PERCONA

www.percona.com

Percona Kubernetes for Percona Server for MongoDB Documentation

Release 1.1.0

Jul 15, 2019

CONTENTS

I Overview 3

II Installation 9

III Configuration 27

IV Reference 57

The Percona Kubernetes Operator for Percona Server for MongoDB automates the creation, modification, or deletion of items in your Percona Server for MongoDB environment. The Operator contains the necessary Kubernetes settings to maintain a consistent Percona Server for MongoDB instance.

The Percona Kubernetes Operators are based on best practices for the configuration of a Percona Server for MongoDB replica set. The Operator provides many benefits but saving time, a consistent environment are the most important.

The operator was developed and tested for the following configurations only:

- Percona Server for MongoDB 3.6 and Percona Server for MongoDB 4.0
- OpenShift 3.11 and OpenShift 4.0

Other options may or may not work.

Backups are not yet supported with Percona Server for MongoDB 4.0. Backups are supported for Percona Server for MongoDB 3.6.

Also, the current PSMDB on Kubernetes implementation does not support Percona Server for MongoDB sharding.

Part I

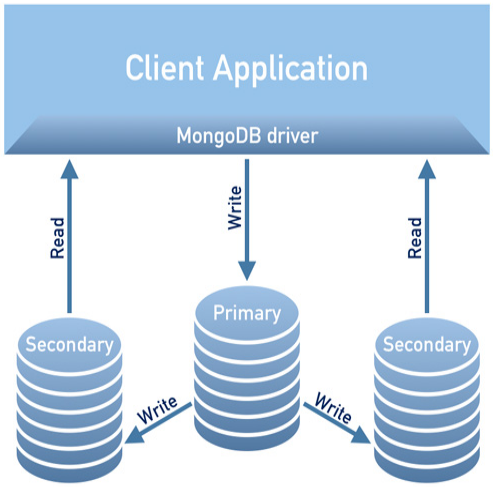
Overview

CHAPTER

ONE

DESIGN OVERVIEW

The design of the operator is tighly bound to the Percona Server for MongoDB replica set, which is briefly described in the following diagram.

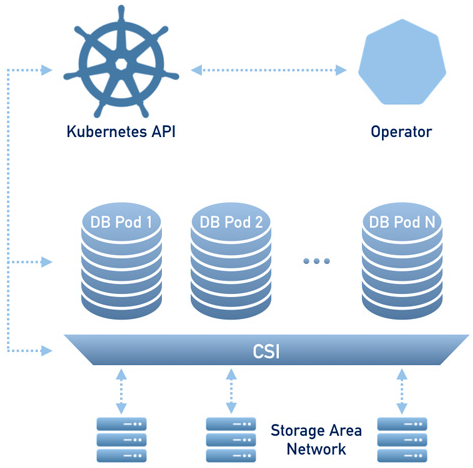


A replica set consists of one primary server and several secondary ones (two in the picture), and the client application accesses the servers via a driver.

To provide high availability the Operator uses node affinity to run MongoDB instances on separate worker nodes if possible, and the database cluster is deployed as a single Replica Set with at least three nodes. If a node fails, the pod with the mongod process is automatically re-created on another node. If the failed node was hosting the primary server, the replica set initiates elections to select a new primary. If the failed node was running the Operator, Kubernetes will restart the Operator on another node, so normal operation will not be interrupted.

Client applications should use a mongo+srv URI for the connection. This allows the drivers (3.6 and up) to retrieve the list of replica set members from DNS SRV entries without having to list hostnames for the dynamically assigned nodes.

Note: The Operator uses security settings which are more secure than the default Percona Server for MongoDB setup. The initial configuration contains default passwords for all needed user accounts, which should be changed in the production environment, as stated in the installation instructions.



To provide data storage for stateful applications, Kubernetes uses Persistent Volumes. A *PersistentVolumeClaim* (PVC) is used to implement the automatic storage provisioning to pods. If a failure occurs, the Container Storage Interface (CSI) should be able to re-mount storage on a different node. The PVC StorageClass must support this feature (Kubernetes and OpenShift support this in versions 1.9 and 3.9 respectively).

The Operator functionality extends the Kubernetes API with *PerconaServerMongoDB* object, and it is implemented as a golang application. Each *PerconaServerMongoDB* object maps to one separate PSMDB setup. The Operator listens to all events on the created objects. When a new PerconaServerMongoDB object is created, or an existing one undergoes some changes or deletion, the operator automatically creates/changes/deletes all needed Kubernetes objects with the appropriate settings to provide a properly operating replica set.

CHAPTER

TWO

SYSTEM REQUIREMENTS

The following platforms are supported:

- OpenShift >=3.11
- Google Kubernetes Engine (GKE)
- Minikube

Part II

Installation

CHAPTER

THREE

INSTALL PERCONA SERVER FOR MONGODB ON KUBERNETES

0. Clone the percona-server-mongodb-operator repository:

```
git clone -b release-1.0.0 https://github.com/percona/percona-server-mongodb-
↪operator
cd percona-server-mongodb-operator
```

Note: It is crucial to specify the right branch with `-b` option while cloning the code on this step. Please be careful.

1. The Custom Resource Definition for PSMDB should be created from the `deploy/crd.yaml` file. The Custom Resource Definition extends the standard set of resources which Kubernetes "knows" about with the new items (in our case resources which are the core of the operator).

```
$ kubectl apply -f deploy/crd.yaml
```

This step should be done only once; the step does not need to be repeated with any other Operator deployments.

2. Add the `psmdb` namespace to Kubernetes, and set the correspondent context for further steps:

```
$ kubectl create namespace psmdb
$ kubectl config set-context $(kubectl config current-context) --namespace=psmdb
```

3. The role-based access control (RBAC) for PSMDB is configured with the `deploy/rbac.yaml` file. Role-based access is based on defined roles and the available actions which correspond to each role. The role and actions are defined for Kubernetes resources in the yaml file. Further details about users and roles can be found in Kubernetes documentation.

```
$ kubectl apply -f deploy/rbac.yaml
```

Note: Setting RBAC requires your user to have cluster-admin role privileges. For example, those using Google Kubernetes Engine can grant user needed privileges with the following command:

```
$ kubectl create clusterrolebinding cluster-admin-binding --clusterrole=cluster-
↪admin --user=$(gcloud config get-value core/account)
```

4. Start the operator within Kubernetes:

```
$ kubectl apply -f deploy/operator.yaml
```

5. Add the MongoDB Users secrets to Kubernetes. Additional names should be placed in the data section of the `deploy/mongodb-users.yaml` file as login name and the base64-encoded passwords for the user accounts (see Kubernetes documentation for details).

   Note: The following command can be used to get base64-encoded password from a plain text string:

   ```
   $ echo -n 'plain-text-password' | base64
   ```

   After editing the yaml file, mongodb-users secrets should be created (or updated with the new passwords) using the following command:

   ```
   $ kubectl apply -f deploy/secrets.yaml
   ```

   More details about secrets can be found in Users.

6. Install cert-manager if it is not up and running yet and apply ssl secrets with the following command:

   Pre-generated certificates are available in the `deploy/ssl-secrets.yaml` secrets file for test purposes, but we strongly recommend avoiding their usage on any production system.

   ```
   $ kubectl apply -f <secrets file>
   ```

7. After the operator is started, Percona Server for MongoDB cluster can be created with the following command:

   ```
   $ kubectl apply -f deploy/cr.yaml
   ```

   The creation process may take some time. The process is over when both operator and replica set pod have reached their Running status:

   ```
   $ kubectl get pods
   NAME                                               READY   STATUS    RESTARTS   AGE
   my-cluster-name-rs0-0                              1/1     Running   0          8m
   my-cluster-name-rs0-1                              1/1     Running   0          8m
   my-cluster-name-rs0-2                              1/1     Running   0          7m
   percona-server-mongodb-operator-754846f95d-sf6h6   1/1     Running   0          9m
   ```

6. Check connectivity to newly created cluster

   ```
   $ kubectl run -i --rm --tty percona-client --image=percona/percona-server-
   ↪mongodb:3.6 --restart=Never -- bash -il
   percona-client:/$ mongo "mongodb+srv://userAdmin:userAdmin123456@my-cluster-name-
   ↪rs0.psmdb.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
   ```

CHAPTER

FOUR

INSTALL PERCONA SERVER FOR MONGODB ON OPENSHIFT

0. Clone the percona-server-mongodb-operator repository:

```
git clone -b release-1.0.0 https://github.com/percona/percona-server-mongodb-
↪operator
cd percona-server-mongodb-operator
```

Note: It is crucial to specify the right branch with `-b` option while cloning the code on this step. Please be careful.

