PERCONA

Percona Kubernetes for Percona Server for MongoDB Documentation

Release 1.4.0

Percona LLC and/or its affiliates 2009-2020

Mar 31, 2020

CONTENTS

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.
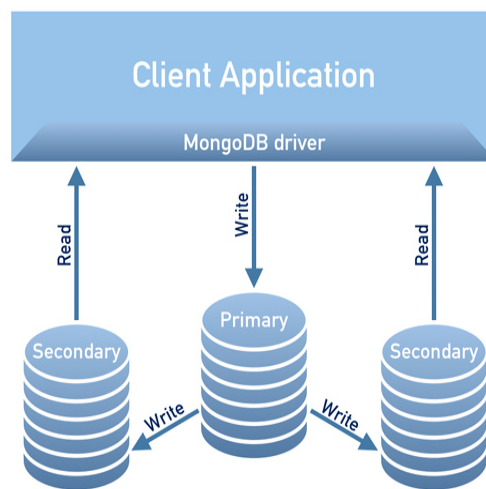
Part I

Overview

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.
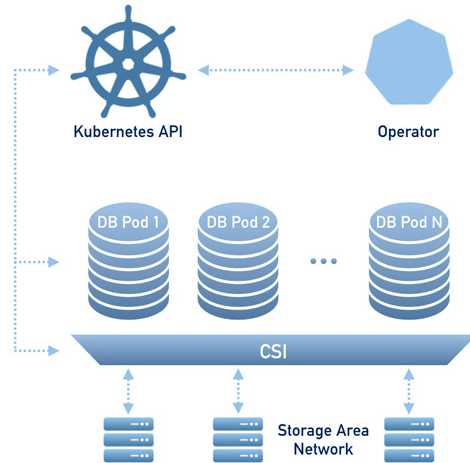
To provide data storage for stateful applications, Kubernetes uses Persistent Volumes. A *PersistentVolumeClaim* (PVC) is used to implement the automatic storage provisioning to pods. If a failure occurs, the Container Storage Interface (CSI) should be able to re-mount storage on a different node. The 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.

SYSTEM REQUIREMENTS

The Operator was developed and tested with Percona Server for MongoDB 3.6, 4.0, and 4.2. Other options may or may not work.

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

Officially supported platforms

The following platforms are supported:

- OpenShift 3.11
- OpenShift 4.2
- Google Kubernetes Engine (GKE) 1.13
- GKE 1.15
- Amazon Elastic Container Service for Kubernetes (EKS) 1.15
- Minikube 1.16

Other Kubernetes platforms may also work but have not been tested.

Resource Limits

A cluster running an officially supported platform contains at least three Nodes, with the following resources:

- 2GB of RAM,
- 2 CPU threads per Node for Pods provisioning,
- at least 60GB of available storage for Private Volumes provisioning.

Note: Use Storage Class with XFS as the default filesystem if possible

to achieve better MongoDB performance.

Platform-specific limitations

The Operator is subsequent to specific platform limitations.

- Minikube doesn't support multi-node cluster configurations because of its local nature, which is in collision with the default affinity requirements of the Operator. To arrange this, the *Install Percona Server for MongoDB on Minikube* instruction includes an additional step which turns off the requirement of having not less than three Nodes.

Part II

Installation

INSTALL PERCONA SERVER FOR MONGODB ON KUBERNETES

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

```
git clone -b 1.4.0 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. These secrets should be placed in the data section of the `deploy/secrets.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. Now certificates should be generated. By default, the Operator generates certificates automatically, and no actions are required at this step. Still, you can generate and apply your own certificates as secrets according to the *TLS instructions*.

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
```

8. Check connectivity to newly created cluster

```
$ kubectl run -i --rm --tty percona-client --image=percona/percona-server-
→mongodb:4.0 --restart=Never -- bash -il
percona-client:/$ mongo "mongodb+srv://userAdmin:userAdmin123456@my-cluster-name-
→rs0.psmdb.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
```

INSTALL PERCONA SERVER FOR MONGODB ON OPENSHIFT

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

```
git clone -b 1.4.0 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.

If you want to manage PSMDB cluster with a non-privileged user, the necessary permissions can be granted by applying the next 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
$ oc adm policy add-cluster-role-to-user psmdb-admin <some-user>
```

