DevOps and Security Lab 4 - Kubernetes advanced

The lab goal is to get familiar with the advanced Kubernetes manifests, then try them on practice and deploy application on Kubernetes cluster as well.

> Individual lab. **Choose this lab flavor if you are familiar with Kubernetes and already know basic k8s specs, did some k8s labs/worked on practice before.**
>
> While working on some tasks, keep in mind that you might wipe your created objects after the task completion before to go the next task in order do not mess some logically conflicted objects.
>
> We are going to work within the local cluster (minikube, k3s, MicroK8s...) but nothing prevents you from deploying a cluster on a real cloud like AWS EKS or On-premise bare-metal solution.

Task 1 - Preparation

1.1. Choose an application that has such characteristics as (*points with "star" are not required but recommended*):

- any non-confidential data like in JSON, XML or TOML format
- application configuration values (or environment variables)
- any secrets (like authorization credentials)
- availability from outside (on Internet)
- has any health checks or/and status endpoints
- data that is stored on distributed storage/volumes
- your app has a logic to perform short temporal work
- envisage some work should be done before the main container is started

It could be any yours application or open-source project that meets the requirements. You are even able to work with different applications in different lab tasks but try to avoid it. However, it's **much better** to work with the single project but it's acceptable to involve only the required application functionality regarding the particular task requirements. Put the git link to your chosen project into report and provide a brief project description for what it is and how it works.

We assume that you already know how to work with k8s basic `specs` tha are offered in the basic lab flavor. Thus, you already have an application deployed with this `specs` or it's easy and quickly for you to create and apply those basic k8s manifests. So, you just should list basic manifests that you applied to your app deployment briefly and proceed with the current advances lab tasks.

1.2. Install and set up the necessary tools:

- kubectl
- minikube or its alternative
- helm

1.3. Get access to Kubernetes Dashboard.

Task 2 - k8s PersistentVolumes

> `configMaps` and `Secrets` allows us to work with read-only data. From the other hand, with `PersistentVolumeClaim` we can work with rewritable data, mount multiple resources to pod, work with different filesystems types and storage types, have an opportunities to recover the data after pod death and so on.

1. Make sure that your application writes a data to a file from app logic work. If not, modify your app to do it.
2. Figure out the necessary `PersistentVolumeClaim` spec fields.
3. Create and apply a new `PersistentVolumeClaim` manifest. Here you have to specify a volume with some data that should be connected to app pod.
4. With `kubectl`, get and describe PVC and PV.
5. Provide a PoC that the data from the `PertistentVolume` is available for your app pod on specified path.
6. Show the case when your app writes the data to the path that is specified on PV. Provide a PoC.
7. Play with different `accessModes` and `Retain` policies within `PersistentVolumeClaim` manifest.
8. Put the results into report.

Task 3 - k8s StatefulSet

> `StatefulSet` is advanced abstraction of Deployment. While Deployment create and deploy pods in parallel and with random names, `StatefulSet` provides an ability to have a stable application deployment environment and to organize the objects deployments priority and order. It's a must have feature for production or replicated projects as databases, message queue or other complicated applications.

1. Figure out the necessary `StatefulSet` spec fields.
2. Deploy your app using `StatefulSet`.
3. With `kubectl`, get and describe sts.
4. Scale and rollout your app replicas through `StatefulSet`.
5. Implement `Startup`, `Liveness` or `Readiness` probes within StatefulSet. Explain their difference.
6. Envisage some work to be done before the core app container is created and started, i.e. add `init` containers.
7. Stop your application pod (make it as died). Add a temporal "rescue" pod using Pod manifest. As for images, one of the best options are `praqma/network-multitool` or `busybox/busybox`. Example:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  volumes:
    - name: <pv_name>
```