1. The Custom Resource Definition for PSMDB should be created from the `deploy/crd.yaml` file. The Custom Resource Definition extends the standard set of resources which Kubernetes "knows" about with the new items, in our case these items are the core of the operator.

   This step should be done only once; it does not need to be repeated with other deployments.

```
$ oc apply -f deploy/crd.yaml
```

Note: Setting Custom Resource Definition requires your user to have cluster-admin role privileges.

   An extra action is required if you want to manage PSMDB cluster with a non-privileged user. Please make shure that cert-manager is already installed. The necessary permissions can be granted by applying the clusterrole:

```
$ oc create clusterrole psmdb-admin --verb="*" --resource=perconaservermongodbs.
↪psmdb.percona.com,perconaservermongodbs.psmdb.percona.com/status,
↪perconaservermongodbbackups.psmdb.percona.com,perconaservermongodbbackups.psmdb.
↪percona.com/status,perconaservermongodbrestores.psmdb.percona.com,
↪perconaservermongodbrestores.psmdb.percona.com/status,issuers.certmanager.k8s.
↪io,certificates.certmanager.k8s.io
$ oc adm policy add-cluster-role-to-user psmdb-admin <some-user>
```

2. Create a new `psmdb` project:

```
$ oc new-project psmdb
```

3. Add role-based access control (RBAC) for PSMDB is configured with the `deploy/rbac.yaml` file. RBAC is based on clearly defined roles and corresponding allowed actions. These actions are allowed on specific Kubernetes resources. The details about users and roles can be found in OpenShift documentation.

```
$ oc apply -f deploy/rbac.yaml
```

4. Start the Operator within OpenShift:

```
$ oc apply -f deploy/operator.yaml
```

4. Add the MongoDB Users secrets to OpenShift. These secrets should be placed in the data section of the `deploy/secrets.yaml` file as login names and base64-encoded passwords (see Kubernetes documentation for details).

Note: The following command can be used to return a base64-encoded password from a plain text string:

```
$ echo -n 'plain-text-password' | base64
```

When you have completed adding the additional information, the secrets should be created or updated with the following command:

```
$ oc apply -f deploy/secrets.yaml
```

More details about secrets can be found in Users.

5. Install cert-manager if it is not up and running yet then generate and apply certificates as secrets according to *TLS document <TLS.html>*:

Pre-generated certificates are available in the `deploy/ssl-secrets.yaml` secrets file for test purposes, but we strongly recommend avoiding their usage on any production system.

```
$ oc apply -f <secrets file>
```

6. Percona Server for MongoDB cluster can be created at any time with the following two steps:

(a) Uncomment the `deploy/cr.yaml` field `#platform:` and edit the field to `platform: openshift`. The result should be like this:

```
apiVersion: psmdb.percona.com/v1alpha1
kind: PerconaServerMongoDB
metadata:
  name: my-cluster-name
spec:
  platform: openshift
...
```

b (optional). In you're using minishift, please adjust antiaffinity policy to `none`

```
   affinity:
     antiAffinityTopologyKey: "none"
...
```

(a) Create/apply the CR file:

```
$ oc apply -f deploy/cr.yaml
```

The creation process will take time. The process is complete when both the operator and the replica set pod have reached their Running status:

```
$ oc get pods
NAME                                               READY   STATUS    RESTARTS   ␣
↪AGE
my-cluster-name-rs0-0                              1/1     Running   0          8m
```

```
my-cluster-name-rs0-1                                1/1     Running   0          8m
my-cluster-name-rs0-2                                1/1     Running   0          7m
percona-server-mongodb-operator-754846f95d-sf6h6     1/1     Running   0          9m
```

3. Check connectivity to newly created cluster. Please note that mongo client command shall be executed inside the container manually.

```
$ oc run -i --rm --tty percona-client --image=percona/percona-server-mongodb:3.6 -
↪-restart=Never -- bash -il
percona-client:/$ mongo "mongodb+srv://userAdmin:userAdmin123456@my-cluster-name-
↪rs0.psmdb.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
```

CHAPTER

FIVE

INSTALL PERCONA SERVER FOR MONGODB ON MINIKUBE

Installing the PSMDB Operator on Minikube is the easiest way to try it locally without a cloud provider. Minikube runs Kubernetes on GNU/Linux, Windows, or macOS system using a system-wide hypervisor, such as VirtualBox, KVM/QEMU, VMware Fusion or Hyper-V. Using it is a popular way to test Kubernetes application locally prior to deploying it on a cloud.

The following steps are needed to run PSMDB Operator on minikube:

0. Install minikube, using a way recommended for your system. This includes the installation of the following three components: #. kubectl tool, #. a hypervisor, if it is not already installed, #. actual minikube package

   After the installation running `minikube start` should download needed virtualized images, then initialize and run the cluster. After Minikube is successfully started, you can optionally run Kubernetes dashboard, which visually represents the state of your cluster. Executing `minikube dashboard` will start the dashboard and open it in your default web browser.

1. Clone the percona-server-mongodb-operator repository:

   ```
   git clone -b release-1.1.0 https://github.com/percona/percona-server-mongodb-
   ↪operator
   cd percona-server-mongodb-operator
   ```

2. Deploy the operator with the following command:

   ```
   kubectl apply -f deploy/bundle.yaml
   ```

3. Edit the `deploy/cr.yaml` file to change the following keys in `replsets` section, which would otherwise prevent running Percona Server for MongoDB on your local Kubernetes installation:

   (a) comment `resources.requests.memory` and `resources.requests.cpu` keys

   (b) set `affinity.antiAffinityTopologyKey` key to `"none"`

   Also, switch `allowUnsafeConfigurations` key to `true`.

4. Now apply the `deploy/cr.yaml` file with the following command:

   ```
   kubectl apply -f deploy/cr.yaml
   ```

5. During previous steps, the Operator has generated several secrets, including the password for the admin user, which you will need to access the cluster. Use `kubectl get secrets` to see the list of Secrets objects (by default Secrets object you are interested in has `my-cluster-name-secrets` name). Then `kubectl get secret my-cluster-name-secrets -o yaml` will return the YAML file with generated secrets, including the `MONGODB_USER_ADMIN` and `MONGODB_USER_ADMIN_PASSWORD` strings, which should look as follows:

```
...
data:
  ...
  MONGODB_USER_ADMIN_PASSWORD: aDAzQ0pCY3NSWEZ2ZUIzS1I=
  MONGODB_USER_ADMIN_USER: dXNlckFkbWlu
```

Here the actual login name and password are base64-encoded, and `echo 'aDAzQ0pCY3NSWEZ2ZUIzS1I=' | base64 --decode` will bring it back to a human-readable form.

6. Check connectivity to a newly created cluster.

   First of all, run percona-client and connect its console output to your terminal (running it may require some time to deploy the correspondent Pod):

```
kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:4.
↪0 --restart=Never -- bash -il
```

   Now run `mongo` tool in the percona-client command shell using the login (which is `userAdmin`) and password obtained from the secret:

```
mongo "mongodb+srv://userAdmin:userAdminPassword@my-cluster-name-rs0.default.svc.
↪cluster.local/admin?replicaSet=rs0&ssl=false"
```

CHAPTER

SIX

SCALE PERCONA SERVER FOR MONGODB ON KUBERNETES AND OPENSHIFT

One of the great advantages brought by Kubernetes and the OpenShift platform is the ease of an application scaling. Scaling a Deployment up or down ensures new Pods are created and set to available Kubernetes nodes.

