**Getting Started with Cluster Division**

Learn about Namespaces.

**We'll cover the following**

* [Why create multiple clusters?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkJXQBZ1r26#Why-create-multiple-clusters?-0)
* [The problem with multiple clusters](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkJXQBZ1r26#The-problem-with-multiple-clusters-1)
* [The Namespaces](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkJXQBZ1r26#The-Namespaces-2)

**Why create multiple clusters?**

Applications and corresponding objects often need to be separated from each other to avoid conflicts and other undesired effects. Up till now, all examples we have seen create one cluster. However, there are situations in which multiple clusters should be created. Some benefits of creating multiple clusters are as follows:

* We might need to separate objects created by different teams. We can, for example, give each team a separate cluster so that they can “experiment” without affecting others.
* We might want to create different clusters that will be used for various purposes. For example, we could have a production and a testing cluster.
* There are many other problems that we tend to solve by creating different clusters. Most of them are based on the fear that some objects will produce adverse effects on others.
* We might be afraid that a team will accidentally replace a production release of an application with an untested beta.
* We might be concerned that performance tests will slow down the whole cluster. Fear is one of the main reasons why we tend to be defensive and conservative. In some cases, it is founded on past experiences. In others, it might be produced by insufficient knowledge of the tools we adopted. More often than not, it is a combination of the two.

**The problem with multiple clusters**

The problem with having many Kubernetes clusters is that each has an operational and resource overhead. Managing one cluster is often far from trivial. Having a few is complicated. Having many can become a nightmare and requires quite a significant investment in hours dedicated to operations and maintenance.

If that overhead is not enough, we must also be aware that each cluster needs resources dedicated to Kubernetes. The more clusters we have, the more resources (CPU, memory, IO) are spent. While that can be said for big clusters as well, the fact remains that the resource overhead of having many smaller clusters is higher than having a single big one.

**The Namespaces**

We are not trying to discourage you from having multiple Kubernetes clusters. In many cases, that is a welcome, if not a required, strategy. However, there is the possibility of using Kubernetes Namespaces instead. In this chapter, we’ll explore ways to split a cluster into different segments as an alternative to having multiple clusters.

# Deploying the First Release

Learn how to deploy our first release to get started with Namespaces.

**We'll cover the following**

* [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8XAGP9E5Px#Looking-into-the-definition)
* [Altering the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8XAGP9E5Px#Altering-the-definition)
* [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8XAGP9E5Px#Verification)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8XAGP9E5Px#Try-it-yourself)

## Looking into the definition

We’ll start by deploying the go-demo-2 application and use it to explore Namespaces. The file is shown below.

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: go-demo-2

annotations:

kubernetes.io/ingress.class: "nginx"

ingress.kubernetes.io/ssl-redirect: "false"

nginx.ingress.kubernetes.io/ssl-redirect: "false"

spec:

rules:

- host: go-demo-2.com

http:

paths:

- path: /demo

pathType: ImplementationSpecific

backend:

service:

name: go-demo-2-api

port:

number: 8080

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: go-demo-2-db

spec:

selector:

matchLabels:

type: db

service: go-demo-2

strategy:

type: Recreate

template:

metadata:

labels:

type: db

service: go-demo-2

vendor: MongoLabs

spec:

containers:

- name: db

image: mongo:3.3

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-db

spec:

ports:

- port: 27017

selector:

type: db

service: go-demo-2

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: go-demo-2-api

spec:

replicas: 3

selector:

matchLabels:

type: api

service: go-demo-2

template:

metadata:

labels:

type: api

service: go-demo-2

language: go

spec:

containers:

- name: api

image: vfarcic/go-demo-2

env:

- name: DB

value: go-demo-2-db

readinessProbe:

httpGet:

path: /demo/hello

port: 8080

periodSeconds: 1

livenessProbe:

httpGet:

path: /demo/hello

port: 8080

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-api

spec:

ports:

- port: 8080

selector:

type: api

service: go-demo-2

The definition is the same as the one we used before, so we’ll skip the explanation of the YAML file. Instead, we’ll jump right away into the deployment.

## Altering the definition

Unlike previous cases, we’ll deploy a specific tag of the application. If this would be a Docker Swarm stack, we’d define the tag of the vfarcic/go-demo-2 image as an environment variable with the default value set to latest.

Unfortunately, Kubernetes does not have that option. Since we don’t believe that it is a good idea to create a different version of the YAML file for each release, we’ll use sed to modify the definition before passing it to kubectl.

Using sed to alter Kubernetes definitions is not a good solution. We should use a templating solution like, for example, [Helm](https://helm.sh/). However, we are focusing purely on Kubernetes.

