Kubernetes in Depth
Lab 2
Persisting data with volumes

1 References

The official reference for Kubernetes volumes:
https://kubernetes.io/docs/concepts/storage/volumes/

Additional references
https://spacelift.io/blog/kubernetes-persistent-volumes

2 Commands covered

3 Lab setup

The root folders for all the various projects here can be found in the `Lab 2` subfolder in the main `labcode` folder of your downloaded zip for this workshop.

4 Configuring a basic app with Docker Compose and Dockerfile

The root folder for this project is: `kub-data-01-starting-setup`
Open some Powershell terminals in this folder to work with it.

We start off with a very simple Node.js API application, consisting of a single `app.js` file. In this file, we create a Node server that listens on port 3000 and defines two endpoints. We handle GET requests to the `/story` route and POST requests to the same route.

For the POST request handler, we extract some text from the incoming request, check if it is empty, and, if it is not empty, append the text to a file located in the specified file path. This file path points to a folder named story, where the text.txt file resides. The text sent by the user in the request is added to this file. For the GET request handler, we read the contents of this file and return the data as the response. This simple application writes data to the `text.txt` file inside the `story` folder, which is initially empty.

We have a Dockerfile to build the Node.js application image and a `docker-compose.yaml` file for building and running the image. The application uses volumes to ensure data in the `story` folder survives container removal and restarts.

Start up the containers specified in the Docker Compose file, with appropriate flags:

```
docker compose up -d --build
```

The YAML specifies a port mapping to the localhost port of 8080. If this port is occupied, you will encounter an error while attempting to start up the container, in which case you should change the port mapping in the YAML:

```
ports:      - localhostport:3000
```

to a different port number that is free on your machine. You can check whether a particular port `xxx` is free from the listing of ports used and the processes that are bound to them with:

```
netstat -aon | findstr xxx
```

Check that the services are up with:

```
docker compose ps
```

Use Postman to send a GET request to:

http://localhost:8080/story

which will return an empty string for `story`

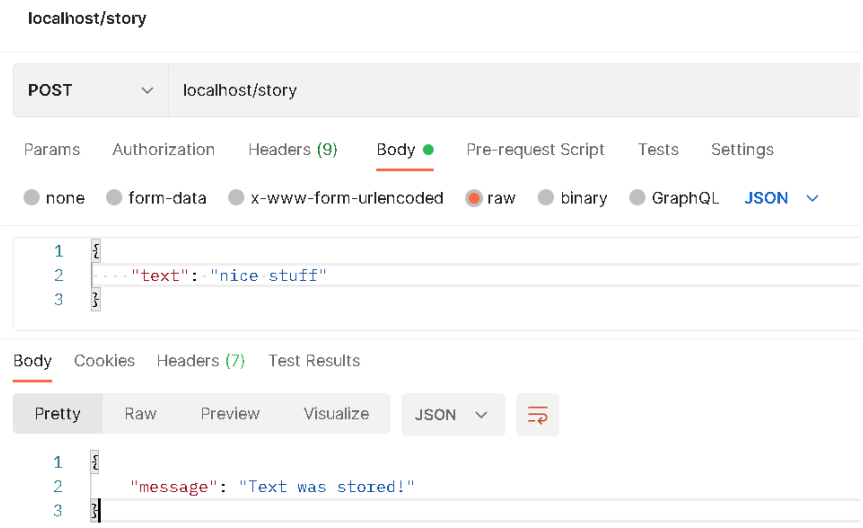
```
{"story":""}
```

If you send a POST request to the same URL with some raw JSON as its body content:

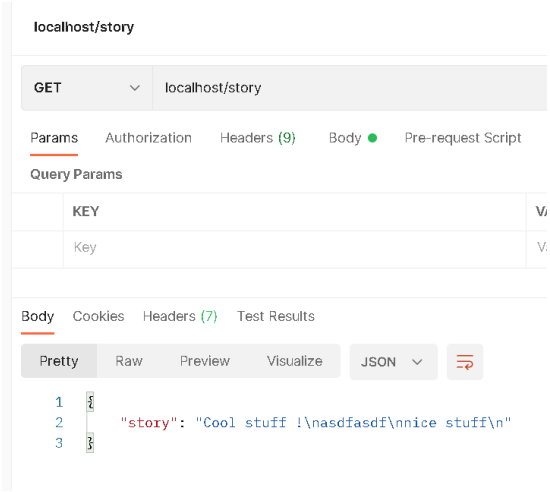
```
{
```

```
    "text" : "nice stuff"
}
```

You should get back a JSON response as follows:



You can repeatedly keep sending POST messages with random text content for the `text` property in the JSON content, and subsequently issuing a GET to the same URL will then return a concatenated string of all these messages.



Since we are using a volume in the Docker Compose for the container, we can remove the container and then regenerate it again with:

```
docker compose down
```

```
docker compose up -d
```

And if we issue another GET request to the same URL: http://localhost:8080/story

We should get back the same response that we got back earlier, since the relevant data (`/app/story/text.txt`) was saved to the named volume: `stories`

You can check that this volume specified in the the Docker Compose YAML was created at the same time when the container was started earlier with:

```
docker volume ls
```

Finally stop the container with:

```
docker compose down
```

5 Creating a deployment with a configuration manifest

Make sure you don't have any existing deployments in Kubernetes from the previous lab session, and if you do, delete them:

```
kubectl get deployments
```

```
kubectl delete deployment xxxx
```

Create two configuration manifest files for a deployment and a service

`deployment.yaml`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: story-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: story
  template:
    metadata:
      labels:
        app: story
    spec:
      containers:
        - name: story
          image: dockerhubaccount/kub-data-demo
```

`service.yaml`

```yaml
apiVersion: v1
kind: Service
metadata:
  name: story-service
spec:
  selector:
    app: story
  type: LoadBalancer
  ports:
    - protocol: "TCP"
      port: 80
      targetPort: 3000
```

Create repo in DockerHub based on the image name specified in `deployment.yaml`

Build the current Dockerfile in the current directory:

```
docker build -t dockerhubaccount/kub-data-demo .
```

Make sure you logged into your DockerHub account; if not, login with:

```
docker login
```

Then push it to create a new repo in your DockerHub account:

```
docker push dockerhubaccount/kub-data-demo
```

We can now apply both the configuration manifest files that we created earlier:

```
kubectl apply -f service.yaml -f deployment.yaml
```

Check that the deployment and its associated service is running:

```
kubectl get deployments

kubectl get services
```

Note here that we are specified a LoadBalancer type for the service. This type typically creates an external load balancer in supported cloud providers (e.g., AWS, GCP, Azure) to route traffic to the service. If the service is running in a local environment, the EXTERNAL-IP might remain <pending> as shown here. Here, <pending> indicates that the load balancer has not yet been assigned an external IP address, which is the case for a local cluster such as Minikube, where external load balancers are not supported.

We can access the service in the same way that we have done before:

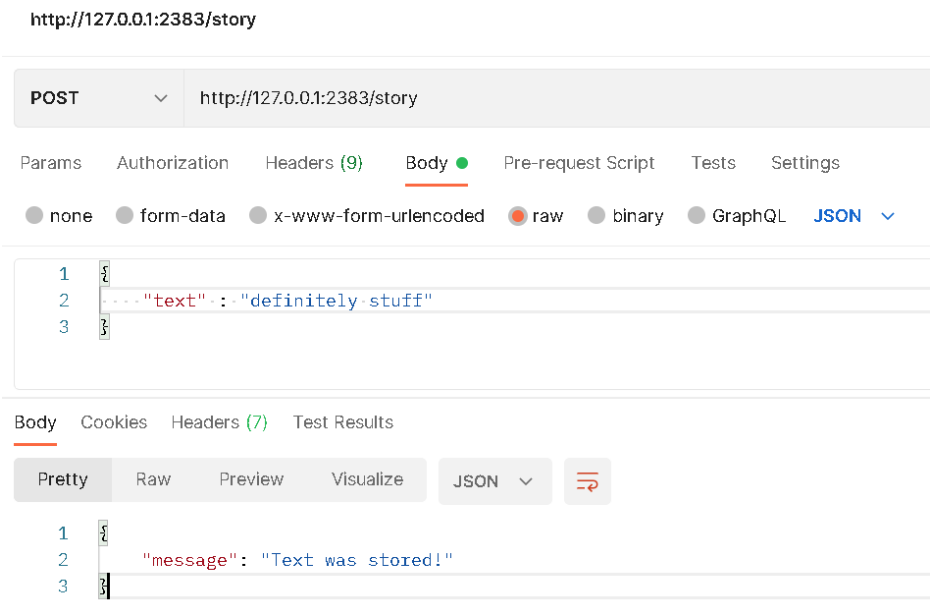
```
minikube service story-service
```

As we have already seen, Minikube provides a specific way to access the port on the NodePort: it creates a tunnel from a port on localhost to the NodePort. This localhost port is randomly assigned from the range of free ports on the local machine. You can then use the loopback address (`127.0.0.1`) or `localhost` at this tunnelling port to forward HTTP traffic to `story-deployment-app`, which automatically is accessed through a browser tab when you use the `minikube service` command.

You will then need to add the path: `/story` to the URL generated by the service to access the API endpoint of the container in the pod, for e.g.

http://127.0.0.1:xxxx/story

You can also use Postman to send POST requests to store random JSON content at that particular URL / port number at the same API endpoint and GET request to get the latest content stored as we had done previously:



And a subsequent GET request will just return a concatenated string of all text messages sent in POST requests so far.

6 Using the emptyDir ephemeral volume

Kubernetes supports many types of volumes. A Pod can use any number of volume types simultaneously. Ephemeral volume types have a lifetime of a pod, but persistent volumes exist beyond the lifetime of a pod. When a pod ceases to exist, Kubernetes destroys ephemeral volumes; however, Kubernetes does not destroy persistent volumes. However, for any kind of volume in a given pod, data is preserved across container restarts: this is true for both ephemeral and persistent volumes.

Kubernetes supports several different kinds of ephemeral volumes for different purposes:
https://kubernetes.io/docs/concepts/storage/ephemeral-volumes/#types-of-ephemeral-volumes

Add in some code to `app.js` to implement an API endpoint whose execution will cause the app to crash.

```
……

app.get('/error', () => {
  process.exit(1);
  });

app.listen(3000);
```

Build the image again with a new tag to indicate a new version:

`docker build -t dockerhubaccount/kub-data-demo:v1 .`

Then push it again to your DockerHub account with:

`docker push dockerhubaccount/kub-data-demo:v1`

Then change `deployment.yaml` to use this latest updated image:

```
…..
…..
    spec:
      containers:
        - name: story
          image: dockerhubaccount/kub-data-demo:v1
```

Then apply the changes with:

`kubectl apply -f deployment.yaml`

Check the pod associated with the deployment and get more details on it with:

`kubectl get pods`

You may need to wait for the previously running pod to be terminated and a new pod to be created based on the updated image

`kubectl describe pod story-deployment`

Check that you can still use GET and POST requests to the API endpoint of the container in the pod via the tunnelling URL /port generated from the previous `minikube service` command, for e.g. http://localhost:xxxx/story

and you can retrieve all the previous content that you stored

If this is not possible, this is likely to be an issue with the service used to provide access to the container port within the pod. You will thus need to stop the previous service with

```
kubectl delete service story-service
```

You may need to use Ctrl-C at the terminal where the tunnel for the service is being enacted.

And then restart it with:

```
kubectl apply -f service.yaml
```

And access it via

```
minikube service story-service
```

Now try and crash the container in the pod by sending a GET request to the API endpoint modification we specified earlier using Postman:

http://127.0.0.1:xxxx/error

If you periodically check the pod status now with:

```
kubectl get pods
```

You will see an initial status of `error` followed by eventual transition back to `running` as Kubernetes restores the pod (due to the requirement of `replicas : 1` in `deployment.yaml`)

Once the pod is back up and running, if you send a GET request to the `/story` API endpoint of the container in the pod via the same service URL, for e.g.
http://127.0.0.1:xxxx/story

You will notice that all previously persisted data is now lost, as the container was removed and restarted.

Make further changes to `deployment.yaml` to add a Kubernetes volume that is mounted to the internal directory of the container where app state changes need to be persisted (in this case `/app/story`, based on code in `app.js` and also in the specification of `docker-compose.yaml`

`deployment.yaml`

