Creating ReplicaSets

In this lesson, first, we will create a ReplicaSet and then retrieve it.

WE'LL COVER THE FOLLOWING

Looking into the Definition

Let's take a look at a ReplicaSet based on the Pod we created in the previous chapter.

```
cat rs/go-demo-2.yml
```

The **output** is as follows.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: go-demo-2
spec:
  replicas: 2
  selector:
    matchLabels:
      type: backend
      service: go-demo-2
  template:
    metadata:
      labels:
        type: backend
        service: go-demo-2
        db: mongo
        language: go
    spec:
      containers:
      - name: db
        image: mongo:3.3
      - name: api
        image: vfarcic/go-demo-2
        env:
```

```
    - name: DB
      value: localhost
    livenessProbe:
      httpGet:
        path: /demo/hello
        port: 8080
```

> The `apiVersion`, `kind`, and `metadata` fields are mandatory with all Kubernetes objects. **ReplicaSet** is no exception, i.e., it is also a Kubernetes object.

- **Line 1:** We specified that the `apiVersion` is `apps/v1`.
- **Line 2-3:** The `kind` is `ReplicaSet` and `metadata` has the `name` key set to `go-demo-2`. We could have extended ReplicaSet `metadata` with labels. However, we skipped that part since they would serve only for informational purposes. They do not affect the behavior of the ReplicaSet.

You should be familiar with the above **three fields** since we already explored them when we worked with Pods. In addition to them, the `spec` section is mandatory as well.

- **Line 5-6:** The first field we defined in the `spec` section is `replicas`. It sets the desired number of replicas of the Pod. In this case, the ReplicaSet should ensure that two Pods should run concurrently. If we did not specify the value of the `replicas`, it would default to `1`.
- **Line 7:** The next `spec` section is the `selector`. We use it to select which pods should be included in the ReplicaSet. It does not distinguish between the Pods created by a ReplicaSet or some other process. In other words, ReplicaSets and Pods are decoupled. If Pods that match the `selector` exist, ReplicaSet will do nothing. If they don't, it will create as many Pods to match the value of the `replicas` field. Not only that ReplicaSet creates the Pods that are missing, but it also monitors the cluster and ensures that the desired number of `replicas` is (almost) always running. In case there are already more running Pods with the matching `selector`, some will be terminated to match the number set in `replicas`.

- **Line 8-10:** We used `spec.selector.matchLabels` to specify a few labels. They must match the labels defined in the `spec.template`. In our case,

  ReplicaSet will look for Pods with `type` set to `backend` and `service` set to `go-demo-2`. If Pods with those labels do not already exist, it'll create them using the `spec.template` section.

- **Line 11-17:** The last section of the `spec` field is the `template`. It is the only required field in the `spec`, and it has the same schema as a Pod specification. At a minimum, the labels of the `spec.template.metadata.labels` section must match those specified in the `spec.selector.matchLabels`. We can set additional labels that will serve informational purposes only. ReplicaSet will make sure that the number of replicas matches the number of Pods with the same labels. In our case, we set `type` and `service` to the same values and added two additional ones (`db` and `language`). It might sound confusing that the `spec.template.spec.containers` field is mandatory. ReplicaSet will look for Pods with the matching labels created by other means. If we already created a Pod with labels `type: backend` and `service: go-demo-2`, this ReplicaSet would find them and would not create a Pod defined in `spec.template`. The main purpose of that field is to ensure that the desired number of `replicas` is running. If they are created by other means, ReplicaSet will do nothing. Otherwise, it'll create them using the information in `spec.template`.

- **Line 18-23:** Finally, the `spec.template.spec` section contains the same `containers` definition we used in the previous chapter. It defines a Pod with two containers (`db` and `api`).

In the previous chapter, we claimed that those two containers should not belong to the same Pod. The same is true for the containers in Pods managed by the ReplicaSet. However, we did not yet have the opportunity to explore ways to allow containers running in different Pods to communicate with each other. So, for now, we'll continue using the same *flawed Pods definition*.

Creating the ReplicaSet

Let's create the ReplicaSet and experience its advantages first hand.