Helm and other third-party products are out of the scope of this course. So, we’ll have to do with a workaround in the form of sed commands.

IMG=vfarcic/go-demo-2

TAG=1.0

cat go-demo-2.yml \

| sed -e \

"s@image: $IMG@image: $IMG:$TAG@g" \

| kubectl create -f –

* We declared environment variables IMG and TAG.
* We cat the YAML file and piped the output to sed. It, in return, replaced image: vfarcic/go-demo-2 with image: vfarcic/go-demo-2:1.0.
* The modified definition was piped to kubectl.

When the -f argument is followed with a dash (-), kubectl uses standard input (stdin) instead of a file. In our case, that input is the YAML definition altered by adding the specific tag (1.0) to the vfarcic/go-demo-2 image.

## Verification

Let’s confirm that the deployment rolled out successfully.

kubectl rollout status \

deploy go-demo-2-api

We’ll check whether the application is deployed correctly by sending an HTTP request. Since the Ingress resource we just created has the host set to go-demo-2.com, we’ll have to “fake” it by adding Host: go-demo-2.com header to the request.

nohup kubectl port-forward service/go-demo-2-api 3000:8080 --address 0.0.0.0 > /dev/null 2>&1 &

#wait a few second before running the following command

curl -H "Host: go-demo-2.com" <http://0.0.0.0:3000/demo/hello>

The **output** is as follows.

hello, release 1.0!

The reason we jumped through so many hoops to deploy a specific release will be revealed soon. For now, we’ll assume that we’re running the first release in production.

## Try it yourself

A list of all the commands used in the lesson is given below.

#update image and create go-demo-2.yml

IMG=vfarcic/go-demo-2

TAG=1.0

cat go-demo-2.yml \

| sed -e \

"s@image: $IMG@image: $IMG:$TAG@g" \

| kubectl create -f -

# Rollout status of go-demo-2.yml

kubectl rollout status \

deploy go-demo-2-api

# Building connection and calling application

nohup kubectl port-forward service/go-demo-2-api 3000:8080 --address 0.0.0.0 > /dev/null 2>&1 &

# Please wait fpr a few second before running the following command:

curl -H "Host: go-demo-2.com" <http://0.0.0.0:3000/demo/hello>

# Exploring Virtual Clusters

Explore the details of running system-level objects and to learn about virtual clusters.

**We'll cover the following**

* [Seeking for system-level objects](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3YkOY6yEw0O#Seeking-for-system-level-objects)
* [The virtual clusters](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3YkOY6yEw0O#The-virtual-clusters)
* [The default namespace](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3YkOY6yEw0O#The-default-namespace)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3YkOY6yEw0O#Try-it-yourself)

## Seeking for system-level objects

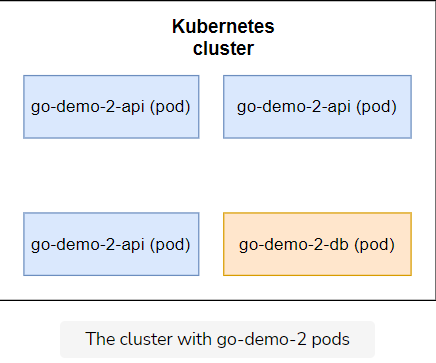
Almost all of the system services are running as Kubernetes objects. Kube DNS is used for deployments. Dashboard, Storage Controller, and nginx Ingress are a few of the system Pods that are currently running in our k3d cluster.

Still, we haven’t seen them yet. Even though we executed kubectl get all quite a few times, there was not a trace of any of those objects. How can that be? Will we see them now if we list all the objects?

kubectl get all

The **output** shows only the objects we created. There are go-demo-2 Deployments, ReplicaSets, Services, and Pods. The only system object we can observe is the kubernetes Service.

Judging from the current information, if we limit our observations to Pods, our cluster can be described through the following illustration.



All in all, our cluster runs a mixture of system-level objects and the objects we created, but only the latter is visible. You might be compelled to execute kubectl get --help hoping that there is an argument that will allow you to retrieve the information about system level objects.

You might think that these objects are hidden from you by default. That’s not the case. They are not hidden. Instead, they do not live in the Namespace we’re looking at.

## The virtual clusters

Kubernetes uses Namespaces to create virtual clusters. When we created the cluster, we got **three Namespaces**. In a way, each Namespace is a cluster within the cluster. They provide scope for names.

So far our experience tells us that we cannot have two of the same types of objects with the same name. There cannot be, for example, two deployments named go-demo-2-api. However, that rule applies only within a Namespace.

