**Getting Started with State Persistence**

Learn about the state persistence of applications and create a cluster.

**We'll cover the following**

* [How can we persist states?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g72ZONEOmYD#How-can-we-persist-states)
* [Kubernetes failure handling](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g72ZONEOmYD#Kubernetes-failure-handling)
* [Creating a Kubernetes cluster](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g72ZONEOmYD#Creating-a-Kubernetes-cluster)
  + [Setting up environment variables](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g72ZONEOmYD#Setting-up-environment-variables)
  + [Creating an S3 bucket](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g72ZONEOmYD#Creating-an-S3-bucket)
  + [Only for Windows users](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g72ZONEOmYD#Only-for-Windows-users)
  + [Cluster creation](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g72ZONEOmYD#Cluster-creation)
  + [Validation](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g72ZONEOmYD#Validation)
  + [A note to Windows users](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g72ZONEOmYD#A-note-to-Windows-users)
  + [Creating Ingress and ELB DNS](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g72ZONEOmYD#Creating-Ingress-and-ELB-DNS)

**How can we persist states?**

**Note:** Having fault-tolerance and high availability is of no use if we lose the application state during rescheduling.

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

The way to preserve the state of our applications depends on their architecture. Some store data in memory and rely on periodic backups. Others are capable of synchronizing data between multiple replicas so that a loss of one instance does not result in loss of data. Most, however, rely on the 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 sit 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 data center. 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. AutoS Scaling GGroups 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 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.

**Creating a Kubernetes cluster**

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

cd cluster

We enter the local copy of the k8s-specs repository, pull the latest code, and go 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.

\*\*Note: \*\* 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:

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**

Finally, now we can 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 increase node-count to 2 and node-size to t2.medium. This will give us more than enough capacity for the exercises to run in this chapter.

**Validation**

Let’s validate the cluster:

1

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.

\*\*Note:\*\*Please wait until the validation checks pass.

**A note to Windows users**

**Note:** This note is for Windows users. kOps is executed inside a container. It has changed the context inside the container, which 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 exports the config to /config/kubecfg.yaml. This path is specified through the environment variable KUBECONFIG and is mounted as config/kubecfg.yaml on the local hard disk. The latter command exports KUBECONFIG locally. Through this 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 want to access the applications we’ll deploy.

kubectl create \

    -f https://raw.githubusercontent.com/kubernetes/kops/master/addons/ingress-nginx/v1.6.0.yaml

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.

# Deploying Stateful Applications without Persisting State

Learn how to deploy Jenkins, a stateful application, without persisting its state.

**We'll cover the following**

* [Deploying Jenkins](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JPn98XQX5nJ#Deploying-Jenkins)
  + [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JPn98XQX5nJ#Looking-into-the-definition)
  + [Creating the resources](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JPn98XQX5nJ#Creating-the-resources)
    - [Creating the Secret](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JPn98XQX5nJ#Creating-the-Secret)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JPn98XQX5nJ#Verification)

We’ll start the exploration by deploying a stateful application without any mechanism to persist its state. That will give us a better insight into the benefits of some of the Kubernetes concepts and resources we’ll use in this chapter.

## Deploying Jenkins

We have already deployed Jenkins a few times. Since it is a stateful application, it is an excellent candidate to serve as a playground.

### Looking into the definition

Let’s look at a definition stored in the pv/jenkins-no-pv.yml file:

cat pv/jenkins-no-pv.yml

The YAML defines the jenkins namespace, an Ingress controller, and a Service. We’re already familiar with those types of resources so we’ll skip explaining them and jump straight to the Deployment definition.

The output of the cat command, (limited to the jenkins Deployment), is as follows:

...

apiVersion: apps/v1

kind: Deployment

metadata:

name: jenkins

namespace: jenkins

spec:

selector:

matchLabels:

app: jenkins

strategy:

type: Recreate

template:

metadata:

labels:

app: jenkins

spec:

containers:

- name: jenkins

image: vfarcic/jenkins

env:

- name: JENKINS\_OPTS

value: --prefix=/jenkins

- name: SECRETS\_DIR

value: /etc/secrets

volumeMounts:

- name: jenkins-creds

mountPath: /etc/secrets

resources:

limits:

memory: 2Gi

cpu: 1

requests:

memory: 1Gi

cpu: 0.5

volumes:

- name: jenkins-creds

secret:

secretName: jenkins-creds

    There’s nothing special about this Deployment. We have already used a very similar one. Besides, by now, you’ll be familiar with the Deployment controllers.

The only thing worth mentioning is that there is only one volume mount and it references a Secret we’re using to provide Jenkins with the initial administrative user. Jenkins is persisting its state in /var/jenkins\_home, and we are not mounting that directory.

### Creating the resources

Let’s create the resources defined in pv/jenkins-no-pv.yml:

kubectl create \

    -f pv/jenkins-no-pv.yml \

    --record --save-config

The output is as follows:

namespace "jenkins" created

ingress "jenkins" created

service "jenkins" created

deployment "jenkins" created

We’ll take a quick look at the events as a way to check that everything was deployed successfully.

kubectl --namespace jenkins get events

The output (limited to relevant parts) is as follows:

2018-03-14 22:36:26 +0100 CET   2018-03-14 22:35:54 +0100 CET   7         jenkins-8768d486-lmv6b.151be70fd682e40d   Pod                 Warning   FailedMount  kubelet, ip-172-20-99-208.us-east-2.compute.internal   MountVolume.SetUp failed for volume "jenkins-creds" : secrets "jenkins-creds" not found

...

#### Creating the Secret

We can see that the setup of the only volume failed since it could not find the secret referenced as jenkins-creds. Let’s create it:

kubectl --namespace jenkins \

    create secret \

    generic jenkins-creds \

    --from-literal=jenkins-user=jdoe \

    --from-literal=jenkins-pass=incognito

### Verification

Now, with the secret jenkins-creds created in the jenkins namespace, we can confirm that the rollout of the Deployment is successful.

kubectl --namespace jenkins \

    rollout status \

    deployment jenkins

We can see from the output, that the deployment "jenkins" is successfully rolled out.

In the next lesson, we will analyze the failure of the Jenkins Deployment due to the lack of persisting state.