```
kubectl create -f rs/go-demo-2.yml
```

We got the response that the `replicaset "go-demo-2"` was `created`. We can confirm that by listing all the ReplicaSets in the cluster.

```
kubectl get rs
```

The **output** is as follows.

```
NAME      DESIRED CURRENT READY AGE
go-demo-2 2       2       0     14s
```

We can see that the desired number of replicas is `2` and that it matches the current value. The value of the `ready` field is still `0` but, after the images are pulled, and the containers are running, it'll change to `2`.

Instead of retrieving all the replicas in the cluster, we can retrieve those specified in the `rs/go-demo-2.yml` file.

```
kubectl get -f rs/go-demo-2.yml
```

The **output** should be the same since, in both cases, there is only one ReplicaSet running inside the cluster.

All the other `kubectl get` arguments we explored in the previous chapter also apply to ReplicaSets or, to be more precise, to all Kubernetes objects. The same is true for `kubectl describe` command.

```
kubectl describe -f rs/go-demo-2.yml
```

The last lines of the **output** are as follows.

```
...
Events:
  Type    Reason            Age  From                   Message
  ----    ------            ---- ----                   -------
  Normal SuccessfulCreate 3m   replicaset-controller Created pod: go-demo-2-v59t5
  Normal SuccessfulCreate 3m   replicaset-controller Created pod: go-demo-2-5fd54
```

Judging by the events, we can see that ReplicaSet created two Pods while

trying to match the desired state with the actual state.

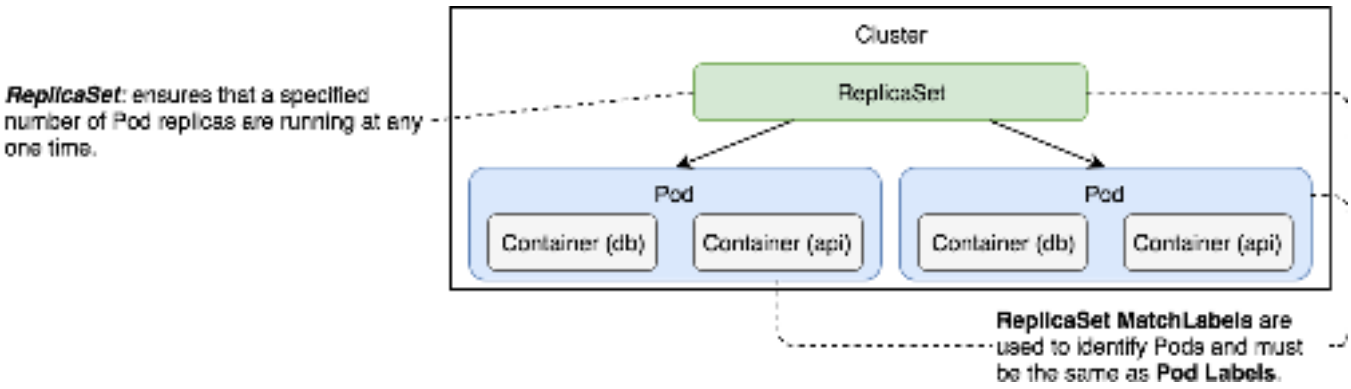
Finally, if you are not yet convinced that the ReplicaSet created the missing Pods, we can list all those running in the cluster and confirm it.

```
kubectl get pods --show-labels
```

To be on the safe side, we used the `--show-labels` argument so that we can verify that the Pods in the cluster match those created by the ReplicaSet.

The **output** is as follows.

```
NAME            READY STATUS  RESTARTS AGE LABELS
go-demo-2-5fd54 2/2   Running 0        6m  db=mongo,language=go,service=go-demo-2,type=backer
go-demo-2-v59t5 2/2   Running 0        6m  db=mongo,language=go,service=go-demo-2,type=backer
```



A ReplicaSet with two replicas of a Pod

The above illustration shows a ReplicaSet with two replicas of a Pod.

Till now, we have successfully replicated a Pod using ReplicaSets. In the next lesson, we will go through the sequential breakdown of this process.