Size of the cluster is controlled by a `size` key in the Custom Resource options configuration, as specified in the Operator Options section. That's why scaling the cluster needs nothing more but changing this option and applying the updated configuration file. This may be done in a specifically saved config, or on the fly, using the following command, which saves the current configuration, updates it and applies the changed version:

```
$ kubectl get psmdb/my-cluster-name -o yaml | sed -e 's/size: 3/size: 5/' | kubectl␣
↪apply -f -
```

In this example we have changed the size of the Percona Server for MongoDB from `3`, which is a minimum recommended value, to `5` nodes.

Note: *Using ''kubectl scale StatefulSet_name'' command to rescale Percona Server for MongoDB is not recommended, as it makes ''size'' configuration option out of sync, and the next config change may result in reverting the previous number of nodes.*

CHAPTER

SEVEN

UPDATE PERCONA SERVER FOR MONGODB OPERATOR

Starting from the version 1.1.0 the Percona Kubernetes Operator for MongoDB allows upgrades to newer versions. The upgrade can be either semi-automatic or manual.

Note: Manual update mode is the recommended way for a production cluster.

Semi-automatic update

1. Edit the `deploy/cr.yaml` file, setting `updateStrategy` key to `RollingUpdate`.
2. Now you should apply a patch to your deployment, supplying necessary image names with a newer version tag. This is done with the `kubectl patch deployment` command. For example, updating to the `1.1.0` version should look as follows:

```
kubectl patch deployment percona-server-mongodb-operator \
   -p'{"spec":{"template":{"spec":{"containers":[{"name":"percona-server-mongodb-
↪operator","image":"percona/percona-server-mongodb-operator:1.1.0"}]}}}}'

kubectl patch psmdb my-cluster-name --type=merge --patch '{
   "spec": {
       "image": "percona/percona-server-mongodb-operator:1.1.0-mongod4.0",
       "backup": { "image": "percona/percona-server-mongodb-operator:1.1.0-backup
↪" }
   }}'
```

3. The deployment rollout will be automatically triggered by the applied patch. You can track the rollout process in real time using the `kubectl rollout status` command with the name of your cluster:

```
kubectl rollout status sts cluster1-pxc
```

Manual update

1. Edit the `deploy/cr.yaml` file, setting `updateStrategy` key to `OnDelete`.
2. Now you should apply a patch to your deployment, supplying necessary image names with a newer version tag. This is done with the `kubectl patch deployment` command. For example, updating to the `1.1.0` version should look as follows:

```
kubectl patch deployment percona-server-mongodb-operator \
   -p'{"spec":{"template":{"spec":{"containers":[{"name":"percona-server-mongodb-
↪operator","image":"percona/percona-server-mongodb-operator:1.1.0"}]}}}}'

kubectl patch psmdb my-cluster-name --type=merge --patch '{
   "spec": {"replsets":{ "image": "percona/percona-server-mongodb-operator:1.1.0-
↪mongod4.0" },
       "mongod": { "image": "percona/percona-server-mongodb-operator:1.1.0-
↪mongod4.0" },
       "backup":   { "image": "percona/percona-server-mongodb-operator:1.1.0-
↪backup" }
   }}'
```

3. Pod with the newer Percona Server for MongoDB image will start after you delete it. Delete targeted Pods manually one by one to make them restart in the desired order:

   (a) Delete the Pod using its name with the command like the following one:

   ```
   kubectl delete pod my-cluster-name-rs0-2
   ```

   (b) Wait until Pod becomes ready:

   ```
   kubectl get pod my-cluster-name-rs0-2
   ```

   The output should be like this:

   ```
   NAME                     READY   STATUS    RESTARTS   AGE
   my-cluster-name-rs0-2    1/1     Running   0          3m33s
   ```

4. The update process is successfully finished when all Pods have been restarted.

CHAPTER

EIGHT

MONITORING

The Percona Monitoring and Management (PMM) provides an excellent solution to monitor Percona Server for MongoDB.

The following steps are needed to install both PMM Client and PMM Server. The PMM Client and PMM Server are preconfigured to monitor Percona Server for MongoDB on Kubernetes or OpenShift.

1. The recommended installation approach is based on using helm - the package manager for Kubernetes, which will substantially simplify further steps. Install helm following its official installation instructions.
2. Using helm, add the Percona chart repository and update the information for the available charts as follows:

```
$ helm repo add percona https://percona-charts.storage.googleapis.com
$ helm repo update
```

3. Use helm to install PMM Server:

```
$ helm install percona/pmm-server --name monitoring --set platform=openshift --
↪set credentials.username=clusterMonitor --set "credentials.
↪password=clusterMonitor123456"
```

   You must specify the correct options in the installation command:

   - `platform` should be either `kubernetes` or `openshift` depending on which platform are you using.
   - `name` should correspond to the `serverHost` key in the `pmm` section of the deploy/cr.yaml file with a "-service" suffix, the default `--name monitoring` part of the command corresponds to a `monitoring-service` value of the `serverHost` key.
   - `credentials.username` should correspond to the `MONGODB_CLUSTER_MONITOR_USER` base64 decoded value of key in the the deploy/secrets.yaml file.
   - `credentials.password` should correspond to a value of the `MONGODB_CLUSTER_MONITOR_PASSWORD` base64 decoded value of key specified in deploy/secrets.yaml secrets file. Note - the password specified in this example is the default development mode password and is not intended to be used on production systems.

4. You must edit and update the `pmm` section in the deploy/cr.yaml file.

   - set `pmm.enabled=true`
   - ensure the `serverHost` (the PMM service name is `monitoring-service` by default) is the same as value specified for the `name` parameter on the previous step, but with an added additional `-service` suffix.
   - make sure the `PMM_USER` and `PMM_PASSWORD` keys in the deploy/secrets.yaml secrets file are the same as base64 decoded equivalent values specified for the `credentials.username` and `credentials.`

`password` parameters on the previous step (if not, fix the value and apply with the `kubectl apply -f deploy/secrets.yaml` command).

When done, apply the edited `deploy/cr.yaml` file:

```
$ kubectl apply -f deploy/cr.yaml
```

5. Check that correspondent Pods are not in a cycle of stopping and restarting. This cycle occurs if there are errors on the previous steps:

```
$ kubectl get pods
$ kubectl logs my-cluster-name-rs0-0 -c pmm-client
```

6. Run the following command:

   `kubectl get service/monitoring-service -o wide`

   In the results, locate the the `EXTERNAL-IP` field. The external-ip address can be used to access PMM via *https* in a web browser, with the login/password authentication, and the browser is configured to show Percona Server for MongoDB metrics.

CHAPTER

NINE

USE DOCKER IMAGES FROM A CUSTOM REGISTRY

Using images from a private Docker registry may required for privacy, security or other reasons. In these cases, Percona Server for MongoDB Operator allows the use of a custom registry This following example of the Operator deployed in the OpenShift environment demonstrates the process:

1. Log into the OpenShift and create a project.

```
$ oc login
Authentication required for https://192.168.1.100:8443 (openshift)
Username: admin
Password:
Login successful.
$ oc new-project psmdb
Now using project "psmdb" on server "https://192.168.1.100:8443".
```

2. You need obtain the following objects to configure your custom registry access:

   - A user token
   - the registry IP address

   You can view the token with the following command:

```
$ oc whoami -t
ADO8CqCDappWR4hxjfDqwijEHei31yXAvWg61Jg210s
```

   The following command returns the registry IP address:

```
$ kubectl get services/docker-registry -n default
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
docker-registry   ClusterIP   172.30.162.173   <none>        5000/TCP   1d
```

3. Use the user token and the registry IP address to login to the registry:

```
$ docker login -u admin -p ADO8CqCDappWR4hxjfDqwijEHei31yXAvWg61Jg210s 172.30.162.
↪173:5000
Login Succeeded
```

