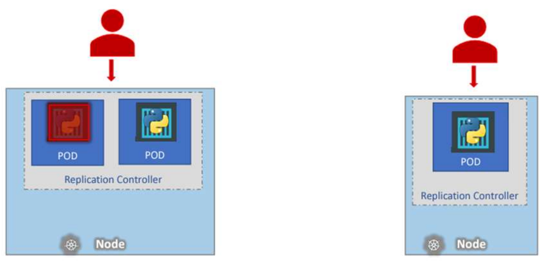
Replicaset

In this lecture, we will discuss about one controller in particular, and that is the Replication Controller. So what is a replica and why do we need a Replication Controller? Let's go back to our first scenario where we had a single pod running our application. What if for some reason our application crashes and the pod fails? Users will no longer be able to access our application. To prevent users from losing access to our application, we would like to have more than one instance or pod running at the same time. That way, if one fails we still have our application running on the other one. The Replication Controller helps us run multiple instances of a single pod in the Kubernetes cluster, thus providing high availability.

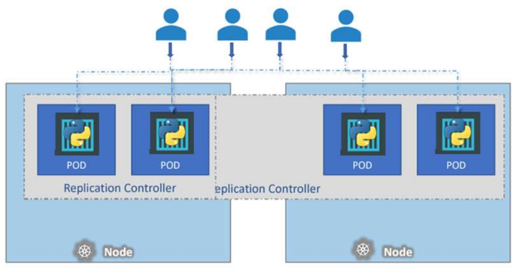
So does that mean you can't use a Replication Controller if you plan to have a single pod? No. Even if you have a single pod, the Replication Controller can help by automatically bringing up a new pod when the existing one fails. Thus, the Replication Controller ensures that the specified number of pods are running at all times even if it's just one or 100.

High Availability



Another reason we need Replication Controller is to create multiple pods to share the load across them. For example, in this simple scenario we have a single pod serving a set of users. When the number of users increase we deploy additional pod to balance the load across the two parts. If the demand further increases and if we were to run out of resources on the first node we could deploy additional parts across the other nodes in the cluster. As you can see, the Replication Controller spans across multiple nodes in the cluster. It helps us balance the load across multiple pods on different nodes as well as scale our application when the demand increases.

Load Balancing & Scaling



It's important to note that there are two similar terms. Replication Controller and Replica Set. Both have the same purpose, but they're not the same. Replication Controller is the older technology that is being replaced by Replica Set. Replica Set is the new recommended way to set up replication. However, whatever we discussed in the previous few slides remain applicable to both these technologies. There are minor differences in the way each works and we will look at that in a bit. As such, we will try to stick to Replica Sets in all of our demos and implementations going forward.

Let us now look at how we create a Replication Controller.

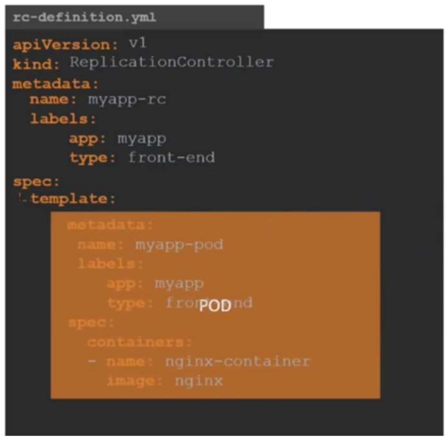
As with the previous lecture, we start by creating a Replication Controller definition file. We will name it **RC-definition.yaml**. As with any Kubernetes definition file, we have four sections: the API version, Kind, Metadata, and Spec.

The API version is specific to what we are creating. In this case, Replication Controller is supported in Kubernetes API version V1, so we will set it as V1. The Kind, as we know, is Replication Controller. Under Metadata, we will add a name, and we will call it My App-RC. We will also add a few labels, app and type, and assign values to them. So far, it has been very similar to how we created a pod in the previous section. The next is the most crucial part of the definition file, and that is the specification written as Spec. For any Kubernetes definition file, the Spec section defines what's inside the object we are creating. In this case, we know that the Replication Controller creates multiple instances of a pod, but what pod?

