Kubernetes Lab 2
Persisting data with volumes

1 References

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

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 Setting up configuration manifest for demo app

The root folder for this project is: `kub-data-01-starting-setup`

We start first by verifying app functionality in Docker first.
Start up the containers specified in the Docker Compose file, with appropriate flags:

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

Check that the services are up with

```
docker compose ps
```

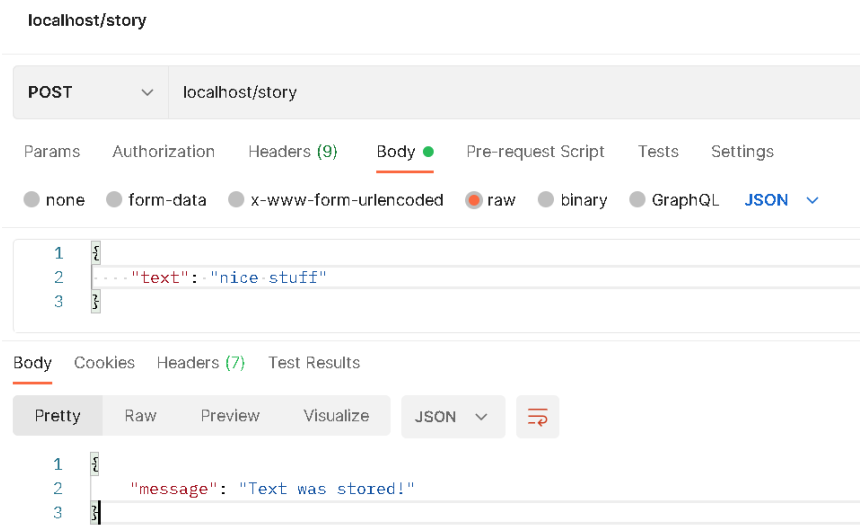
Use browser or REST client (Postman is the best) to send request to:

http://localhost/story

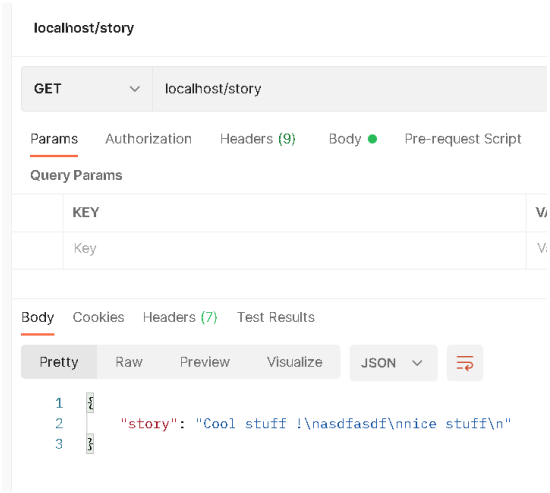
which will return an empty string for `story`

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

If you send a POST request to the same URL with some raw JSON as its body content, you should get back a JSON response as follows:



You can repeatedly keep sending POST messages with text content and 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
```

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

Finally stop it again with:

```
docker compose down
```

Make sure you don't have any existing deployments in Kubernetes and delete any running ones:

```
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 .
```

Then push it the repo you created earlier:

```
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 is running:

```
kubectl get deployments
```