Inside a cluster, we can have many of same object types with the same name as long as they belong to different Namespaces.

## The default namespace

So far, we had the impression that we are operating on the level of a k3d Kubernetes cluster. That was a wrong assumption. All this time we were inside one Namespace of all the possible Namespaces in the cluster. To be more concrete, all the commands we executed thus far created objects in the default Namespace.

Namespaces are so much more than scopes for object names.

* They allow us to split a cluster among different groups of users.
* Each of those Namespaces can have different permissions and resources quotas.
* There are quite a few other things we can do if we combine Namespaces with other Kubernetes services and concepts.

However, we’ll ignore permissions, quotas, policies, and other things we did not yet explore. We’ll focus on Namespaces alone.

We’ll start by exploring the pre-defined Namespaces first.

# Exploring the Existing Namespaces

Explore and discuss briefly the existing Namespaces.

**We'll cover the following**

* [Getting the existing namespaces](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnEWo62BqNW#Getting-the-existing-namespaces)
* [The default Namespace](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnEWo62BqNW#The-default-Namespace)
* [The kube-public Namespace](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnEWo62BqNW#The-kube-public-Namespace)
* [The kube-system Namespace](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnEWo62BqNW#The-kube-system-Namespace)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnEWo62BqNW#Try-it-yourself)

Now that we know that our cluster has multiple Namespaces, let’s explore them a bit.

## Getting the existing namespaces

We can list all the Namespaces through the kubectl get namespaces command. As with the most of the other Kubernetes objects and resources, we can also use a shortcut ns instead of the full name.

kubectl get ns

The **output** is as follows.

NAME STATUS AGE

default Active 23m

kube-node-lease Active 23m

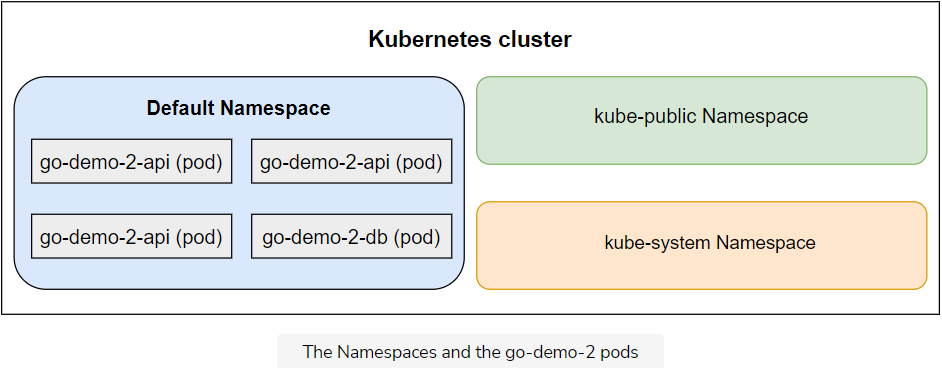
kube-public Active 23m

kube-system Active 23m

We can see that three Namespaces were set up automatically when we created the cluster.

## The default Namespace

The default Namespace is the one we used all this time. If we do not specify otherwise, all the kubectl commands will operate against the objects in the default Namespace. That’s where our go-demo-2 application is running. Even though we were not aware of its existence, we now know that’s where the objects we created are placed.



There are quite a few ways to specify a Namespace. For now, we’ll use the --namespace argument. It is one of the global options that is available for all kubectl commands.

## The kube-public Namespace

The command that will retrieve all the objects from the kube-public Namespace is as follows.

kubectl --namespace kube-public get all

The **output** states that “No resources found”. That’s disappointing, isn’t it? Kubernetes does not use the kube-public Namespace for its system-level objects. All the objects we created are in the default Namespace.

The kube-public Namespace is readable by all users from all Namespaces.

The primary reason for kube-public's existence is to provide space where we can create objects that should be visible throughout the whole cluster.

A good example is ConfigMaps. When we create one in, let’s say, the default Namespace, it is accessible only by the other objects in the same Namespace. Those residing somewhere else would be oblivious of its existence. If we’d like such a ConfigMap to be visible to all objects no matter where they are, we’d put it into the kube-public Namespace instead. We won’t use this Namespace much (if at all).

## The kube-system Namespace

The kube-system Namespace is critical. Almost all the objects and resources Kubernetes needs are running inside kube-system Namespace.

We can check that by executing the command that follows.

kubectl --namespace kube-system get all

We retrieved all the objects and resources running inside the kube-system Namespace. The **output** is as follows.