```
    spec:
      containers:
        - name: story
          image: dockerhubaccount/kub-data-demo:v1
          volumeMounts:
            - mountPath: /app/story
              name: story-volume
      volumes:
```

```
        - name: story-volume
          emptyDir: {}
```

There are many volume types in Kubernetes, here we are using the `emptydir`
https://kubernetes.io/docs/concepts/storage/volumes/#emptydir

Now we apply again the YAML file:

```
kubectl apply -f deployment.yaml
```

Check the pod associated with the deployment and get more details on it with:

```
kubectl get pods
```

Once the pod is back up and running, send a GET request to the `/story` API endpoint of the container in the pod via the same service URL, for e.g.
http://127.0.0.1:xxxx/story

Note that for the first GET access to the API endpoint after the new pod associated with the new deployment configuration is up and running, you initially get an error with regards to `failed to open a file`, because the code in `app.js` attempts to access `text.txt`, which in this case does not exist because the existing `text.txt` in the `/app/story` folder (which is already included in the custom image through the initial Docker build) is overwritten by the empty mounted Kubernetes volume that we mapped to this folder.

However, you should still be able to send some POST requests to store new data in this file, and subsequently you can retrieve this data via a GET request as we have done previously.

Next, try and crash the container in the pod by sending a GET request to the API endpoint we specified earlier:

http://127.0.0.1:xxxx/error

If you periodically check the pod status now with:

```
kubectl get pods
```

You will see an initial status of `error` followed by eventual transition back to `running` as Kubernetes restores the pod (due to the requirement of `replicas : 1` in `deployment.yaml`)

Once the pod is back up and running, if you send a GET request to the `/story` API endpoint of the container in the pod via the same service URL, for e.g.
http://127.0.0.1:xxxx/story

You will be able to get back all the data you had stored previously.

This demonstrates that the Kubernetes `emptyDir` volume persists across container removals and restarts within a pod. This is in fact true for all volumes, regardless of whether they are ephemeral or persistent.

6.1 Behavior of service LoadBalancer routing and emptyDir

This `emptyDir` has a drawback when it comes to multiple pod replicas. The `emptyDir` volume is created per pod and exists only while the pod is running. Each replica is an independent pod, and Kubernetes will create a unique `emptyDir` volume for each pod instance. This design ensures that data written to an `emptyDir` volume in one pod isn't accessible from another pod, which is expected behavior.

These volumes are ephemeral, meaning the data in each `emptyDir` is tied to the lifecycle of its pod. If a pod is deleted or restarted, its associated `emptyDir` volume and all its data are lost, but this does not affect other replicas' volumes.

Change `deployment.yaml` to maintain more than 1 replica, for e.g. 3 replicas

```
…..

metadata:
  name: story-deployment
spec:
  replicas: 3
  selector:

….
```

Apply it again with:

```
kubectl apply -f deployment.yaml
```

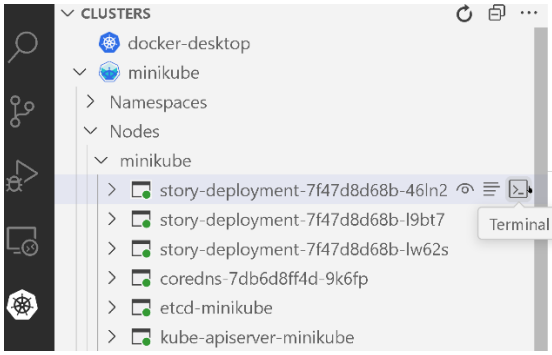
Check that you now have all 3 replica pods running with:

```
kubectl get pods
```

You can open a terminal shell into all of these 3 replica pods through the Powershell terminal by using the exact name of the pod through the command below

```
kubectl exec -it story-deployment-xxxx-yyyy -- /bin/sh
```

Alternatively, if you have the Kubernetes extension installed in VS Code, you can also open a terminal there through the terminal icons for that pod entry in the listing of pods for the Minikube node.



The shell will open in the `/app` directory of the container (as this is the `WORKDIR` designated in the Dockerfile used to generate the image of the single container running in the pod). You can then navigate to the `/app/story` subdirectory to check for the presence of a `text.txt`, and then checking its contents (if it exists) with the standard Linux `cat text.txt` command.

This is the file created to hold new content coming from POST request to the container. Only of the 3 pod replicas will have this file: the original pod replica before the additional 2 additional new replicas were created.
The other 2 pod replicas will have the `app/story` subdirectory initially empty since this folder is bound to a completely new mounted Kubernetes volume for their case. This newly mounted volume will be initially empty and therefore will overwrite the existing `text.txt` in their `/app/story` folder

Now send a GET request to the same API endpoint at the URL exposed by the tunnelling to the service

http://127.0.0.1:xxxx/story

The LoadBalancer service will randomly redirect this GET request to one of the 3 pod replicas.
If it is to the original pod replica, you will obtain back the content of `text.txt` that was there from previous POST request content sent out. If it was routed instead to any of the 2 other new pod replicas that do not have a `text.txt`, then you will obtain back the message:

```
{"message":"Failed to open file."}
```

Continue to send some consecutive POST requests with new JSON content. These will now end up being stored in `text.txt` in the `app/story` of one of the 3 existing pod replicas (either the original one or one of the 2 new replicas just created), depending on how the LoadBalancer service routes these requests.

You can verify this again by checking `/app/story` in the terminal shell in all of the 3 pod replicas and checking the contents of `text.txt` (if it exists) with the standard Linux `cat text.txt` command.

Now intentionally access the API endpoint to crash the container
http://127.0.0.1:xxxx/error

Once again, the LoadBalancer will route this request to any one of the 3 pod replicas causing that pod to crash. Check how many of the pods have gone down with:

```
kubectl get pods
```

Keep sending GET requests to the same API endpoint until you have 2 pod replicas crashed (Status: Error) and then send a GET request to the original `/story` endpoint

http://127.0.0.1:xxxx/story

This time, you are likely to get back content that is different from previously because there is now only one pod replica still running and this is likely to be different from the pod replica that the LoadBalancer was previously routing requests to.

To summarize the results of what we have seen so far:

- The `emptyDir` volume is created per pod. Each replica is an independent pod, and Kubernetes will create a unique `emptyDir` volume for each pod instance.
- Each `emptyDir` volume for each pod is different from the others, and they do not affect each other. This ensures that data written to an `emptyDir` volume in one pod isn't accessible from another pod.
- The behavior of a LoadBalancer service is that it routes all incoming HTTP requests to it (regardless of method type: GET, POST, PUT etc) to one of the available pod replicas that it exposes using a randomized round-robin load balancing. However, successive HTTP requests can end up being routed to the same pod replica, depending on a variety of external situations (such as internal DNS configuration for the pod replicas, etc)
- The Service will not replicate or broadcast the same request to all pods. Each pod handles its own incoming requests

If your POST requests need to store content and consistency is important (i.e. all the POST content should end up in the same storage or volume), then the use of emptyDir volume and non-deterministic routing of the LoadBalancer service will cause problems.

7 Using the hostpath persistent volume

A simple and quick work around the issue mentioned previously is to use the `hostpath` volume instead. A `hostPath` volume mounts a file or directory from the host node's filesystem into a Pod.
https://kubernetes.io/docs/concepts/storage/volumes/#hostpath

It is the most basic form of persistent volume to be demonstrated for a single node Kubernetes cluster such as Minikube
https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes

Configure `deployment.yaml` to reflect this:

```
…
…
…
      volumes:
        - name: story-volume
          hostPath:
            path: "some directory on host node"
```

```
            type: DirectoryOrCreate
```

The `hostPath` value should be an actual directory on the worker node in the Kubernetes cluster hosting the pod and is conceptually similar to bind mounts in Docker storage system.

For a Kubernetes cluster involving a Minikube node that was started using a Docker driver, i.e.:
`minikube start --driver=docker`
the `hostPath` value specifies a path on the local file system of the Minikube node (which in this case is the host machine running the node that hosts the pod)
https://minikube.sigs.k8s.io/docs/handbook/persistent_volumes/

A suitable value in this case (where Minikube is being used) might be a subdirectory in the top level `/data` directory, for e.g.

```
            path: /data/tempstorage
```

We will use this value for this lab, so place it into your `deployment.yaml`

The `type` value defines how Kubernetes handles the specified directory. The option `DirectoryOrCreate` means that if the `/data/tempstorage` directory does not exist, it will be created as an empty directory.

To navigate the local file system on the Minikube node, you can open a SSH connection to the Minikube node with SSH in a Powershell terminal:

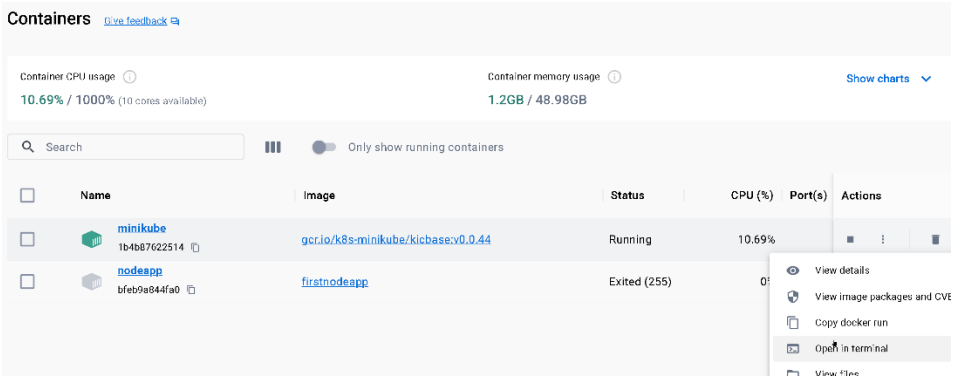
```
minikube ssh
```

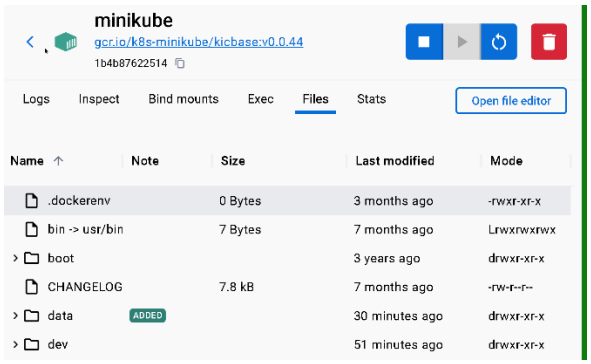
You will enter the shell under the docker user account. To elevate yourself to root, you can type:

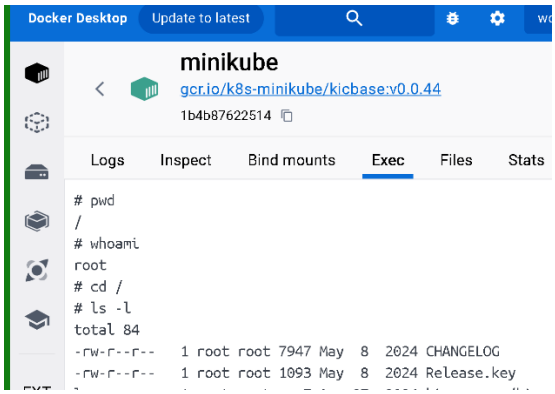
```
sudo -i
```

You can then navigate and examine the local file system using the standard Linux commands, such as `cd` and `ls`.
Verify that there is a top level `/data` directory which you can use for the `hostpath` value.

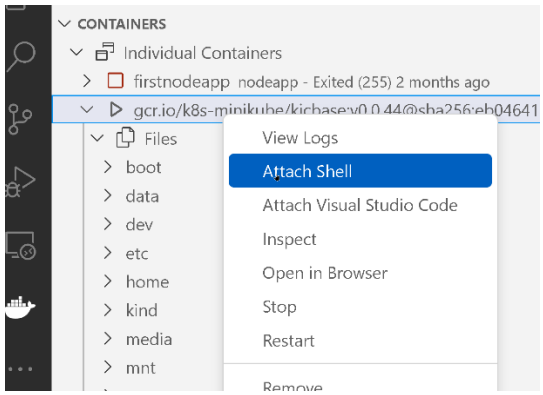
If you have Docker Desktop installed and you are running Minikube with Docker as the driver, the Minikube node will be visible as a running container in the Containers section of the main dashboard UI. Clicking on the ellipsis menu gives you list of options, including viewing the file system as well as opening a terminal in the container.







Alternatively, if you have the Docker extension installed on VS Code, you can also use the UI for this extension to view the local file system as well as open a terminal into the container.



NOTE: The file system browsers on the Docker extension of VS Code as well as Docker Desktop UI may sometimes not reflect the latest file system changes, so its best to double check from the CLI through standard Linux commands like `ls -l` or `cat` to examine directory or file contents.

When you start Minikube with the Docker driver, Minikube creates a Docker container that acts as a single-node Kubernetes cluster. In this configuration, the outer Docker daemon on your host system manages the Minikube container, while the inner Kubernetes system within that container manages the pods as Docker containers.

In the shell terminal inside the Minikube node, you can type: `docker ps`
and view all the other containers that are available which implement the various key components and workload objects in the Kubernetes architecture; such as pods, coredns, etcd, kube-apiserver, kube-controller, etc.

SIDE NOTE: If you are using Docker Desktop (installed via WSL) to run the Kubernetes cluster (instead of Minikube), then `hostPath` can reference an actual path on you're the Windows host file system:
https://stackoverflow.com/questions/71018631/kubernetes-on-docker-for-windows-persistent-volume-with-hostpath-gives-operatio

https://stackoverflow.com/questions/54073794/kubernetes-persistent-volume-on-docker-desktop-windows

In the example above, the `path` value needs to be specified in a certain way, for e.g. if the bind mount path on the Windows host is `C:\workshoplabs\storage`, then the `path` value in `deployment.yaml` should be as follows:

```
path: /run/desktop/mnt/host/c/workshoplabs /storage
```

Once you have decided the appropriate correct value for `hostPath.path`, (whether a Windows host directory or a directory on the Minikube local file system), you can again the configuration for the deployment of the 3 pod replicas with:

```
kubectl apply -f deployment.yaml
```

After a while, check that you now have 3 new running replica pods (with corresponding new names) and the previous 3 pods have been terminated

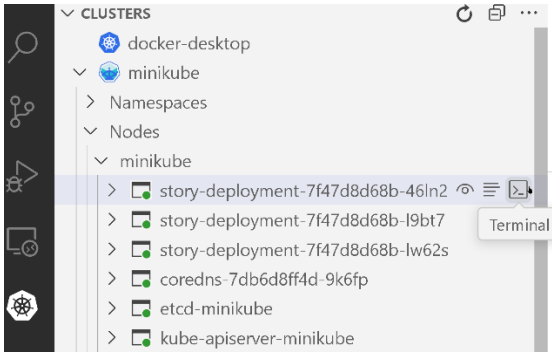
```
kubectl get pods
```

With the application of the new deployment, the new `hostPath` volume will be bound to `/app/story` in the containers of all the 3 newly started pods, erasing any previous content there.

Just as before, you can verify this by opening a terminal shell into all of these 3 replica pods through the Powershell terminal by using the exact name of the pod through the command below

```
kubectl exec -it story-deployment-xxxx-yyyy -- /bin/sh
```

Alternatively, if you have the Kubernetes extension installed in VS Code, you can also open a terminal there through the terminal icons for that pod entry in the listing of pods for the Minikube node.



The shell will open in the `/app` directory of the container (as this is the `WORKDIR` designated in the Dockerfile used to generate the image of the single container running in the pod). You can then navigate to the `/app/story` subdirectory and this folder should currently be empty for all 3 new replica pods.

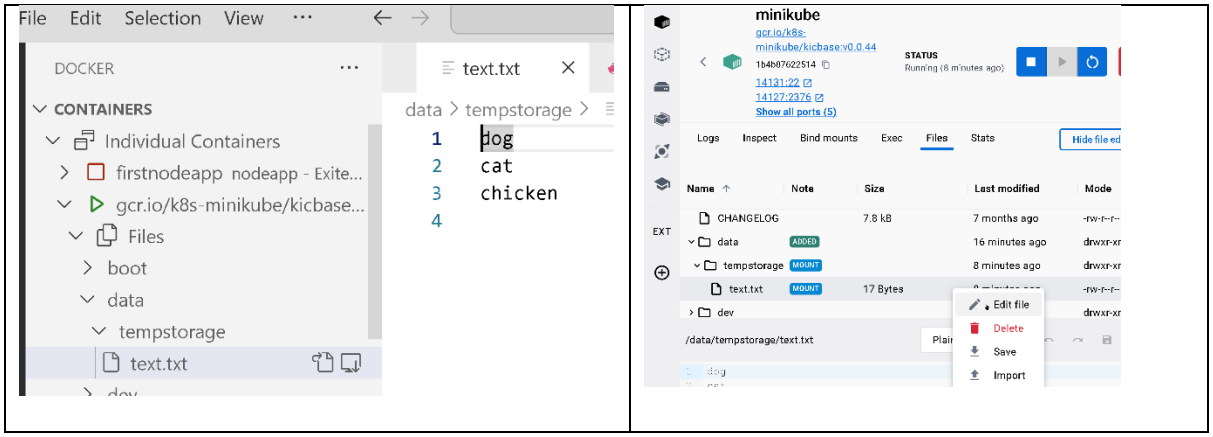
Now send a GET request to the previous API endpoint at the URL exposed by the service, http://127.0.0.1:xxxx/story

As expected, you will get back this message since `/app/story` in the containers of all the 3 newly started pods is empty.