If you have a cert-manager installed, then you have to execute two more commands to be able to manage certificates with a non-privileged user:

```
$ oc create clusterrole cert-admin --verb="*" --resource=iissuers.certmanager.k8s.
↪io,certificates.certmanager.k8s.io
$ oc adm policy add-cluster-role-to-user cert-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
```

5. 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
```

After editing the yaml file, 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*.

6. Now certificates should be generated. By default, the Operator generates certificates automatically, and no actions are required at this step. Still, you can generate and apply your own certificates as secrets according to the *TLS instructions*.

7. 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:4.0 -
→-restart=Never -- bash -il
percona-client:/$ mongo "mongodb+srv://userAdmin:userAdmin123456@my-cluster-name-
→rs0.psmdb.svc.cluster.local/admin?replicaSet=rs0&ssl=false"
```

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, run `minikube start --memory=4096 --cpus=3` (parameters increase the virtual machine limits for the CPU cores and memory, to ensure stable work of the Operator). Being executed, this command will download needed virtualized images, then initialize and run the cluster. After Minikube is successfully started, you can optionally run the 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 1.4.0 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. Because minikube runs locally, the default `deploy/cr.yaml` file should be edited to adapt the Operator for the the local installation with limited resources. Change the following keys in the `replsets` section:

   (a) comment `resources.requests.memory` and `resources.requests.cpu` keys (this will fit the Operator in minikube default limitations)

   (b) set `affinity.antiAffinityTopologyKey` key to `"none"` (the Operator will be unable to spread the cluster on several nodes)

   Also, switch `allowUnsafeConfigurations` key to `true` (this option turns off the Operator's control over the cluster configuration, making it possible to deploy Percona Server for MongoDB as a one-node cluster).

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"
   ```

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.*

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.

Note: Only the incremental update to a nearest minor version is supported (for example, update from 1.2.0 to 1.3.0). To update to a newer version, which differs from the current version by more than one, make several incremental updates sequentially.

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 `1.4.0` 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:1.4.0"}]}}}}'

kubectl patch psmdb my-cluster-name --type=merge --patch '{
   "metadata": {"annotations":{ "kubectl.kubernetes.io/last-applied-configuration
→": "{\"apiVersion\":\"pxc.percona.com/v1-3-0\"}" }},
   "spec": {
       "image": "percona/percona-server-mongodb-operator:1.4.0-mongod4.0",
       "backup": { "image": "percona/percona-server-mongodb-operator:1.4.0-backup
→" },
       "pmm": { "image": "percona/percona-server-mongodb-operator:1.4.0-pmm" }
   }}'
```

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 `1.4.0` 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:1.4.0"}]}}}}'

kubectl patch psmdb my-cluster-name --type=merge --patch '{
   "metadata": {"annotations":{ "kubectl.kubernetes.io/last-applied-configuration
↪": "{\"apiVersion\":\"pxc.percona.com/v1-3-0\"}" }},
   "spec": {"replsets":{ "image": "percona/percona-server-mongodb-operator:1.4.0-
↪mongod4.0" },
      "mongod": { "image": "percona/percona-server-mongodb-operator:1.4.0-
↪mongod4.0" },
      "backup":   { "image": "percona/percona-server-mongodb-operator:1.4.0-
↪backup" },
      "pmm": { "image": "percona/percona-server-mongodb-operator:1.4.0-pmm" }
   }}'
```

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.

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.

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@sha256:4b41c7149d6968a6b61c11e7af7cfea2d67057179716e2c08ba9f7f12459c902
Trying to pull repository docker.io/perconalab/percona-server-mongodb-operator ...
sha256:4b41c7149d6968a6b61c11e7af7cfea2d67057179716e2c08ba9f7f12459c902: Pulling␣
↪from docker.io/perconalab/percona-server-mongodb-operator
Digest: sha256:4b41c7149d6968a6b61c11e7af7cfea2d67057179716e2c08ba9f7f12459c902
Status: Image is up to date for docker.io/perconalab/percona-server-mongodb-
↪operator@sha256:4b41c7149d6968a6b61c11e7af7cfea2d67057179716e2c08ba9f7f12459c902
```

