of an object that kubectl can use to accurately and efficiently track set membership across `apply` operations.

```
KUBECTL_APPLYSET=true kubectl apply -f <directory> --prune --applyset=<name>
```

By default, the type of the ApplySet parent object used is a Secret. However, ConfigMaps can also be used in the format: `--applyset=configmaps/<name>`. When using a Secret or ConfigMap, kubectl will create the object if it does not already exist.

It is also possible to use custom resources as ApplySet parent objects. To enable this, label the Custom Resource Definition (CRD) that defines the resource you want to use with the following: `applyset.kubernetes.io/is-parent-type: true`. Then, create the object you want to use as an ApplySet parent (kubectl does not do this automatically for custom resources). Finally, refer to that object in the applyset flag as follows: `--applyset=<resource>.<group>/<name>` (for example, `widgets.custom.example.com/widget-name`).

With ApplySet-based pruning, kubectl adds the `applyset.kubernetes.io/part-of=<parentID>` label to each object in the set before they are sent to the server. For performance reasons, it also collects the list of resource types and namespaces that the set contains and adds these in annotations on the live parent object. Finally, at the end of the apply operation, it queries the API server for objects of those types in those namespaces (or in the cluster scope, as applicable) that belong to the set, as defined by the `applyset.kubernetes.io/part-of=<parentID>` label.

Caveats and restrictions:

- Each object may be a member of at most one set.
- The `--namespace` flag is required when using any namespaced parent, including the default Secret. This means that ApplySets spanning multiple namespaces must use a cluster-scoped custom resource as the parent object.
- To safely use ApplySet-based pruning with multiple directories, use a unique ApplySet name for each.

How to view an object

You can use `kubectl get` with `-o yaml` to view the configuration of a live object:

```
kubectl get -f <filename|url> -o yaml
```

How apply calculates differences and merges changes

Caution:

A *patch* is an update operation that is scoped to specific fields of an object instead of the entire object. This enables updating only a specific set of fields on an object without reading the object first.

When `kubectl apply` updates the live configuration for an object, it does so by sending a patch request to the API server. The patch defines updates scoped to specific fields of the live object configuration. The `kubectl apply` command calculates this patch request using the configuration file, the live configuration, and the `last-applied-configuration` annotation stored in the live configuration.

Merge patch calculation

The `kubectl apply` command writes the contents of the configuration file to the `kubectl.kubernetes.io/last-applied-configuration` annotation. This is used to identify fields that have been removed from the configuration file and need to be cleared from the live configuration. Here are the steps used to calculate which fields should be deleted or set:

1. Calculate the fields to delete. These are the fields present in `last-applied-configuration` and missing from the configuration file.
2. Calculate the fields to add or set. These are the fields present in the configuration file whose values don't match the live configuration.

Here's an example. Suppose this is the configuration file for a Deployment object:

application/update_deployment.yaml Copy application/update_deployment.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deploymentspec:  selector:    matchLabels:      app: nginx  template:    metadata:      lab
```

Also, suppose this is the live configuration for the same Deployment object:

```
apiVersion: apps/v1
kind: Deploymentmetadata:  annotations:    # ...    # note that the annotation does not contain replicas    # because it was not u
```

Here are the merge calculations that would be performed by `kubectl apply`:

1. Calculate the fields to delete by reading values from `last-applied-configuration` and comparing them to values in the configuration file. Clear fields explicitly set to null in the local object configuration file regardless of whether they appear in the `last-applied-configuration`. In this example, `minReadySeconds` appears in the `last-applied-configuration` annotation, but does not appear in the configuration file. **Action:** Clear `minReadySeconds` from the live configuration.
2. Calculate the fields to set by reading values from the configuration file and comparing them to values in the live configuration. In this example, the value of `image` in the configuration file does not match the value in the live configuration. **Action:** Set the value of `image` in the live configuration.
3. Set the `last-applied-configuration` annotation to match the value of the configuration file.
4. Merge the results from 1, 2, 3 into a single patch request to the API server.

Here is the live configuration that is the result of the merge:

```
apiVersion: apps/v1
kind: Deploymentmetadata:  annotations:    # ...    # The annotation contains the updated image to nginx 1.16.1,    # but does not
```

How different types of fields are merged

How a particular field in a configuration file is merged with the live configuration depends on the type of the field. There are several types of fields:

- *primitive*: A field of type string, integer, or boolean. For example, `image` and `replicas` are primitive fields. **Action:** Replace.
- *map*, also called *object*: A field of type map or a complex type that contains subfields. For example, `labels`, `annotations`,`spec` and `metadata` are all maps. **Action:** Merge elements or subfields.
- *list*: A field containing a list of items that can be either primitive types or maps. For example, `containers`, `ports`, and `args` are lists. **Action:** Varies.

When `kubectl apply` updates a map or list field, it typically does not replace the entire field, but instead updates the individual subelements. For instance, when merging the `spec` on a Deployment, the entire `spec` is not replaced. Instead the subfields of `spec`, such as `replicas`, are compared and merged.

Merging changes to primitive fields

Primitive fields are replaced or cleared.

Note:

- is used for "not applicable" because the value is not used.

Field in object configuration file	Field in live object configuration	Field in last-applied-configuration	Action
Yes	Yes	-	Set live to configuration file value.
Yes	No	-	Set live to local configuration.
No	-	Yes	Clear from live configuration.
No	-	No	Do nothing. Keep live value.

Merging changes to map fields

Fields that represent maps are merged by comparing each of the subfields or elements of the map:

Note:

- is used for "not applicable" because the value is not used.

Key in object configuration file	Key in live object configuration	Field in last-applied-configuration	Action
Yes	Yes	-	Compare sub fields values.
Yes	No	-	Set live to local configuration.
No	-	Yes	Delete from live configuration.
No	-	No	Do nothing. Keep live value.

Merging changes for fields of type list

Merging changes to a list uses one of three strategies:

- Replace the list if all its elements are primitives.
- Merge individual elements in a list of complex elements.
- Merge a list of primitive elements.

The choice of strategy is made on a per-field basis.

Replace the list if all its elements are primitives

Treat the list the same as a primitive field. Replace or delete the entire list. This preserves ordering.

Example: Use `kubectl apply` to update the `args` field of a Container in a Pod. This sets the value of `args` in the live configuration to the value in the configuration file. Any `args` elements that had previously been added to the live configuration are lost. The order of the `args` elements defined in the configuration file is retained in the live configuration.

```
# last-applied-configuration value
    args: ["a", "b"]

# configuration file value    args: ["a", "c"]# live configuration    args: ["a", "b", "d"]# result after merge    args: ["a", "c"
```

Explanation: The merge used the configuration file value as the new list value.

Merge individual elements of a list of complex elements:

Treat the list as a map, and treat a specific field of each element as a key. Add, delete, or update individual elements. This does not preserve ordering.

This merge strategy uses a special tag on each field called a `patchMergeKey`. The `patchMergeKey` is defined for each field in the Kubernetes source code: types.go When merging a list of maps, the field specified as the `patchMergeKey` for a given element is used like a map key for that element.

Example: Use `kubectl apply` to update the `containers` field of a PodSpec. This merges the list as though it was a map where each element is keyed by `name`.

```
# last-applied-configuration value
    containers:
    - name: nginx
      image: nginx:1.16
    - name: nginx-helper-a # key: nginx-helper-a; will be deleted in result
      image: helper:1.3
    - name: nginx-helper-b # key: nginx-helper-b; will be retained
      image: helper:1.3

# configuration file value    containers:    - name: nginx      image: nginx:1.16    - name: nginx-helper-b      image: helper:1.3
```

Explanation:

- The container named "nginx-helper-a" was deleted because no container named "nginx-helper-a" appeared in the configuration file.
- The container named "nginx-helper-b" retained the changes to `args` in the live configuration. `kubectl apply` was able to identify that "nginx-helper-b" in the live configuration was the same "nginx-helper-b" as in the configuration file, even though their fields had different values (no `args` in the configuration file). This is because the `patchMergeKey` field value (name) was identical in both.
- The container named "nginx-helper-c" was added because no container with that name appeared in the live configuration, but one with that name appeared in the configuration file.
- The container named "nginx-helper-d" was retained because no element with that name appeared in the last-applied-configuration.

Merge a list of primitive elements

As of Kubernetes 1.5, merging lists of primitive elements is not supported.

Note:

Which of the above strategies is chosen for a given field is controlled by the `patchStrategy` tag in types.go If no `patchStrategy` is specified for a field of type list, then the list is replaced.

Default field values

The API server sets certain fields to default values in the live configuration if they are not specified when the object is created.

Here's a configuration file for a Deployment. The file does not specify `strategy`:

application/simple_deployment.yaml 

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deploymentspec:  selector:    matchLabels:      app: nginx  minReadySeconds: 5  template:
```

