

## 1. Deploy Postgres database using PVC & PV cluster.

Firstly we need to Dockerize PostgreSQL, for this Kubernetes can pull Docker images from a registry and deploy them based on the configuration. We can create our own Docker image or we can use the official image from Docker Hub. For this task we will be using the latest postgres image.

Now let's start by creating our Connection Configuration and Secretes:

- We need to store some connection configuration for the PostgreSQL instance using the Kubernetes secrets config. This will make sure that the sensitive information (say database credentials) are not stored in plain text.
- This provider stores the secretes as base64 strings by default, so let's create the configuration for the secretes. Let's say the password to our database is "password123", then let's convert our password to base64 strings format using the following command:
  - `echo "password123" | base64`

```
> echo "password123" | base64
cGFZc3dvcmQxMjMK
```

- After this step, let's create a secret config file named "postgres-secretes.yml" and apply it to the cluster. We need to add these lines as in snapshot below:

```
GNU nano 4.8                                postgres-secretes.yml
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secret-config
  type: Opaque
data:
  password: cGFZc3dvcmQxMjMK
```

- Now we can apply this config and then verify that the contents are stored correctly using the following commands:
  - `kubectl apply -f postgres-secretes.yml`

```
> kubectl apply -f postgres-secretes.yml
secret/postgres-secret-config created
```

- *kubectl get secret postgres-secret-config -o yaml*

```
> kubectl get secret postgres-secret-config -o yaml
apiVersion: v1
data:
  password: cGFzc3dvcmQxMjMK
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"password":"cGFzc3dvcmQxMjMK"},"kind":"Secret","metadata":{"annotations":{},"name":"postgres-secret-config"},"namespace":"default"}}
  creationTimestamp: "2021-11-28T01:55:21Z"
  name: postgres-secret-config
  namespace: default
  resourceVersion: "8928"
  uid: 38845e8b-d8a4-4851-b333-645d128cc694
type: Opaque
```

Now let's create a PersistentVolumeClaim (PVC) and PersistentVolume (PV) cluster.

- We have to create a permanent file storage for database data because the Docker instance doesn't persist any information when the container no longer exists (by default). And the solution to this is to mount a filesystem to store the data. Kubernetes has different configuration formats for those operations.
- First, we need to create a PersistentVolume manifest that describes the type of volumes we want to use. We define the configuration for the PersistentVolume as in snapshot below:

```
GNU nano 4.8 pv-volume.yml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgres-pv-volume
  labels:
    type: local
    app: postgres
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/mnt/data"
```

- In the above configuration, we have instructed to reserve 5GB of read-write storage at `/mnt/data` on the cluster's node.
- Now, we can apply it and check that the persistent volume is available using the following command:
  - `kubectl apply -f pv-volume.yml`

```
> kubectl apply -f pv-volume.yml
persistentvolume/postgres-pv-volume created
```

- `kubectl get pv postgres-pv-volume`

```
> kubectl get pv postgres-pv-volume
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS	REASON	AGE
postgres-pv-volume	5Gi	RWX	Retain	Available		manual		8s

- Then, after this we create a PersistentVolumeClaim that requests the usage for that particular PersistentVolume type based on the same storage class. Also importantly we need to follow up with a PersistentVolumeClaim configuration that matches the details of the previous manifest. We define the configuration of the PersistentVolumeClaim as in snapshot below:

```
GNU nano 4.8 pv-claim.yml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pv-claim
  labels:
    app: postgres
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

- In the above configuration, we have requested a PersistentVolumeClaim for 1GB of data using the same storage class name. This is an important parameter because it enables Kubernetes to reserve 1GB of the available 5GB of the same storage class for this claim.