4. Use the Docker commands to pull the needed image by its SHA digest:

```
$ docker pull docker.io/perconalab/percona-server-mongodb-
↪operator@sha256:69c935ac93d448db76f257965470367683202f725f50d6054eae1c3d2e731b9a
Trying to pull repository docker.io/perconalab/percona-server-mongodb-operator ...
sha256:69c935ac93d448db76f257965470367683202f725f50d6054eae1c3d2e731b9a: Pulling␣
↪from docker.io/perconalab/percona-server-mongodb-operator
Digest: sha256:69c935ac93d448db76f257965470367683202f725f50d6054eae1c3d2e731b9a
Status: Image is up to date for docker.io/perconalab/percona-server-mongodb-
↪operator@sha256:69c935ac93d448db76f257965470367683202f725f50d6054eae1c3d2e731b9a
```

5. The following method can push an image to the custom registry for the example OpenShift PSMDB project:

```
$ docker tag \
    docker.io/perconalab/percona-server-mongodb-
↪operator@sha256:69c935ac93d448db76f25796547036768320 2f725f50d6054eae1c3d2e731b9a␣
↪\
    172.30.162.173:5000/psmdb/percona-server-mongodb-operator:0.2.1-mongod3.6
$ docker push 172.30.162.173:5000/psmdb/percona-server-mongodb-operator:0.2.1-
↪mongod3.6
```

6. Verify the image is available in the OpenShift registry with the following command:

```
$ oc get is
NAME                                DOCKER REPO                                                ␣
↪                           TAGS              UPDATED
percona-server-mongodb-operator   docker-registry.default.svc:5000/psmdb/percona-
↪server-mongodb-operator  0.2.1-mongod3.6  2 hours ago
```

7. When the custom registry image is available, edit the the `image:` option in `deploy/operator.yaml` configuration file with a Docker Repo + Tag string (it should look like‘‘docker-registry.default.svc:5000/psmdb/percona-server-mongodb-operator:0.2.1-mongod3.6‘‘)

Note: If the registry requires authentication, you can specify the `imagePullSecrets` option for all images.

8. Repeat steps 3-5 for other images, and update corresponding options in the `deploy/cr.yaml` file.
9. Now follow the standard Percona Server for MongoDB Operator installation instruction.

Percona certified images

Following table presents Percona's certified images to be used with the Percona Server for MongoDB Operator:

Image	Digest
percona/percona-server-mongodb-operator:0.3.0	69d2018790ed14de1a79bef1fd7afc5fb91b57374f1e4ca33e5f48996646bb3e
percona/percona-server-mongodb-operator:0.3.0-mongod3.6.10	a02a10c9e0bc36fac2b1a7e1215832c5816abfbbe0018fca61d133835140b4e8
percona/percona-server-mongodb-operator:0.3.0-mongod4.0.6	0849fee6073e85414ca36d4f394046342d623292f03e9d3afd5bd5b02e6df812
percona/percona-server-mongodb-operator:0.3.0-backup	5a32ddf1194d862b5f6f3826fa85cc4f3c367ccd8e69e501f27b6bf94f7e3917
perconalab/pmm-client:1.17.1	f762cda2eda9ef17bfd1242ede70ee72595611511d8d0c5c46931ecbc968e9af

Part III

Configuration

CHAPTER

TEN

USERS

During installation, the Operator requires Kubernetes Secrets to be deployed before the Operator is started. The name of the required secrets can be set in `deploy/cr.yaml` under the `spec.secrets` section.

Unprivileged users

There are no unprivileged (general purpose) user accounts created by default. If you need general purpose users, please run commands below:

```
$ kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:3.6␣
↪--restart=Never -- bash -il
mongodb@percona-client:/$ mongo "mongodb+srv://userAdmin:userAdmin123456@my-cluster-
↪name-rs0.psmdb.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
rs0:PRIMARY> db.createUser({
    user: "myApp",
    pwd: "myAppPassword",
    roles: [
      { db: "myApp", role: "readWrite" }
    ],
    mechanisms: [
       "SCRAM-SHA-1"
    ]

})
```

Now check the newly created user:

```
$ kubectl run -i --rm --tty percona-client --image=percona/percona-server-mongodb:3.6␣
↪--restart=Never -- bash -il
mongodb@percona-client:/$ mongo "mongodb+srv://myApp:myAppPassword@my-cluster-name-
↪rs0.psmdb.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
rs0:PRIMARY> use myApp
rs0:PRIMARY> db.test.insert({ x: 1 })
rs0:PRIMARY> db.test.findOne()
```

MongoDB System Users

Default Secret name: `my-cluster-name-mongodb-users`

Secret name field: `spec.secrets.users`

The operator requires system-level MongoDB users to automate the MongoDB deployment.

Warning: These users should not be used to run an application.

User Purpose	Username Secret Key	Password Secret Key
Backup/Restore	MONGODB_BACKUP_USER	MONGODB_BACKUP_PASSWORD
Cluster Admin	MONGODB_CLUSTER_ADMIN_USER	MONGODB_CLUSTER_ADMIN_PASSWORD
Cluster Monitor	MONGODB_CLUSTER_MONITOR_USER	MONGODB_CLUSTER_MONITOR_PASSWORD
User Admin	MONGODB_USER_ADMIN_USER	MONGODB_USER_ADMIN_PASSWORD

Backup/Restore - MongoDB Role: backup, clusterMonitor, restore

Cluster Admin - MongoDB Role: clusterAdmin

Cluster Monitor - MongoDB Role: clusterMonitor

User Admin - MongoDB Role: userAdmin

Development Mode

To make development and testing easier, `deploy/mongodb-users.yaml` secrets file contains default passwords for MongoDB system users.

The development-mode credentials from `deploy/mongodb-users.yaml` are:

Secret Key	Secret Value
MONGODB_BACKUP_USER	backup
MONGODB_BACKUP_PASSWORD	backup123456
MONGODB_CLUSTER_ADMIN_USER	clusterAdmin
MONGODB_CLUSTER_ADMIN_PASSWORD	clusterAdmin123456
MONGODB_CLUSTER_MONITOR_USER	clusterMonitor
MONGODB_CLUSTER_MONITOR_PASSWORD	clusterMonitor123456
MONGODB_USER_ADMIN_USER	userAdmin
MONGODB_USER_ADMIN_PASSWORD	userAdmin123456

Warning: Do not use the default MongoDB Users in production!

MongoDB Internal Authentication Key (optional)

Default Secret name: `my-cluster-name-mongodb-key`

Secret name field: `spec.secrets.key`

By default, the operator will create a random, 1024-byte key for MongoDB Internal Authentication if it does not already exist. If you would like to deploy a different key, create the secret manually before starting the operator.

CHAPTER

ELEVEN

CUSTOM RESOURCE OPTIONS

The operator is configured via the spec section of the deploy/cr.yaml file. This file contains the following spec sections:

Key	Value type	Default	Description
platform	string	kubernetes	Override/set the Kubernetes platform: *kubernetes* or *openshift*. Set openshift on OpenShift 3.11+
version	string	`3.6.8`	The Dockerhub tag of percona/percona-server-mongodb to deploy
secrets	subdoc		Operator secrets section
replsets	array		Operator MongoDB Replica Set section
pmm	subdoc		Percona Monitoring and Management section
mongod	subdoc		Operator MongoDB Mongod configuration section
backup	subdoc		Percona Server for MongoDB backups section

Secrets section

Each spec in its turn may contain some key-value pairs. The secrets one has only two of them:

Key	Value Type	Example	Description
key	string	`my-cluster-name-mongodb-key`	The secret name for the MongoDB Internal Auth Key. This secret is auto-created by the operator if it doesn't exist.
users	string	`my-cluster-name-mongodb-users`	The secret name for the MongoDB users required to run the operator. **This secret is required to run the operator.**

Replsets section

The replsets section controls the MongoDB Replica Set.