NAME READY STATUS RESTARTS AGE

pod/coredns-7448499f4d-jhnnx 1/1 Running 0 2m19s

pod/local-path-provisioner-5ff76fc89d-6lkrx 1/1 Running 0 2m19s

pod/metrics-server-86cbb8457f-bh7lv 1/1 Running 0 2m19s

pod/helm-install-traefik-crd-plhxt 0/1 Completed 0 2m19s

pod/helm-install-traefik-9cgf4 0/1 Completed 1 2m19s

pod/svclb-traefik-r5wpl 2/2 Running 0 70s

pod/traefik-97b44b794-hzdng 1/1 Running 0 71s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/kube-dns ClusterIP 10.43.0.10 <none> 53/UDP,53/TCP,9153/TCP 2m31s

service/metrics-server ClusterIP 10.43.219.121 <none> 443/TCP 2m30s

service/traefik LoadBalancer 10.43.202.156 172.19.0.3 80:31337/TCP,443:32605/TCP 72s

NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE

daemonset.apps/svclb-traefik 1 1 1 1 1 <none> 71s

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/coredns 1/1 1 1 2m31s

deployment.apps/local-path-provisioner 1/1 1 1 2m31s

deployment.apps/metrics-server 1/1 1 1 2m30s

deployment.apps/traefik 1/1 1 1 72s

NAME DESIRED CURRENT READY AGE

replicaset.apps/coredns-7448499f4d 1 1 1 2m20s

replicaset.apps/local-path-provisioner-5ff76fc89d 1 1 1 2m20s

replicaset.apps/metrics-server-86cbb8457f 1 1 1 2m20s

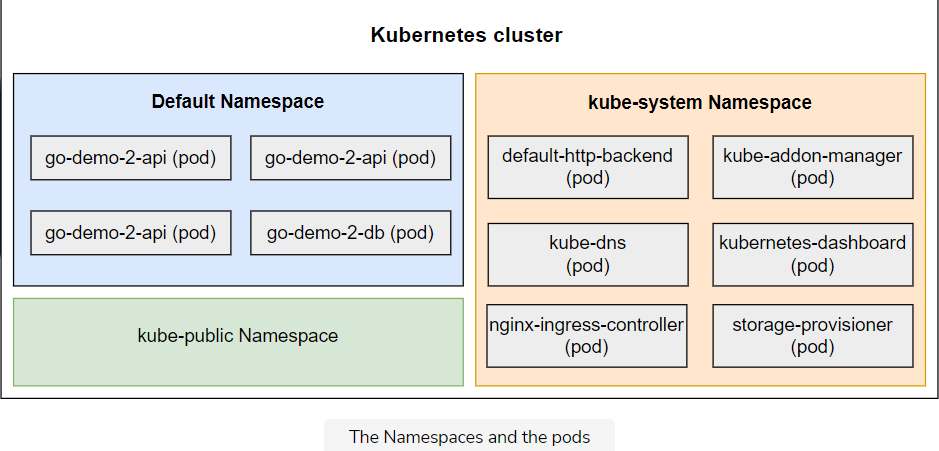
replicaset.apps/traefik-97b44b794 1 1 1 72s

NAME COMPLETIONS DURATION AGE

job.batch/helm-install-traefik-crd 1/1 64s 2m29s

job.batch/helm-install-traefik 1/1 70s 2m29s

As we can see, quite a few things are running inside the kube-system Namespace.



As long as the system works as expected, there isn’t much need to do anything inside the kube-system Namespace. The real fun starts when we create new Namespaces.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl get ns

kubectl --namespace kube-public get all

kubectl --namespace kube-system get all

# Creating a New Namespace

Learn how to create a new Namespace and switch the kubectl context to it.

**We'll cover the following**

* [Understanding the scenario](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnE9QwyD9jY#Understanding-the-scenario)
* [Exploring the options](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnE9QwyD9jY#Exploring-the-options)
* [Creating a Namespace](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnE9QwyD9jY#Creating-a-Namespace)
* [Context switching](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnE9QwyD9jY#Context-switching)
* [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnE9QwyD9jY#Verification)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnE9QwyD9jY#Try-it-yourself)
  + [Troubleshooting Tips for minikube](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/BnE9QwyD9jY#Troubleshooting-Tips-for-minikube)

## Understanding the scenario

Currently, we’re running the release 1.0 of the go-demo-2 application. We can consider it the production release. Now, let’s say that the team in charge of the application just made a new release. They ran unit tests and built the binary. They produced a new Docker image and tagged it as vfarcic/go-demo-2:2.0.

What they didn’t do is run functional, performance, and other types of tests that require a running application. The new release is still not ready to be deployed to production so we cannot yet execute a rolling update and replace the production release with the new one. We need to finish running the tests, and for that we need the new release running in parallel with the old one.

