DEPLOYING POSTGRESQL AS A STATEFULSET IN KUBERNETES

There are different types of applications, from single large applications to microservices-based applications that cater to different needs. When it comes to the states of those applications, there are two states:

- **Stateless applications** can be run independently in isolation without any knowledge of past transactions.
- **Stateful applications** have full knowledge of the past information (state).

Most applications we use are stateful applications, and their state data may consist of user preferences, recent activity, database transactions, credentials, settings, etc.

Kubernetes provides StatefulSets when creating a stateful application in a Kubernetes cluster. Managing states within a containerized environment has become even more significant with the popularity of deploying database clusters in Kubernetes.

In this article, we will focus on how to deploy a PostgreSQL database on a Kubernetes cluster using StatefulSets.

(This article is part of our Kubernetes Guide. Use the right-hand menu to navigate.)

What is Kubernetes StatefulSets?

StatefulSet is a Kubernetes workload API object that can be used to manage stateful applications.

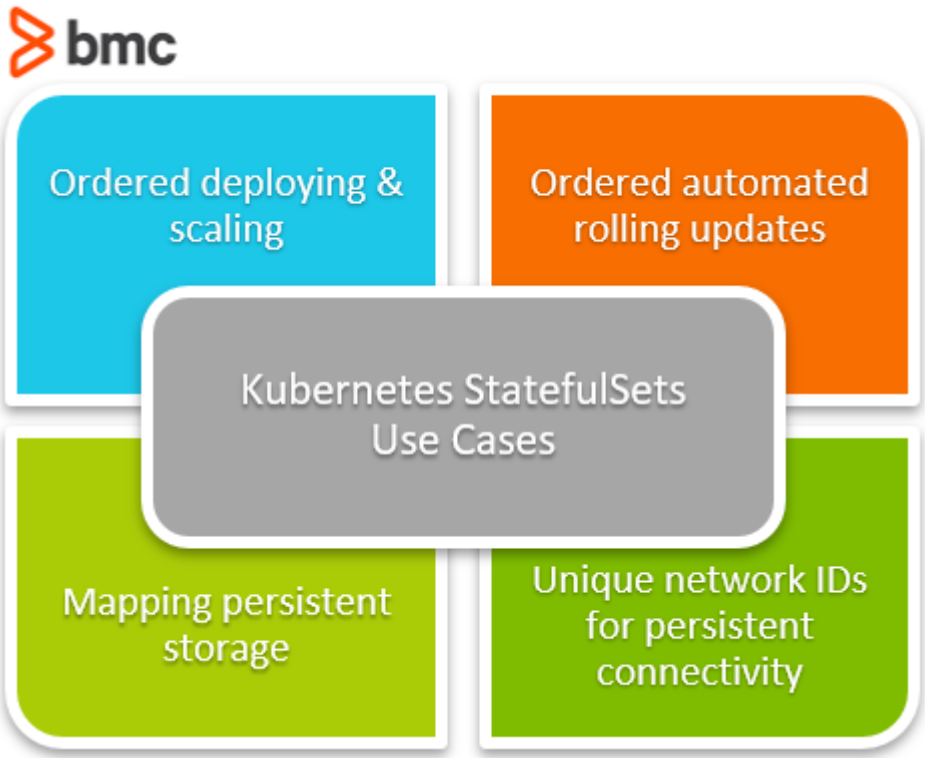
In a normal deployment, the user does not care how the pods are scheduled as long as it doesn't have a negative impact on the deployed application. However, there is the need to properly identify

pods to preserve the state in stateful applications with persistent storage.

StatefulSet provides this functionality by creating pods with a persistent identifier that will pertain to its value across rescheduling. This way, a pod will get correctly mapped to the storage volumes even if it is recreated, and the application state will be preserved.

StatefulSets use cases

There are several use cases for StatefulSets.



Ordered deploying & scaling

When an application relies on multiple containers, the ordered approach to scaling ensures that dependent containers are created in an orderly manner at deployments and scaling scenarios.

Ordered automated rolling updates

Updating applications or microservices that are dependent should also be updated in an orderly manner. Additionally, an update should not affect the functionality.

Therefore, users can decide the order in which the applications or microservices should be updated by using a StatefulSet.

