# Exploring the Existing Namespaces

In this lesson, we will explore and discuss briefly about the existing Namespaces.

> **WE'LL COVER THE FOLLOWING** ^
>
> - Getting the Existing Namespaces
> - The default Namespace
> - The kube-public Namespace
> - The kube-system Namespace

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 Minikube 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.



The Namespaces and the go-demo-2 Pods
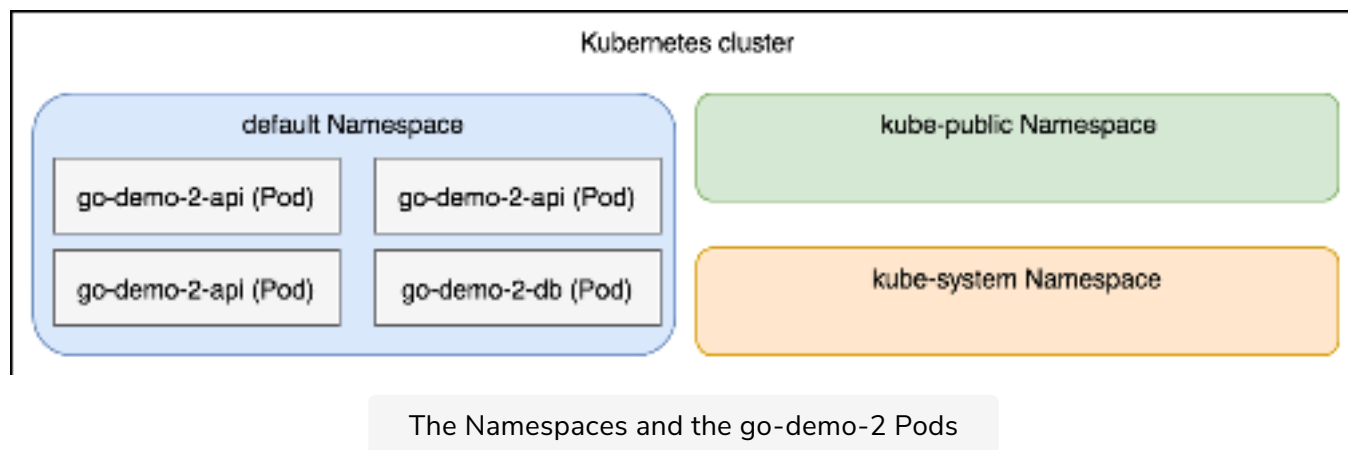
There are quite a few ways to specify a Namespace. For now, we'll use the `--namespace` argument. It is one of the global options that is available for all `kubectl` commands.

# 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` were `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-fb8b8dccf-54ppw                     1/1     Running   1          4h32m
pod/coredns-fb8b8dccf-vgxjk                     1/1     Running   1          4h32m
pod/default-http-backend-6864bbb7db-7mtdc       1/1     Running   0          4h32m
pod/etcd-minikube                               1/1     Running   0          4h31m
pod/kube-addon-manager-minikube                 1/1     Running   0          4h31m
pod/kube-apiserver-minikube                     1/1     Running   0          4h31m
pod/kube-controller-manager-minikube            1/1     Running   0          4h31m
pod/kube-proxy-6w4xt                            1/1     Running   0          4h32m
pod/kube-scheduler-minikube                     1/1     Running   0          4h31m
pod/kubernetes-dashboard-79dd6bfc48-ztpd6       1/1     Running   4          4h32m
pod/nginx-ingress-controller-586cdc477c-r9b2m   1/1     Running   0          4h32m
pod/storage-provisioner                         1/1     Running   0          4h32m

NAME                           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
service/default-http-backend   NodePort    10.102.50.182   <none>        80:30001/TCP
service/kube-dns               ClusterIP   10.96.0.10      <none>        53/UDP,53/TCP,9153/T
service/kubernetes-dashboard   ClusterIP   10.102.236.69   <none>        80/TCP

NAME                       DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTO
daemonset.apps/kube-proxy  1         1         1       1            1           <none>

NAME                                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/coredns                     2/2     2            2           4h32m
deployment.apps/default-http-backend        1/1     1            1           4h32m
deployment.apps/kubernetes-dashboard        1/1     1            1           4h32m
```

```
deployment.apps/nginx-ingress-controller   1/1      1                    1          4h32m

NAME                                              DESIRED   CURRENT   READY   AGE
replicaset.apps/coredns-fb8b8dccf                    2         2         2      4h32m
replicaset.apps/default-http-backend-6864bbb7db      1         1         1      4h32m
replicaset.apps/kubernetes-dashboard-79dd6bfc48      1         1         1      4h32m
replicaset.apps/nginx-ingress-controller-586cdc477c  1         1         1      4h32m
```

As we can see, quite a few things are running inside the `kube-system` Namespace. For example, we knew that there is an nginx Ingress controller, but this is the first time we saw its objects. It consists of a Replication Controller `nginx-ingress-controller`, and the Pod it created, `nginx-ingress-controller-fxrhn`.



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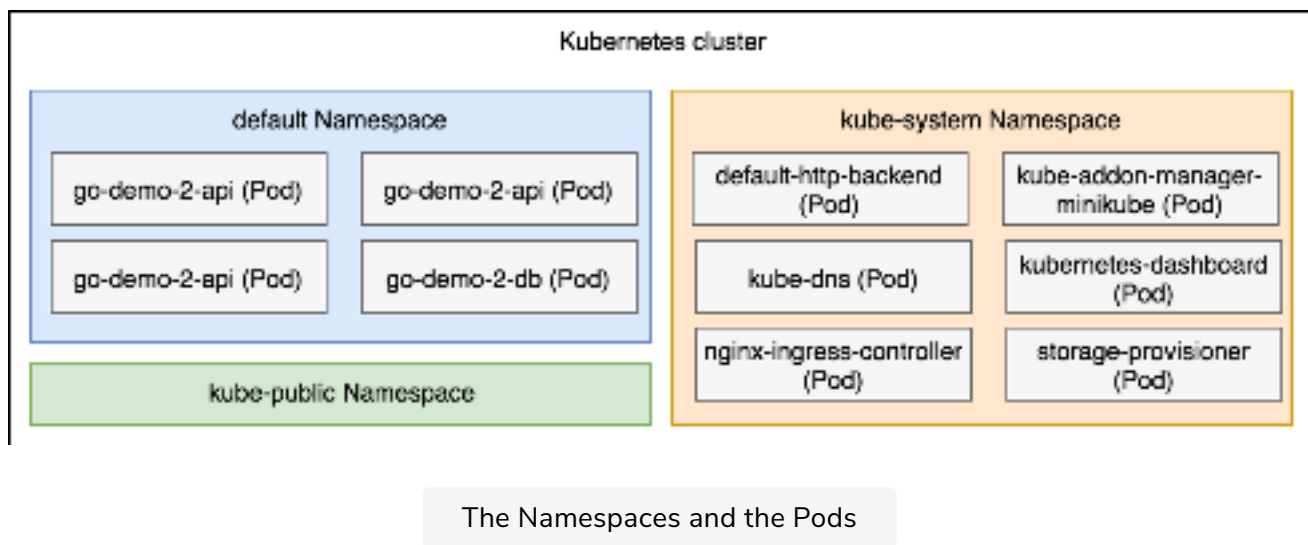
As long as the system works as expected, there isn't much need to do anything inside the `kube-system` Namespace. The real fun starts when we create new Namespaces.

In the next lesson, we will create a Namespace and deploy to it.