## Exploring the options

We could, for example, create a new cluster that would be used only for testing purposes. While that is indeed a good option in some situations, in others it might be a waste of resources. Moreover, we’d face the same challenge in the testing cluster. There might be multiple new releases that need to be deployed and tested in parallel.

Another option could be to create a new cluster for each release that is to be tested. That would create the necessary separation and maintain the freedom we strive for. However, that is slow. Creating a cluster takes time. Even though it might not look like much, wasting ten minutes (if not more) only on that is too much time. Even if you disagree and you think that ten minutes is not that much, such an approach would be too expensive.

Every cluster has a resource overhead that needs to be paid. While the overall size of a cluster affects the resource overhead, the number of clusters affects it even more. It’s more expensive to have many smaller clusters than a big one. On top of all that, there is the operational cost. While it is often not proportional to the number of clusters, it still increases.

Having a separate cluster for all our testing needs is not a bad idea. We shouldn’t discard it, just as we should consider creating (and destroying) a new cluster for each new release. However, before you start creating new Kubernetes clusters, we’ll explore how we might accomplish the same goals with a single cluster and with the help of Namespaces.

## Creating a Namespace

First things first. We need to create a new Namespace before we can use it.

kubectl create ns testing

kubectl get ns

The **output** of the latter command is as follows.

NAME STATUS AGE

default Active 4h36m

kube-node-lease Active 4h36m

kube-public Active 4h36m

kube-system Active 4h36m

testing Active 3s

We can see that the new Namespace “testing” was created.

We can continue using the --namespace argument to operate within the newly created Namespace. However, writing --namespace with every command is tedious. Instead, we’ll create a new context.

kubectl config set-context testing \

--namespace testing \

--cluster k3d-mycluster \

--user admin@k3d-mycluster

We created a new context called testing. It is the same as the mycluster context, except that it uses the testing Namespace.

kubectl config view

The **output**, limited to the relevant parts, is as follows.

...

contexts:

- context:

cluster: k3d-mycluster

user: admin@k3d-mycluster

name: k3d-mycluster

- context:

cluster: mycluster

namespace: testing

user: admin@k3d-mycluster

name: testing

...

We can see that there are two contexts. Both are set to use the same “mycluster” cluster with the same “mycluster” user. The only difference is that one does not have the Namespace set, meaning that it will use the default. The other has it set to “testing”.

## Context switching

Now that we have two contexts, we can switch to testing. The

kubectl config use-context testing

We switched to the testing context that uses the Namespace of the same name. From now on, all the kubectl commands will be executed within the context of the testing Namespace. That is, until we change the context again, or use the --namespace argument.

## Verification

To be on the safe side, we’ll confirm that nothing is running in the newly created Namespace.

kubectl get all

The **output** shows that no resources were found.

If we repeat the same command with the addition of the --namespace=default argument, we’ll see that the go-demo-2 objects we created earlier are still running.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create ns testing

kubectl get ns

kubectl config set-context testing \

--namespace testing \

--cluster k3d-mycluster \

--user admin@k3d-mycluster

kubectl config view

kubectl config use-context testing

kubectl get all

### Troubleshooting Tips for minikube

If you are working with minkube locally you can create a new context in this way.

kubectl config set-context testing \

--namespace testing \

--cluster minikube \

--user minikube

# Deploying to a New Namespace

Learn to deploy a release to our newly created Namespace.

**We'll cover the following**

* [Altering the deployment definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gxMNm5LBO2j#Altering-the-deployment-definition)
* [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gxMNm5LBO2j#Verification)
* [Concluding remarks](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gxMNm5LBO2j#Concluding-remarks)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gxMNm5LBO2j#Try-it-yourself)

## Altering the deployment definition

As we explained in [Creating a new Namespace](https://www.educative.io/pageeditor/10370001/5920988434792448/5066859332239360) lesson, the main objective of the deployment is to provide a means to test the release. It should remain hidden from our users.

The users should be oblivious to the existence of the new Deployment and continue using the release 1.0 until we are confident that 2.0 works as expected.

IMG=vfarcic/go-demo-2

TAG=2.0

DOM=go-demo-2.com

cat go-demo-2.yml \

| sed -e \

"s@image: $IMG@image: $IMG:$TAG@g" \

| sed -e \

"s@host: $DOM@host: $TAG\.$DOM@g" \

| kubectl create -f –

Just as before, we used sed to alter the image definition. This time, we’re deploying the tag 2.0.

Apart from changing the image tag, we also modified the host. This time, the Ingress resource will be configured with the host 2.0.go-demo-2.com. That will allow us to test the new release using that domain while our users will continue seeing the production release 1.0 through the domain go-demo-2.com.