```
{"message":"Failed to open file."}
```

Now store some persistent content through appropriate POST messages, and verify that all these content has been stored through a GET request through your REST client.

Check the content of `/data/tempstorage` directory (the directory mapped to our `hostPath` volume in our example) on the Minikube single node using any one the methods we described earlier: minikube ssh, Docker Desktop UI functionality or VS Code Docker extension. Verify that there is `text.txt` containing all the content you had sent earlier using your POST requests.



NOTE: The file system browsers on the Docker extension of VS Code as well as Docker Desktop UI may sometimes not reflect the latest file system changes, so its best to double check from the CLI through standard Linux commands like `ls -l` or `cat` to examine directory or file contents.

Back in the shell terminals of the 3 pod replicas, verify that there is also a `text.txt` in all of their containers in the `/app/story` folder, and that this file also contains exactly the same content as the one in the `/data/tempstorage` (the directory mapped to the `hostPath` volume)

Send a GET request to the previous API endpoint at the URL exposed by the service:
http://127.0.0.1:xxxx/story

and verify that you are able to retrieve the same content.

Now send multiple HTTP requests to crash the containers in the pods using Postman:

http://127.0.0.1:xxxx/error

until all or at least 2 of the pods are down. Check with:

```
kubectl get pods
```

Finally, verify that you will be able to get back the previously stored data with a GET request to:

http://127.0.0.1:xxxx/story

7.1 Mounting Minikube hostPath directory to local Windows file system

You can further extend on what we have done so far by mapping the local file path within the Minikube node to a path on the local Windows host system with a command in this format:

```
minikube mount Windows-path:path-on-Minikube-node
```

This establishes a bind mount between both paths.
https://minikube.sigs.k8s.io/docs/handbook/mount/

To perform this, you will need to delete the current deployment first:

```
kubectl delete deployment story-deployment
```

Create a new directory on your Windows local host with any name of your choice. You will use this directory to bind to the `/data/tempstorage` path in the Minikube node file system.

Navigate to this new directory in a Powershell terminal and create a process that will perform this bind mount with:

```
minikube mount ${pwd}:/data/tempstorage
```

A successful mounting operation will display the relevant messages and lock at this point.

In a separate Powershell terminal in the original project folder, create the deployment again with:

```
kubectl apply -f deployment.yaml
```

Store some data into the app via the usual POST requests to the same API endpoint and verify that they are stored correctly with a corresponding GET request.

Back in Windows, using File Explorer, check the content of the new directory. You should see a file `text.txt` there, which was written into the corresponding folder `/data/tempstorage` in the Minikube node. This file should contain all the content that you sent in your POST requests to the app in the container.

Create a random file `mystuff.txt` in this new directory using File Explorer and populate it with some random content.

Connect back to the Minikube node and verify that this file has been synchronized correctly to `/data/tempstorage` with exactly the same random content that you populated it with

To terminate the mount, press Ctrl-C in the Powershell terminal where you typed the `minikube mount` command.

In general, we will not use a `hostPath` persistent volume due to the security issues and also potential inconsistencies.

- In multi-node clusters, pods scheduled on different nodes won't have access to the same hostPath, leading to inconsistencies.
- Using hostPath volumes can expose sensitive parts of the host filesystem to pods, potentially leading to security vulnerabilities. It's crucial to ensure that pods using hostPath volumes are trusted and that the hostPaths are carefully managed to prevent unauthorized access.

However, hostPath volumes can be a quick work around which are suitable primarily for testing data persistence of apps on single-node clusters.

8 Defining and applying a persistent volume (PV) and persistent volume claim (PVC)

The conventional and widely accepted approach for persistent storage in Kubernetes in real life applications is through Persistent Volumes (PV) and Persistent Volume Claims (PVC).

A Persistent Volume (PV) can been provisioned by an administrator or dynamically provisioned using a StorageClass. PVs are a cluster resource, similar to a node or namespace, and they exist independently of the Pod lifecycle, meaning they can persist data even after the Pod using them is deleted.

A Persistent Volume Claim (PVC) is a request for storage by a Pod. It abstracts the details of the underlying PV, enabling Pods to request storage without needing to know its specifics. The PVC defines the storage requirements, such as size, access modes, and optionally, the StorageClass.

Key Characteristics:

a) Dynamic Binding: When a PVC is created, Kubernetes automatically binds it to an available PV that satisfies the PVC's requirements.
b) Portability: PVCs allow developers to request storage without being tied to a specific storage backend, improving portability.
c) Lifecycle Integration: PVCs are bound to a specific Pod or application but are not tied to the Pod's lifecycle. Data persists even if the Pod is deleted, as long as the PVC is not removed.

Create a new configuration manifest file to define a PersistentVolume (PV)

`host-pv.yaml`

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: host-pv
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem
  storageClassName: standard
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /pvdata
    type: DirectoryOrCreate
```

Some of the fields here are also found in the `deployment.yaml` and have the same meaning. Other key elements here are:

`hostPath` - creates a standalone HostPath volume detached from any pod. However, it remains tied to the node it resides on as we are still working within the Minikube single node Kubernetes cluster.
`path` - the directory on the host node to use as the storage location. When a pod uses this PV, one of its local directories will be bound to this directory on the node where the pod is scheduled. Here,

we are using `/pvdata`, which is a completely different directory from the `/data/tempstorage` that we used earlier for our normal `hostPath` volume
`volumeMode` - Two modes are supported: Filesystem and Block. For this setup, we use Filesystem, as it corresponds to a folder on the host node's filesystem.
`accessModes` - This allows listing one or more supported access modes, such as ReadWriteOnce, ReadOnlyMany, or ReadWriteMany. Each mode has specific use cases:

- ReadWriteOnce: Allows read-write access by multiple pods on the same node.
- ReadOnlyMany: Allows read-only access by multiple nodes.
- ReadWriteMany: Allows read-write access by multiple nodes.

For `hostpath`, only ReadWriteOnce is supported, as the volume is node-specific.

`storageClassName` - specifies the StorageClass that a PersistentVolumeClaim (PVC) must use to bind to the PV. A StorageClass provides a way to define and manage different types of storage with specific configurations and parameters and can be configured specifically for different cloud providers

Create a configuration manifest for a PV claim (PVC) related to this PV you created earlier

`host-pvc.yaml`

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: host-pvc
spec:
  volumeName: host-pv
  accessModes:
    - ReadWriteOnce
  storageClassName: standard
  resources:
    requests:
      storage: 1Gi
```

Most of the elements here are replicated from the PV. The most important are:

`volumeName` - This explicitly binds the PersistentVolumeClaim to a specific PersistentVolume with the specified name (host-pv). If this is not specified, Kubernetes dynamically provisions or selects a matching PersistentVolume automatically
`storageClassName` – This must match what was specified in the PV that this PVC attempts to bind to. In this case, this is standard.
`storage` – The storage capacity should either match or be less than the storage specified in the PV that this PVC attempts to bind to.

We now need to make some changes to the `deployment.yaml` to connect a pod to the PVC so that it is able to reach out to the associated PV to access it.

`deployment.yaml`