**Analyzing Failure of the Stateful Application**

Understand failure of the Jenkins application due to a lack of handling the state persistence.

**We'll cover the following**

* [Opening Jenkins in a browser](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xljqDDA05g9#Opening-Jenkins-in-a-browser)
* [Creating a job](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xljqDDA05g9#Creating-a-job)
* [Simulating and analyzing failure](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xljqDDA05g9#Simulating-and-analyzing-failure)

**Opening Jenkins in a browser**

Now that everything is up and running, we can open Jenkins UI in a browser.

open "http://$CLUSTER\_DNS/jenkins"

**Note:** Git Bash might not be able to use the open command. If that’s the case, please replace the open command with echo. As a result, you’ll get the full address that should be opened directly in your browser of choice.

**Creating a job**

Please click the “Log in” link, type “jdoe” as the User, and “incognito” as the Password. When finished, click the “Log in” button.

Now that we are authenticated as the jdoe administrator, we can proceed and create a job. This will generate a state we can use to explore what happens when a stateful application fails.

Please click the “Create new jobs” link, type “my-job” as the item name, select “Pipeline” as the job type, and press the “OK” button.

You’ll be presented with the job configuration screen. There’s no need to do anything here since we are not, at the moment, interested in any specific Pipeline definition. It’s enough to click the “Save” button.

**Simulating and analyzing failure**

Next, we’ll simulate a failure by terminating a java process running inside the Pod created by the jenkins Deployment. To do that, we need to find out the name of the Pod.

kubectl --namespace jenkins \

    get pods \

    --selector=app=jenkins \

    -o json

We retrieve the Pods from the jenkins namespace, filter them with the selector api=jenkins, and format the output as json.

The output (limited to the relevant parts) is as follows:

{

  "apiVersion": "v1",

    "items": [

    {

      ...

      "metadata": {

        ...

        "name": "jenkins-8768d486-lmv6b",

        ...

We can see that the name is inside the metadata entry of one of the items. We can use that to formulate jsonpath that will retrieve only the name of the Pod.

POD\_NAME=$(kubectl \

    --namespace jenkins \

    get pods \

    --selector=app=jenkins \

    -o jsonpath="{.items[\*].metadata.name}")

echo $POD\_NAME

The name of the Pod is now stored in the environment variable POD\_NAME.

The output of the latter command is as follows:

jenkins-8768d486-lmv6b

Now that we know the name of the Pod hosting Jenkins, we can proceed and terminate the java process.

kubectl --namespace jenkins \

    exec -it $POD\_NAME pkill java

The container fails once we terminate the Jenkins process. We already know from experience that a failed container inside a Pod will be recreated. As a result, we have a short downtime, but Jenkins is running once again.

Let’s see what happened to the job we created earlier.

open "http://$CLUSTER\_DNS/jenkins"

As expected, “my-job” is nowhere to be found. The container hosting /var/jenkins\_home directory fails, and is replaced with a new one. The state we created is lost.

We already saw in the [“Volumes”](https://www.educative.io/collection/page/5376908829130752/4742963282313216/4769265662033920) chapter that we can mount a volume in an attempt to preserve the state across failures.

However, in the past, we used emptyDir, which mounts a local volume. Even though that’s better than nothing, such a volume exists only as long as the server it is stored in is up and running. If the server fails, the state stored in emptyDir would be gone.

Such a solution would be only slightly better than not using any volume. By using a local disk we would only postpone the inevitable we’d eventually get to the same situation. We’d be left wondering why we lost everything we created in Jenkins.

In the next lesson, we will learn to create AWS Volumes.

# Creating AWS Volumes

Exploring the options to persist states

**We'll cover the following**

* [Exploring the Options to Persist State](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#Exploring-the-Options-to-Persist-State)
  + [Local storage](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#Local-storage)
  + [External storage](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#External-storage)
    - [Elastic File System (EFS)](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#Elastic-File-System-EFS)
    - [Elastic Block Store (EBS)](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#Elastic-Block-Store-EBS)
    - [Our Choice](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#Our-Choice)
* [Creating an EBS volume](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#Creating-an-EBS-volume)
  + [Looking into the description](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#Looking-into-the-description)
  + [Retrieving the availability zones](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#Retrieving-the-availability-zones)
  + [Setting up the environment variables](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#Setting-up-the-environment-variables)
  + [Creating volumes](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#Creating-volumes)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLNA4zmrXwz#Verification)

## Exploring the Options to Persist State

If we want to persist a state that will survive even server failures, we have two options to choose.

### Local storage

We could, for example, store data locally and replicate it to multiple servers. This way, a container could use local storage, knowing that the files are available on all the servers. Such a setup would be too complicated if we’d like to implement the process ourselves. We could use one of the volume drivers for that. However, we’ll opt for a more commonly used method to persist the state across failures. We’ll use external storage.

### External storage

Since we’re running our cluster in AWS, we can choose between [S3](https://aws.amazon.com/s3/), [Elastic File System (EFS)](https://aws.amazon.com/efs/), and [Elastic Block Store](https://aws.amazon.com/ebs/).

S3 is meant to be accessed through its API and is not suitable as a local disk replacement. This leaves us with EFS and EBS.

#### Elastic File System (EFS)

Elastic File System (EFS) has a distinct advantage: it can be mounted to multiple EC2 instances spread across multiple availability zones. It is the closest we can get to fault-tolerant storage. Even if a whole zone (data center) fails, we’ll still be able to use EFS in the rest of the zones used by our cluster. However, this comes at a cost. EFS introduces a performance penalty. It is, after all, a network file system (NFS), which entails higher latency.

#### Elastic Block Store (EBS)

**Elastic Block Store (EBS)** is the fastest storage we can use in AWS. Its data access latency is very low, therefore making it the best choice when performance is the primary concern. The downside is availability. It doesn’t work in multiple availability zones. Failure of one will mean downtime, at least until the zone is restored to its operational state.

#### Our Choice

We’ll choose EBS for our storage needs. Jenkins depends heavily on IO, and we need data access to be as fast as possible. However, there is another reason for such a choice. EBS is fully supported by Kubernetes. EFS will come but it is still in the experimental stage. As a bonus advantage, EBS is much cheaper than EFS.

