Analyzing Failure of the Stateful Application

In this lesson, we will analyze the failure of Jenkins application due to lack of handling the state persistence.

WE'LL COVER THE FOLLOWING

- Opening Jenkins in a Browser
  - 📝 A note to Windows users
- Creating a Job
- 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"
```

📝 A note to Windows users

> 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 `jdoe` administrator, we can proceed and create a job. That will generate a state that 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

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 killing `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 retrieved the Pods from the `jenkins` Namespace, filtered them with the selector `api=jenkins`, and formatted the output as `json`.

The **output**, limited to the relevant parts, as is follows.

```
{
  "apiVersion": "v1",
    "items": [
    {
      ...
      "metadata": {
        ...
        "name": "jenkins-8768d486-lmv6b",
        ...
```

We can see that the name is inside `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 kill the `java` process.

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

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

Let's see what happened to the job we created earlier. I'm sure you know the answer, but we'll check it anyway.

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

As expected, *my-job* is nowhere to be found. The container that was hosting `/var/jenkins_home` directory failed, and it was replaced with a new one. The state we created is lost.

We already saw in the *Volumes* chapter that we can mount a volume in an attempt to preserve 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 would fail, the state stored in `emptyDir` would be gone.

Such a solution would be only slightly better than not using any volume. By using local disk we would only postpone the inevitable, and, sooner or later, we'd get to the same situation. We'd be left wondering why we lost everything we created in Jenkins. We can do better than that.

In the next lesson, we will explore creating AWS Volumes.