```
      persistentVolumeClaim:
        claimName: <PVC name that assosiated to the crashed pods>
  containers:
  - image: busybox
    name: busybox
    volumeMounts:
      - mountPath: /mnt
        name: <pv_name>
```

8. Then go to the temporal container shell. And copy the mounted data to your host disk. After this temporal container could be deleted.

> Note for steps 5-7. If your Persistent Volume Claim Storage Class policy is defined as RWO , it means that PV could not be mounted both to the crashed pod and to the busybox pod at the same time. In this case, you have to remove the crashed container and only then mount the particular PV to the busybox. You will not lose the PVs with data until you haven't removed StatefulSet and PVCs. Also It seems that you also can not mount multiple PVs to the single busybox pod if these volumes are in the different geographical regions.

8. Put the results into report.

Task 4 - k8s Job

> `Job` is Kubernetes manifest to designed for running some time-limited work and then stop. Job creates a pod that do some work for a relatively short time. After work is successfully, `Job` removes the pod and it's gone itself. If this time-limited pod fails `Job` will replace it and try to start again.

1. Figure out the necessary `Job` spec fields.
2. Prepare and apply a `Job` manifest that creates and run a new temporal pod that to some work and then it's gone. For example, it could be some unit test or a load script that sends loading to your application database or pod filesystem.
3. With `kubectl`, get and describe your `Jobs`.
4. Put the results into report.

Bonus: schedule your Jobs with CronJobs. Demonstrate a PoC.

Task 5 - k8s Ingress

> Ingress is a resource that defines the rules for routing external traffic for the cluster. This is a Kubernetes API object that manages external access to services by defining the host, path, and other necessary information.

1. Figure out the necessary `Ingress` spec fields.
2. Create `Ingress` spec, bind it with your `Service` and apply.
3. With `kubectl`, get and describe your `Ingress`.
4. Get access to your app from outside network using `Ingress.host` value.
5. Question: if we don't have an opportunity to use `Ingress`, how we can enable access to our app from outside directly?
6. Put the results into report.

Task 6 - k8s Autoscaling and Scheduler

> Autoscaling is feature that allows a cluster to automatically increase or decrease the number of nodes, or adjust pod resources, in response to demand. From the other hand, Scheduler in Kubernetes is responsible for distributing Pods across worker nodes in the cluster. The main task of the scheduler is to optimize the placement of pods, taking into account the available resources on the nodes, the requirements of each pods, and various other factors.

1. Figure out the necessary `HorizontalPodAutoscaler`, `VerticalPodAutoscaler` and `*PodDisruptionBudget*` specs fields.
2. Answer when `HorizontalPodAutoscaler` and `VerticalPodAutoscaler` are more suitable for.
3. Set up `minAllowed` and `maxAllowed` for vertical autoscaling and `averageUtilization` for horizontal autoscaling.
4. Apply `HorizontalPodAutoscaler` and `VerticalPodAutoscaler`. Provide a PoC of these features work.
5. Create and apply `PodDisruptionBudget`. Set `minAvailable` or `maxUnavailable` fields. Answer when both of them are more suitable.
6. With `kubectl`, get and describe your `HorizontalPodAutoscaler`, `VerticalPodAutoscaler` and `PodDisruptionBudget`.
7. Put the results into report.

Task 7 - k8s RBAC

> RBAC (Role-based access control) is a system for allocating access rights to various objects in a Kubernetes cluster.

1. Figure out the necessary `ServiceAccount`, `ClusterRole` and `*ClusterRoleBinding*` specs fields.
2. Create a new `ServiceAccount` dedicated to your app deployment.
3. Create `ClusterRole` and `ClusterRoleBinding` manifests, connect them to the custom `ServiceAccount` then.
4. With `kubectl`, get and describe your Service Accounts, Roles and Bindings.
5. Provide a Poc that your custom `ServiceAccount` has a different permissions to cluster resources rather than default one.
6. Question: why to create a custom `ServiceAccounts` is a good practice? Why it's useful?
7. Put the results into report.

Task 8 - CRD

