Getting Started with State Persistence

In this lesson, we will discuss briefly about the state persistence of applications and create a cluster.

WE'LL COVER THE FOLLOWING

How to Persist State?

> Having fault-tolerance and high-availability is of no use if we lose application state during rescheduling.

Having state is unavoidable, and we need to preserve it no matter what happens to our applications, servers, or even a whole datacenter.

The way to preserve the state of our applications depends on their architecture. Some are storing data in-memory and rely on periodic backups. Others are capable of synchronizing data between multiple replicas, so that loss instance of one does not result in loss of data. Most, however, are relying on disk to store their state. We'll focus on that group of stateful applications.

If we are to build fault-tolerant systems, we need to make sure that failure of any part of the system is recoverable. Since speed is of the essence, we cannot rely on manual operations to recuperate from failures. Even if we could, no one wants to be the person sitting in front of a screen, waiting for something to fail, only to bring it back to its previous state.

Kubernetes Failure Handling

We already saw that Kubernetes would, in most cases, recuperate from a failure of an application, of a server, or even of a whole datacenter. It'll reschedule Pods to healthy nodes. We also experienced how AWS and kops accomplish more or less the same effect on the infrastructure level. Auto-scaling groups will recreate failed nodes and, since they are provisioned with kops startup processes, new instances will have everything they need, and they will join the cluster.

The only thing that prevents us from saying that our system is (mostly) highly available and fault tolerant is the fact that we did not solve the problem of persisting state across failures. That's the subject we'll explore next.

We'll try to preserve our data no matter what happens to our stateful applications or the servers where they run.

> ℹ All the commands from this chapter are available in the 15-pv.sh Gist.

Creating A Kubernetes Cluster

We'll start by recreating a similar cluster as the one we used in the previous chapter.

```
cd k8s-specs

git pull

cd cluster
```

We entered the local copy of the `k8s-specs` repository, pulled the latest code, and went into the `cluster` directory.

Setting Up Environment Variables

In the previous chapter, we stored the environment variables we used in the `kops` file. Let's take a quick look at them.

```
cat kops
```

The **output**, without the keys, is as follows.

```
export AWS_ACCESS_KEY_ID=...
export AWS_SECRET_ACCESS_KEY=...
export AWS_DEFAULT_REGION=us-east-2
export ZONES=us-east-2a,us-east-2b,us-east-2c
export NAME=devops23.k8s.local
export KOPS_STATE_STORE=s3://devops23-1520933480
```

By storing the environment variables in a file, we can fast-track the process by loading them using the `source` command.

> Please ensure that your copy of the file has all the lines starting with `export` . If that's not the case, please update it accordingly.

```
source kops
```

Creating an S3 Bucket

Now that the environment variables are set, we can proceed to create an S3 bucket.

```
export BUCKET_NAME=devops23-$(date +%s)

aws s3api create-bucket \
    --bucket $BUCKET_NAME \
    --create-bucket-configuration \
    LocationConstraint=$AWS_DEFAULT_REGION

export KOPS_STATE_STORE=s3://$BUCKET_NAME
```

Only for Windows Users

The command that creates the `kops` alias is as follows. Execute it only if you are a **Windows user**.

are a **Windows user**.

```
alias kops="docker run -it --rm \
    -v $PWD/devops23.pub:/devops23.pub \
    -v $PWD/config:/config \
    -e KUBECONFIG=/config/kubecfg.yaml \
    -e NAME=$NAME -e ZONES=$ZONES \
    -e AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID \
    -e AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY \
    -e KOPS_STATE_STORE=$KOPS_STATE_STORE \
    vfarcic/kops"
```

Cluster Creation

Now we can, finally, create a new Kubernetes cluster in AWS.

```
kops create cluster \
    --name $NAME \
    --master-count 3 \
    --master-size t2.small \
    --node-count 2 \
    --node-size t2.medium \
    --zones $ZONES \
    --master-zones $ZONES \
    --ssh-public-key devops23.pub \
    --networking kubenet \
    --yes
```

If we compare that command with the one we executed in the previous chapter, we'll notice only a few minor changes. We increased `node-count` to `2` and `node-size` to `t2.medium`. That will give us more than enough capacity for the exercises we'll run in this chapter.

Validation

Let's validate the cluster.

```
kops validate cluster
```

Assuming that enough time passed since we executed `kops create cluster`, the **output** should indicate that the `cluster devops23.k8s.local is ready`.

> Please wait until the validation checks pass.

A note to Windows users

A note to Windows users

> Kops was executed inside a container. It changed the context inside the container that is now gone. As a result, your local `kubectl` context was left intact. We'll fix that by executing `kops export kubecfg --name ${NAME}` and `export KUBECONFIG=$PWD/config/kubecfg.yaml`. The first command exported the config to `/config/kubecfg.yaml`. That path was specified through the environment variable `KUBECONFIG` and is mounted as `config/kubecfg.yaml` on local hard disk. The latter command exports `KUBECONFIG` locally. Through that variable, `kubectl` is now instructed to use the configuration in `config/kubecfg.yaml` instead of the default one. Before you run those commands, please give AWS a few minutes to create all the EC2 instances and for them to join the cluster. After waiting and executing those commands, you'll be all set.

Creating Ingress and ELB DNS

We'll need Ingress if we'd like to access the applications we'll deploy.

```
kubectl create \
    -f https://raw.githubusercontent.com/kubernetes/kops/master/addons/ingress-nginx/v1.6.0.y
```

Ingress will not help us much without the ELB DNS, so we'll get that as well.

```
CLUSTER_DNS=$(aws elb \
    describe-load-balancers | jq -r \
    ".LoadBalancerDescriptions[] \
    | select(.DNSName \
    | contains (\"api-devops23\") \
    | not).DNSName")

echo $CLUSTER_DNS
```

The **output** of the latter command should end with `us-east-2.elb.amazonaws.com`.

Finally, now that we are finished with the cluster setup, we can go back to the repository root directory.

```
cd ..
```

In the next lesson, we will deploy stateful applications without persisting their state.