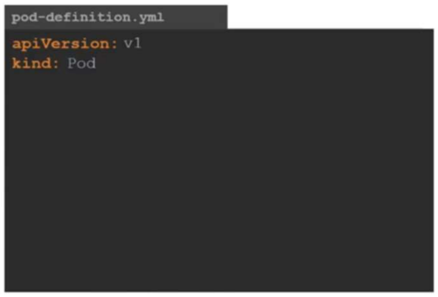
We create a template section under Spec, to provide a pod template to be used by the Replication Controller to create replicas.

Now, how do we define the pod template? It's not that hard because we have already done that in the previous exercise. Remember, we created a pod definition file in the previous exercise; we could reuse the contents of the file to populate the template section.

Move all the contents of the pod definition file into the template section of the Replication Controller, except for the first few lines which are API version and Kind.



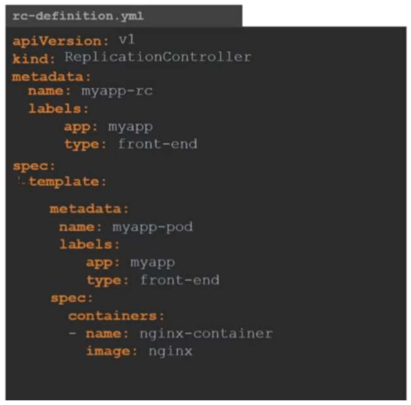
```
rc-definition.yml
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
      app: myapp
      type: front-end
spec:
 - template:
      metadata:
       name: myapp-pod
       labels:
          app: myapp
          type: front-end
      spec:
        containers:
        - name: nginx-container
          image: nginx
```



```
pod-definition.yml
apiVersion: v1
kind: Pod
```

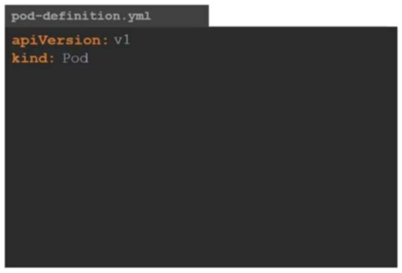
Remember, whatever we move must be under the template section, meaning they should be indented to the right and have more spaces before them than the template line itself. They should be children of the template section.

Looking at our file now, we now have two Metadata sections. One is for the Replication Controller and another for the pod, and we have two Spec sections, one for each. We have nested two definition files together, the Replication Controller being the parent and the pod definition being the child.



```
rc-definition.yml
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
      app: myapp
      type: front-end
spec:
 - template:

      metadata:
       name: myapp-pod
       labels:
          app: myapp
          type: front-end
      spec:
        containers:
        - name: nginx-container
          image: nginx
```



```
pod-definition.yml
apiVersion: v1
kind: Pod
```

Now there is something still missing. We haven't mentioned how many replicas we need in the Replication Controller. For that, add another property to the Spec called replicas and input the number of replicas you need under it. Remember that the template and replicas are direct children of Spec sections, so they are siblings and must be on the same vertical line, which means having an equal number of spaces before them.

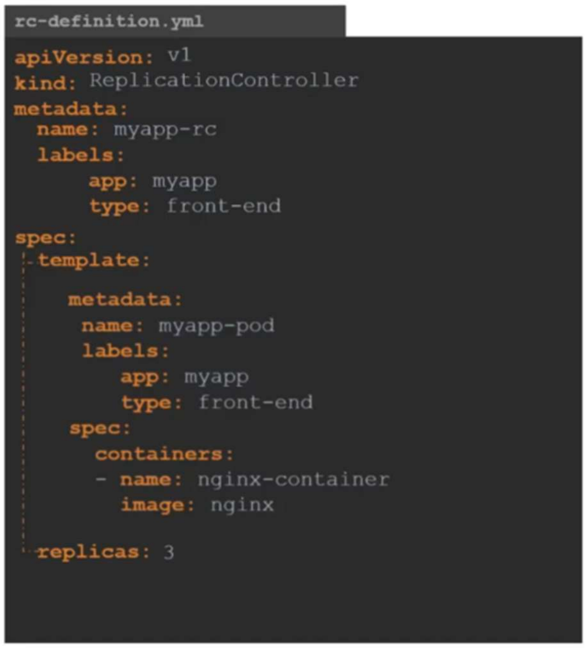
Once the file is ready, run the kubectl create command and input the file using the -F parameter. The Replication Controller is created. When the Replication Controller is created, it first creates the pod using the pod definition template as many as required, which is three in this case.