- Now we can apply it and check that the persistent volume claim is bounded using the following commands:
  - `kubectl apply -f pv-claim.yml`

```
> kubectl apply -f pv-claim.yml

persistentvolumeclaim/postgres-pv-claim created
```

- `kubectl get pvc postgres-pv-claim`

```
> kubectl get pvc postgres-pv-claim
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
postgres-pv-claim	Bound	postgres-pv-volume	5Gi	RWX	manual	8s

Next, it's now ready for deployment. We need to issue a deployment config for our instance that uses the settings from the post-secret-config secret name. We also need to reference PersistentVolume and PersistentVolumeClaim that we created earlier. We define the configuration of the deployment as in the snapshot below:

```
GNU nano 4.8 postgres-deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      volumes:
        - name: postgres-pv-storage
          persistentVolumeClaim:
            claimName: postgres-pv-claim
      containers:
        - name: postgres
          image: postgres:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 5432
          env:
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: postgres-secret-config
                  key: password
            - name: PGDATA
              value: /var/lib/postgresql/data/pgdata
          volumeMounts:
            - mountPath: /var/lib/postgresql/data
              name: postgres-pv-storage
```

- We can see that we have combined all the configurations together that we defined earlier with the secret config and the persistent volume mounts. We used the `apiVersion: apps/v1` deployment config, which requires us to specify a few lines, such as selector and metadata fields. Then, we added details of the container image and the image pull policy. This is all very important to ensure that we have the right volume and secrets used for that container.
- Now, finally we can apply the deployment and check that it is up and available using the following commands:
  - `kubectl apply -f postgres-deployment.yml`

```
> kubectl apply -f postgres-deployment.yml
deployment.apps/postgres created
```

- `kubectl get deployments`

```
> kubectl get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
postgres      1/1     1            1           97s
```

- We can use the following to check if the PVC is connected to the PV successfully:
  - `kubectl get pvc`

```
> kubectl get pvc
NAME            STATUS   VOLUME          CAPACITY   ACCESS MODES   STORAGECLASS   AGE
postgres-pv-claim  Bound   postgres-pv-volume  5Gi        RWX            manual         4h8m
```

- The status column shows that the claim is Bound, which means they are connected.
- We can see that there is a postgres deployment in the list and it's ready and very much healthy.

## 2. Deploy Postgres Client in cluster (psql).

We can also now create a service to expose the PostgreSQL server first. For this we have several options, like configuring a different port or exposing the Nordport. We will use NordPort for this task, which will expose the service on the Node's IP at a static port.

For this let's create a file named as *postgres-service.yml* and use the following service manifest in the yml file as in the snapshot below:

```
GNU nano 4.8 postgres-service.yml
apiVersion: v1
kind: Service
metadata:
  name: postgres
  labels:
    app: postgres
spec:
  type: NodePort
  ports:
    - port: 5432
  selector:
    app: postgres
```

Here we have used the postgres app selector to connect the postgres deployment as a NordPort service. This will open the host and port the postgres server pair to the <node\_server\_ip>:<nord\_port>.

We can now apply the service and check if it's available, with its assigned port using the following commands:

- *kubectl apply -f postgres-service.yml*

```
> kubectl apply -f postgres-service.yml

service/postgres created
```

- *kubectl get service postgres*

```
> kubectl get service postgres

NAME         TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
postgres     NodePort    10.101.12.244 <none>         5432:30061/TCP   7s
```

Now let's install Postgres Client (psql) on our system, we will use apt to install it using the following command:

→ *sudo apt install postgresql-client-12*

```
> sudo apt install postgresql-client-12
[sudo] password for rikeshkarma:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libpq5 postgresql-client-common
Suggested packages:
  postgresql-12 postgresql-doc-12
The following NEW packages will be installed:
  libpq5 postgresql-client-12 postgresql-client-common
0 upgraded, 3 newly installed, 0 to remove and 0 not upgraded.
Need to get 1,192 kB of archives.
After this operation, 4,360 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://us.archive.ubuntu.com/ubuntu focal-security/main amd64 libpq5 amd64 12.9-0ubuntu0.20.04.1 [117 kB]
Get:2 http://us.archive.ubuntu.com/ubuntu focal-security/main amd64 postgresql-client-common all 214ubuntu0.1 [28.2 kB]
Get:3 http://us.archive.ubuntu.com/ubuntu focal-security/main amd64 postgresql-client-12 amd64 12.9-0ubuntu0.20.04.1 [1,047 kB]
Fetched 1,192 kB in 3s (394 kB/s)
Selecting previously unselected package libpq5:amd64.
(Reading database ... 214640 files and directories currently installed.)
Preparing to unpack .../libpq5_12.9-0ubuntu0.20.04.1_amd64.deb...
```

We actually don't need to install this PostgreSQL client if we are connecting our database through kubectl, but we need this client's installation if we have to connect to the database using the Postgre Client i.e psql.

We should be able to connect to the Postgres database internally using the following commands (using kubectl):

- *kubectl get pods*