Given the requirements and what Kubernetes offers, the choice is obvious. We’ll use EBS, even though we might run into trouble if the availability zone where our Jenkins will run goes down. In such a case, we’d need to migrate EBS volume to a healthy zone. There’s no such thing as a perfect solution.

We are jumping ahead of ourselves. We’ll keep Kubernetes aside for a while and concentrate on creating an EBS volume.

## Creating an EBS volume

Each EBS volume is tied to an availability zone. Unlike EFS, the Elastic Block Store cannot span multiple zones. So, the first thing we need to do is to find out which zones worker nodes are running in.

### Looking into the description

We can get the above information by describing the EC2 instances belonging to the security group nodes.devops23.k8s.local.

aws ec2 describe-instances

The output (limited to the relevant parts) is as follows:

{

  "Reservations": [

    {

      "Instances": [

        {

          ...

          "SecurityGroups": [

            {

              "GroupName": "nodes.devops23.k8s.local",

              "GroupId": "sg-33fd8c58"

            }

          ],

          ...

          "Placement": {

            "Tenancy": "default",

            "GroupName": "",

            "AvailabilityZone": "us-east-2a"

          },

          ...

We can see that the information is inside the Reservations.Instances array. To get the zone, we need to filter the output by the SecurityGroups.GroupName field. The zone name is located in the Placement.AvailabilityZone field.

### Retrieving the availability zones

The command that filters and retrieves the availability zones of the worker nodes is as follows:

aws ec2 describe-instances \

    | jq -r \

    ".Reservations[].Instances[] \

    | select(.SecurityGroups[]\

    .GroupName==\"nodes.$NAME\")\

    .Placement.AvailabilityZone"

The output is as follows:

us-east-2a

us-east-2c

We can see that the two worker nodes are located in the zones us-east-2a and us-east-2c.

### Setting up the environment variables

The commands that retrieve the zones of the two worker nodes and store them in environment variables is as follows:

aws ec2 describe-instances \

    | jq -r \

    ".Reservations[].Instances[] \

    | select(.SecurityGroups[]\

    .GroupName==\"nodes.$NAME\")\

    .Placement.AvailabilityZone" \

    | tee zones

AZ\_1=$(cat·zones·|·head·-n·1)

AZ\_2=$(cat·zones·|·tail·-n·1)

We retrieve the zones and store the output in the zones file. Further on, we retrieve the first row with the head command and store it in the environment variable AZ\_1. Similarly, we store the last (the second) row in the variable AZ\_2.

### Creating volumes

Now we have all the information we need to create a few volumes.

**Note:** The command that follows requires a relatively newer version of aws. IPlease update your AWS CLI binary to the latest version if it fails.

VOLUME\_ID\_1=$(aws·ec2·create-volume·\

····--availability-zone·$AZ\_1·\

····--size·10·\

····--volume-type·gp2·\

····--tag-specifications·"ResourceType=volume,Tags=[{Key=KubernetesCluster,Value=$NAME}]"·\

····|·jq·-r·'.VolumeId')

VOLUME\_ID\_2=$(aws ec2 create-volume \

    --availability-zone $AZ\_1 \

    --size 10 \

    --volume-type gp2 \

    --tag-specifications "ResourceType=volume,Tags=[{Key=KubernetesCluster,Value=$NAME}]" \

    | jq -r '.VolumeId')

VOLUME\_ID\_3=$(aws ec2 create-volume \

    --availability-zone $AZ\_2 \

    --size 10 \

    --volume-type gp2 \

    --tag-specifications "ResourceType=volume,Tags=[{Key=KubernetesCluster,Value=$NAME}]" \

    | jq -r '.VolumeId')

We execute the aws ec2 create-volume command three times. As a result, we create three EBS volumes. Two of them are in one zone, while the third is in another. They all have 10 GB of space. We choose gp2 as the type of the volumes. The other types either require bigger sizes or are more expensive. When in doubt, gp2 is usually the best choice for EBS volumes.

We also define a tag that will help us distinguish the volumes dedicated to this cluster from those we might have in our AWS account for other purposes.

Finally, jq filters the output so that only the volume ID is retrieved. The results are stored in the environment variables VOLUME\_ID\_1, VOLUME\_ID\_2, and VOLUME\_ID\_3.

### Verification

Let’s take a quick look at one of the IDs we stored as an environment variable:

echo $VOLUME\_ID\_1

The output is as follows:

vol-092b8980b1964574a

Finally, to be on the safe side, we’ll list the volume that matches the ID and thus confirm that the EBS is created.

aws ec2 describe-volumes \

    --volume-ids $VOLUME\_ID\_1

The output is as follows:

{

  "Volumes": [

    {

      "AvailabilityZone": "us-east-2c",

      "Attachments": [],

      "Tags": [

        {

          "Value": "devops23.k8s.local",

          "Key": "KubernetesCluster"