Mapping persistent storage

When considering databases, persistent storage is the most critical part as applications need to store data. With a StatefulSet, users can:

1. Define which pods correspond to each persistent storage
2. Create resilient application deployments

Using unique network identifiers to create persistent network connectivity

With unique identifiers, network users can manage and route traffic for specific pods without worrying about IP changes at rescheduling. This provides greater control over the network communications between pods by providing the ability to configure persistent routing, policies, and security configs for desired pods.

Even with these benefits, StatefulSets do not provide a solution for all requirements. For instance, StatefulSets are not interchangeable with deployments or ReplicaSets—these are instead geared to stateless configurations.

Drawbacks of StatefulSets

StatefulSets also come with a set of limitations that users should be aware of before deploying the application.

- The storage for a StatefulSet must be provisioned either by a PersistentVolume Provisioner based on the storage class or pre-provisioned.
- Scaling or deleting pods will not affect the underlying persistent storage in order to ensure data safety. The provisioned volumes will remain within Kubernetes.
- The user needs to create a headless service manually to ensure network identity in StatefulSets.
- SatefulSets does not guarantee the termination of current pods when the StatefulSet is deleted. So, best practice is that the user implement an SOP to scale the StatefulSet to zero pods before deleting.
- Rolling updates with the default pod management policy may cause issues when deploying if a pod is broken (due to an application config error, bad binary, etc.). In such instances, users need to:
    - Manually revert to a previous deployment template.
    - Delete the broken Pods before attempting to rerun the new updates.

Setting up a StatefulSet in a Kubernetes cluster

Now that we have a basic understanding of a StatefulSet, let's look at a sample StatefulSet deployment.

StatefulSets are ideal for database deployments. In this example, we will create a PostgreSQL deployment as a StatefulSet with a persistent storage volume.

postgresql.yaml

```
# PostgreSQL StatefulSet
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgresql-db
spec:
  serviceName: postgresql-db-service
  selector:
```

```
    matchLabels:
      app: postgresql-db
  replicas: 2
  template:
    metadata:
      labels:
        app: postgresql-db
    spec:
      containers:
        - name: postgresql-db
          image: postgres:latest
          volumeMounts:
            - name: postgresql-db-disk
              mountPath: /data
          env:
            - name: POSTGRES_PASSWORD
              value: testpassword
            - name: PGDATA
              value: /data/pgdata
  # Volume Claim
  volumeClaimTemplates:
    - metadata:
        name: postgresql-db-disk
      spec:
        accessModes:
        resources:
          requests:
            storage: 25Gi
```

In the above YAML file, we have defined a simple StatefulSet to deploy a PostgreSQL database. We are creating a StatefulSet called postgresql-db with two pods (replicas: 2).

Additionally, we are creating a Persistent Volume using the volumeClaimTemplate and using it in the StatefulSet to store the PostgreSQL data. The default Persistent Volume provisioner will provision the volume, and we can deploy this by running the following command.

```
kubectl apply -f postgresql.yaml
```

Result:

```
❯ kubectl apply -f postgresql.yaml
statefulset.apps/postgresql-db created
```

Now we have successfully created a PostgreSQL StatefulSet yet need a service to expose it outside of the Kubernetes cluster. That can be done by creating a service that points to the StatefulSet.

postgresql-service.yaml

```
# PostgreSQL StatefulSet Service
apiVersion: v1
```

```
kind: Service
metadata:
  name: postgres-db-lb
spec:
  selector:
    app: postgresql-db
  type: LoadBalancer
  ports:
    - port: 5432
      targetPort: 5432
```

This will create a load balancer, and the service will expose our PostgreSQL database using the selector "app: postgresql-db." You can create the service by running the below command.

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

Result:

```
> kubectl apply -f postgresql-service.yaml
service/postgres-db-lb created
```

Now let's see if both the StatefulSet and the service are successfully created in Kubernetes by running the following command.

```
kubectl get all
```

Result:

```
> kubectl get all
NAME                  READY   STATUS    RESTARTS   AGE
pod/postgresql-db-0   1/1     Running   0          3m24s
pod/postgresql-db-1   1/1     Running   0          2m55s

NAME                     TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)          AGE
service/kubernetes       ClusterIP      10.96.0.1      <none>         443/TCP          21m
service/postgres-db-lb   LoadBalancer   10.111.253.4   10.111.253.4   5432:30975/TCP   3m7s

NAME                             READY   AGE
statefulset.apps/postgresql-db   2/2     3m24s
```

The result indicates that two pods are created and running with a load balancer service exposing the StatefulSet via IP 10.111.253.4 using port 5432.