## Verification

Let’s confirm that the rollout finished.

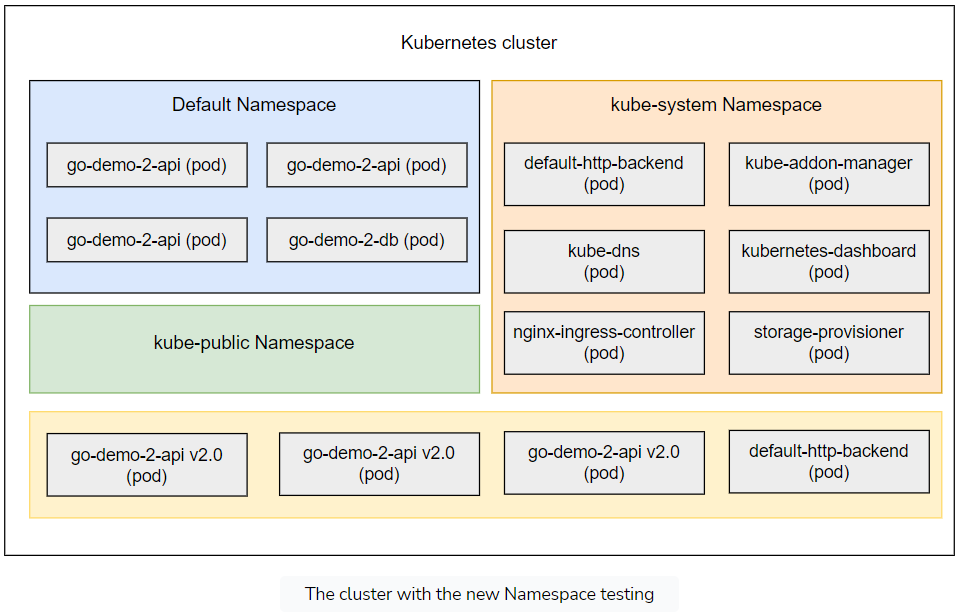
kubectl rollout status \

deploy go-demo-2-api

The **output** is as follows.

deployment "go-demo-2-api" successfully rolled out

As you can see, we rolled out the Deployment go-demo-2-api, along with some other resources. That means that we have two sets of the same objects with the same name. One is running in the default Namespace, while the other (release 2.0) is running in the testing Namespace.



Before we celebrate the successful deployment of the new release without affecting production, we should verify that both are indeed working as expected.

If we send a request to go-demo-2.com, we should receive a response from the release 1.0 running in the default Namespace.

nohup kubectl port-forward -n ingress-nginx service/ingress-nginx-controller 3000:80 --address 0.0.0.0 > /dev/null 2>&1 &

curl -H "Host: go-demo-2.com" <http://0.0.0.0:3000/demo/hello>

The **output** is as follows.

hello, release 1.0!

If, on the other hand, we send a request to 2.0.go-demo-2.com, we should get a response from the release 2.0 running in the testing Namespace.

curl -H "Host: 2.0.go-demo-2.com" <http://0.0.0.0:3000/demo/hello>

The **output** is as follows.

hello, release 2.0!

## Concluding remarks

The result we accomplished through different Namespaces is very similar to what we’d expect by using separate clusters. The main difference is that we did not need to complicate things by creating a new cluster. We saved time and resources by using a new Namespace instead.

If this would be a “real world” situation, we’d run functional and other types of tests using the newly deployed release. Hopefully, those tests would be automated, and they would last for only a few minutes. We’ll skip the testing part since it’s not within the scope of this chapter. Instead, we’ll imagine that the tests were executed and that they were successful.

Communication is an important subject when working with Namespaces, so in the next lesson, we’ll spend a few moments exploring it.

## Try it yourself

A list of all the commands used in the lesson is given below.

IMG=vfarcic/go-demo-2

TAG=2.0

DOM=go-demo-2.com

cat go-demo-2.yml \

| sed -e \

"s@image: $IMG@image: $IMG:$TAG@g" \

| sed -e \

"s@host: $DOM@host: $TAG\.$DOM@g" \

| kubectl create -f -

kubectl rollout status \

deploy go-demo-2-api

nohup kubectl port-forward -n ingress-nginx service/ingress-nginx-controller 3000:80 --address 0.0.0.0 > /dev/null 2>&1 &

curl -H "Host: go-demo-2.com" "http://0.0.0.0:3000/demo/hello"

curl -H "Host: 2.0.go-demo-2.com" "http://0.0.0.0:3000/demo/hello"