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@sha256:4b41c7149d6968a6b61c11e7af7cfea2d67057179716e2c08ba9f7f12459c902↩
↪\
    172.30.162.173:5000/psmdb/percona-server-mongodb-operator:1.3.0-mongod3.6
$ docker push 172.30.162.173:5000/psmdb/percona-server-mongodb-operator:1.3.0-
↪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  1.4.0-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:1.4.0-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:1.4.0	fcae74acdc26a065e3d25f272a6be088daa6dd6f254207368e048ce492bcc1c0
percona/percona-server-mongodb-operator:1.4.0-mongod3.6	1532e1930a6aa89c56821f2a4248a0902971357fcfff3ebe336ad44e217c1aa6
percona/percona-server-mongodb-operator:1.4.0-mongod4.0	d37a2b8c29707e521ad838939571f9587f7863e0927ac513096fbdf20e4728b7
percona/percona-server-mongodb-operator:1.4.0-mongod4.2	d79a68524efb48d06e79e84b50870d1673cdfecc92b043d811e3a76cb0ae05ab
percona/percona-server-mongodb-operator:1.4.0-backup	a7c661789afa45b5ccbab5ec288557b0863fd0e3b2697d113ced639b98183905
percona/percona-server-mongodb-operator:1.4.0-pmm	bf0cdfd9f9971964cb720a92e99da1a75367cf6a07deec9367ca6b80e78b0f89
percona/percona-server-mongodb-operator:1.3.0	d6abd625833fe3f3cae49721b7600bab5eeeaba78129df4796218a7ce170260d
percona/percona-server-mongodb-operator:1.3.0-mongod3.6	4b41c7149d6968a6b61c11e7af7cfea2d67057179716e2c08ba9f7f12459c902
percona/percona-server-mongodb-operator:1.3.0-mongod4.0	cbe42483639e15b0c3f916f237664b63d552d7a15090025a3c130e62aa2f04b7
percona/percona-server-mongodb-operator:1.3.0-backup	1c79e370edf0391e7cba0b0d63d94a8cfc4bb699018e3508a2140a2c198c83c5
percona/percona-server-mongodb-operator:1.3.0-pmm	28bbb6693689a15c407c85053755334cd25d864e632ef7fed890bc85726cfb68
percona/percona-server-mongodb-operator:1.2.0	fe8699da9ec2f5a2461ecc0e0ff70913ce4c9f053f86992e5a0236597871187b
percona/percona-server-mongodb-operator:1.2.0-mongod3.6	ec-cbfe8682db0b88656a0db59df773172f232f8f65bd8a203782de625a4b32bf
percona/percona-server-mongodb-operator:1.2.0-mongod4.0	baf07ebf9774832999238c03d3c713cca17e7e91d68aeefd93c04a90c5bf8619
percona/percona-server-mongodb-operator:1.2.0-backup	1c79e370edf0391e7cba0b0d63d94a8cfc4bb699018e3508a2140a2c198c83c5
percona/percona-server-mongodb-operator:1.2.0-pmm	28bbb6693689a15c407c85053755334cd25d864e632ef7fed890bc85726cfb68
percona/percona-server-mongodb-operator:1.1.0	d5155898cd19bb70a4d100bb60bfb39d8c9de82c33a908d30fd7caeca1385fc3
percona/percona-server-mongodb-operator:1.1.0-mongod3.6	b3a653b5143a7a60b624c825da8190af6e2e15dd3bc1baee24a7baaeaa455719
percona/percona-server-mongodb-operator:1.1.0-mongod4.0	6af85917a86a838c0ef14b923336f8b150e31a85978b537157d71fed857ae723
percona/percona-server-mongodb-operator:1.1.0-backup	1c79e370edf0391e7cba0b0d63d94a8cfc4bb699018e3508a2140a2c198c83c5
percona/percona-server-mongodb-operator:1.0.0	10a545afc94b7d0040bdbfeed5f64b332861dad190639baecc2989c94284efd1
percona/percona-server-mongodb-operator:1.0.0-mongod3.6.12	31a06ecdd74746d4ff7fe48ae06fd76b461f2a7730de3bd17d7ee4f9d0d2d1e5
percona/percona-server-mongodb-operator:1.0.0-mongod4.0.9	6743dc153c073477fc64db0ccf9a63939d2d949ca37d5bc2848bbc3e5ccd8a7a
percona/percona-server-mongodb-operator:1.0.0-backup	c799d3efcb0b42cdf50c47aea8b726e3bbd8199547f438cffd70be6e2722feec