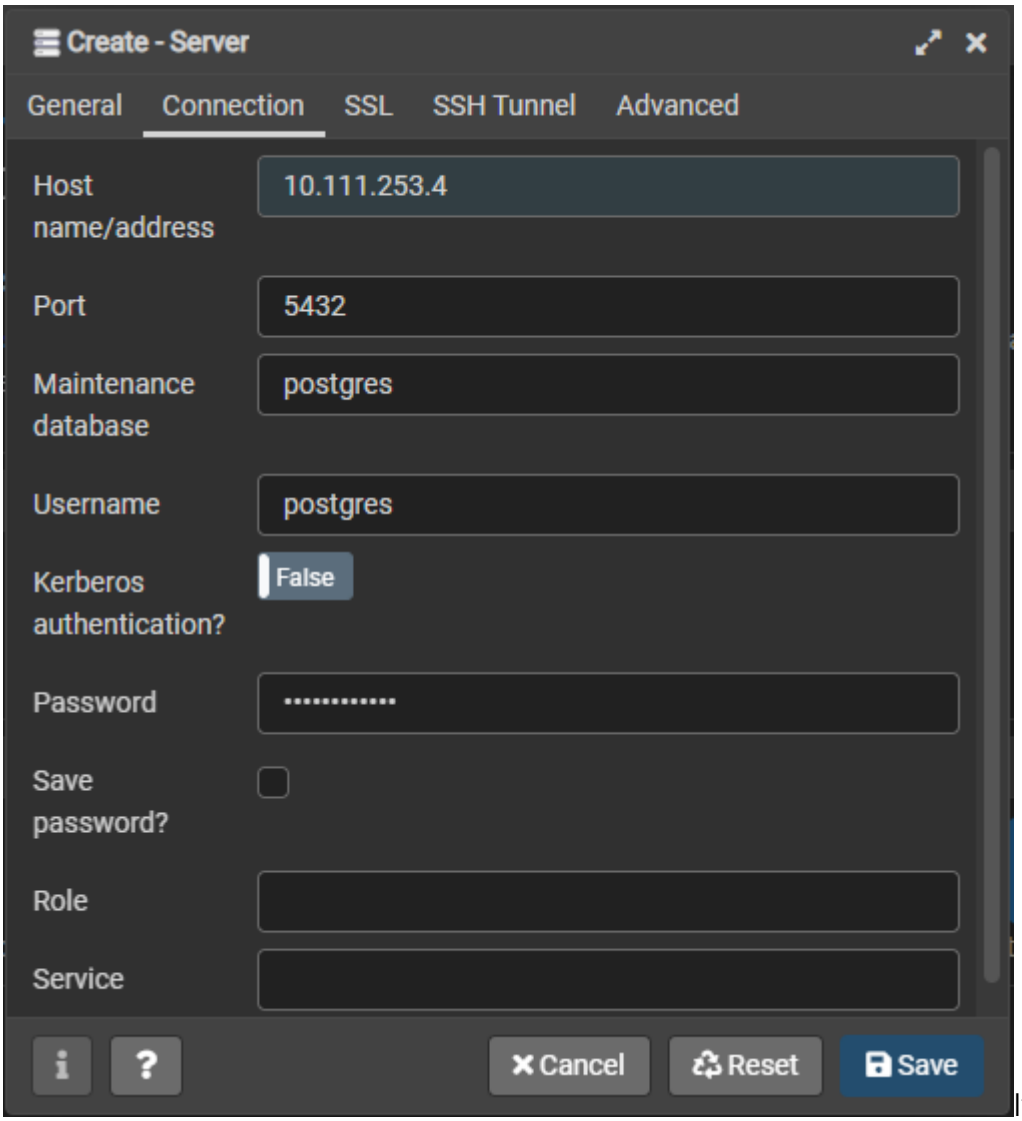
Testing the connectivity

Having deployed PostgreSQL, we need to verify that we can access it without any issues. We will use the pgAdmin4 client to initialize the connection:

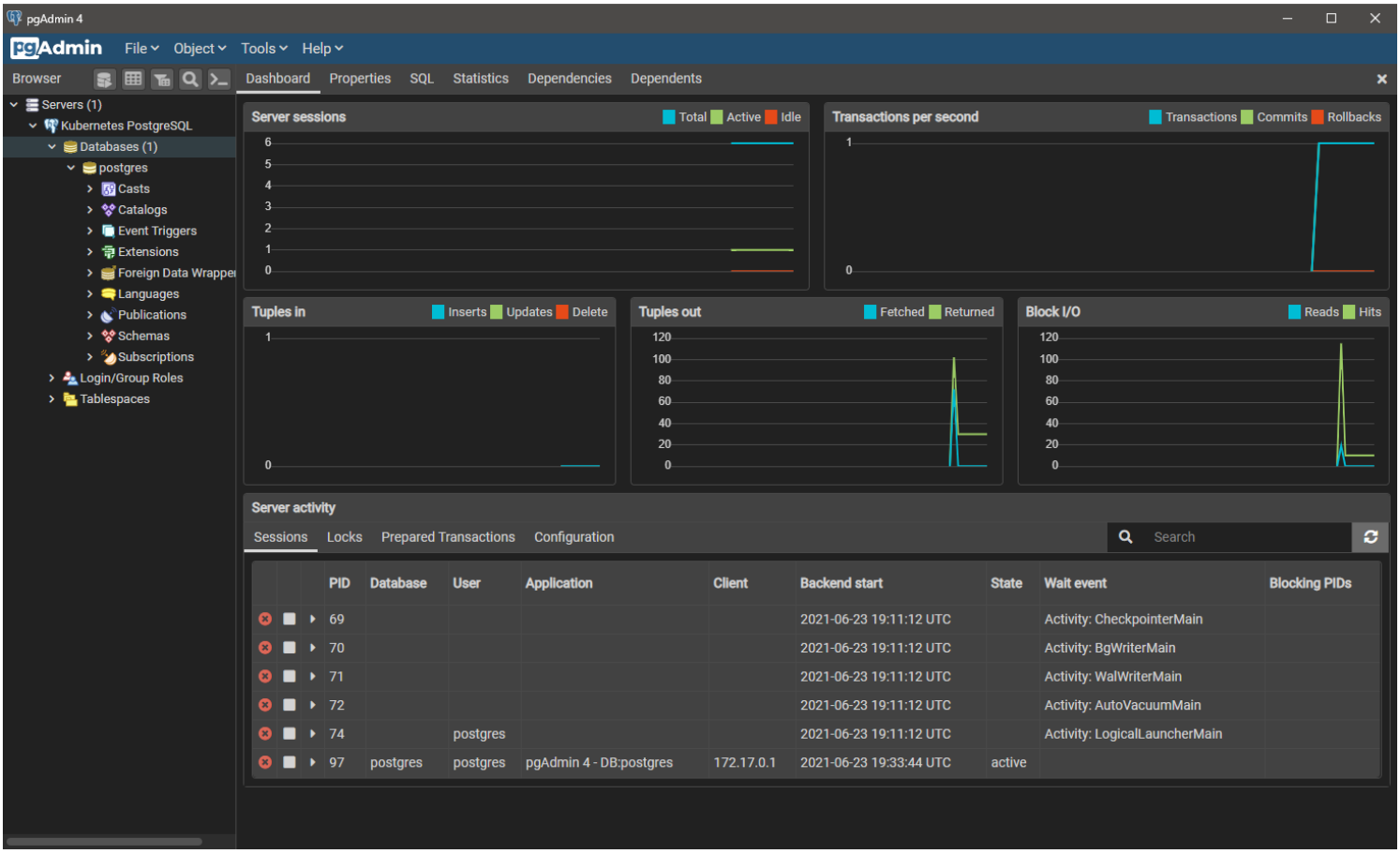
1. Download the pgAdmin client in your environment.
2. Connect and try to initialize a connection.

With the above deployment, we will use the external IP of the postgres-db-lb service (10.111.253.4) with port 5432. Since we only defined a password in our environment variables for the PostgreSQL

StatefulSet, the configuration will have the default username "postgres" with the password we defined.



