

KubeCon + CloudNativeCon 2024

Join us for three days of incredible opportunities to collaborate, learn and share with the cloud native community.

[Buy your ticket now! 12 - 15 November | Salt Lake City](#)



 Search this site

- ▶ Documentation
- ▶ Getting started
- ▶ Concepts
- ▼ Tasks
 - ▶ Install Tools
 - ▶ Administer a Cluster
 - ▶ Configure Pods and Containers
 - ▶ Monitoring, Logging, and Debugging
 - ▼ Manage Kubernetes Objects
 - Declarative Management of Kubernetes Objects Using Configuration Files
 - Declarative Management of Kubernetes Objects Using Kustomize
 - Managing Kubernetes Objects Using Imperative Commands

[Kubernetes Documentation](#) / [Tasks](#) / [Manage Kubernetes Objects](#) / [Update API Objects in Place Using kubectl patch](#)

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:

- [Killercoda](#)
- [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](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: patch-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
```

```
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - name: patch-demo-ctr
        image: nginx
    tolerations:
      - effect: NoSchedule
        key: dedicated
        value: test-team
```

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
```

kubernetes

Documentation

TrainingPartnersCommunityCase StudiesVersion

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:

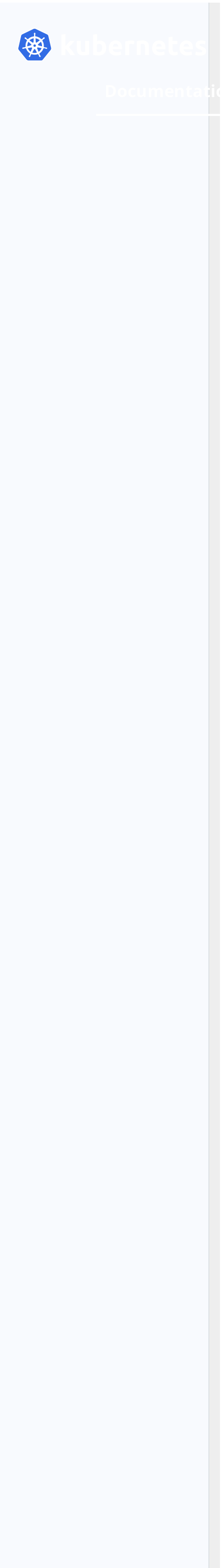
```
containers:
- image: redis
  ...
- image: nginx
  ...
```

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,1,rep"`
  ...
}
```

kubernetes

Documentation

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
```

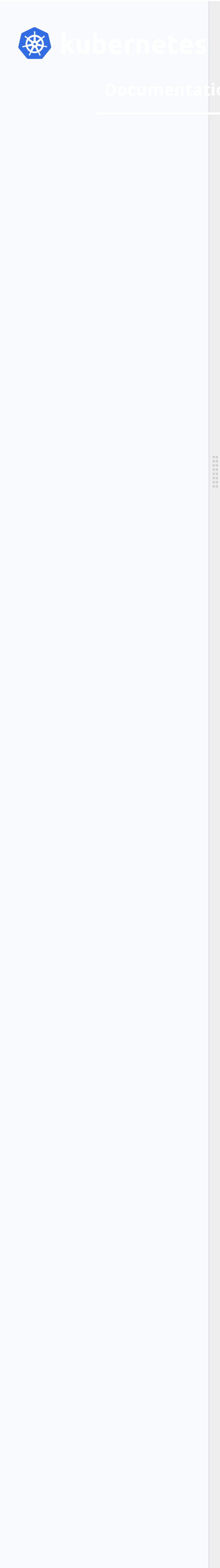
View the patched Deployment:

```
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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: retainkeys-demo
spec:
  selector:
    matchLabels:
      app: nginx
  strategy:
    rollingUpdate:
      maxSurge: 30%
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: retainkeys-demo-ctr
          image: nginx
```

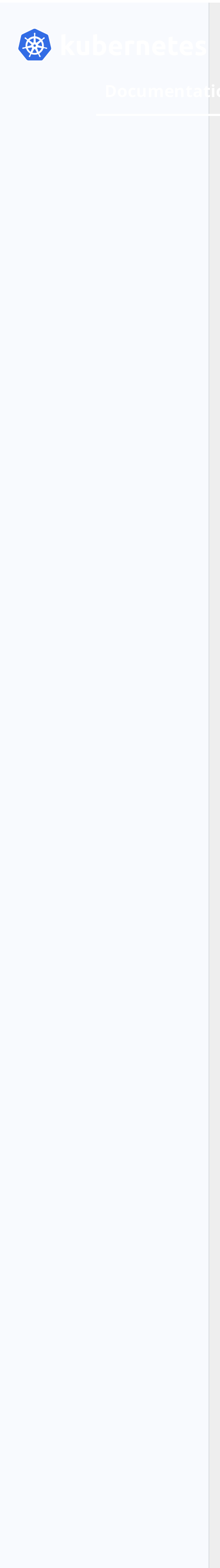
Create the deployment:

```
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
```