Key	Value Type	Example	De
name	string	`rs 0`	Th
size	int	3	Th
storageClass	string		Se
arbiter.enabled	boolean	`f`	En
arbiter.size	int		Th

Table 11.1 – continu

Key	Value Type	Example	De
arbiter.afinity.antiAffinityTopologyKey	string	`kubernetes.io/hostname`	Th
arbiter.tolerations.key	string	`node.alpha.kubernetes.io/unreachable`	Th
arbiter.tolerations.operator	string	`Exists`	Th
arbiter.tolerations.effect	string	`NoExecute`	Th
arbiter.tolerations.tolerationSeconds	int	`6000`	Th
arbiter.priorityClassName	string	`high priority`	Th
arbiter.annotations.iam.amazonaws.com/role	string	`role-arn`	Th
arbiter.labels	label	`rack: rack-22`	Th
arbiter.nodeSelector	label	`disktype:ssd`	Th
expose.enabled	boolean	`false`	En
expose.exposeType	string	`ClusterIP`	the
resources.limits.cpu	string		Ku
resources.limits.memory	string		Ku
resources.limits.storage	string		Ku
resources.requests.cpu	string		Th
resources.requests.memory	string		Th
volumeSpec.emptyDir	string	`{}`	Th
volumeSpec.hostPath.path	string	`/data`	Ku
volumeSpec.hostPath.type	string	`Directory`	Th
volumeSpec.persistentVolumeClaim.storageClassName	string	`standard`	Th
volumeSpec.persistentVolumeClaim.accessModes	array	`[ "ReadWriteOnce" ]`	Th
volumeSpec.resources.requests.storage	string	`3Gi`	Th
affinity.antiAffinityTopologyKey	string	`kubernetes.io/hostname`	Th
tolerations.key	string	`node.alpha.kubernetes.io/unreachable`	Th
tolerations.operator	string	`Exists`	Th
tolerations.effect	string	`NoExecute`	Th
tolerations.tolerationSeconds	int	`6000`	Th
annotations.iam.amazonaws.com/role	string	`role-arn`	Th
labels	label	`rack: rack-22`	Th
nodeSelector	label	`disktype: ssd`	Th
podDisruptionBudget.maxUnavailable	int	`1`	Th
podDisruptionBudget.minAvailable	int	`1`	Th

PMM Section

The `pmm` section in the deploy/cr.yaml file contains configuration options for Percona Monitoring and Management.

Key	Value Type	Example	Description
enabled	boolean	`false`	Enables or disables monitoring Percona Server for MongoDB with PMM
image	string	`perconalab/pmm-client:1.17.1`	PMM Client docker image to use
serverHost	string	`monitoring-service`	Address of the PMM Server to collect data from the Cluster

Mongod Section

The largest section in the deploy/cr.yaml file contains the Mongod configuration options.

backup section

The `backup` section in the deploy/cr.yaml file contains the following configuration options for the regular Percona Server for MongoDB backups.

Key	Value Type	Example	Description
annotations.iam.amazonaws.com/role	string	`role-arn`	The AWS IAM role for the backup storage nodes
labels	label	`rack: rack-22`	The Kubernetes affinity labels for the backup storage nodes
nodeSelector	label	`disktype: ssd`	The Kubernetes nodeSelector affinity constraint for the backup storage nodes
coordinator.requests.storage	string	`1Gi`	The Kubernetes Persistent Volume size for the MongoDB Coordinator container
coordinator.requests.storageClass	string	`aws-gp2`	Sets the Kubernetes Storage Class to use with the MongoDB Coordinator container
coordinator.debug	string	`false`	Enables or disables debug mode for the MongoDB Coordinator operation
tasks.name	string	`sat-night-backup`	The backup name
tasks.enabled	boolean	`true`	Enables or disables this exact backup
tasks.schedule	string	`0 0 * * 6`	The scheduled time to make a backup, specified in the crontab format
tasks.storageName	string	`st-us-west`	The name of the S3-compatible storage for backups, configured in the *storages* subsection
tasks.compressionType	string	`gzip`	The backup compression format

CHAPTER

TWELVE

PROVIDING BACKUPS

Percona Server for MongoDB Operator allows taking cluster backup in two ways. *Scheduled backups* are configured in the deploy/cr.yaml file to be executed automatically at the selected time. *On-demand backups* can be done manually at any moment. Both ways use the Percona Backup for MongoDB tool.

The backup process is controlled by the Backup Coordinator daemon residing in the Kubernetes cluster alongside the Percona Server for MongoDB, while actual backup images are stored separately on any Amazon S3 or S3-compatible storage.

Making scheduled backups

Since backups are stored separately on the Amazon S3, a secret with `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` should be present on the Kubernetes cluster. These keys should be saved to the deploy/backup-s3.yaml file and applied with the appropriate command, e.g. `kubectl apply -f deploy/backup-s3.yaml` (for Kubernetes).

A backup schedule is defined in the `backup` section of the deploy/cr.yaml file. This section contains three subsections:

- `storages` contains data needed to access the S3-compatible cloud to store backups.
- `coordinator` configures the Kubernetes limits and claims for the Percona Backup for MongoDB Coordinator daemon.
- `tasks` schedules backups (the schedule is specified in crontab format).

This example uses Amazon S3 storage for backups:

```
...
backup:
  enabled: true
  version: 0.3.0
  ...
  storages:
    s3-us-west:
      type: s3
      s3:
        bucket: S3-BACKUP-BUCKET-NAME-HERE
        region: us-west-2
        credentialsSecret: my-cluster-name-backup-s3
  tasks:
   - name: daily-s3-us-west
     enabled: true
     schedule: "0 0 * * *"
     storageName: s3-us-west
```

```
    compressionType: gzip
  ...
```

Note: If you use some S3-compatible storage instead of the original Amazon S3, one more key is needed in the `s3` subsection: the `endpointUrl`, which points to the actual cloud used for backups and is specific to the cloud provider.

For example, the Google Cloud key is the following:

```
endpointUrl: https://storage.googleapis.com
```

The options within these three subsections are further explained in the Operator Options.

One option which should be mentioned separately is `credentialsSecret` which is a Kubernetes secret for backups. The Sample backup-s3.yaml can be used as a template to create this secret object. Verify that the yaml contains the proper `name` value which must be equal to the one specified for `credentialsSecret`, i.e. `my-cluster-name-backup-s3` for example, and also proper `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` keys. After editing is finished, secrets object should be created or updated using the following command:

```
$ kubectl apply -f deploy/backup-s3.yaml
```

Making on-demand backup

To make on-demand backup, user should run the PBM Control tool inside of the coordinator container, supplying it with needed options, like in the following example:

```
kubectl run -it --rm pbmctl --image=percona/percona-server-mongodb-operator:0.3.0-
↪backup-pbmctl --restart=Never -- \
  run backup \
  --server-address=<cluster-name>-backup-coordinator:10001 \
  --storage <storage> \
  --compression-algorithm=gzip \
  --description=my-backup
```

Don't forget to specify the name of your cluster instead of the `<cluster-name>` part of the Backup Coordinator URL (the same cluster name is specified in the deploy/cr.yaml file). Also the `<storage>` placeholder should be substituted with the storage name, which is located in the `backups\storages` subsection in deploy/cr.yaml file.

Restore the cluster from a previously saved backup

To restore a previously saved backup you must specify the backup name. A list of the available backups can be obtained from the Backup Coordinator as follows (you must use the correct Backup Coordinator's URL and the correct storage name for your environment):

```
kubectl run -it --rm pbmctl --image=percona/percona-server-mongodb-operator:0.3.0-
↪backup-pbmctl --restart=Never -- list backups --server-address=<cluster-name>-
↪backup-coordinator:10001
```

Now, restore the backup, substituting the cluster-name and storage values and using the selected backup name instead of `backup-name`:

```
kubectl run -it --rm pbmctl --image=percona/percona-server-mongodb-operator:0.3.0-
↪backup-pbmctl --restart=Never -- \
  run restore \
  --server-address=<cluster-name>-backup-coordinator:10001 \
  --storage <storage> \
  backup-name
```

CHAPTER

THIRTEEN

CREATING A PRIVATE S3-COMPATIBLE CLOUD FOR BACKUPS

As it is mentioned in backups any cloud storage which implements the S3 API can be used for backups. The one way to setup and implement the S3 API storage on Kubernetes or OpenShift is Minio - the S3-compatible object storage server deployed via Docker on your own infrastructure.