To view the list of created Replication Controllers, run the kubectl get Replication Controller command, and you will see the Replication Controller listed.

We can also see the desired number of replicas or pods, the current number of replicas, and how many of them are ready in the output.

If you would like to see the pods that were created by the Replication Controller, run the kubectl get pods command, and you will see three pods running.

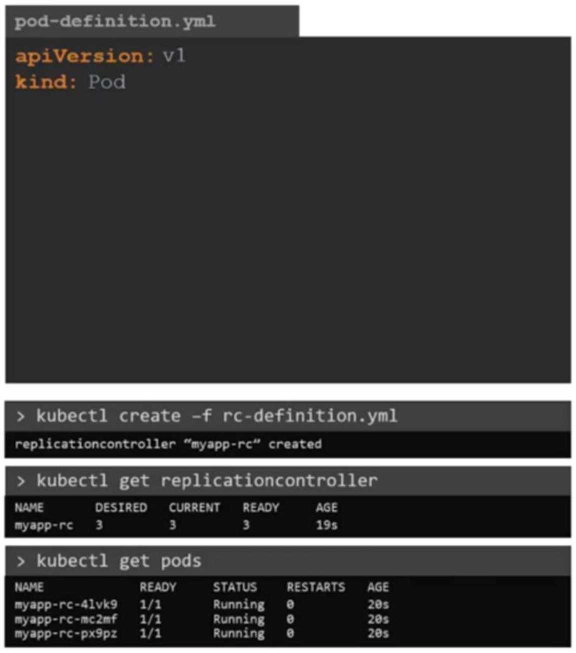
Note that all of them are starting with the name of the Replication Controller, which is My App-RC, indicating that they're all created automatically by the Replication Controller.



```yaml
rc-definition.yml
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
      app: myapp
      type: front-end
spec:
 -template:

    metadata:
     name: myapp-pod
     labels:
         app: myapp
         type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

  replicas: 3
```



```yaml
pod-definition.yml
apiVersion: v1
kind: Pod
```

```
> kubectl create -f rc-definition.yml
replicationcontroller "myapp-rc" created
```

```
> kubectl get replicationcontroller
NAME       DESIRED   CURRENT   READY   AGE
myapp-rc   3         3         3       19s
```

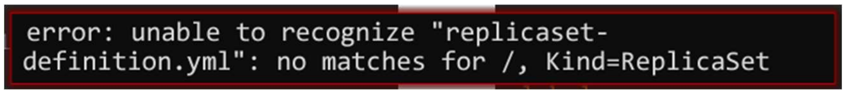
```
> kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
myapp-rc-4lvk9   1/1     Running   0          20s
myapp-rc-mc2mf   1/1     Running   0          20s
myapp-rc-px9pz   1/1     Running   0          20s
```

Let us now look at Replica Set.

It is very similar to Replication Controller, as usual, first we have API version, Kind, Metadata, and Spec.

The API version though is a bit different. It is apps/V1, which is different from what we had before for Replication Controller, which was just V1.

If you get this wrong, you are likely to get an error that looks like this. It would say no match for kind Replica Set because the specified Kubernetes API version has no support for Replica Set.



```
error: unable to recognize "replicaset-
definition.yml": no matches for /, Kind=ReplicaSet
```

The Kind would be Replica Set, and we add name and labels in the Metadata. The specification section looks very similar to Replication Controller. It has a template section where we provide the pod definition as before.

So I'm going to copy contents over from pod definition file and we have the number of replicas, which is set to three.

However, there is one major difference between Replication Controller and Replica Set.

Replica Set requires a selector definition. The selector section helps the Replica Set identify what pods fall under it.

But why would you have to specify what pods fall under it if you have provided the contents of the pod definition file itself in the template? It's because Replica Set can also manage pods that were not created as part of the Replica Set creation. Say, for example, there were pods created before the creation of the Replica Set that match labels specified in the selector; the Replica Set will also take those pods into consideration when creating the replicas.