```
> kubectl get pods

NAME                                READY   STATUS    RESTARTS   AGE
postgres-7f6bbb7dcf-cdt4v          1/1     Running   0           4h24m
```

- In the snapshot we can see that the name of our PostgreSQL pod is postgres-7f6bbb7dcf-cdt4v, so the next command we execute will be:
  - *kubectl exec -it postgres-7f6bbb7dcf-cdt4v -- psql -U postgres*

```
> kubectl exec -it postgres-7f6bbb7dcf-cdt4v -- psql -U postgres
psql (14.1 (Debian 14.1-1.pgdg110+1))
Type "help" for help.

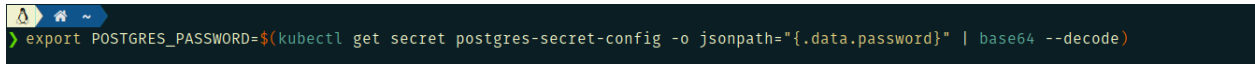
postgres=# _
```

### 3. Connect Postgres database from Postgres Client using core-dns's host name.

Above we have connected to the Postgres database internally using Kubectl, and now let's connect to our Postgres database from Postgres Client using core-dns's host name.

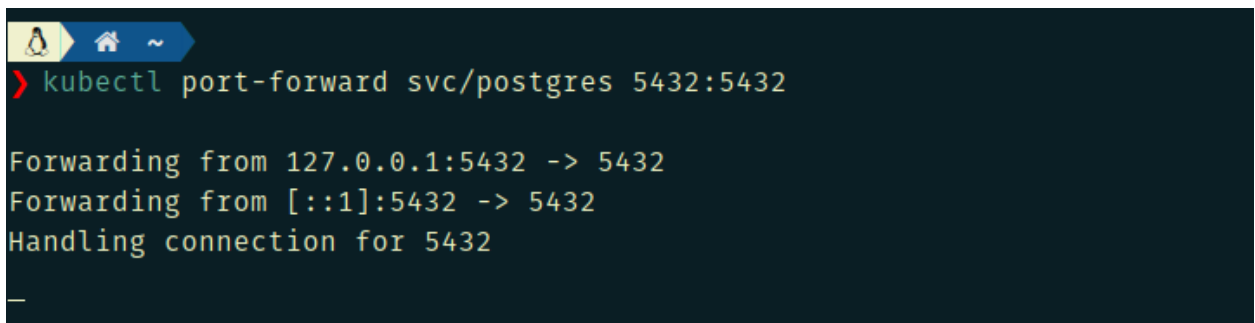
Firstly let's export the POSTGRES\_PASSWORD environment variable to be able to connect/ log into the PostgreSQL instance because we have stored the password in the secretes as base64 strings: Let's use the following command:

- `export POSTGRES_PASSWORD=$(kubectl get secret postgres-secret-config -o jsonpath="{.data.password}" | base64 --decode)`



```
> export POSTGRES_PASSWORD=$(kubectl get secret postgres-secret-config -o jsonpath="{.data.password}" | base64 --decode)
```

- This will store our base64 password in the POSTGRES\_PASSWORD.
- Now we can open another terminal and type the following command to forward the Postgres port: (This is optional if we have created a service already)
  - `kubectl port-forward svc/postgres 5432:5432`



```
> kubectl port-forward svc/postgres 5432:5432

Forwarding from 127.0.0.1:5432 -> 5432
Forwarding from [::1]:5432 -> 5432
Handling connection for 5432
—
```

- After this the system will start handling the port connection.
- We can now minimize the port-forwarding window and return to the previous one and type the following command to connect to psql (a postgresSQL client):
  - `PGPASSWORD="$POSTGRES_PASSWORD" psql --host 127.0.0.1 -U postgres -d postgres -p 5432`



```
PGPASSWORD="$POSTGRES_PASSWORD" psql --host 127.0.0.1 -U postgres -d postgres -p 5432
psql (12.9 (Ubuntu 12.9-0ubuntu0.20.04.1), server 14.1 (Debian 14.1-1.pgdg110+1))
WARNING: psql major version 12, server major version 14.
        Some psql features might not work.
Type "help" for help.