        }

      ],

      "Encrypted": false,

      "VolumeType": "gp2",

      "VolumeId": "vol-092b8980b1964574a",

      "State": "available",

      "Iops": 100,

      "SnapshotId": "",

      "CreateTime": "2018-03-14T21:47:13.242Z",

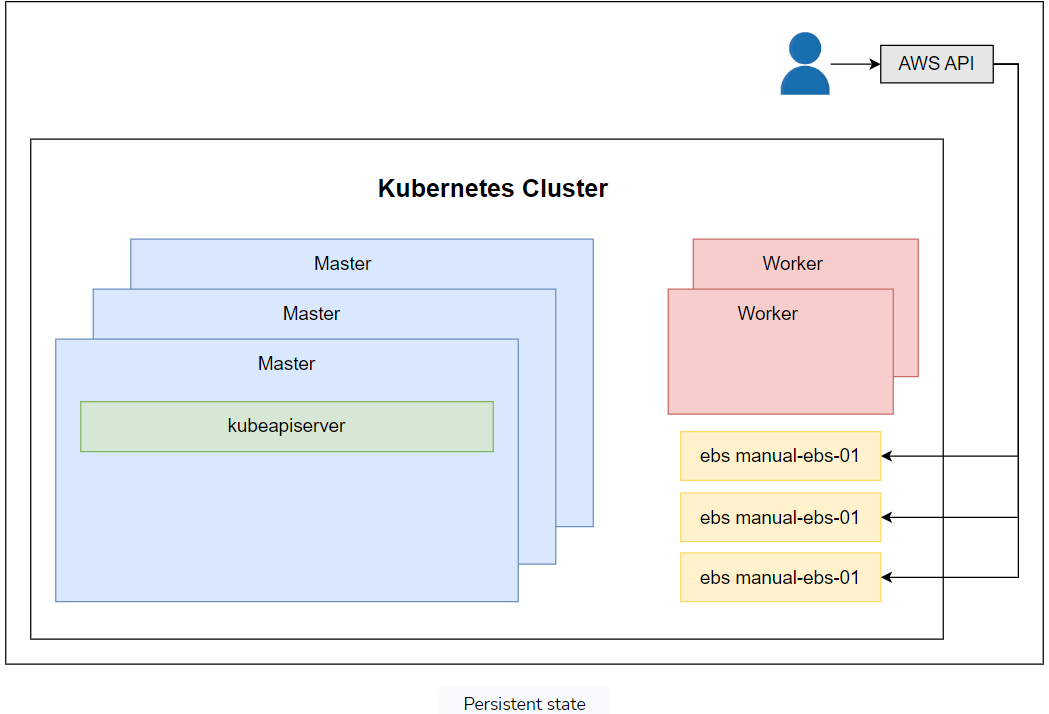
      "Size": 10

    }

  ]

}

Now that the EBS volumes are indeed available and in the same zones as the worker nodes, we can proceed and create Kubernetes persistent volumes.



In the next lesson, we will create Kubernetes PersistentVolumes.

# Creating Kubernetes Persistent Volumes

In this lesson, we will understand and create three Kubernetes persistent Volumes.

**We'll cover the following**

* [Understanding PersistentVolumes](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlWRNNkwWgE#Understanding-PersistentVolumes)
  + [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlWRNNkwWgE#Looking-into-the-definition)
    - [Exploring the spec section (Lines 7–15)](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlWRNNkwWgE#Exploring-the-spec-section-Lines-715)
  + [Exploring other storage platforms](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlWRNNkwWgE#Exploring-other-storage-platforms)
* [Creating the PersistentVolume](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlWRNNkwWgE#Creating-the-PersistentVolume)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlWRNNkwWgE#Verification)

## Understanding PersistentVolumes

The fact that we have a few EBS volumes available does not mean that Kubernetes knows about their existence. We need to add PersistentVolumes that will act as a bridge between our Kubernetes cluster and AWS EBS volumes.

**PersistentVolumes** allow us to abstract details of how storage is provided (e.g., EBS) from how it is consumed. Just like volumes, PersistentVolumes are resources in a Kubernetes cluster. The main difference is that their lifecycle is independent of individual Pods that are using them.

### Looking into the definition

Let’s look at a definition that will create a few PersistentVolumes:

cat pv/pv.yml

The output (limited to the first of the three volumes) is as follows:

kind: PersistentVolume

apiVersion: v1

metadata:

  name: manual-ebs-01

  labels:

    type: ebs

spec:

  storageClassName: manual-ebs

  capacity:

    storage: 5Gi

  accessModes:

    - ReadWriteOnce

  awsElasticBlockStore:

    volumeID: REPLACE\_ME\_1

    fsType: ext4

...

#### Exploring the spec section (Lines 7–15)

The spec section features a few interesting details:

**Line 8:** We set manual-ebs as the storage class name. We’ll see later what its function is. For now, just remember the name.

**Lines 9–10:** We define that the storage capacity is 5Gi. It does not need to be the same as the capacity of the EBS we created earlier, as long as it is not bigger. Kubernetes will try to match PersistentVolume with, in this case, EBS that has a similar, if not the same capacity. Since we have only one EBS volume with 10 GB, it is the closest (and the only) match to the PersistentVolume request of 5Gi. Ideally, persistent volumes capacity should match EBS size, but we want to demonstrate that any value equal to or less than the actual size should do.

**Lines 11–12:** We specify that the access mode should be ReadWriteOnce. This means that we’ll be able to mount the volume as read-write only once. Only one Pod will be able to use it at any given moment. Such a strategy fits us well since EBS cannot be mounted to multiple instances. Our choice of the access mode is not truly a choice but more an acknowledgment of the way how EBS works. The alternative modes are ReadOnlyMany and ReadWriteMany. Both modes would result in volumes that could be mounted to multiple Pods, either as read-only or read-write. Those modes would be more suitable for NFS. For example, EFS can be mounted by multiple instances.

**Line 13-15:** The spec fields we explored so far are common to all PersistentVolume types. Besides those, there are entries specific to the actual volume we associate with a Kubernetes PersistentVolume. Since we’re going to use EBS, we specify awsElasticBlockStore with the volume ID and file system type. Since we could not know in advance what the ID is of our EBS volume, the definition has the value set to REPLACE\_ME. Later on, we’ll replace it with the ID of the EBS we created earlier.

### Exploring other storage platforms

We can also specify many other types of storage platforms:

* If this cluster would run on Azure, we could use azureDisk or azureFile.
* In Google Compute Engine (GCE), it would be GCEPersistentDisk.
* We could have set up Glusterfs.
* If this cluster was running in an on-prem data center, it would probably be nfs.

There are quite a few others we could use but, since we’re running the cluster in AWS, many would not work, while others could be too difficult to set up.

Since EBS is already available, we’ll just roll with it. All in all, this cluster is in AWS, and awsElasticBlockStore is the easiest, if not the best choice.

## Creating the PersistentVolume

Now that we have an understanding of the YAML definition, we can proceed and create the PersistentVolume.

cat·pv/pv.yml·\

····|·sed·-e·\

····"s@REPLACE\_ME\_1@$VOLUME\_ID\_1@g"·\

····|·sed·-e·\

····"s@REPLACE\_ME\_2@$VOLUME\_ID\_2@g"·\

····|·sed·-e·\

····"s@REPLACE\_ME\_3@$VOLUME\_ID\_3@g"·\

····|·kubectl·create·-f·-·\

····--save-config·--record

We use cat to output the contents of the pv/pv.yml file and pipe it into sed commands which, in turn, replace the REPLACE\_ME\_\* strings with the IDs of the EBS volumes we created earlier. The result is sent to the kubectl create command that created the Persistent Volumes. As a result, we can see from the output that all three PersistentVolumes wearere created.