Create the object using `kubectl apply`:

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

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

The output shows that the API server set several fields to default values in the live configuration. These fields were not specified in the configuration file.

```
apiVersion: apps/v1
kind: Deployment# ...spec:  selector:    matchLabels:      app: nginx  minReadySeconds: 5  replicas: 1 # defaulted by apiserver  s
```

In a patch request, defaulted fields are not re-defaulted unless they are explicitly cleared as part of a patch request. This can cause unexpected behavior for fields that are defaulted based on the values of other fields. When the other fields are later changed, the values defaulted from them will not be updated unless they are explicitly cleared.

For this reason, it is recommended that certain fields defaulted by the server are explicitly defined in the configuration file, even if the desired values match the server defaults. This makes it easier to recognize conflicting values that will not be re-defaulted by the server.

Example:

```
# last-applied-configuration
spec:  template:    metadata:      labels:        app: nginx    spec:      containers:      - name: nginx        image: nginx:1.14
```

Explanation:

1. The user creates a Deployment without defining `strategy.type`.
2. The server defaults `strategy.type` to `RollingUpdate` and defaults the `strategy.rollingUpdate` values.
3. The user changes `strategy.type` to `Recreate`. The `strategy.rollingUpdate` values remain at their defaulted values, though the server expects them to be cleared. If the `strategy.rollingUpdate` values had been defined initially in the configuration file, it would have been more clear that they needed to be deleted.
4. Apply fails because `strategy.rollingUpdate` is not cleared. The `strategy.rollingupdate` field cannot be defined with a `strategy.type` of `Recreate`.

Recommendation: These fields should be explicitly defined in the object configuration file:

- Selectors and PodTemplate labels on workloads, such as Deployment, StatefulSet, Job, DaemonSet, ReplicaSet, and ReplicationController
- Deployment rollout strategy

How to clear server-defaulted fields or fields set by other writers

Fields that do not appear in the configuration file can be cleared by setting their values to `null` and then applying the configuration file. For fields defaulted by the server, this triggers re-defaulting the values.

How to change ownership of a field between the configuration file and direct imperative writers

These are the only methods you should use to change an individual object field:

- Use `kubectl apply`.
- Write directly to the live configuration without modifying the configuration file: for example, use `kubectl scale`.

Changing the owner from a direct imperative writer to a configuration file

Add the field to the configuration file. For the field, discontinue direct updates to the live configuration that do not go through `kubectl apply`.

Changing the owner from a configuration file to a direct imperative writer

As of Kubernetes 1.5, changing ownership of a field from a configuration file to an imperative writer requires manual steps:

- Remove the field from the configuration file.
- Remove the field from the `kubectl.kubernetes.io/last-applied-configuration` annotation on the live object.

Changing management methods

Kubernetes objects should be managed using only one method at a time. Switching from one method to another is possible, but is a manual process.

Note:

It is OK to use imperative deletion with declarative management.

Migrating from imperative command management to declarative object configuration

Migrating from imperative command management to declarative object configuration involves several manual steps:

1. Export the live object to a local configuration file:

   ```
   kubectl get <kind>/<name> -o yaml > <kind>_<name>.yaml
   ```

2. Manually remove the `status` field from the configuration file.

   Note:

   This step is optional, as `kubectl apply` does not update the status field even if it is present in the configuration file.