I will elaborate on this in the next slide, but before we get into that, I would like to mention that the selector is one of the major differences between Replication Controller and Replica Set.

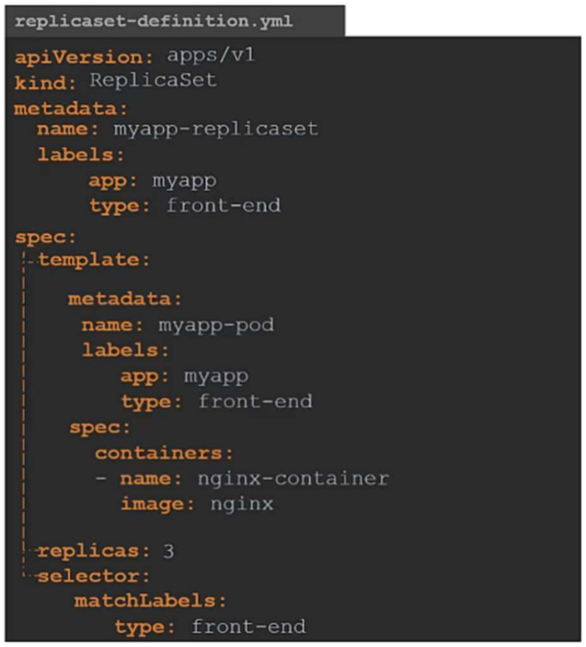
The selector is not a required field in the case of a Replication Controller, but it is still available. When you skip it, as we did in the previous slide, it assumes it to be the same as the labels provided in the pod definition file.

In the case of Replica Set, user input is required for this property, and it has to be written in the form of matched labels as shown here. The match labels selectors simply match the labels specified under it to the labels on the pod.

The Replica Set selector also provides many other options for matching labels that were not available in a Replication Controller.

And as always, to create a Replica Set, run the kubectl create command providing the definition file as input, and to see the created replicas, run the kubectl get Replica Set command.

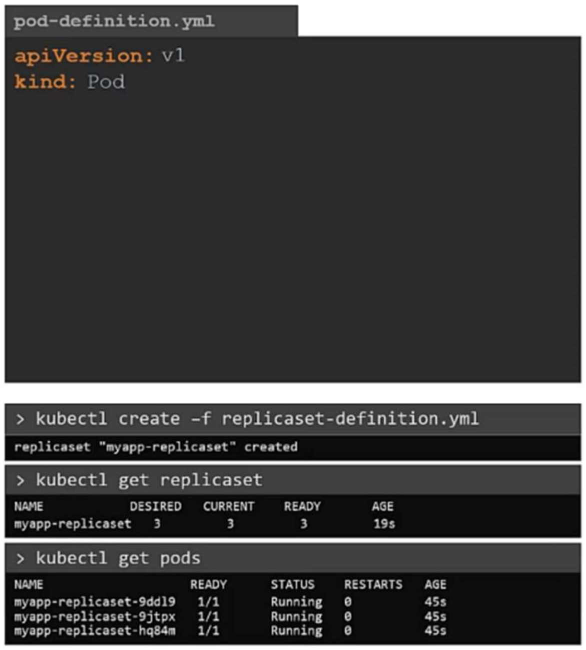
To get a list of pods, simply run the kubectl get pods command.



```
replicaset-definition.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:

    metadata:
      name: myapp-pod
      labels:
         app: myapp
         type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

  replicas: 3
  selector:
     matchLabels:
        type: front-end
```



```
pod-definition.yml
apiVersion: v1
kind: Pod
```

```
> kubectl create -f replicaset-definition.yml
replicaset "myapp-replicaset" created
> kubectl get replicaset
NAME              DESIRED   CURRENT   READY     AGE
myapp-replicaset   3         3         3         19s
> kubectl get pods
NAME                       READY     STATUS    RESTARTS   AGE
myapp-replicaset-9dd19      1/1      Running   0          45s
myapp-replicaset-9jtpx      1/1      Running   0          45s
myapp-replicaset-hq84m      1/1      Running   0          45s
```

labels and selectors

So what is the deal with labels and selectors? Why do we label our pods and objects in Kubernetes?