### Verification

Let’s look at the PersistentVolumes currently available in our cluster:

kubectl·get·pv

The output is as follows:

NAME          CAPACITY ACCESS MODES RECLAIM POLICY STATUS    CLAIM STORAGECLASS REASON AGE

manual-ebs-01 5Gi      RWO          Retain         Available       manual-ebs          11s

manual-ebs-02 5Gi      RWO          Retain         Available       manual-ebs          11s

manual-ebs-03 5Gi      RWO          Retain         Available       manual-ebs          11s

It should come as no surprise that we have three volumes.

The status column is the most interesting part

**Claiming Persistent Volumes**

Learn why and how to claim a PersistentVolume.

**We'll cover the following**

* [Usage of Persistent Volume](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLnX1l0kK1w#Usage-of-Persistent-Volume)
* [How can we claim PersistentVolumes?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLnX1l0kK1w#How-can-we-claim-PersistentVolumes)
  + [Creating the claim](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLnX1l0kK1w#Creating-the-claim)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLnX1l0kK1w#Verification)

**Usage of Persistent Volume**

Kubernetes PersistentVolumes are useless if no one uses them. They exist only as objects with relation to, in our case, specific EBS volumes. They are waiting for someone to claim them through the PersistentVolumeClaim resource.

Just like Pods that can request specific resources like memory and CPU, PersistentVolumeClaims can request particular sizes and access modes. Both are, in a way, consuming resources, even though of different types. Just as Pods should not specify on which node they should run, PersistentVolumeClaims cannot define which volume they should mount. Instead, the Kubernetes scheduler will assign them a volume depending on the claimed resources.

**How can we claim PersistentVolumes?**

We’ll use pv/pvc.yml to explore how we can claim a PersistentVolume.

cat pv/pvc.yml

The output is as follows:

kind: PersistentVolumeClaim

apiVersion: v1

metadata:

  name: jenkins

  namespace: jenkins

spec:

  storageClassName: manual-ebs

  accessModes:

    - ReadWriteOnce

  resources:

    requests:

      storage: 1Gi

The YAML file defines a PersistentVolumeClaim with the storage class name manual-ebs. This is the same class as the PersistentVolumes manual-ebs-\* we created earlier. The access mode and the storage request also match what we defined for the PersistentVolume.

Please note that we do not specify which volume we’d like to use. Instead, this claim specifies a set of attributes (storageClassName, accessModes, and storage). Any of the volumes in the system that match those specifications might be claimed by the PersistentVolumeClaim named jenkins. Keep in mind that resources do not have to be the exact match. Any volume with the same, or larger amount of storage is considered a match. A claim for 1Gi can be translated to at least 1Gi. In our case, a claim for 1Gi matches all three PersistentVolumes since they are set to 5Gi.

**Creating the claim**

Now that we have explored the definition of the claim, we can proceed and create it.

kubectl create -f pv/pvc.yml \

    --save-config --record

The output indicates that the persistentvolumeclaim "jenkins" is created.

**Verification**

Let’s list the claims and see what we get:

kubectl·--namespace·jenkins·\

····get·pvc

The output is as follows:

NAME    STATUS VOLUME        CAPACITY ACCESS MODES STORAGECLASS AGE

jenkins Bound  manual-ebs-02 5Gi      RWO          manual-ebs   17s

We can see from the output that the status of the claim is bound. This means that the claim found a persistent matching volume and bounded it. We can confirm that by listing the volumes.

kubectl get pv

The output is as follows:

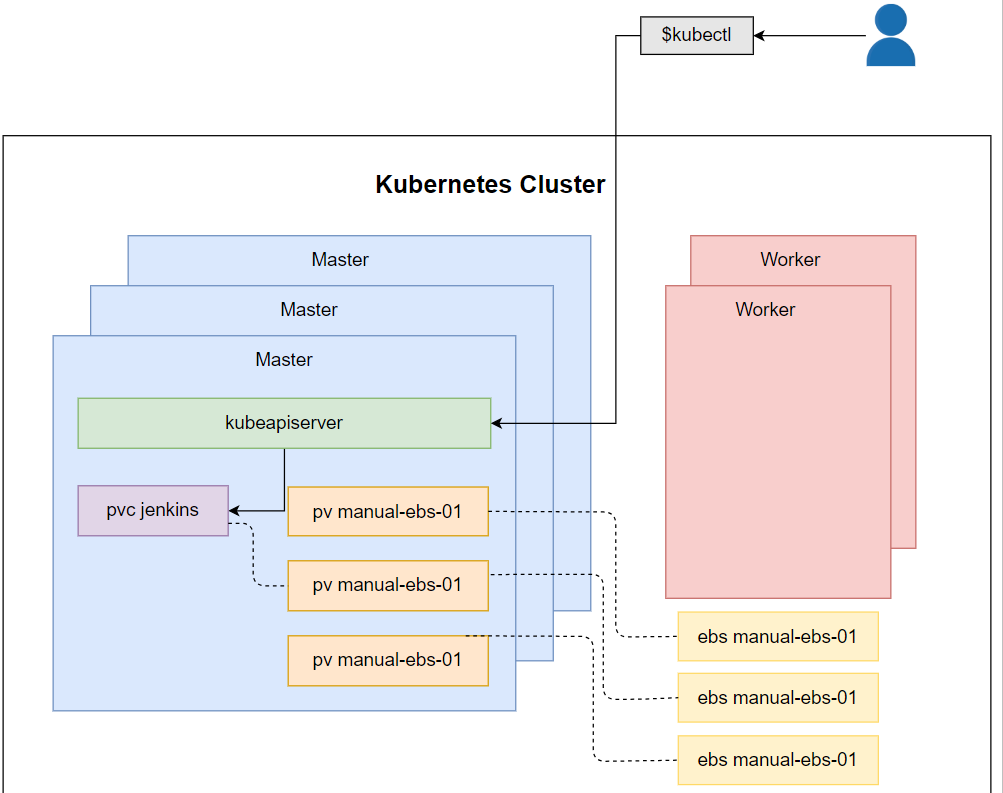
CLAIM           STORAGECLASS REASON AGE

manual-ebs-01 5Gi      RWO          Retain         Available                 manual-ebs          7m

manual-ebs-02 5Gi      RWO          Retain         Bound     jenkins/jenkins manual-ebs          7m

manual-ebs-03 5Gi      RWO          Retain         Available                 manual-ebs          7m

We can see that one of the volumes (manual-ebs-02) changed the status from available to bound. That is the volume bound to the claim we created a moment ago. We can see that the claim comes from jenkins Namespace and jenkins PersistentVolumeClaim.