DEPLOY PERCONA SERVER FOR MONGODB WITH SERVICE BROKER

Percona Service Broker provides the Open Service Broker object to facilitate the operator deployment within high-level visual tools. Following steps are needed to use it while installing the Percona Server for MongoDB on the OpenShift platform:

1. The Percona Service Broker is to be deployed based on the `percona-broker.yaml` file. To use it you should first enable the Service Catalog, which can be done as follows:

```
$ oc patch servicecatalogapiservers cluster --patch '{"spec":{"managementState":
↪"Managed"}}' --type=merge
$ oc patch servicecatalogcontrollermanagers cluster --patch '{"spec":{
↪"managementState":"Managed"}}' --type=merge
```

When Service Catalog is enabled, download and install the Percona Service Broker in a typical OpenShift way:

```
$ oc apply -f https://raw.githubusercontent.com/Percona-Lab/percona-dbaas-cli/
↪broker/deploy/percona-broker.yaml
```

Note: This step should be done only once; the step does not need to be repeated with any other Operator deployments. It will automatically create and setup the needed service and projects catalog with all necessary objects.

2. Now login to your OpenShift Console Web UI and switch to the percona-service-broker project. You can check its Pod running on a correspondent page:
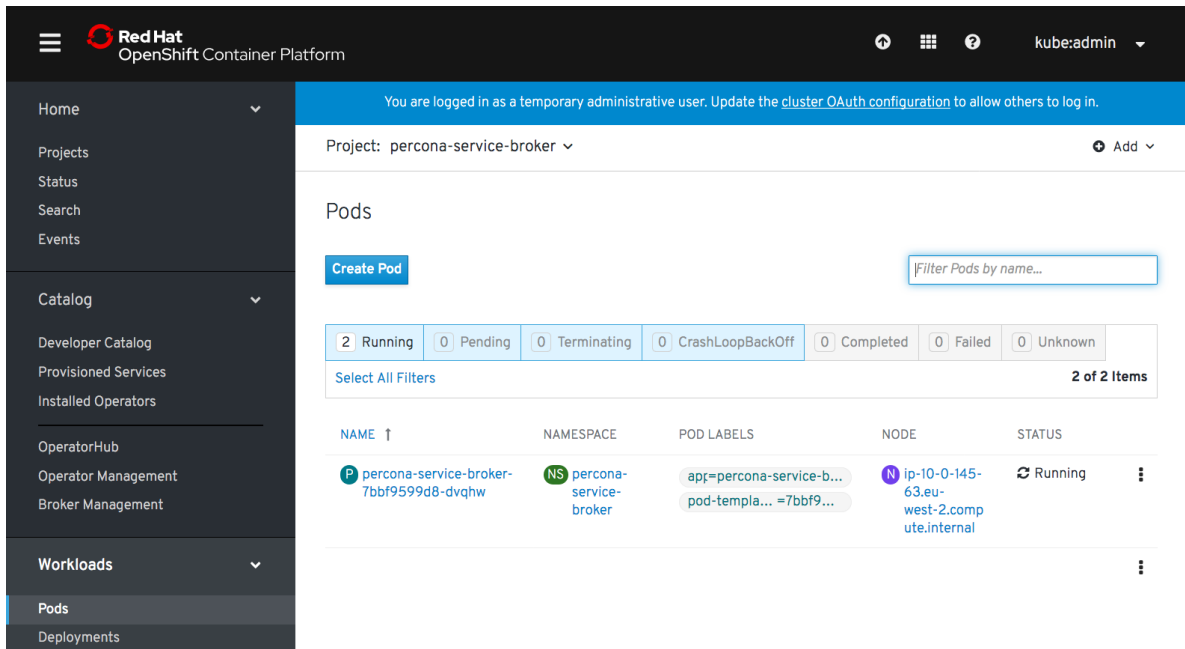
Now switch to the Developer Catalog and select Percona Kubernetes Operator for MongoDB:
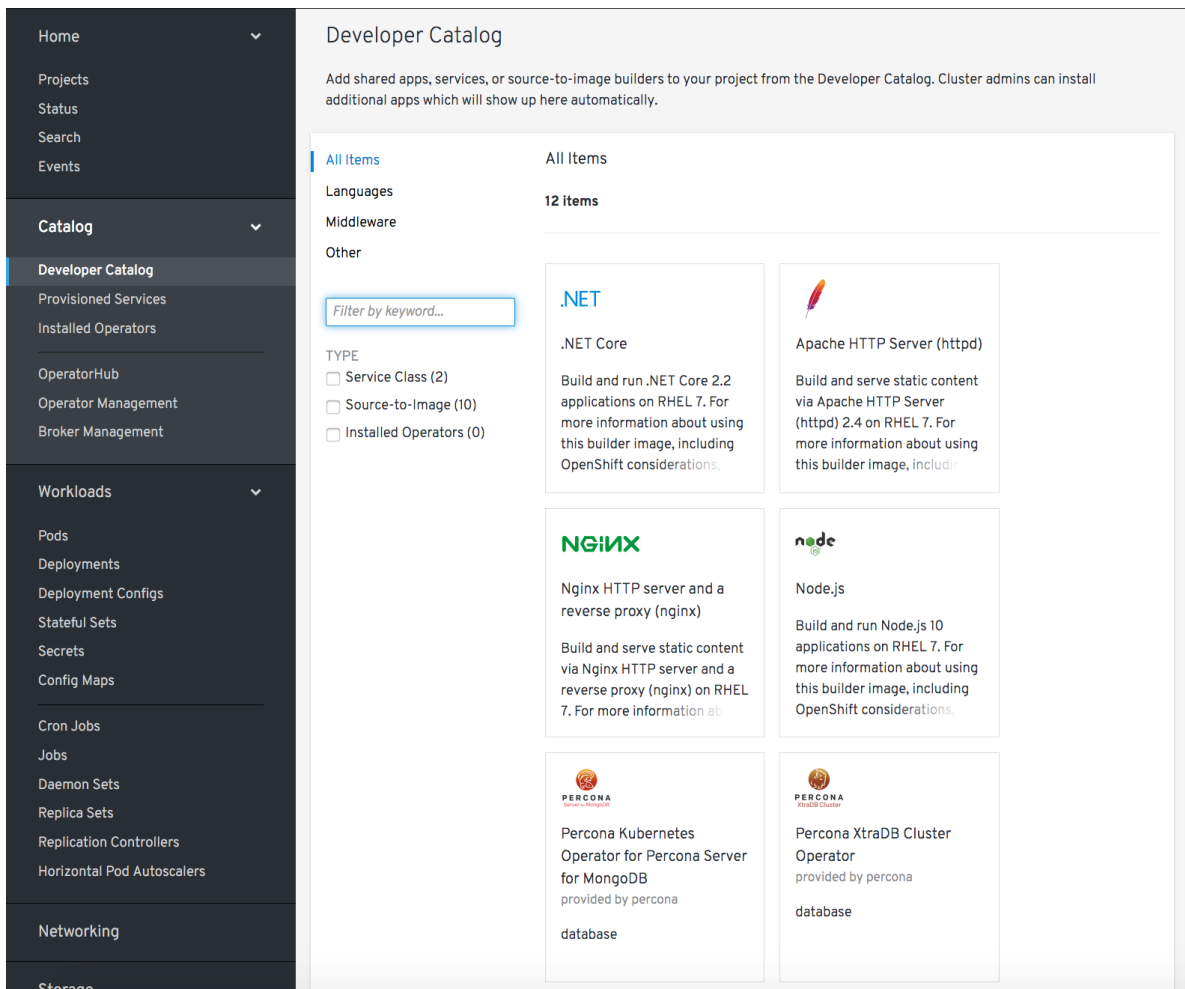
Choose `Percona Kubernetes Operator for Percona Server for MongoDB` item. This will lead you to the Operator page with the *Create Service Instance* button.

3. Clicking the *Create Service Instance* button guides you to the next page:

The two necessary fields are *Service Instance Name* and *Cluster Name*, which should be unique for your project.

4. Clicking the *Create* button gets you to the *Overview* page, which reflects the process of the cluster creation process:

You can also track Pods to see when they are deployed and track any errors.

Part III

Configuration

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:4.0␣
↪--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:4.0␣
↪--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.