------------------- yaml -------------------

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: go-demo-2

annotations:

kubernetes.io/ingress.class: "nginx"

ingress.kubernetes.io/ssl-redirect: "false"

nginx.ingress.kubernetes.io/ssl-redirect: "false"

spec:

rules:

- host: go-demo-2.com

http:

paths:

- path: /demo

pathType: ImplementationSpecific

backend:

service:

name: go-demo-2-api

port:

number: 8080

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: go-demo-2-db

spec:

selector:

matchLabels:

type: db

service: go-demo-2

strategy:

type: Recreate

template:

metadata:

labels:

type: db

service: go-demo-2

vendor: MongoLabs

spec:

containers:

- name: db

image: mongo:3.3

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-db

spec:

ports:

- port: 27017

selector:

type: db

service: go-demo-2

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: go-demo-2-api

spec:

replicas: 3

selector:

matchLabels:

type: api

service: go-demo-2

template:

metadata:

labels:

type: api

service: go-demo-2

language: go

spec:

containers:

- name: api

image: vfarcic/go-demo-2

env:

- name: DB

value: go-demo-2-db

readinessProbe:

httpGet:

path: /demo/hello

port: 8080

periodSeconds: 1

livenessProbe:

httpGet:

path: /demo/hello

port: 8080

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-api

spec:

ports:

- port: 8080

selector:

type: api

service: go-demo-2

# Communicating between Namespaces

Learn to establish communication between Namespaces.

**We'll cover the following**

* [Creating a pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g28k8Bvj9qk#Creating-a-pod)
* [Establishing the Communication](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g28k8Bvj9qk#Establishing-the-Communication)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g28k8Bvj9qk#Try-it-yourself)

## Creating a pod

We’ll create an alpine-based Pod that we’ll use to demonstrate communication between Namespaces.

Run the following commands in the widget.

kubectl config use-context k3d-mycluster

kubectl run test --image=alpine sleep 10000

We switched to the mycluster context (default Namespace) and created a Pod with a container based on the alpine image. We let it sleep for a long time. Otherwise, the container would be without a process and would stop almost immediately.

Before we proceed, we should confirm that the Pod is indeed running.

kubectl get pod test

The **output** is as follows.

NAME READY STATUS RESTARTS AGE

test 1/1 Running 0 10m

Please wait a few moments if, in your case, the Pod is not yet ready.

## Establishing the Communication

Before we proceed, we’ll install curl inside the container in the test Pod.

kubectl exec -it test -- apk add -U curl

We already explored communication between objects in the same Namespace. Since the test Pod is running in the default Namespace, we can, for example, reach the go-demo-2-api Service by using the Service name as a DNS name.

kubectl exec -it test -- curl "http://go-demo-2-api:8080/demo/hello"

The **output** is as follows.

hello, release 1.0!

We got the response from the release 1.0 because that’s the one running in the same Namespace. Does that mean that we cannot reach Services from other Namespaces?

When we create a Service, it creates a few DNS entries. One of them corresponds to the name of the Service.

So, the go-demo-2-api Service created a DNS based on that name. Actually, the full DNS entry is go-demo-2-api.svc.cluster.local. Both resolve to the same service go-demo-2-api which, in this case, runs in the default Namespace.

The third DNS entry we got is in the format <service-name>.<namespace-name>.svc.cluster.local. In our case, that is go-demo-2-api.default.svc.cluster.local. Or, if we prefer a shorter version, we could use go-demo-2-api.default.

In most cases, there is no good reason to use the <service-name>.<namespace-name> format when communicating with Services within the same Namespace.

The primary objective behind the existence of the DNSes with the Namespace name is when we want to reach services running in a different Namespace.

If we’d like to reach go-demo-2-api running in the testing Namespace from the test Pod in the default Namespace, we should use the go-demo-2-api.testing.svc.cluster.local DNS or, even better, the shorter version go-demo-2-api.testing.

kubectl exec -it test -- curl <http://go-demo-2-api.testing:8080/demo/hello>

This time, the **output** is different.

hello, release 2.0!

Kube DNS used the DNS suffix testing to deduce that we want to reach the Service located in that Namespace. As a result, we got the response from the release 2.0 of the go-demo-2 application.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl config use-context k3d-mycluster

kubectl run test --image=alpine sleep 10000

kubectl get pod test

kubectl exec -it test -- apk add -U curl

kubectl exec -it test -- curl "http://go-demo-2-api:8080/demo/hello"

kubectl exec -it test -- curl <http://go-demo-2-api.testing:8080/demo/hello>

# Deleting a Namespace and All Its Objects

Learn how to delete the Namespace and everything associated with it.