**Note:** If a PersistentVolumeClaim cannot find a matching volume, it will remain unbound indefinitely unless we add a new PersistentVolume with the matching specifications.

We still haven’t accomplished our goal. The fact that we claimed a volume does not mean anyone uses it. On the other hand, our Jenkins needs to persist its state.

In the next lesson, we’ll join our PersistentVolumeClaim with a Jenkins container.

**Creating Deployment for Attaching Claimed Volumes to Pods**

Learn to create a Jenkins deployment for attaching claimed volumes to Pods and look into the sequence of associated events.

**We'll cover the following**

* [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVoBGA5YLN2#Looking-into-the-definition)
* [Deploying resources](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVoBGA5YLN2#Deploying-resources)
* [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVoBGA5YLN2#Verification)
* [The sequential break down](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVoBGA5YLN2#The-sequential-break-down)

In the previous lesson, we looked into claiming the persistent volumes. The next step is to attach these claimed volumes to pods.

**Looking into the definition**

Let’s look into a Jenkins definition:

cat pv/jenkins-pv.yml

The relevant parts of the output as as follows:

...

apiVersion: apps/v1

kind: Deployment

metadata:

  name: jenkins

  namespace: jenkins

spec:

  ...

  template:

    ...

    spec:

      containers:

      - name: jenkins

        ...

        volumeMounts:

        - name: jenkins-home

          mountPath: /var/jenkins\_home

        ...

      volumes:

      - name: jenkins-home

        persistentVolumeClaim:

          claimName: jenkins

      ...

You’ll notice that this time, we add a new volume jenkins-home, which references the PersistentVolumeClaim called jenkins. From the container’s perspective, the claim is a volume.

**Deploying resources**

Let’s deploy Jenkins resources and confirm that everything works as expected:

kubectl apply \

    -f pv/jenkins-pv.yml \

    --record

The output is as follows:

namespace "jenkins" configured

ingress "jenkins" configured

service "jenkins" configured

deployment "jenkins" configured

**Verification**

We’ll wait until the Deployment rolls out before proceeding with a test that will confirm whether Jenkins state is now persisted.

kubectl --namespace jenkins \

    rollout status \

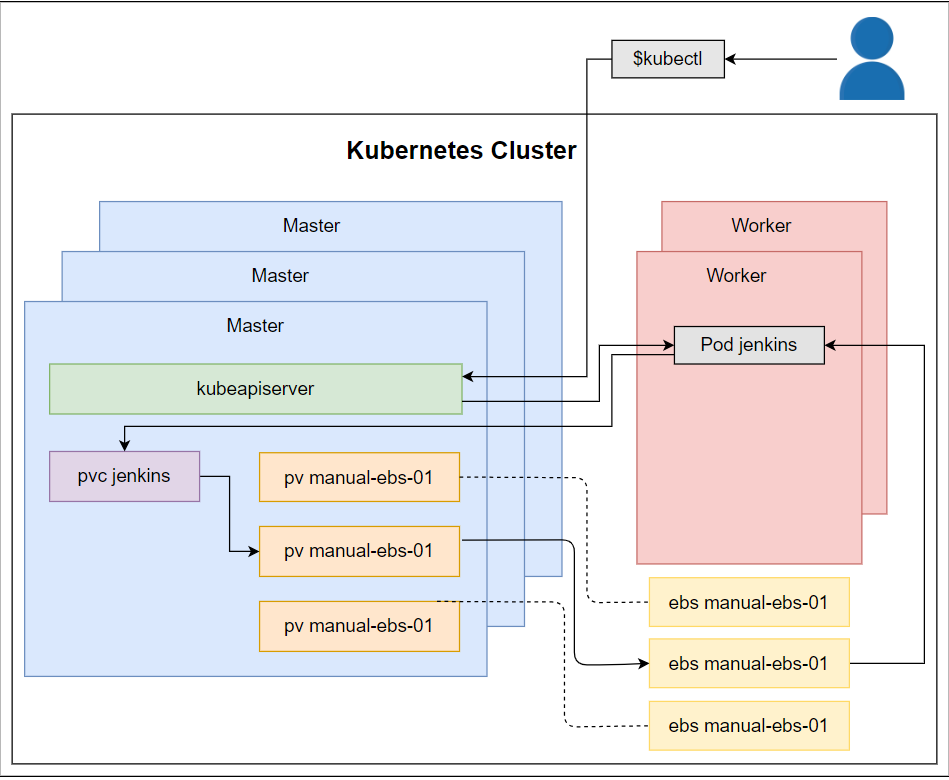
    deployment jenkins

Once the rollout is finished, we’ll see a message stating that the deployment "jenkins" is successfully rolled out.

We send a request to the Kubernetes API to create a Deployment. As a result, we get a ReplicaSet that, in turn, creates the jenkins Pod. It mounts the PersistentVolumeClaim, which is bound to the PersistenceVolume, that is tied to the EBS volume. As a result, the EBS volume was mounted to the jenkins container running in a Pod.

**The sequential break down**

A simplified version of the sequence of events is depicted in the below illustration.



1. We executed the kubectl command.
2. The kubectl sends a request to kube-apiserver to create the resources defined in pv/jenkins-pv.yml.
3. Among others, the jenkins Pod is created in one of the worker nodes.
4. Since the jenkins container in the Pod has a PersistentVolumeClaim, it mounted it as a logical volume.
5. The PersistentVolumeClaim is already bound to one of the PersistentVolumes.
6. The PersistentVolume is associated with one of the EBS volumes.
7. The EBS volume is mounted as a physical volume to the jenkins Pod.

