# Solution M6: Observability and Troubleshooting

For this challenge we will assume that we are working as the **root** user on the control plane node of a three node **Kubernetes** cluster. The pod network plugin is of no importance, so it can be any of the available. We will execute all tasks in order but keep in mind that every other task expects that the artefacts from the previous have been deleted

The original files are in the **in/** folders and the resulting ones in the **out/** folders

# Task 1

**Challenge:**

Try to solve **scenario 2** and make the application working again

**Solution:**

Okay, having the file (**scenario-2.yaml**), we may explore it a bit

We may or may not find anything disturbing there

So, instead of exploring it, let’s try to deploy it and see what will happen

**kubectl apply -f scenario-2.yaml**

Good, we can see that there are some issues being reported

It appears that the **readiness-http** pod has two invalid volume mounts - **html**

After a check, we will notice that there is a volume named **data** and not **html**

Let’s change the two mounts from **html** to **data** (on row **15** and row **39**)

Save and close and try to deploy again

This time it looks successful but let’s check

**kubectl get pods,svc**

There is an issue with the pod. Let’s check

**kubectl describe pod readiness-http**

It seems that there is an issue with the init container – a wrong call – **bash** (which is not available in **alpine**) instead of **sh**

Open the file and correct this (row **11**). It should become

**command: ["/bin/sh", "-c"]**

Save and close and try to deploy again but first delete the old resources

**kubectl delete -f scenario-2.yaml**

**kubectl apply -f scenario-2.yaml**

And check again

**kubectl get pods,svc**

Again, there is an issue with the pod. Let’s check

**kubectl describe pod readiness-http**

Hm, there is a problem with the init container again but is not that obvious

Let’s check the logs

**kubectl logs readiness-http -c init-data**

Aha, there is an issue with the arguments passed to the **/bin/sh** command

Open the file again end change the **13th** row to

**- echo '(Almost) Always Ready to Serve ;)' > /data/index.html**

Save and close and try to deploy again but first delete the old resources

**kubectl delete -f scenario-2.yaml**

**kubectl apply -f scenario-2.yaml**

And check again

**kubectl get pods,svc**

Again, there is an issue with the pod. Let’s check

**kubectl describe pod readiness-http**

Again, there is a problem with the init container again but is not that obvious

Let’s check the logs

**kubectl logs readiness-http -c init-data**

Aha, there is an issue with the path where we are creating the file – the folder is not existent

If do another round over the manifest, we will notice that the path should be **/usr/share/nginx/html** or we should correct the volume mount to point to **/data** instead. Let’s go with the first correction

Open the file and correct this (row **13**). It should become

**- echo '(Almost) Always Ready to Serve ;)' > /usr/share/nginx/html/index.html**

Save and close and try to deploy again but first delete the old resources

**kubectl delete -f scenario-2.yaml**

**kubectl apply -f scenario-2.yaml**

If we check again after a while

**kubectl get pods,svc**

We will notice that now the pod looks okay

Now, check if the pod is registered with the service

**kubectl describe service readiness-cmd**

No, it is not. If we explore the label selector that the service is using, we will notice that it is different than what the pod has. We can correct either of those

Let’s change the name, label and selector of the service to **readiness-http** instead of **readiness-cmd**

But first, delete the resources

**kubectl delete -f scenario-2.yaml**

Then apply the changes. Now, the service manifest will become

apiVersion: v1

kind: Service

metadata:

  name: readiness-http

  labels:

    app: readiness-http

spec:

  type: NodePort

  ports:

  - port: 80

    nodePort: 30001

    protocol: TCP

  selector:

    app: readiness-http

Save and close and deploy again

**kubectl apply -f scenario-2.yaml**

If we check again after a while

**kubectl get pods,svc**

We will see that everything looks okay

Again, check if the pod is registered with the service

**kubectl describe service readiness-http**

Yes, it is there

Open a browser tab and navigate to http://<cluster-ip>:30001

Our application should be there 😊

# Task 2

**Challenge:**

Try to solve **scenario 3** and make the application working again

**Solution:**

Okay, having the file (**scenario-3.yaml**), we may explore it a bit

We may or may not find anything disturbing there

So, instead of exploring it, let’s try to deploy it and see what will happen

**kubectl apply -f scenario-3.yaml**

Everything seems just fine but let’s check anyway

**kubectl get pods,svc**

If repeat again after a while, we will notice that the pod is keep restarting

Let’s check

**kubectl describe pod startup-mixed**

Aha, it looks that the startup probe is failing

Delete the resources

**kubectl delete -f scenario-3.yaml**

Open the manifest and check

It seems that the startup probe checks too early. We may add an **initialDelaySeconds** option there

Considering the code in the **cont-sidecar-postpone** – it sleeps for **20** seconds and then creates the file that the startup probe is looking for, it seems that **25** second of initial delay is more than enough

The code of the startup probe should become

    startupProbe:

      exec:

        command:

        - cat

        - /check/healthy.html

      initialDelaySeconds: 25

      failureThreshold: 3

      periodSeconds: 5

Save and close the file. Deploy it again

**kubectl apply -f scenario-3.yaml**

Everything seems just fine but let’s check anyway

**kubectl get pods,svc**

If repeat again after a while, we will notice that the pod is keep restarting

Let’s check

**kubectl describe pod startup-mixed**

Aha, again the startup probe is failing. If we look again at the manifest, we will notice that it looks for the file in a wrong place. Change the code to

        command:

        - cat

        - /usr/share/nginx/html/healthy.html

Delete and redeploy the resources

**kubectl delete -f scenario-3.yaml**

**kubectl apply -f scenario-3.yaml**

Everything seems just fine but let’s check anyway

**kubectl get pods,svc**

If we repeat again after a while, we will notice that the pod is finally running

But if we repeat again, we will notice that again something is causing a restart of the main container

This time, it is the liveness probe

If we look at the manifest more carefully, we will notice that it is a HTTP probe, and it is looking in a wrong place

Change the code to

    livenessProbe:

      httpGet:

        path: /healthy.html

        port: 80

Delete and redeploy the resources

**kubectl delete -f scenario-3.yaml**

**kubectl apply -f scenario-3.yaml**

Everything seems just fine but let’s check anyway

**kubectl get pods,svc**

If we repeat again after a while, we will notice a restart and if we dig further, we will notice that it is due to the liveness probe, but it is by design

Now, pick a moment when everything is running and check if the service knows about the pod

**kubectl describe service startup-mixed**

It appears that the pod never gets registered with the service

If we dig a bit further, we will notice that there is an issue with the label selector of the service

It states **app=startup-nixed** instead of **app=startup-mixed**

Let’s correct it and redeploy the objects

**kubectl apply -f scenario-3.yaml**

Now, when the pod is considered live, it will be registered with the service and our application should be reachable

Open a browser tab and navigate to http://<cluster-ip>:30001

Hm, it is not working …

If we check again, we will see that the port where the service is sending client requests is not the correct one

Change it from **8080** to **80** and redeploy again

Return to the browser (when the pod is considered live and registered with the service) and check again

It is there and working (most of the time) 😊

Of course, it will still crash but it is by design

# Task 3

**Challenge:**

Try to solve **scenario 4** and make the application working again

**Solution:**

As this is a more complex set of objects and manifests, let’s explore them first and try to tackle as many issues as possible, before deploying them

After a brief exploration, we should have the following understanding about the files

* **pvss.yaml** is for creating **three** persistent volumes each with **different size** and **NFS** based
* **ss.yaml** is manifest of a stateful set with **four replicas** and persistent volume claims are expecting volumes with **size of 1 Gi**
* **svcssnp.yaml** is a manifest of a “public” service for the stateful set
* **svcss.yaml** is manifest of a headless service for the stateful set

Based on what we saw, we should do first some sort of alignment – number of persistent volumes and the respective claims (based on the replica count)

As the persistent volumes are infrastructure dependent, we should consider them as the leading object

So, we should shrink the replica count (in the **ss.yaml file**) to **three** (to match the persistent volumes)

Next, we should edit the persistent volumes manifest (**pvss.yaml**) and align their sizes to the expected by the stateful set – **1 Gi**

Being there, we should notice also that the access mode is not the same for all – it should become **ReadWriteOnce**

There is one more issue here. One of the volumes is pointing to another folder - **/bata/nfs/k8spvc** instead of **/data/nfs/k8spvc**. We should change this one as well

So far, we should have something working (or at least close to working)

However, we should also have some infrastructure available, before we try to deploy it:

* we need an **NFS** server reachable by all nodes under the name **nfs-server**
* there should be three exports - **/data/nfs/k8spva**, **/data/nfs/k8spvb** and **/data/nfs/k8spvc**

Once, we have done the above corrections and have the required infrastructure, we can try to deploy the first two manifests with

**kubectl apply -f pvss.yaml**

**kubectl apply -f ss.yaml**

And then check how the resource creation is going

**kubectl get pods,pv,pvc,statefulset,svc**

Everything seems to be fine

Now, lest check again the manifest for the headless service (**svcss.yaml**)

Aha, here the label selector is wrong – it is **app: factc** instead of **app: facts**

Let’s correct this and deploy it

**kubectl apply -f svcss.yaml**

And then check how the resources so far

**kubectl get pods,pv,pvc,statefulset,svc**

Everything seems to be fine. We have the headless service as well

Before we continue, let’s see if the pods are registered with the headless service

**kubectl describe service facts**

Yes, they are listed there

Last step is to check again the public service (**svcssnp.yaml**) and deploy it if it fine

Hm, now checking it again, it seems that there are issues here

For example, the label selector seems to be wrong – it is **app: fact** instead of **app: facts**

We should correct this at once

There something else, the service type is set to **ClusterIP** and yet, two lines bellow, a **nodePort** parameter is set

We should either make it a **NodePort** or **ClusterIP** service

As this is the so-called public service, perhaps we would like to be able to reach it

So, the **NodePort** seems to be the right choice

Apply this correction as well and deploy the manifest

**kubectl apply -f svcssnp.yaml**

Check that it also appears amongst the other resources

**kubectl get pods,pv,pvc,statefulset,svc**

And then check if the pods are registered with it

**kubectl describe service factsnp**

Yes, they are in the list

Now, open a browser tab and navigate to http://<cluster-ip>:30001

Finally, our application is working! Yay 😊