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

Patch your Deployment:

```
kubectl patch deployment retainkeys-demo --type strategic --patch-file patch-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: Invalid value: "": spec.strategy.rollingUpdate: Forbidden: may not be set when a value is defined for spec.strategy.rollingUpdate
```

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-retainkeys.yaml
```

Examine the content of the Deployment:

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

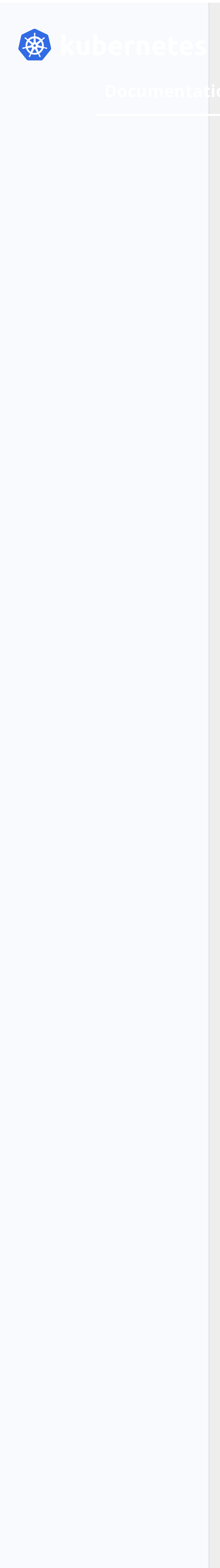
The output shows that the strategy object in the Deployment does not contain the `rollingUpdate` key anymore:

```
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 the 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'
```


 **kubernetes**

Documentation


```
kubectl patch deployment patch-demo --patch-file patch-file.json
kubectl patch deployment patch-demo --patch '{"spec": {"template": {"spe
```

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

 **FEATURE STATE:** Kubernetes v1.24 [alpha]

The flag `--subresource=[subresource-name]` is used with `kubectl` commands like `get`, `patch`, `edit` and `replace` to fetch and update `status` and `scale` subresources of the resources (applicable for `kubectl` version v1.24 or more). This flag is used with all the API resources (built-in and CRs) that have `status` or `scale` subresource. Deployment is one of the examples which supports these subresources.

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

application/deployment.yaml 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Create the Deployment:

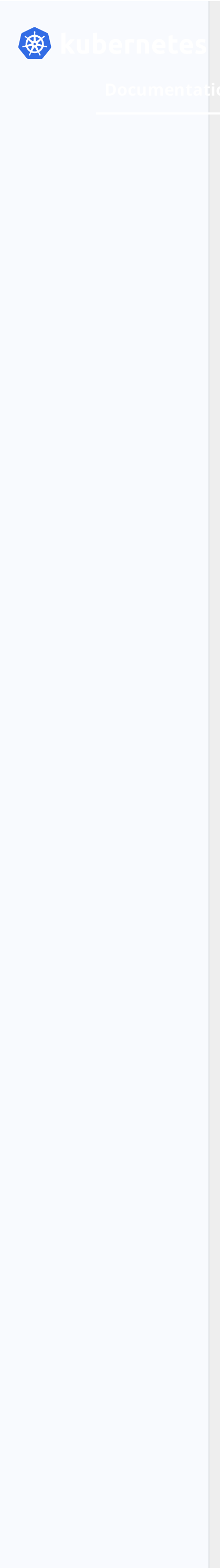
```
kubectl apply -f https://k8s.io/examples/application/deployment.yaml
```

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='scale'
```

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](#).

kubernetes

Documentation

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](#)

Feedback

Was this page helpful?

Yes

No

Last modified June 02, 2024 at 2:43 AM PST: [Modify the image node-hello to hello-app \(#46582\) \(d5b194da5b\)](#)



© 2024 The Kubernetes Authors | Documentation Distributed under CC BY 4.0



© 2024 The Linux Foundation ®. All rights reserved. The Linux Foundation has registered trademarks and uses trademarks. For a list of trademarks of The Linux Foundation, please see our Trademark Usage page

ICP license: 京ICP备17074266号-3