And access it via the associated service

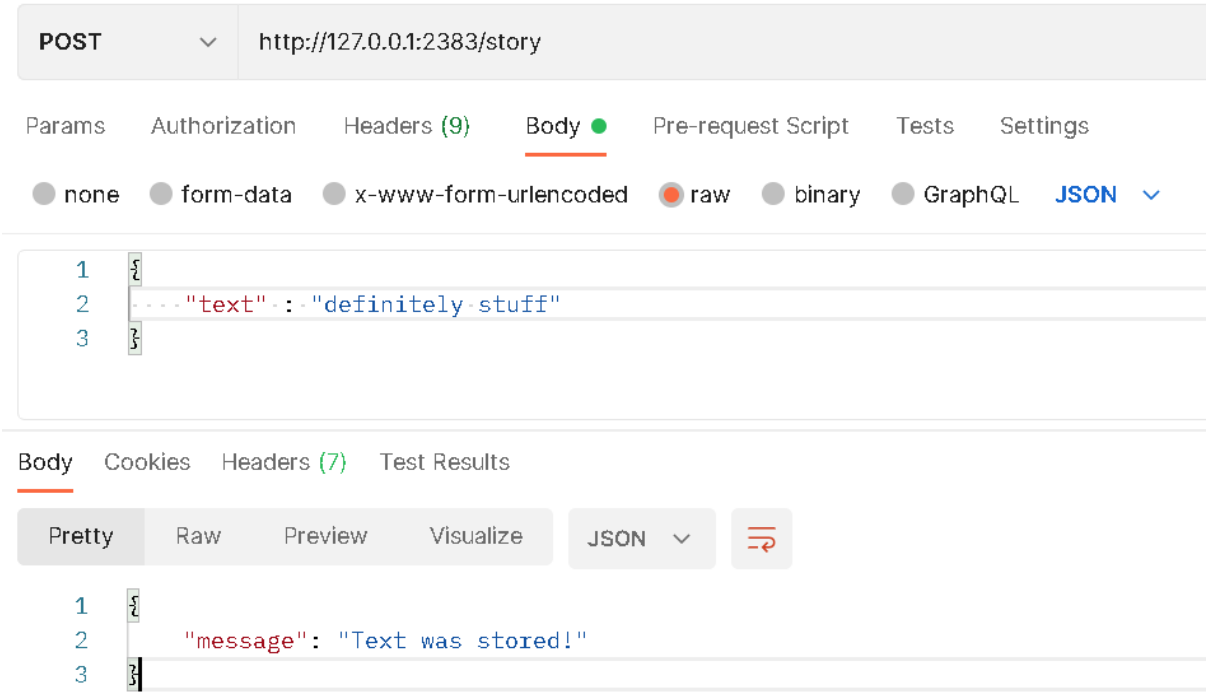
```
minikube service story-service
```

Which returns a tunnel for the story-service. 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://127.0.0.1:2383/story

You can also use Postman to send a POST request to story data at that particular URL / port number at the same API endpoint as we had done previously:

http://127.0.0.1:2383/story



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

5 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

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 service URL generated from the previous lab, for e.g.
http://127.0.0.1:5867/story
and the actions you can perform are still exactly the same.

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 we specified earlier:

http://127.0.0.1:5867/error

Its best to use the REST client to send this GET request to this API endpoint, as we want to guarantee that exactly only one HTTP request is send in order to just crash one of the 2 currently running pod replicas. If you try to send the HTTP request to this API endpoint through a browser tab, you may end up crashing both the pods (and thus nullifying the demo) since the browser will automatically periodically resend HTTP requests to this API endpoint when it fails to connect (thereby potentially causing both pods to crash).

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`)

Now if you send a GET request back the API endpoint of the container in the pod via the service URL generated from the previous lab, for e.g.
http://127.0.0.1:5867/story

You will be able to access it but 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 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`

```
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
```

Now access the app via the URL / port number exposed by the service. 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://127.0.0.1:5867/error

Its best to use the REST client to send this GET request to this API endpoint, as we want to guarantee that exactly only one HTTP request is send in order to just crash the currently running pod replica

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`)

Now if you send a GET request back the API endpoint of the container in the pod via the service URL generated from the previous lab, for e.g.
http://127.0.0.1:5867/story

You will be able to get back all the data you had stored previously, proving 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 hostpath persistent volume

This `emptyDir` has a drawback when it comes to multiple replicas:

Change `deployment.yaml` to more than 1 replica

```
.....
```

```
metadata:
  name: story-deployment
spec:
  replicas: 2
  selector:

....
```

Apply it again with:

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

Check that you now have 2 replica pods with:

```
kubectl get pods
```

Now send a GET request to the same API endpoint at the URL exposed by the service,
http://127.0.0.1:5867/story
You should get back all the data stored there previously.

Now intentionally access the API endpoint to crash the container
http://127.0.0.1:5867/error

You should see that one of the pods go down, while the other keeps running:

```
kubectl get pods
```

At this point of time, sending a GET request to the original API endpoint returns:

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

This is because the request is now redirected to the only running pod, which does not have the volume associated with it and therefore does not retain any data.
However, once both pods are running (check again with `kubectl get pods`), sending a GET request to the original API endpoint will now return all our original data as it is redirected to the correct pod which returns the data.

