Verifying the State Persistence and Exploring the Failures

In this lesson, we will verify the state persistence of our Jenkins deployment and explore different failures that can occur.

WE'LL COVER THE FOLLOWING

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 is persisted across failures.

```
open "http://$CLUSTER_DNS/jenkins"
```

We opened 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}")
```

Killing a Process

With the name of the Pod stored in the environment variable `POD_NAME`, we can proceed and kill `java` process that's running Jenkins.

```
kubectl --namespace jenkins \
    exec -it $POD_NAME pkill java
```

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

Verification

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

```
open "http://$CLUSTER_DNS/jenkins"
```

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

Exploring the Failures

What if a Node Gets Killed?

What If a Node Gets Killed?

If instead of destroying the container, we terminated the server where the Pod is running, the result, from the functional perspective, would be the same. The Pod would be rescheduled to a healthy node. Jenkins would start again and restore its state from the EBS volume. Or, at least, that's what we'd hope. 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 re-bound 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'd have fifty percent chances 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 are 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 less, we might be in a complicated position if our system is small and needs, let's say, less 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, you might reconsider that and switch to six t2.large servers. Sure, more nodes mean more resource overhead spent on operating systems, Kubernetes system Pods, and few other things. However, we believe that is compensated with bigger stability of your 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 master nodes, and the failed worker nodes will be recreated in healthy zones. However, we'd run into trouble with our stateful services. Kubernetes would not be able 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 be replicating 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 would 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 Elastic File System (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 is still in 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 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.

> 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 explored how to manage static persistent volumes, we'll try to accomplish the same results using 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.