> CRD (Custom Resource Definition) is a special resource in Kubernetes that allows you to operate with any data. This is usually a table with the names of the elements and column types. The scope of CRD usage is wide.

1. Figure out the necessary spec `CustomResourceDefinition` fields.
2. Develop a data structure for your CRD that might be useful to insert and affect on application work.
3. Convert your data like table to CRD and apply it.
4. With `kubectl`, get and describe your `CRD`.
5. With `kubectl`, operate with record creation, getting list or records and record deletion.
6. Put the results into report.

Bonus: provide a PoC where CRD usage is useful for your app and allows to change/modify app configutations.

Task 9 - k8s Helm Chart

> Helm is a Kubernetes package manager that allows to organize, pack, upload, download, deploy and wipe Kubernetes objects in fast and effective way. Chart is a unit of Helm package.

1. After installing the `helm` tool, create a helm chart for your application, i.e. summarize and join all necessary manifest that you developed during the lab and pack all of them in helm package.
2. Generate resources and check that helm package yaml syntax is correct with `helm template` command.
3. Upload your own chart to any helm registry (*optional*).
4. Install on your k8s cluster any helm chart that you like with `helm` command and make sure that it's running. *Hint: you can join this subtask with Bonus 1.*
5. Wipe deployment and remove this helm chart with `helm` command.
6. Put the results into report.

Bonus 1 : play with variables override through values.yaml . For example, with different image pull policies or image tag.

Bonus 2: set up and show in practice cases with helm hooks usage.

Bonus 1 - k8s Monitoring

> `ServiceMonitor` is used together with the Prometheus operator stack for more convenient and automated monitoring of resources in Kubernetes. This resource is designed to simplify the process of collecting metrics from various services.

1. Figure out the necessary `ServiceMonitor` spec fields.
2. Deploy kube-prometheus-stack in other namespace.
3. Create `ServiceMonitor` for your app metrics scrapping.
4. With `kubectl`, get and describe your `ServiceMonitor`.
5. Find a way how to get and view your application metrics in Prometheus.
6. Using `kube-prometheus-stack` functionality, enable monitoring of k8s nodes, provide a PoC.
7. Try to apply `PodMonitor` spec.
8. Put the results into report.

Bonus 2 - GitOps

Recently, the concept of GitOps CD pipelines has been gaining popularity. The point is to deploy (and then monitor and manage them) Kubernetes deployments (in fact, there are helm charts) through the special configurations in the git repository automatically based on pre-defined updating triggers and policies.

Try to configure your helm chart deployment via GitOps CD. ArgoCD and Flux are the most popular and welldeveloped to date solution.

Bonus 3 - Secure Cluster Policies

> One of important tasks of real k8s cluster provision is to make it secure. There are various ways to do it: apply third-party solutions for cluster policies audit, make secure linux containers and so on.

1. For your StatefulSet/Deployment, try to enable such settings:

```
podSecurityContext:
  enabled: true
  runAsUser: 1001
  runAsGroup: 1001
  runAsNonRoot: true
  seccompProfile:
    type: RuntimeDefault
  fsGroup: 1001
  fsGroupChangePolicy: "OnRootMismatch"

  ---
containerSecurityContext:
  enabled: true
  runAsUser: 1001
  runAsGroup: 1001
  runAsNonRoot: true
  seccompProfile:
    type: RuntimeDefault
  privileged: false
  procMount: Default
  readOnlyRootFilesystem: true
  allowPrivilegeEscalation: false
  capabilities:
    drop:
    - ALL
```

2. Verify that with these settings your application is still working.
3. Try to add such settings via `values.yaml` in helm chart.
4. Research such security tools concepts as Falco and Kyverno. Try to deploy them and use on practice.

Bonus 4 - Multi nodes cluster

In production we use multiple nodes within the k8s cluster for sure. Find any way to deploy multi nodes cluster. Then, learn, play and use the following k8s functions:

1. Cluster AutoScaler deployment.
2. `DaemonSet` spec.
3. `Affinity` and `anti-affinity` rules.
4. Add a priority classes and apply `PriorityClass` spec.