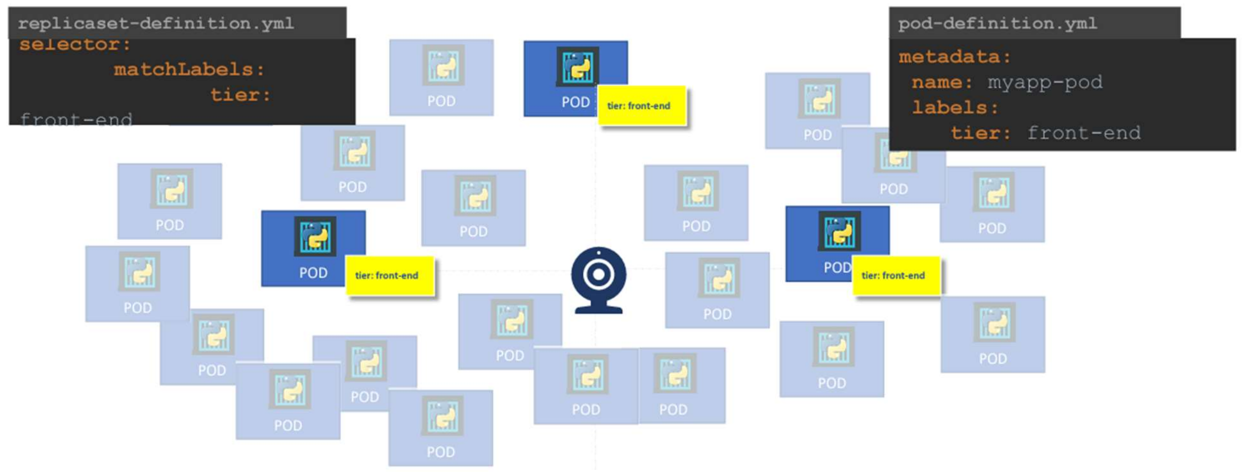
Let us look at a simple scenario. Say we deployed three instances of our front-end web application as three pods. We would like to create a Replication Controller or Replica Set to ensure that we have three active pods at any time. And yes, that is one of the use cases of Replica Sets. You can use it to monitor existing pods if you have them already created, as it is in this example. In case they were not created, the Replica Set will create them for you.

The role of the Replica Set is to monitor the pods and if any of them were to fail, deploy new ones. The Replica Set is, in fact, a process that monitors the pods.

Now, how does the Replica Set know what pods to monitor? There could be hundreds of other pods in the cluster running different applications. This is where labeling our pods during creation comes in handy. We could now provide these labels as a filter for the Replica Set. Under the selector section, we use the match labels filter and provide the same label that we used while creating the pods. This way, the Replica Set knows which pods to monitor.

The same concept of labels and selectors is used in many other places throughout Kubernetes

Labels and Selectors



```
replicaset-definition.yml
selector:
        matchLabels:
                tier:
front-end
```

```
pod-definition.yml
metadata:
 name: myapp-pod
 labels:
    tier: front-end
```

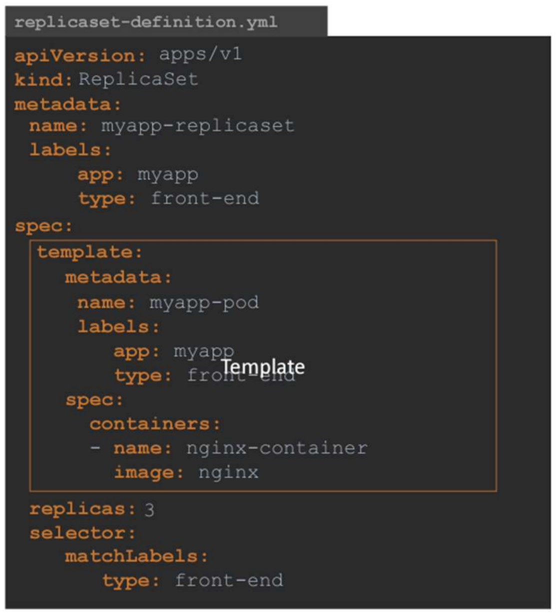
Now, let me ask you a question along the same lines.