Setting up Minio to be used with Percona Server for MongoDB Operator backups involves following steps:

1. Install Minio in your Kubernetes or OpenShift environment and create the correspondent Kubernetes Service as follows:

```
helm install \
  --name minio-service \
  --set accessKey=some-access-key \
  --set secretKey=some-secret-key \
  --set service.type=ClusterIP \
  --set configPath=/tmp/.minio/ \
  --set persistence.size=2G \
  --set environment.MINIO_REGION=us-east-1 \
  stable/minio
```

   Don't forget to substitute default `some-access-key` and `some-secret-key` strings in this command with actual unique key values. The values can be used later for access control. The `storageClass` option is needed if you are using the special Kubernetes Storage Class for backups. Otherwise, this setting may be omitted. You may also notice the `MINIO_REGION` value which is may not be used within a private cloud. Use the same region value here and on later steps (`us-east-1` is a good default choice).

2. Create an S3 bucket for backups:

```
kubectl run -i --rm aws-cli --image=perconalab/awscli --restart=Never -- \
 bash -c 'AWS_ACCESS_KEY_ID=some-access-key \
 AWS_SECRET_ACCESS_KEY=some-secret-key \
 AWS_DEFAULT_REGION=us-east-1 \
  /usr/bin/aws \
  --endpoint-url http://minio-service:9000 \
  s3 mb s3://operator-testing'
```

   This command creates the bucket named `operator-testing` with the selected access and secret keys (substitute `some-access-key` and `some-secret-key` with the values used on the previous step).

3. Now edit the backup section of the deploy/cr.yaml file to set proper values for the `bucket` (the S3 bucket for backups created on the previous step), `region`, `credentialsSecret` and the `endpointUrl` (which should point to the previously created Minio Service).

```
...
backup:
  enabled: true
```

```
  version: 0.3.0
  ...
  storages:
    minio:
      type: s3
      s3:
        bucket: operator-testing
        region: us-east-1
        credentialsSecret: my-cluster-name-backup-minio
        endpointUrl: http://minio-service:9000
  ...
```

The option which should be specially mentioned is `credentialsSecret` which is a Kubernetes secret for backups. Sample backup-s3.yaml can be used to create this secret object. Check that the object contains the proper `name` value and is equal to the one specified for `credentialsSecret`, i.e. `my-cluster-name-backup-minio` in the backup to Minio example, and also contains the proper `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` keys. After you have finished editing the file, the secrets object are created or updated when you run the following command:

```
$ kubectl apply -f deploy/backup-s3.yaml
```

4. When the setup process is completed, making the backup is based on a script. Following example illustrates how to make an on-demand backup:

```
kubectl run -it --rm pbmctl --image=percona/percona-server-mongodb-operator:0.3.0-
↪backup-pbmctl --restart=Never -- \
  run backup \
  --server-address=<cluster-name>-backup-coordinator:10001 \
  --storage <storage> \
  --compression-algorithm=gzip \
  --description=my-backup
```

Don't forget to specify the name of your cluster instead of the `<cluster-name>` part of the Backup Coordinator URL (the cluster name is specified in the deploy/cr.yaml file). Also substitute `<storage>` with the actual storage name located in a subsection inside of the `backups` in the deploy/cr.yaml file. In the earlier example this value is `minio`.

5. To restore a previously saved backup you must specify the backup name. With the proper Backup Coordinator URL and storage name, you can obtain a list of the available backups:

```
kubectl run -it --rm pbmctl --image=percona/percona-server-mongodb-operator:0.3.0-
↪backup-pbmctl --restart=Never -- list backups --server-address=<cluster-name>-
↪backup-coordinator:10001
```

Now, restore the backup, using backup name instead of the `backup-name` parameter:

```
kubectl run -it --rm pbmctl --image=percona/percona-server-mongodb-operator:0.3.0-
↪backup-pbmctl --restart=Never -- \
  run restore \
  --server-address=<cluster-name>-backup-coordinator:10001 \
  --storage <storage> \
  backup-name
```

CHAPTER

FOURTEEN

ENABLING REPLICA SET ARBITER NODES

Percona Server for MongoDB replication model is based on elections, when nodes of the Replica Set choose which node becomes the primary node. Elections are the reason to avoid an even number of nodes in the cluster. The cluster should have at least three nodes. Normally, each node stores a complete copy of the data, but there is also a possibility, to reduce disk IO and space used by the database, to add an arbiter node. An arbiter cannot become a primary and does not have a complete copy of the data. The arbiter does have one election vote and can be the odd number for elections. The arbiter does not demand a persistent volume.

Percona Server for MongoDB Operator has the ability to create Replica Set Arbiter nodes if needed. This feature can be configured in the Replica Set section of the deploy/cr.yaml file:

- set `arbiter.enabled` option to `true` to allow Arbiter nodes,
- use `arbiter.size` option to set the desired amount of the Replica Set nodes which should be Arbiter ones instead of containing data.

CHAPTER

FIFTEEN

EXPOSING CLUSTER NODES WITH DEDICATED IP ADDRESSES

When Kubernetes creates Pods, each Pod has an IP address in the internal virtual network of the cluster. Creating and destroying Pods is a dynamic process, therefore binding communication between Pods to specific IP addresses would cause problems as things changes over time as a result of the cluster scaling, maintenance, etc.. Due to this changing environment, you should connect to Percona Server for MongoDB via Kubernetes internal DNS names in URI (e.g. `mongodb+srv://userAdmin:userAdmin123456@<cluster-name>-rs0.<namespace>.svc.cluster.local/admin?replicaSet=rs0&ssl=false`). It is strictly recommended.

Sometimes you cannot communicate to the Pods using the Kubernetes internal DNS names. To make Pods of the Replica Set accessible, Percona Server for MongoDB Operator can assign a Kubernetes Service to each Pod.

This feature can be configured in the Replica Set section of the deploy/cr.yaml file:

- set 'expose.enabled' option to 'true' to allow exposing Pods via services,
- set 'expose.exposeType' option specifying the IP address type to be used:
  - `ClusterIP` - expose the Pod's service with an internal static IP address. This variant makes MongoDB Pod only reachable from within the Kubernetes cluster.
  - `NodePort` - expose the Pod's service on each Kubernetes node's IP address at a static port. ClusterIP service, to which the node port will be routed, is automatically created in this variant. As an advantage, the service will be reachable from outside the cluster by node address and port number, but the address will be bound to a specific Kubernetes node.
  - `LoadBalancer` - expose the Pod's service externally using a cloud provider's load balancer. Both ClusterIP and NodePort services are automatically created in this variant.

If this feature is enabled, URI looks like `mongodb://userAdmin:userAdmin123456@<ip1>:<port1>,<ip2>:<port2>,<ip3>:<port3>/admin?replicaSet=rs0&ssl=false` All IP adresses should be *directly* reachable by application.

CHAPTER

SIXTEEN

BINDING PERCONA SERVER FOR MONGODB COMPONENTS TO SPECIFIC KUBERNETES/OPENSHIFT NODES

The operator does a good job of automatically assigning new pods to nodes to achieve balanced distribution across the cluster. There are situations when you must ensure that pods land on specific nodes: for example, for the advantage of speed on an SSD-equipped machine, or reduce costs by choosing nodes in the same availability zone.

The appropriate (sub)sections (`replsets`, `replsets.arbiter`, and `backup`) of the deploy/cr.yaml file contain the keys which can be used to do assign pods to nodes.

Node selector

The `nodeSelector` contains one or more key-value pairs. If the node is not labeled with each key-value pair from the Pod's `nodeSelector`, the Pod will not be able to land on it.

The following example binds the Pod to any node having a self-explanatory `disktype: ssd` label:

```
nodeSelector:
  disktype: ssd
```

Affinity and anti-affinity

Affinity defines eligible pods that can be scheduled on the node which already has pods with specific labels. Anti-affinity defines pods that are not eligible. This approach is reduces costs by ensuring several pods with intensive data exchange occupy the same availability zone or even the same node or, on the contrary, to spread the pods on different nodes or even different availability zones for high availability and balancing purposes.

