Week 4 - Part 2

Deploying the TodoApp in Kubernetes

High level components

Just the same as the Docker application last week:

- Spring Boot app
- MySQL Database

Properties we want

- Internet accessible in a conventional way
- Persistent storage that doesn't dissapear when we move the database
- Scalable
- Recovery from failure

All the Kubernetes objects we use in this session will be directly supporting one of these properties or behaviours.

This is going to get complicated.

Don't worry if it doesn't all make sense today.

There are many layers of concepts at play here.

A fresh namespace

Kubernetes namespaces provide some separation between objects in environments where *many different users* and teams may work on *many different projects*.

- Contain service accounts
- Are access controlled via role based permissions
- Prevent naming conflicts
- Allow many users to work in their own spaces
- Namespaced DNS resolution
- Deletes all namespaced objects when deleted itself

```
$ kubectl create namespace todoapp-demo

            Or

$ kubectl apply -f - << EOF

apiVersion: v1
kind: Namespace
metadata:
  name: todoapp-demo

EOF
```

->

```
namespace "todoapp-demo" created
```

Very little exists here yet

```
$ kubectl config set-context $(kubectl config current-context) -n todoapp-
demo

$ kubectl get all
No resources found

$ kubectl get serviceaccount
NAME      SECRETS   AGE
default   1         7s
```

Persistent storage

We're going to use a **PersistentVolumeClaim** to request a blob of storage for our database.

This will either bind to an existing persistent volume known by Kubernetes or in OKE may cause a new **OCI Storage Volume** to be created for us.

Important bits:

- `storageClassName: oci`
- `requests: storage: 50Gi`

```
$ kubectl apply -f - << EOF

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-storage-claim
spec:
  storageClassName: oci
  selector:
    matchLabels:
      failure-domain.beta.kubernetes.io/zone: "AD-1"
  accessModes:
  - ReadWriteOnce
  resources:

    requests:
      storage: 50Gi

EOF
```

Wait for creation ..

```
$ kubectl get pvc
NAME                    STATUS
mysql-storage-claim     Pending
```

```
NAME                    STATUS
mysql-storage-claim     Bound
```

Woohoo!

Kubernetes provisioned a new Block Storage volume (network attached storage) in OCI and made it known to the cluster. **It has not been attached anywhere yet**.

Database root password secret

To illustrate how secrets can be used, lets make sure keep the MySQL root password away from prying eyes and only mount it in where it is needed.

A Kubernetes secret will store encoded (not encrypted) values in Secrets which can be mounted as files or used as variables where necessary.

NOTE: these are stored in plain text in the Kubernetes data store, they provide only rudimentary protection for values. If you want really secrets, look at systems like Vault.

Create the secret:

```
$ kubectl create secret generic mysql-vars --from-literal=password=secret
secret "mysql-vars" created
```

Describe it:

```
$ kubectl describe secret mysql-vars
Name:         mysql-vars
Namespace:    ben
Labels:       <none>
Annotations:  <none>
Type:  Opaque
Data
====
password:  6 bytes
```

Deploy the database

You may have heard about Kubernetes Pods. For our Database we're going to use a higher-level abstraction called a "Deployment".

A Deployment manages a number of replicas of a pod and will redeploy them if their configuration changes or scale them up or down responsibly depending on the desired replica count.

They are also very useful for controlling upgrade and downgrade (configuration changes).

```
$ kubectl apply -f - << EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  strategy:
    type: Recreate                  # terminate BEFORE starting a new one!


  selector:
    matchLabels:
      app: mysql-app                # matches the label used in the template

  template:
    metadata:
      labels:
        app: mysql-app              # no convention for these - but they must
match
    spec:
      ...
```

```
containers:
  - name: mysql
    image: mysql:5.7
    env:
    - name:  MYSQL_DATABASE         # database to create automatically
      value: uob
    - name: MYSQL_ROOT_PASSWORD     # heres that root password again
      valueFrom:
      secretKeyRef:
        name: mysql-vars
        key: password
    ports:
    - name: mysql                   # named port
      containerPort: 3306
    volumeMounts:
    - name: mysql-vol
      mountPath: /var/lib/mysql
      subPath: mysql
    volumes:
    - name: mysql-vol               # attach to our persistent storage
      persistentVolumeClaim:
        claimName: mysql-storage-claim
```

Now we wait for it to create..

```
$ kubectl get pod --watch
NAME                     READY     STATUS              RESTARTS   AGE

mysql-55ffff8667-xsjz5   0/1       ContainerCreating   0          34s
mysql-55ffff8667-xsjz5   1/1       Running             0          1m
```

Notice the pod name, it is a combination of:

- deployment name
- the suffix of the replica set created by the deployment
- the suffix of the pod created by the replica set (only 1 replica here)

This helps to keep things uniquely named.

Improving the MySQL connection access

Without doing anything else, we *could* connect to the MySQL pod by using one of the following approaches:

- Find the pod IP and connect to it directly on the mysql port
- Use the pod IP DNS name `<ip>.<namespace>.pod.cluster.local`

These work and are often appropriate for single pods. However we can do better by using Kubernetes "Services".

A "Service" will allow us to create a persistent DNS name that outlives the pod itself. The DNS name is resolvable from any other Kubernetes pod and is namespaced as well.

```
$ kubectl apply -f - << EOF
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
    targetPort: mysql          # that named port
  selector:
    app: mysql-app             # the same selector used by the deployment
EOF
```

Connect!

Exec into our MySQL pod (or any other pod with a MySQL client)

```
$ kubectl exec -ti mysql-ccb8f69c7-llsqv /bin/bash
```

```
$ mysql -h mysql.todoapp-demo.svc.cluster.local -u root -D uob -p
Enter password: (secret)

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
...
```

Database: ✅

Deploying the Todo App

We're also going to use a deployment for this, but this time we're going to use the replica features. This will allow us to scale up and down and do upgrades and downgrades fairly easily.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: todoapp
spec:
  replicas: 2               # start with 2 copies of the app
  selector:
    matchLabels:
      app: todoapp-app      # different label compared to mysql
    ...
```

Defining properties in the pod spec:

```
containers:
- name: todoapp
  image: iad.ocir.io/uobtestaccount1/todoapp:latest
  args: [
      "-Dspring.datasource.url=jdbc:mysql://mysql.todoapp-
demo.svc.cluster.local:3306/uob",
      "-Dspring.datasource.username=root",
      "-Dspring.datasource.password=$(MYSQL_ROOT_PASSWORD)",
  ]
  env:
  - name: MYSQL_ROOT_PASSWORD
    valueFrom:
      secretKeyRef:
        name: mysql-vars
        key: password
  ports:
  - name: web
    containerPort: 8080
```

Notes

1. By using args instead of a config map for the properties, we get redeployment of the app for free when we change the properties.
2. We can use environment variables in the args - very useful.
3. Our replicas can both connect to the database concurrently.

Putting a service in front

Similar to MySQL, we'll put a Service with its own Cluster IP in front of the app. Requests will be balanced between the replicas.

```
$ kubectl apply -f - << EOF
apiVersion: v1
kind: Service
metadata:
  name: todoapp-svc
spec:
  ports:
  - port: 8080
    targetPort: web
  selector:
    app: todoapp-app            # same selector as the service
EOF
```