In the Replica Set specification section, we learned that there are three sections: template, replicas, and the selector. We need three replicas, and we have updated our selector based on our discussion in the previous slide.

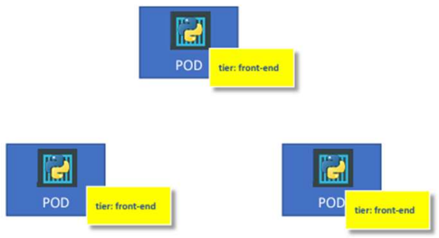
Say, for instance, we have the same scenario as in the previous slide where we have three existing pods that were already created, and we need to create a Replica Set to monitor the pods to ensure there are a minimum of three running at all times.

When the Replication Controller is created, it is not going to deploy a new instance of the pod, as three of them with matching labels are already created. In that case, do we really need to provide a template section in the Replica Set specification, since we are not expecting the Replica Set to create a new pod on deployment?

Yes, we do, because in case one of the pods were to fail in the future, the Replica Set needs to create a new one to maintain the desired number of pods. And for the Replica Set to create a new pod, the template definition section is required.



```
replicaset-definition.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
 name: myapp-replicaset
 labels:
     app: myapp
     type: front-end
spec:
  template:
    metadata:
     name: myapp-pod
     labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
 replicas: 3
 selector:
    matchLabels:
       type: front-end
```



Let's now look at how we scale the Replica Set.

Say we started with three replicas, and in the future, we decided to scale to six. How do we update our Replica Set to scale to six replicas? Well, there are multiple ways to do it.

The first is to update the number of replicas in the definition file to six. Then run the kubectl replace command to specify the same file using the -F parameter, and that will update the Replica Set to have six replicas.

The second way to do it is to run the kubectl scale command, use the replicas parameter to provide the new number of replicas, and specify the same file as input. You may either input the definition file or provide the Replica Set name in the type name format. However, remember that using the file name as input will not result in the number of replicas being updated automatically in the file. In other words, the number of replicas in the Replica Set's definition file will still be three, even though you scaled your Replica Set to have six replicas using the kubectl scale command and the file as input.

There are also options available for automatically scaling the Replica Set based on load, but that is an advanced topic, and we will discuss it at a later time.

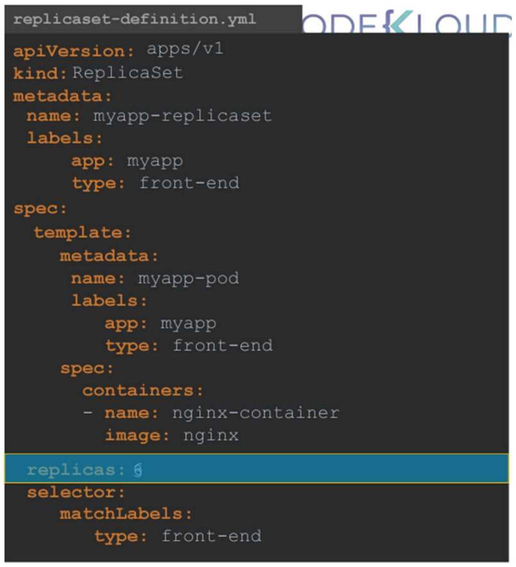
Scale

```
> kubectl replace -f replicaset-definition.yml
```

```
> kubectl scale --replicas=6 -f replicaset-definition.yml
```

```
> kubectl scale --replicas=6 replicaset myapp-replicaset
```

TYPE NAME



```
replicaset-definition.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
         app: myapp
         type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

  replicas: 6
  selector:
    matchLabels:
       type: front-end
```

commands

```
> kubectl create -f replicaset-definition.yml
```

```
> kubectl get replicaset
```

```
> kubectl delete replicaset myapp-replicaset
```

*Also deletes all underlying PODs

```
> kubectl replace -f replicaset-definition.yml
```

```
> kubectl scale -replicas=6 -f replicaset-definition.yml
```