In the next lesson, we will verify the persistence of the state and explore different types of failures that can occur.

**Verifying State Persistence and Exploring Failures**

Verify the state persistence of our Jenkins deployment and explore different failures that can occur.

**We'll cover the following**

* [Verifying the state persistence](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DYLBZgVDry#Verifying-the-state-persistence)
  + [Creating a new job](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DYLBZgVDry#Creating-a-new-job)
  + [Getting the Pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DYLBZgVDry#Getting-the-Pod)
  + [Terminating a process](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DYLBZgVDry#Terminating-a-process)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DYLBZgVDry#Verification)
* [Exploring the failures](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DYLBZgVDry#Exploring-the-failures)
  + [What if a node gets terminated?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DYLBZgVDry#What-if-a-node-gets-terminated)
  + [More servers, more tolerance](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DYLBZgVDry#More-servers-more-tolerance)
  + [What if the availability zone fails?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DYLBZgVDry#What-if-the-availability-zone-fails)
  + [Exploring the solutions](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DYLBZgVDry#Exploring-the-solutions)

**Verifying the state persistence**

Now that Jenkins is up and running, we’ll execute a similar set of steps as before and validate that the state has persisted across failures.

open "http://$CLUSTER\_DNS/jenkins"

We have opened the Jenkins home screen.

**Creating a new job**

If you are not authenticated, please click the “Log in” link and type “jdoe” as the User and “incognito” as the Password. Click the “log in” button.

You’ll see the “Create new jobs” link. Click it. Type “my-job” as the item name, select “Pipeline” as the job type, and click the “OK” button. Once inside the job configuration screen, all we have to do is click the “Save” button. An empty job will be enough to test persistence.

**Getting the Pod**

Now we need to find out the name of the Pod created through the jenkins Deployment.

POD\_NAME=$(kubectl \

    --namespace jenkins \

    get pod \

    --selector=app=jenkins \

    -o jsonpath="{.items[\*].metadata.name}")

**Terminating a process**

With the name of the Pod stored in the environment variable POD\_NAME, we can proceed and terminate the java process that’s running Jenkins.

kubectl --namespace jenkins \

    exec -it $POD\_NAME pkill java

We teminated the Jenkins process and therefore simulated failure of the container. As a result, Kubernetes detected the failure and recreated the container.

**Verification**

A minute later, we can open the Jenkins home screen again and check whether the state (the job we created) is preserved.

1

open "http://$CLUSTER\_DNS/jenkins"

As you can see, the job is still available, therefore proving that we successfully mounted the EBS volume as the directory where Jenkins preserves its state.

**Exploring the failures**

**What if a node gets terminated?**

If instead of destroying the container, we terminate the server where the Pod is running, the result would be the same from the functional perspective. The Pod would be rescheduled to a healthy node. Jenkins would start again and restore its state from the EBS volume. However, such behavior is not guaranteed to happen in our cluster.

We have only two worker nodes distributed in two (out of three) availability zones. If the node that hosted Jenkins failed, we’d be left with only one node. To be more precise, we’d have only one worker node running in the cluster until the Auto Scaling Group detects that an EC2 instance is missing and recreates it. During those few minutes, the single node we’re left with is not in the same zone. As we already mentioned, each EBS instance is tied to a zone, and the one we mounted to the Jenkins Pod would not be associated with the zone where the other EC2 instance is running.

As a result, the PersistentVolume could not rebound the EBS volume and, therefore, the failed container could not be recreated until the failed EC2 instance is recreated.

The chances are that the new EC2 instance would not be in the same zone as the one where the failed server was running. Since we’re using three availability zones, and one of them already has an EC2 instance, AWS would recreate the failed server in one of the other two zones. We have a fifty percent chance that the new EC2 would be in the same zone as the one where the failed server was running. Those are not good odds.

**More servers, more tolerance**

In the real world scenario, we’d probably have more than two worker nodes. Even a slight increase to three nodes would give us a very good chance that the failed server would be recreated in the same zone. Auto-Scaling Groups are trying to distribute EC2 instances more or less equally across all the zones. However, that is not guaranteed to happen. A good minimum number of worker nodes would be six.

The more servers we have, the higher the chances that the cluster is fault-tolerant. That is especially true if we are hosting stateful applications. As it goes, we almost certainly have those. There’s hardly any system that does not have a state in some form or another.

If it’s better to have more servers than fewer, we might be in a complicated position if our system is small and needs, let’s say fewer than six servers. In such cases, we’d recommend running smaller VMs. If, for example, you planned to use three t2.xlarge EC2 instances for worker nodes, we might reconsider that and switch to six t2.large servers. More nodes mean more resource overhead spent on operating systems, Kubernetes system Pods, and few other things. However, we believe that is compensated with a more stable cluster.

**What if the availability zone fails?**

There is still one more situation we might encounter. A whole availability zone (data center) might fail. Kubernetes will continue operating correctly. It’ll have two instead of three primary nodes, and the failed worker nodes will be recreated in healthy zones. However, we’d run into trouble with our stateful services. Kubernetes would be unable to reschedule those that were mounted to EBS volumes from the failed zone. We’d need to wait for the availability zone to come back online, or we’d need to move the EBS volume to a healthy zone manually. The chances are that, in such a case, the EBS would not be available and, therefore, could not be moved.

**Exploring the solutions**

We could create a process that would replicate data in (near) real-time between EBS volumes spread across multiple availability zones, but that also comes with a downside. Such an operation would be expensive and likely slow down state retrieval while everything is fully operational. Should we choose lower performance over high availability? Is the increased operational overhead worth the trouble? The answer to those questions will differ from one use case to another.

There is yet another option. We could use the [Elastic File System (EFS)](https://aws.amazon.com/efs/) instead of EBS. But that would also impact performance since EFS tends to be slower than EBS. On top of that, there is no production-ready EFS support in Kubernetes. At the time of this writing, the [efs provisioner](https://github.com/kubernetes-incubator/external-storage/tree/master/aws/efs" \t "_blank) is still in the beta phase. By the time you read this, things might have changed. Or maybe they didn’t. Even when the efs provisioner becomes stable, it will still be a slower and more expensive solution than EBS.