If all the details are correct, the connection will be initiated when you click on "Save," and the user will be able to see the connected database.



Using ConfigMap in StatefulSet

In our earlier example, we defined the environment variables with the StatefulSet YAML.

However, the best practice would be to separate the environment variables using ConfigMaps and call the ConfigMap from the StatefulSet deployment. This makes it easier to manage and maintain each component of the deployment.

So, let's create a ConfigMap and modify the StatefulSet YAML as shown below.

postgresql-configmap.yaml

```
# PostgreSQL StatefulSet ConfigMap
apiVersion: v1
kind: ConfigMap
metadata:
  name: postgres-db-config
  labels:
    app: postgresql-db
data:
  POSTGRES_DB: testdb
  POSTGRES_USER: testdbuser
  POSTGRES_PASSWORD: testdbuserpassword
  PGDATA: /data/pgdata
```

In the above ConfigMap, we have extended our environment variable to specify a PostgreSQL

database, user, password, and data store. Now let's create the configMap and view the configurations using this command:

```
kubectl apply -f .postgresql-configmap.yaml
```

Result:

```
> kubectl apply -f .\postgresql-configmap.yaml
configmap/postgres-db-config created
```

We can get the information of the created ConfigMap using the describe function:

```
kubectl describe configmap postgres-db-config
```

Result:

```
> kubectl describe configmap postgres-db-config
Name:         postgres-db-config
Namespace:    default
Labels:       app=postgresql-db
Annotations:  <none>

Data
====
PGDATA:
----
/data/pgdata
POSTGRES_DB:
----
testdb
POSTGRES_PASSWORD:
----
testdbuserpassword
POSTGRES_USER:
----
testdbuser
Events:  <none>
```

The next step is to modify the StatefulSet to call the data from the ConfigMap. That can be done by using the envFrom field to point to the above ConfigMap. This will also enable us to create a StatefulSet using the data in the ConfigMap.

postgresql.yaml

```
# PostgreSQL StatefulSet - ConfigMap
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgresql-db
```

```
spec:
  serviceName: postgresql-db-service
  selector:
    matchLabels:
      app: postgresql-db
  replicas: 2
  template:
    metadata:
      labels:
        app: postgresql-db
    spec:
      containers:
        - name: postgresql-db
          image: postgres:latest
          volumeMounts:
            - name: postgresql-db-disk
              mountPath: /data
          # Config from ConfigMap
          envFrom:
          - configMapRef:
              name: postgres-db-config
  # Volume Claim
  volumeClaimTemplates:
    - metadata:
        name: postgresql-db-disk
      spec:
        accessModes:
        resources:
          requests:
            storage: 25Gi
```

With our Persistent Volumes, deletions in the underlying database will be preserved even in case of Pod errors. As a StatefulSet, the state of Pods will also be preserved, and they will get assigned to the desired volumes correctly when recreated.

We can identify the pods using the following command:

`kubectl get pvc`

Result:

```
> kubectl  get pvc
NAME                                 STATUS   VOLUME                                     CAPACITY   AC
CESS MODES   STORAGECLASS   AGE
postgresql-db-disk-postgresql-db-0   Bound    pvc-16800078-b0e0-4911-a7c0-9e2ead637453   25Gi       RW
O            standard       4m12s
postgresql-db-disk-postgresql-db-1   Bound    pvc-165ed65c-a1f3-4a85-a3aa-bf214c53b767   25Gi       RW
O            standard       4m5s
```

Managing state is crucial to app functionality

Kubernetes StatefulSets allows users to easily create and manage stateful applications or services within a Kubernetes cluster. However, these StatefulSets configurations involve some complexity—so you must carefully plan your deployments before them carrying out.

Additionally, StatefulSets are the ideal solution for dealing with database applications, payment services, etc., where managing state is a crucial part of the application functionality.

Related reading

- BMC DevOps Blog
- Kubernetes Multi-Clusters: How & Why To Use Them
- Kubernetes Best Practices for Enhanced Cluster Efficiency
- 3 Kubernetes Patterns for Cloud Native Applications
- The State of Containers Today
- Containerized Machine Learning: An Intro to ML in Containers