Percona Server for MongoDB Operator provides two approaches for doing this:

- simple way to set anti-affinity for Pods, built-in into the Operator,
- more advanced approach based on using standard Kubernetes constraints.

Simple approach - use antiAffinityTopologyKey of the Percona Server for MongoDB Operator

Percona Server for MongoDB Operator provides an `antiAffinityTopologyKey` option, which may have one of the following values:

- `kubernetes.io/hostname` - Pods will avoid residing within the same host,
- `failure-domain.beta.kubernetes.io/zone` - Pods will avoid residing within the same zone,

- `failure-domain.beta.kubernetes.io/region` - Pods will avoid residing within the same region,
- `none` - no constraints are applied.

The following example forces Percona Server for MongoDB Pods to avoid occupying the same node:

```
affinity:
  antiAffinityTopologyKey: "kubernetes.io/hostname"
```

Advanced approach - use standard Kubernetes constraints

The previous method can be used without special knowledge of the Kubernetes way of assigning Pods to specific nodes. Still, in some cases, more complex tuning may be needed. In this case, the `advanced` option placed in the deploy/cr.yaml file turns off the effect of the `antiAffinityTopologyKey` and allows the use of the standard Kubernetes affinity constraints of any complexity:

```
affinity:
   advanced:
     podAffinity:
       requiredDuringSchedulingIgnoredDuringExecution:
       - labelSelector:
           matchExpressions:
           - key: security
             operator: In
             values:
             - S1
         topologyKey: failure-domain.beta.kubernetes.io/zone
     podAntiAffinity:
       preferredDuringSchedulingIgnoredDuringExecution:
       - weight: 100
         podAffinityTerm:
           labelSelector:
             matchExpressions:
             - key: security
               operator: In
               values:
               - S2
           topologyKey: kubernetes.io/hostname
     nodeAffinity:
       requiredDuringSchedulingIgnoredDuringExecution:
         nodeSelectorTerms:
         - matchExpressions:
           - key: kubernetes.io/e2e-az-name
             operator: In
             values:
             - e2e-az1
             - e2e-az2
       preferredDuringSchedulingIgnoredDuringExecution:
       - weight: 1
         preference:
           matchExpressions:
           - key: another-node-label-key
             operator: In
             values:
             - another-node-label-value
```

See explanation of the advanced affinity options in Kubernetes documentation.

Tolerations

Tolerations allow Pods having them to be able to land onto nodes with matching *taints*. Toleration is expressed as a `key` with and `operator`, which is either `exists` or `equal` (the equal variant requires a corresponding `value` for comparison).

Toleration should have a specified `effect`, such as the following:

- `NoSchedule` - less strict
- `PreferNoSchedule`
- `NoExecute`

When a *taint* with the `NoExecute` effect is assigned to a node, any pod configured to not tolerating this *taint* is removed from the node. This removal can be immediate or after the `tolerationSeconds` interval. The following example defines this effect and the removal interval:

```
tolerations:
- key: "node.alpha.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

The Kubernetes Taints and Toleratins contains more examples on this topic.

Priority Classes

Pods may belong to some *priority classes*. This flexibility allows the scheduler to distinguish more and less important Pods when needed, such as the situation when a higher priority Pod cannot be scheduled without evicting a lower priority one. This ability can be accomplished by adding one or more PriorityClasses in your Kubernetes cluster, and specifying the `PriorityClassName` in the deploy/cr.yaml file:

```
priorityClassName: high-priority
```

See the Kubernetes Pods Priority and Preemption documentation to find out how to define and use priority classes in your cluster.

Pod Disruption Budgets

Creating the Pod Disruption Budget is the Kubernetes method to limit the number of Pods of an application that can go down simultaneously due to *voluntary disruptions* such as the cluster administrator's actions during a deployment update. Distribution Budgets allow large applications to retain their high availability during maintenance and other administrative activities. The `maxUnavailable` and `minAvailable` options in the deploy/cr.yaml file can be used to set these limits. The recommended variant is the following:

```
podDisruptionBudget:
   maxUnavailable: 1
```

CHAPTER

SEVENTEEN

LOCAL STORAGE SUPPORT FOR THE PERCONA SERVER FOR MONGODB OPERATOR

Among the wide rage of volume types, supported by Kubernetes, there are two volume types which allow Pod containers to access part of the local filesystem on the node the *emptyDir* and *hostPath*.

emptyDir

A Pod emptyDir volume is created when the Pod is assigned to a Node. The volume is initially empty and is erased when the Pod is removed from the Node. The containers in the Pod can read and write the files in the emptyDir volume.

The `emptyDir` options in the deploy/cr.yaml file can be used to turn the emptyDir volume on by setting the directory name.

The `emptyDir` is useful when you use Percona Memory Engine.

hostPath

A hostPath volume mounts an existing file or directory from the host node's filesystem into the Pod. If the pod is removed, the data persists in the host node's filesystem.

The `volumeSpec.hostPath` subsection in the deploy/cr.yaml file may include `path` and `type` keys to set the node's filesystem object path and to specify whether it is a file, a directory, or something else (e.g. a socket):

```
volumeSpec:
  hostPath:
    path: /data
    type: Directory
```

Please note, you must created the hostPath manually and should have following attributes:

- access permissions
- ownership
- SELinux security context

The `hostPath` volume is useful when you perform manual actions during the first run and require improved disk performance. Consider using the tolerations settings to avoid a cluster migration to different hardware in case of a reboot or a hardware failure.

More details can be found in the official hostPath Kubernetes documentation.

CHAPTER

EIGHTEEN

TRANSPORT LAYER SECURITY (TLS)

The Percona Kubernetes Operator for PSMDB uses Transport Layer Security (TLS) cryptographic protocol for the following types of communication:

- Internal - communication between PSMDB instances in the cluster
- External - communication between the client application and the cluster

The internal certificate is also used as an authorization method.

TLS security can be configured in two ways: Percona Server for MongoDB Operator can use a *cert-manager* for automatic certificates generation, but also supports manual certificates generation. The following subsections cover these two ways to configure TLS security with the Operator, as well as explains how to temporarily disable it if needed.

- *Install and use the* cert-manager
  - *About the* cert-manager
  - *Installation of the* cert-manager
- *Generate certificates manually*
- *Run PSMDB without TLS*

Install and use the *cert-manager*

About the *cert-manager*

A *cert-manager* is a Kubernetes certificate management controller which widely used to automate the management and issuance of TLS certificates. It is community-driven, and open source.

When you have already installed *cert-manager* and deploy the operator, the operator requests a certificate from the *cert-manager*. The *cert-manager* acts as a self-signed issuer and generates certificates. The Percona Operator self-signed issuer is local to the operator namespace. This self-signed issuer is created because PSMDB requires all certificates are issued by the same CA.

The creation of the self-signed issuer allows you to deploy and use the Percona Operator without creating a clusterissuer separately.

Installation of the *cert-manager*

The steps to install the *cert-manager* are the following:

- Create a namespace
- Disable resource validations on the cert-manager namespace
- Install the cert-manager.

The following commands perform all the needed actions:

```
kubectl create namespace cert-manager
kubectl label namespace cert-manager certmanager.k8s.io/disable-validation=true
kubectl apply -f https://raw.githubusercontent.com/jetstack/cert-manager/release-0.7/
↪deploy/manifests/cert-manager.yaml
```

After the installation, you can verify the *cert-manager* by running the following command:

```
kubectl get pods -n cert-manager
```

The result should display the *cert-manager* and webhook active and running.

Generate certificates manually

To generate certificates manually, follow these steps:

1. Provision a Certificate Authority (CA) to generate TLS certificates
2. Generate a CA key and certificate file with the server details
3. Create the server TLS certificates using the CA keys, certs, and server details

The set of commands generate certificates with the following attributes:

- `Server-pem` - Certificate
- `Server-key.pem` - the private key
- `ca.pem` - Certificate Authority

You should generate certificates twice: one set is for external communications, and another set is for internal ones. A secret created for the external use must be added to `cr.yaml/spec/secretsName`. A certificate generated for internal communications must be added to the `cr.yaml/spec/sslInternalSecretName`.

Supposing that your cluster name is `my-cluster-name-rs0`, the instructions to generate certificates manually are as follows:

```
CLUSTER_NAME=my-cluster-name-rs0
cat <<EOF | cfssl gencert -initca - | cfssljson -bare ca
  {
    "CN": "Root CA",
    "key": {
      "algo": "rsa",
      "size": 2048
    }
  }
EOF

cat <<EOF > ca-config.json
  {
    "signing": {
      "default": {
        "expiry": "87600h",
```

```
        "usages": ["signing", "key encipherment", "server auth", "client auth"]
      }
    }
  }
EOF

cat <<EOF | cfssl gencert -ca=ca.pem  -ca-key=ca-key.pem -config=./ca-config.json - |␣
↪cfssljson -bare server
  {
    "hosts": [
      "${CLUSTER_NAME}",
      "*.${CLUSTER_NAME}"
    ],
    "CN": "${CLUSTER_NAME/-rs0}",
    "key": {
      "algo": "rsa",
      "size": 2048
    }
  }
EOF
cfssl bundle -ca-bundle=ca.pem -cert=server.pem | cfssljson -bare server

kubectl create secret generic my-cluster-name-ssl-internal --from-file=tls.crt=server.
↪pem --from-file=tls.key=server-key.pem --from-file=ca.crt=ca.pem --type=kubernetes.
↪io/tls

cat <<EOF | cfssl gencert -ca=ca.pem  -ca-key=ca-key.pem -config=./ca-config.json - |␣
↪cfssljson -bare client
  {
    "hosts": [
      "${CLUSTER_NAME}",
      "*.${CLUSTER_NAME}"
    ],
    "CN": "${CLUSTER_NAME/-rs0}",
    "key": {
      "algo": "rsa",
      "size": 2048
    }
  }
EOF

kubectl create secret generic my-cluster-name-ssl --from-file=tls.crt=client.pem --
↪from-file=tls.key=client-key.pem --from-file=ca.crt=ca.pem --type=kubernetes.io/tls
```

Run PSMDB without TLS

Omitting TLS is also possible, but we recommend that you run your cluster with the TLS protocol enabled.

TLS protocol can be disabled (e.g. for demonstration purposes) by editing the `cr.yaml/spec/allowUnstafeConfigurations` setting to `true`.

CHAPTER

NINETEEN

DATA AT REST ENCRYPTION

Data at rest encryption in Percona Server for MongoDB is supported by the Operator since version 1.1.0.

..note:: “Data at rest” means inactive data stored as files, database records, etc.

Following options the `mongod` section of the `deploy/cr.yaml` file should be edited to turn this feature on:

1. The `security.enableEncryption` key should be set to `true` (the default value).
2. The `security.encryptionCipherMode` key should specify proper cipher mode for decryption. The value can be one of the following two variants: * `AES256-CBC` (the default one for the Operator and Percona Server for

   MongoDB)

   - `AES256-GCM`
3. `security.encryptionKeySecret` should specify a secret object with the encryption key:

   ```
   mongod:
     ...
     security:
       ...
       encryptionKeySecret: my-cluster-name-mongodb-encryption-key
   ```

   Encryption key secret will be created automatically if it doesn’t exist. If you would like to create it yourself, take into account that the key must be a 32 character string encoded in base64.

Part IV

Reference

CHAPTER

TWENTY

KUBERNETES FOR PERCONA SERVER FOR MONGODB 1.1.0 RELEASE NOTES

Percona Kubernetes Operator for Percona Server for MongoDB 1.1.0

Percona announces the general availability of *Percona Kubernetes Operator for Percona Server for MongoDB* 1.1.0 on July 15, 2019. This release is now the current GA release in the 1.1 series. Install the Kubernetes Operator for Percona Server for MongoDB by following the instructions. Please see the GA release announcement.

The Operator simplifies the deployment and management of the Percona Server for MongoDB in Kubernetes-based environments. It extends the Kubernetes API with a new custom resource for deploying, configuring and managing the application through the whole life cycle.

The Operator source code is available in our Github repository. All of Percona's software is open-source and free.

New features and improvements:

- Now the Percona Kubernetes Operator allows upgrading Percona Server for MongoDB to newer versions, either in semi-automatic or in manual mode.
- Also, two modes are implemented for updating the Percona Server for MongoDB `mongod.conf` configuration file: in *automatic configuration update* mode Percona Server for MongoDB Pods are immediately re-created to populate changed options from the Operator YAML file, while in *manual mode* changes are held until Percona Server for MongoDB Pods are re-created manually.
- Percona Server for MongoDB data-at-rest encryption is now supported by the Operator to ensure that encrypted data files cannot be decrypted by anyone except those with the decryption key.
- A separate service account is now used by the Operator's containers which need special privileges, and all other Pods run on default service account with limited permissions.
- User secrets are now generated automatically if don't exist: this feature especially helps reduce work in repeated development environment testing and reduces the chance of accidentally pushing predefined development passwords to production environments.
- The Operator is now able to generate TLS certificates itself which removes the need in manual certificate generation.
- The list of officially supported platforms now includes the Minikube, which provides an easy way to test the Operator locally on your own machine before deploying it on a cloud.
- Also, Google Kubernetes Engine 1.14 and OpenShift Platform 4.1 are now supported.

Percona Server for MongoDB is an enhanced, open source and highly-scalable database that is a fully-compatible, drop-in replacement for MongoDB Community Edition. It supports MongoDB protocols and drivers. Percona Server for MongoDB extends MongoDB Community Edition functionality by including the Percona Memory Engine, as well as several enterprise-grade features. It requires no changes to MongoDB applications or code.

Help us improve our software quality by reporting any bugs you encounter using our bug tracking system.

Percona Kubernetes Operator for Percona Server for MongoDB 1.0.0

Percona announces the general availability of *Percona Kubernetes Operator for Percona Server for MongoDB* 1.0.0 on May 29, 2019. This release is now the current GA release in the 1.0 series. Install the Kubernetes Operator for Percona Server for MongoDB by following the instructions. Please see the GA release announcement. All of Percona's software is open-source and free.

The Percona Kubernetes Operator for Percona Server for MongoDB automates the lifecycle of your Percona Server for MongoDB environment. The Operator can be used to create a Percona Server for MongoDB replica set, or scale an existing replica set.

The Operator creates a Percona Server for MongoDB replica set with the needed settings and provides a consistent Percona Server for MongoDB instance. The Percona Kubernetes Operators are based on best practices for configuration and setup of the Percona Server for MongoDB.

The Kubernetes Operators provide a consistent way to package, deploy, manage, and perform a backup and a restore for a Kubernetes application. Operators deliver automation advantages in cloud-native applications and may save time while providing a consistent environment.

The advantages are the following:

- Deploy a Percona Server for MongoDB environment with no single point of failure and environment can span multiple availability zones (AZs).
- Deployment takes about six minutes with the default configuration.
- Modify the Percona Server for MongoDB size parameter to add or remove Percona Server for MongoDB replica set members
- Integrate with Percona Monitoring and Management (PMM) to seamlessly monitor your Percona Server for MongoDB
- Automate backups or perform on-demand backups as needed with support for performing an automatic restore
- Supports using Cloud storage with S3-compatible APIs for backups
- Automate the recovery from failure of a Percona Server for MongoDB replica set member
- TLS is enabled by default for replication and client traffic using Cert-Manager
- Access private registries to enhance security
- Supports advanced Kubernetes features such as pod disruption budgets, node selector, constraints, tolerations, priority classes, and affinity/anti-affinity
- You can use either PersistentVolumeClaims or local storage with hostPath to store your database
- Supports a replica set Arbiter member
- Supports Percona Server for MongoDB versions 3.6 and 4.0

Installation

Installation is performed by following the documentation installation instructions for Kubernetes and OpenShift.

INDEX

Symbols

1.0.0 (release notes), 60
1.1.0 (release notes), 59