postgres=# \l

               List of databases
  Name      | Owner   | Encoding | Collate  | Ctype    | Access privileges
-----+-----+-----+-----+-----+-----
 postgres   | postgres | UTF8     | en_US.utf8 | en_US.utf8 | 
 template0  | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres +
            |         |         |         |         | postgres=Ctc/postgres
 template1  | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres +
            |         |         |         |         | postgres=Ctc/postgres
(3 rows)

postgres=# \q
```

- After this we observe that the psql command prompt appears, and PostgreSQL is ready to receive our input.

#### 4. Create a database(internship) and a few tables in the database.

To create a database in Postgresql, we need to execute the CREATE DATABASE statement followed by our database name. In our case it will be:

- *CREATE DATABASE internship;*

```
postgres=# CREATE DATABASE internship;
CREATE DATABASE
```

- We can use the “\l” command to view the databases present.

```
postgres=# \l

               List of databases
  Name      | Owner   | Encoding | Collate  | Ctype    | Access privileges
-----+-----+-----+-----+-----+-----
 internship  | postgres | UTF8     | en_US.utf8 | en_US.utf8 | 
 postgres   | postgres | UTF8     | en_US.utf8 | en_US.utf8 | 
 template0  | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres +
            |         |         |         |         | postgres=Ctc/postgres
 template1  | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres +
            |         |         |         |         | postgres=Ctc/postgres
(4 rows)
```

- And to use “\c *internship*” to connect to the database internship.

```
postgres=# \c internship
psql (12.9 (Ubuntu 12.9-0ubuntu0.20.04.1), server 14.1 (Debian 14.1-1.pgdg110+1))
WARNING: psql major version 12, server major version 14.
        Some psql features might not work.
You are now connected to database "internship" as user "postgres".
internship=# _
```

- Now to create we need to execute the CREATE TABLE statement i.e

```
CREATE TABLE table_name (
    column_name TYPE [column_constraint],
    [table_constraint,]
);
```

- This is the simplified basic syntax for the command.
- In our case let's create two tables named "manufacturer" and "supplies" using the following commands:
  - For manufacturer:

```
CREATE TABLE manufacturer (
    id INT PRIMARY KEY,
    name VARCHAR,
    products VARCHAR,
    quality_level int
);
```

```
internship=# CREATE TABLE manufacturer (
internship(# id INT PRIMARY KEY,
internship(# name VARCHAR,
internship(# products VARCHAR,
internship(# quality_level int
internship(# );
CREATE TABLE
```

- For supplies:

```
CREATE TABLE supplies (
    id INT PRIMARY KEY,
    name VARCHAR,
    description VARCHAR,
    manufacturer VARCHAR,
    color VARCHAR
);
```

```
internship=# CREATE TABLE supplies (
internship(# id INT PRIMARY KEY,
internship(# name VARCHAR,
internship(# description VARCHAR,
internship(# manufacturer VARCHAR,
internship(# color VARCHAR
internship(# );
CREATE TABLE
```

- We can use the “\d” command to list the tables in our connected database.

```
internship=# \d
               List of relations
 Schema |      Name      | Type  | Owner
-----+-----+-----+-----
 public | manufacturer   | table | postgres
 public | supplies       | table | postgres
(2 rows)
```

- Also we can use “\d table\_name” to also view the columns and its values of the table.
  - \d manufacturers

```
internship=# \d manufacturer
               Table "public.manufacturer"
  Column      |      Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
 id           | integer        |           | not null |
 name        | character varying |           |          |
 products    | character varying |           |          |
 quality_level | integer        |           |          |
Indexes:
    "manufacturer_pkey" PRIMARY KEY, btree (id)
```

- \d supplies

```
internship=# \d supplies
               Table "public.supplies"
  Column      |      Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
 id           | integer        |           | not null |
 name        | character varying |           |          |
 description  | character varying |           |          |
 manufacturer | character varying |           |          |
 color       | character varying |           |          |
Indexes:
    "supplies_pkey" PRIMARY KEY, btree (id)
```

We have successfully created a database(internship) and a few tables in the internship database.