3. Set the `kubectl.kubernetes.io/last-applied-configuration` annotation on the object:

   ```
   kubectl replace --save-config -f <kind>_<name>.yaml
   ```

4. Change processes to use `kubectl apply` for managing the object exclusively.

Migrating from imperative object configuration to declarative object configuration

1. Set the `kubectl.kubernetes.io/last-applied-configuration` annotation on the object:

   ```
   kubectl replace --save-config -f <kind>_<name>.yaml
   ```

2. Change processes to use `kubectl apply` for managing the object exclusively.

Defining controller selectors and PodTemplate labels

Warning:

Updating selectors on controllers is strongly discouraged.

The recommended approach is to define a single, immutable PodTemplate label used only by the controller selector with no other semantic meaning.

Example:

```
selector:
  matchLabels:
      controller-selector: "apps/v1/deployment/nginx"
template:  metadata:    labels:      controller-selector: "apps/v1/deployment/nginx"
```

What's next

- Managing Kubernetes Objects Using Imperative Commands
- Imperative Management of Kubernetes Objects Using Configuration Files
- Kubectl Command Reference
- Kubernetes API Reference

Perform a Rollback on a DaemonSet

This page shows how to perform a rollback on a DaemonSet.

Before you begin

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version 1.7.

To check the version, enter `kubectl version`.

You should already know how to perform a rolling update on a DaemonSet.

Performing a rollback on a DaemonSet

Step 1: Find the DaemonSet revision you want to roll back to

You can skip this step if you only want to roll back to the last revision.

List all revisions of a DaemonSet:

```
kubectl rollout history daemonset <daemonset-name>
```

This returns a list of DaemonSet revisions:

```
daemonsets "<daemonset-name>"
REVISION        CHANGE-CAUSE
1               ...
2               ...
...
```

- Change cause is copied from DaemonSet annotation `kubernetes.io/change-cause` to its revisions upon creation. You may specify `--record=true` in `kubectl` to record the command executed in the change cause annotation.

To see the details of a specific revision:

```
kubectl rollout history daemonset <daemonset-name> --revision=1
```

This returns the details of that revision:

```
daemonsets "<daemonset-name>" with revision #1
Pod Template:
Labels:       foo=bar
Containers:
app:
 Image:        ...
 Port:         ...
 Environment:  ...
 Mounts:       ...
Volumes:      ...
```

Step 2: Roll back to a specific revision

```
# Specify the revision number you get from Step 1 in --to-revision
kubectl rollout undo daemonset <daemonset-name> --to-revision=<revision>
```

If it succeeds, the command returns:

```
daemonset "<daemonset-name>" rolled back
```

Note:

If `--to-revision` flag is not specified, kubectl picks the most recent revision.

Step 3: Watch the progress of the DaemonSet rollback

`kubectl rollout undo daemonset` tells the server to start rolling back the DaemonSet. The real rollback is done asynchronously inside the cluster control plane.

To watch the progress of the rollback:

```
kubectl rollout status ds/<daemonset-name>
```

When the rollback is complete, the output is similar to:

```
daemonset "<daemonset-name>" successfully rolled out
```

Understanding DaemonSet revisions

In the previous `kubectl rollout history` step, you got a list of DaemonSet revisions. Each revision is stored in a resource named ControllerRevision.

To see what is stored in each revision, find the DaemonSet revision raw resources:

```
kubectl get controllerrevision -l <daemonset-selector-key>=<daemonset-selector-value>
```

This returns a list of ControllerRevisions:

```
NAME                               CONTROLLER                     REVISION   AGE
<daemonset-name>-<revision-hash>   DaemonSet/<daemonset-name>     1          1h
<daemonset-name>-<revision-hash>   DaemonSet/<daemonset-name>     2          1h
```

Each ControllerRevision stores the annotations and template of a DaemonSet revision.

`kubectl rollout undo` takes a specific ControllerRevision and replaces DaemonSet template with the template stored in the ControllerRevision. `kubectl rollout undo` is equivalent to updating DaemonSet template to a previous revision through other commands, such as `kubectl edit` or `kubectl apply`.

Note:

DaemonSet revisions only roll forward. That is to say, after a rollback completes, the revision number (`.revision` field) of the ControllerRevision being rolled back to will advance. For example, if you have revision 1 and 2 in the system, and roll back from revision 2 to revision 1, the ControllerRevision with `.revision: 1` will become `.revision: 3`.

Troubleshooting

- See troubleshooting DaemonSet rolling update.