**We'll cover the following**

* [The cascaded deletion](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g7z535NQRwZ#The-cascaded-deletion)
  + [When it Comes in Handy?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g7z535NQRwZ#When-it-Comes-in-Handy?)
* [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g7z535NQRwZ#Verification)
* [Command for CDP](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g7z535NQRwZ#Command-for-CDP)
* [Destroying everything](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g7z535NQRwZ#Destroying-everything)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g7z535NQRwZ#Try-it-yourself)

## The cascaded deletion

Another handy feature of the Namespaces is their cascading effect. If, for example, we delete the testing Namespace, all the objects and the resources running inside it will be removed as well.

kubectl delete ns testing

kubectl -n testing get all

We deleted the testing Namespace and retrieved all the objects residing in it. If the resources were not deleted very quickly, the **output** is as follows.

NAME READY STATUS RESTARTS AGE

po/go-demo-2-api-56dfb69dbd-8w6rf 0/1 Terminating 0 2m

po/go-demo-2-api-56dfb69dbd-hrr4b 0/1 Terminating 0 2m

po/go-demo-2-api-56dfb69dbd-ws855 0/1 Terminating 0 2m

po/go-demo-2-db-5b49cc946b-xdd6v 0/1 Terminating 0 2m

Please note that, in your case, the output might show more objects. If that’s the case, you were too fast, and Kubernetes did not yet have time to remove them.

After a second or two, the only objects in the testing Namespace are the Pods with the status “terminating”. Once the grace period is over, they will be removed as well. The Namespace is gone, and everything we created in it was removed as well.

### When it Comes in Handy?

The ability to remove a Namespace and all the objects and the resources it hosts is especially useful when we want to create temporary objects. A good example would be continuous deployment (CDP) processes. We can create a Namespace to build, package, test, and do all the other tasks our pipeline requires. Once we’re finished, we can simply remove the Namespace. Otherwise, we would need to keep track of all the objects we created and make sure that they are removed before we terminate the CDP pipeline.

## Verification

Now that the Namespace hosting our release 2.0 is gone, we might want to double-check that the production release (1.0) is still running.

kubectl get all

The **output** should show the go-demo-2 Deployments, ReplicaSets, Pods, and Services since we are still using the default context.

To be on the safe side, we’ll check that a request coming from the go-demo-2.com domain still returns a response from the release 1.0.

nohup kubectl port-forward service/go-demo-2-api 3000:8080 --address 0.0.0.0 > /dev/null 2>&1 &

# please wait for a few second before running the following command:

curl -H "Host: go-demo-2.com" <http://0.0.0.0:3000/demo/hello>

As expected, the response is “hello, release 1.0!”.

## Command for CDP

If this were a continuous deployment pipeline, the only thing left would be to execute rolling updates that would change the image of the production release to vfarcic/go-demo-2:2.0. The command could be as follows.

kubectl set image \

deployment/go-demo-2-api \

api=vfarcic/go-demo-2:2.0 \

--record

That was all about the deletion process of Namespaces.

## Destroying everything

The chapter is finished, and that means that we are about to remove the cluster.

k3d cluster delete mycluster –all

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl delete ns testing

kubectl -n testing get all

kubectl get all

nohup kubectl port-forward service/go-demo-2-api 3000:8080 --address 0.0.0.0 > /dev/null 2>&1 &

# Please wait fpr a few second before running the following command:

curl -H "Host: go-demo-2.com" "http://0.0.0.0:3000/demo/hello"

kubectl set image \

deployment/go-demo-2-api \

api=vfarcic/go-demo-2:2.0 \

--record

k3d cluster delete mycluster –all

# What's Next?

Recap what we have learned so far and what we are going to learn next.

**We'll cover the following**

* [Summary](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JE0PB3PQ8go#Summary)

## Summary

Deploying test releases as part of a continuous deployment process is not the only usage of Namespaces. There can be many other situations when they are useful. We could, for example, give a separate Namespace to each team in our organization. Or we could split the cluster into Namespaces based on the type of applications (e.g., monitoring, continuous-deployment, back-end, and so on).

All in all, Namespaces are a handy way to separate the cluster into different sections. Some of the Namespaces we’ll create will be long-lasting while others, like testing Namespace from our examples, will be short-lived.

The real power behind Namespaces comes when they are combined with authorization policies and constraints. However, we did not yet explore those subjects so, for now, we’ll need to limit our Namespaces experience to their basic form.

ℹ️ If you’d like to know more about Namespaces, please explore [Namespace v1 core](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#namespace-v1-core) API documentation.

The next chapter covers everything about securing the Kubernetes clusters.