```
……
……
          volumeMounts:
            - mountPath: /app/story
              name: story-volume
      volumes:
        - name: story-volume
          persistentVolumeClaim:
            claimName: host-pvc
…..
……
```

Check for the default storage class name with:

```
kubectl get storageclasses
```

We need to ensure the name associated with the default storage class (which should be `standard`) is specified correctly in `host-pv.yaml` and `host-pvc.yaml`
This should already have been done earlier, but we can double check again here.

`host-pv.yaml`

```
……

  volumeMode: Filesystem
  storageClassName: standard
  accessModes:

…….
```

`host-pvc.yaml`

```
……

  accessModes:
    - ReadWriteOnce
  storageClassName: standard
  resources:


…….
```

A StorageClass is a resource that defines a set of parameters and behaviors for dynamically provisioning PersistentVolumes (PVs). It provides a way to abstract the details of storage provisioning, enabling users to request storage without having to understand or specify the underlying storage infrastructure. This abstraction is particularly useful in dynamic environments where storage needs

can vary. StorageClasses allow the definition of specific parameters such as disk type, replication factor, or performance class (e.g., SSDs vs. HDDs), depending on the underlying storage provider (AWS EBS, Google Persistent Disk, etc)

Now we are ready to define the PV and have our existing deployment claim it via a matching PVC.

First start off by applying the persistent volume configuration manifest to create the PV:

```
kubectl apply -f host-pv.yaml
```

Check that the persistent volume has been created:

```
kubectl get pv
```

Then apply the volume claim:

```
kubectl apply -f host-pvc.yaml
```

And check for its creation with:

```
kubectl get pvc
```

```
kubectl get pv
```

The key point to note is that output for both PVC and PV will indicate that the PVC is now correctly bound to the PV, and should be able to access it. So any pods that use this PVC will be able to access the PV, in particular, the pods we will deploy next in `deployment.yaml`.

Check that the pods for the previous deployment are still active and running:

```
kubectl get pods
```

Finally apply the configuration manifest for the deployment again:

```
kubectl apply -f deployment.yaml
```

Check that the new pods are up and running first, and the previous pods are terminated:

```
kubectl get pods
```

If you check in the file system of the Minikube node using any of the methods we have seen earlier, you should see a creation of a new directory `/pvdata` corresponding to `hostPath` volume in `host-pv.yaml`.

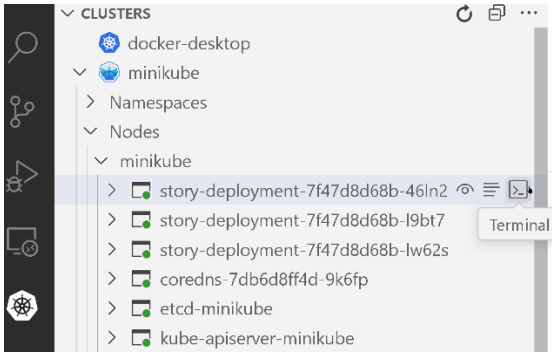
NOTE: The file system browsers on the Docker extension of VS Code as well as Docker Desktop UI may sometimes not reflect the latest file system changes, so its best to double check from the CLI through standard Linux commands like `ls -l` or `cat` to examine directory or file contents.

With the application of the PVC claim in the new deployment, the new `hostPath` volume for the corresponding PV will be bound to `/app/story` in the containers of all the 3 newly started pods, erasing any previous content there.

Just as before, you can verify this by opening a terminal shell into all of these 3 replica pods through the Powershell terminal by using the exact name of the pod through the command below

```
kubectl exec -it story-deployment-xxxx-yyyy -- /bin/sh
```

Alternatively, if you have the Kubernetes extension installed in VS Code, you can also open a terminal there through the terminal icons for that pod entry in the listing of pods for the Minikube node.



The shell will open in the `/app` directory of the container (as this is the `WORKDIR` designated in the Dockerfile used to generate the image of the single container running in the pod). You can then navigate to the `/app/story` subdirectory and this folder should currently be empty for all 3 new replica pods.

Now send a GET request to the previous API endpoint at the URL exposed by the service, http://127.0.0.1:xxxx/story

As expected, you will get back this message since `/app/story` in the containers of all the 3 newly started pods is empty.

```
{"message":"Failed to open file."}
```

Now store some persistent content through appropriate POST messages, and verify that all these content has been stored through a GET request through Postman.

Now verify that `/pvdata` on the Minikube node contains `text.txt` with this new content.

Also verify that `/app/story` in the containers of all the 3 newly started pods (which is bound to the PV linked to `/pvdata`) also contains `text.txt` with this new content

Now send multiple HTTP requests to crash the containers in the pods using Postman:

http://127.0.0.1:xxxx/error

until all or at least 2 of the pods are down. Check with:

```
kubectl get pods
```

Finally, verify that you will be able to get back the previously stored data with a GET request to:

http://127.0.0.1:xxxx/story

once at least one of the pods are back up and running again.

This was the same behavior that we saw previously: which is data is persisted between container / pod removals.

Now we will completely remove all pods and restart them again by deleting the deployment:

```
kubectl delete deployment story-deployment
```

Restart the deployment again with:

```
kubectl apply -f deployment.yaml
```

Once all 3 pods in the deployment are up and running, verify that `/app/story` in the containers of all the 3 newly started pods (which is bound to the PV linked to `/pvdata`) will now have `text.txt` with the previous content

Also verify that you can retrieve this data through a GET request through your REST client to the original API endpoint:
http://127.0.0.1:xxxx/story

For a multi-node Kubernetes cluster (which is not applicable here for Minikube that is a single node cluster), the PV volume can be further configured to be persisted across multiple pods in multiple nodes (rather than multiple pods on a single node as is the case here).

One common solution for that use case is to use Network File System (NFS) or a cloud provider's persistent storage (e.g., AWS EFS, GCP Filestore). We will see this in a later lab.

9 Incorporating environment variables into manifests

Environment variables play a crucial role in application configuration and management, offering a dynamic and efficient way to provide settings and sensitive information (like secrets) to applications running in Kubernetes clusters. Their key use cases are:

a) Environment variables allow applications to adapt to different environments (development, testing, production) without modifying the application code by providing a flexible way to inject configuration details that can vary based on the deployment context.
b) Sensitive information like API keys, database credentials, and tokens can be securely injected using environment variables.
c) Changing configurations via environment variables (e.g., updating database connection strings) avoids the need to rebuild and redeploy applications.

We can incorporate environment variables into a configuration manifest, the same that we can in a Dockerfile or Docker Compose file.

Replace the hardcoded directory name in `app.js` with a value obtained from an environment variable:

```
……

const app = express();

const filePath = path.join(__dirname, process.env.STORY_FOLDER, 'text.txt');

app.use(bodyParser.json());

……
```

Build a new image that incorporate this code change with a new tag as well:

```
docker build -t dockerhubname/kub-data-demo:v2 .
```

and when this is completed, push it up to your DockerHub account:

```
docker push dockerhubname/kub-data-demo:v2
```

Then modify `deployment.yaml` to provide a value for this environment variable and specify the updated image to pull when creating the container. Here we use a different value for this folder to store the `text.txt` file.

```
…..

      containers:
        - name: story
          image: dockerhubaccount/kub-data-demo:v2
          env:
            - name: STORY_FOLDER
              value: 'newstory'
          volumeMounts:
            - mountPath: /app/newstory
              name: story-volume


……
```

Finally apply it with:

```
kubectl apply -f deployment.yaml
```

Check for all the new pods that are created to be running and the old ones to be terminated:

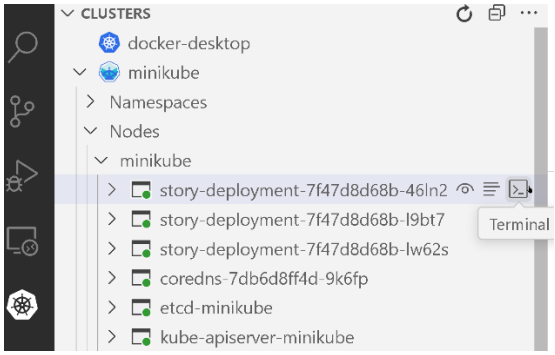
```
kubectl get pods
```

As our deployment already is bound to a PV at `/pvdata` via the existing PVC, the file `text.txt` will now be restored to the bound folder in the file system of the 3 new replica pods, except this time this bound folder will incorporate the value from the environment variable, making it: `/app/newstory`

Just as before, you can verify this by opening a terminal shell into all of these 3 replica pods through the Powershell terminal by using the exact name of the pod through the command below

```
kubectl exec -it story-deployment-xxxx-yyyy -- /bin/sh
```

Alternatively, if you have the Kubernetes extension installed in VS Code, you can also open a terminal there through the terminal icons for that pod entry in the listing of pods for the Minikube node.



The shell will open in the `/app` directory of the container (as this is the `WORKDIR` designated in the Dockerfile used to generate the image of the single container running in the pod). You can then navigate to the `/app/newstory` subdirectory verify that there is `text.txt` present in the PV `/pvdata` on the Minikube node.

Add some new content through appropriate POST messages, and verify that the contents of `text.txt` in the `/app/newstory` subdirectory of all the 3 pod replicas as well as in the PV `/pvdata` on the Minikube node are updated as well, as expected.

9.1 Using ConfigMap to specify environment variables

In Kubernetes, a ConfigMap is a way to manage configuration data as key-value pairs, separate from the application code. It can be used as an alternative to configure environment variables for applications running within containers to access configuration values.

Add a new file that will serve as the configuration manifest for a ConfigMap

`environment.yaml`

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: data-store-env
data:
  folder: 'newstory'
```

```
  # more additional key: value pairs
```

Now apply it with:

```
kubectl apply -f environment.yaml
```

Check that it is created with:

```
kubectl get configmaps
```

Modify `deployment.yaml` to use this ConfigMap

```
.....

          env:
            - name: STORY_FOLDER
              valueFrom:
                configMapKeyRef:
                  name: data-store-env
                  key: folder
                  # returns value: 'newstory'
          volumeMounts:
            - mountPath: /app/newstory
              name: story-volume

......
```

Now apply the latest changes with:

```
kubectl apply -f deployment.yaml
```

Check for all the new pods that are created to be running and the old ones to be terminated:

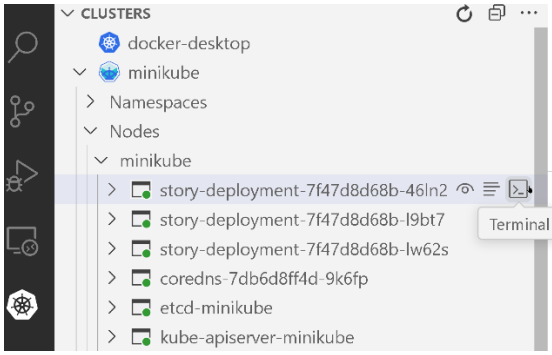
```
kubectl get pods
```

Finally, double check that everything is still working exact as before.

Open a terminal shell into all of these 3 replica pods through the Powershell terminal by using the exact name of the pod through the command below

```
kubectl exec -it story-deployment-xxxx-yyyy -- /bin/sh
```

Alternatively, if you have the Kubernetes extension installed in VS Code, you can also open a terminal there through the terminal icons for that pod entry in the listing of pods for the Minikube node.



The shell will open in the `/app` directory of the container (as this is the `WORKDIR` designated in the Dockerfile used to generate the image of the single container running in the pod). You can then navigate to the `/app/newstory` subdirectory verify that there is `text.txt` present in the PV `/pvdata` on the Minikube node.

Add some new content through appropriate POST messages, and verify that the contents of `text.txt` in the `/app/newstory` subdirectory of all the 3 pod replicas as well as in the PV `/pvdata` on the Minikube node are updated as well, as expected.

Now that we are done with this lab session, we can remove the single running deployment and the pods associated with it:

```
kubectl delete deployment story-deployment
```

We can also stop the Minikube tunnelling for the service that exposes our deployment by pressing Ctrl-C and also deleting the service concerned.

```
kubectl delete service story-service
```

10 END