Maybe you’ll decide to ditch EBS (and EFS) in favor of some other type of persistent storage. There are many different options you can choose. We won’t explore them since an in-depth comparison of all the popular solutions would require much more space than what we have left.

**Note:** All in all, every solution has pros and cons, and none would fit all use cases. For good or bad, we’ll stick with EBS for the remainder of this course.

Now that we’ve explored how to manage persistent static volumes, we’ll try to accomplish the same results using a dynamic approach. But before we do that, we’ll see what happens when some of the resources we created are removed.

In the next lesson, we will play around with the created deployment by removing the resources and exploring the effects.

# Removing Resources and Exploring Effects

Practice with the Deployment by removing the created resources and exploring the effects of removal.

**We'll cover the following**

* [Deleting the Deployment](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gxmZK5R2N8l#Deleting-the-Deployment)
* [Deleting the PersistentVolumeClaim](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gxmZK5R2N8l#Deleting-the-PersistentVolumeClaim)
  + [Reclaim policies](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gxmZK5R2N8l#Reclaim-policies)
    - [The Retain reclaim policy](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gxmZK5R2N8l#The-Retain-reclaim-policy)
    - [Other types of reclaim policy](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gxmZK5R2N8l#Other-types-of-reclaim-policy)
* [Deleting the resources](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gxmZK5R2N8l#Deleting-the-resources)

## Deleting the Deployment

Let’s delete the jenkins Deployment:

1

2

kubectl·--namespace·jenkins·delete·\

····deploy·jenkins

The output shows us that the deployment "jenkins" is deleted.

Did anything happen with the PersistentVolumeClaim and the PersistentVolume?

1

2

3

kubectl --namespace jenkins get pvc

kubectl get pv

The combined output of both commands is as follows:

1

2

3

4

5

6

7

NAME    STATUS VOLUME        CAPACITY ACCESS MODES STORAGECLASS   AGE

jenkins Bound  manual-ebs-02 5Gi      RWO          manual-ebs     57s

NAME          CAPACITY ACCESS MODES RECLAIM POLICY STATUS    CLAIM           STORAGECLASS REASON AGE

manual-ebs-01 5Gi      RWO          Retain         Available jenkins/jenkins manual-ebs          10m

manual-ebs-02 5Gi      RWO          Retain         Bound     jenkins/jenkins manual-ebs          10m

manual-ebs-03 5Gi      RWO          Retain         Available jenkins/jenkins manual-ebs          10m

Even though we have removed the Jenkins Deployment and the Pod that used the claim, both the PersistentVolumeClaim and PersistentVolumes are intact. The manual-ebs-01 volume is still bound to the jenkins claim.

## Deleting the PersistentVolumeClaim

What would happen if we remove the PersistentVolumeClaim jenkins?

kubectl·--namespace·jenkins·\

····delete·pvc·jenkins

The output shows that the persistentvolumeclaim "jenkins" is deleted.

Now, let’s see what happens to the PersistentVolumes:

kubectl get pv

The output is as follows:

NAME          CAPACITY ACCESS MODES RECLAIM POLICY STATUS   CLAIM           STORAGECLASS REASON AGE

manual-ebs-01 5Gi      RWO          Retain         Available jenkins/jenkins manual-ebs          10m

manual-ebs-02 5Gi      RWO          Retain         Released  jenkins/jenkins manual-ebs          10m

manual-ebs-03 5Gi      RWO          Retain         Available jenkins/jenkins manual-ebs          10m

This time, the manual-ebs-2 volume is Released.

### Reclaim policies

This might be a good moment to explain the Retain policy applied to the PersistentVolumes we created.

**Note:** TheReclaimPolicy defines what should be done with a volume after it’s released from its claim.

The policy was applied the moment we deleted the PersistentVolumeClaim that was bound to manual-ebs-02. When we created the PersistentVolumes, we did not specify ReclaimPolicy, so the volumes were assigned the default policy which is Retain.

#### The Retain reclaim policy

The Retain reclaim policy enforces manual reclamation of the resource. When the PersistentVolumeClaim is deleted, the PersistentVolume still exists, and the volume is considered released. But it is not yet available for other claims because the previous claimant’s data remains on the volume. In our case, that data is Jenkins state. If we’d like this PersistentVolume to become available, we’d need to delete all the data on the EBS volume.

Since we’re running the cluster in AWS, it is easier to delete than to recycle resources, so we’ll remove the released PersistentVolume instead of trying to clean everything we generated inside the EBS. Actually, we’ll remove all the volumes since we are about to explore how we can accomplish the same effects dynamically.

#### Other types of reclaim policy

The other two reclaim policies are Recycle and Delete. Recycle is considered deprecated so we won’t waste time explaining it. The Delete policy requires dynamic provisioning, but we’ll postpone the explanation until we explore that topic.

## Deleting the resources

Let’s delete some stuff:

kubectl·delete·-f·pv/pv.yml

The output is as follows:

persistentvolume "manual-ebs-01" deleted

persistentvolume "manual-ebs-02" deleted

persistentvolume "manual-ebs-03" deleted

We can see that all three PersistentVolumes are deleted. However, only Kubernetes resources are removed. We still need to manually delete the EBS volumes.

If we go to our AWS console, you’ll see that all three EBS volumes are now in the Available state and waiting to be mounted. We’ll delete them all.

aws·ec2·delete-volume·\

····--volume-id·$VOLUME\_ID\_1

aws ec2 delete-volume \

    --volume-id $VOLUME\_ID\_2

aws ec2 delete-volume \

    --volume-id $VOLUME\_ID\_3

We are finished with the manual creation of PersistentVolumes. If we’d use this approach to volume management, the cluster administrator would need to ensure that there is always an extra number of available volumes that can be used by new claims. It is tedious work that often results in having more volumes than we need. On the other hand, if we don’t have a sufficient number of available (unused) volumes, we’re risking that someone will create a claim that will not find a suitable volume to mount.

Manual volume management is sometimes unavoidable, especially if you choose to use on-prem infrastructure combined with NFS.

AWS is all about dynamic resource provisioning. In the next lesson, we’ll look into the dynamic provision of PersistentVolumes.