To work around this issue, we can use the `hostpath` volume instead.
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 machine"
            type: DirectoryOrCreate
```

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

For a Kubernetes cluster involving a Minikube node that was started using a Docker driver, i.e.:
`minikube start --driver=docker`

then 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
```

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 `path`, you can again apply it with:

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

Check that you now have 2 replica pods with:

```
kubectl get pods
```

Now send a GET request to the same API endpoint at the URL exposed by the service,
http://127.0.0.1:5867/story

With the application of a new deployment, all previous data in any volumes will be erased and you will get back the previous message:

```
{"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.

Now send a HTTP request to crash a container using the REST client:

http://127.0.0.1:5867/error

and check that one of the pods is still down with:

```
kubectl get pods
```

and at the same time verify that you will be able to get back the previously stored data with a GET request to:
http://127.0.0.1:5867/story

For a Kubernetes cluster involving a Minikube node that was started using a Docker driver, i.e.:
`minikube start --driver=docker`

you can verify that the contents of `/app/story` was copied over to your specified `hostPath` value (which is a path on the local file system of the Minikube node)

Open a SSH connection to the Minikube node with SSH and navigate to this specified path:

```
minikube ssh
```

```
docker@minikube:~$ pwd
/home/docker
docker@minikube:~$ cd /data
docker@minikube:/data$ ls -l
total 4
drwxr-xr-x 2 root root 4096 Aug 11 08:14 tempstorage
docker@minikube:/data$ cd tempstorage
docker@minikube:/data/tempstorage$ ls -l
total 4
-rw-r--r-- 1 root root 20 Aug 11 08:15 text.txt
docker@minikube:/data/tempstorage$ cat text.txt
…
….
….
docker@minikube:/data/tempstorage$ docker ps
…..
….
…..
docker@minikube:/data/tempstorage$ exit
logout
```

Notice that with the `docker ps` command inside the Minikube node (which itself is running within a Docker container), you can see the various deployments, pods and other Kubernetes objects that implement the Kubernetes infrastructure running as containers.

You can further extend on this 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 get deployments

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 transferred correctly to `/data/tempstorage`

```
minikube ssh
```

```
docker@minikube:~$ cd /data/tempstorage/
docker@minikube:/data/tempstorage$ ls -l
total 1
-rw-rw-rw- 1 docker docker 55 Aug 11 09:53 mystuff.txt
-rw-rw-rw- 1 docker docker 16 Aug 11 09:50 text.txt
docker@minikube:/data/tempstorage$ cat mystuff.txt
…..
…..
docker@minikube:/data/tempstorage$ exit
logout
```

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

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: /data
    type: DirectoryOrCreate
```

Create a configuration manifest for a PV claim (PVC) related to the 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
```

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 storage classes with:

```
kubectl get storageclasses
```

We need to ensure the name associated with the default (standard) is included in `host-pv.yaml` and `host-pvc.yaml`

`host-pv.yaml`

```
……

  volumeMode: Filesystem
  storageClassName: standard
  accessModes:

…….
```

`host-pvc.yaml`

```
……

  accessModes:
    - ReadWriteOnce
  storageClassName: standard
  resources:


…….
```

First start off by applying the persistent volume configuration manifest:

```
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`.

Finally apply the configuration manifest for the deployment again:

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

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.

Now send a HTTP request to crash a container using the REST client to the specified API endpoint for this purpose:

http://127.0.0.1:5867/error

and check that one of the pods is still down with:

```
kubectl get pods
```

and at the same time verify that you will be able to get back the previously stored data with a GET request to the original API endpoint:
http://127.0.0.1:5867/story

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

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

```
kubectl get deployments

kubectl delete deployment story-deployment
```

Finally start the deployment again with:

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

Once both pods in the deployment are up and running, verify that you can retrieve the previous stored data through a GET request through your REST client to the original API endpoint:
http://127.0.0.1:5867/story

8 Incorporating environment variables into manifests

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:

```
…..

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

……
```

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
```

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

Now send a HTTP request to crash a container using the REST client to the specified API endpoint for this purpose:

http://127.0.0.1:5867/error

and check that one of the pods is still down with:

```
kubectl get pods
```

and at the same time verify that you will be able to get back the previously stored data with a GET request to the original API endpoint:
http://127.0.0.1:5867/story

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

Add a new `environment.yaml` file that will serve as the configuration manifest for a ConfigMap

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: data-store-env
data:
  folder: 'story'
  # more additional key: value pairs
```

Now apply it with:

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

Check that it is created with:

```
kubectl get configmap
```

Modify `deployment.yaml` to use this ConfigMap

```
.....

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

......
```

Now 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
```

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

Now send a HTTP request to crash a container using the REST client to the specified API endpoint for this purpose:

http://127.0.0.1:5867/error

and check that one of the pods is still down with:

```
kubectl get pods
```

and at the same time verify that you will be able to get back the previously stored data with a GET request to the original API endpoint:
http://127.0.0.1:5867/story

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

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

```
kubectl get deployments

kubectl delete deployment story-deployment
```

9 END