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
ClusterSer-viceDNSSuffix	string	`svc. cluster. local`	The (non-standard) cluster domain to be used as a suffix of the Service name.
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	secrets.key
Value Type	string
Example	`my-cluster-name-mongodb-key`
Description	The secret name for the MongoDB Internal Auth Key. This secret is auto-created by the operator if it doesn't exist.

Key	secrets.users
Value Type	string
Example	`my-cluster-name-mongodb-users`
Description	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.

Continued on next page

Table 12.1 – continued from previous page

Key	replsets.name
Value Type	string
Example	`rs 0`
Description	The name of the MongoDB Replica Set

Key	replsets.size
Value Type	int
Example	3
Description	The size of the MongoDB Replica Set, must be >= 3 for High-Availability

Key	replsets.affinity.antiAffinityTopologyKey
Value Type	string
Example	`kubernetes.io/hostname`
Description	The Kubernetes topologyKey node affinity constraint for the Replica Set nodes

Key	replsets.affinity.advanced
Value Type	subdoc
Example	
Description	In cases where the pods require complex tuning the *advanced* option turns off the `topologykey` effect. This setting allows the standard Kubernetes affinity constraints of any complexity to be used

Key	replsets.tolerations.key
Value Type	string
Example	`node.alpha.kubernetes.io/unreachable`
Description	The Kubernetes Pod tolerations key for the Replica Set nodes

Key	replsets.tolerations.operator
Value Type	string
Example	`Exists`
Description	The Kubernetes Pod tolerations operator for the Replica Set nodes

Key	replsets.tolerations.effect
Value Type	string
Example	`NoExecute`
Description	The Kubernetes Pod tolerations effect for the Replica Set nodes

Key	replsets.tolerations.tolerationSeconds
Value Type	int
Example	`6000`
Description	The Kubernetes Pod tolerations time limit for the Replica Set nodes

Key	replsets.priorityClassName
Value Type	string
Example	`high priority`
Description	The Kuberentes Pod priority class for the Replica Set nodes

Key	replsets.annotations.iam.amazonaws.com/role
Value Type	string

Continued on next page

Table 12.1 – continued from previous page

Example	`role-arn`
Description	The AWS IAM role for the Replica Set nodes
Key	labels
Value Type	label
Example	`rack:  rack-22`
Description	The Kubernetes affinity labels for the Replica Set nodes
Key	replsets.nodeSelector
Value Type	label
Example	`disktype:  ssd`
Description	The Kubernetes nodeSelector affinity constraint for the Replica Set nodes
Key	replsets.podDisruptionBudget.maxUnavailable
Value Type	int
Example	`1`
Description	The Kubernetes Pod distribution budget limit specifying the maximum value for unavailable Pods
Key	replsets.podDisruptionBudget.minAvailable
Value Type	int
Example	`1`
Description	The Kubernetes Pod distribution budget limit specifying the minimum value for available Pods
Key	replsets.expose.enabled
Value Type	boolean
Example	`false`
Description	Enable or disable exposing MongoDB Replica Set nodes with dedicated IP addresses
Key	replsets.expose.exposeType
Value Type	string
Example	`ClusterIP`
Description	The IP address type to be exposed
Key	replsets.arbiter.enabled
Value Type	boolean
Example	`false`
Description	Enable or disable creation of Replica Set Arbiter nodes within the cluster
Key	replsets.arbiter.size
Value Type	int
Example	`1`
Description	The number of Replica Set Arbiter nodes within the cluster
Key	replsets.arbiter.afinity.antiAffinityTopologyKey
Value Type	string
Example	`kubernetes.io/hostname`
Description	The Kubernetes topologyKey node affinity constraint for the Arbiter
Key	replsets.arbiter.affinity.advanced
Value Type	subdoc

Continued on next page

Table 12.1 – continued from previous page

Example	
Description	In cases where the pods require complex tuning the *advanced* option turns off the `topologykey` effect. This setting allows the standard Kubernetes affinity constraints of any complexity to be used
Key	replsets.arbiter.tolerations.key
Value Type	string
Example	`node.alpha.kubernetes.io/unreachable`
Description	The Kubernetes Pod tolerations key for the Arbiter nodes
Key	replsets.arbiter.tolerations.operator
Value Type	string
Example	`Exists`
Description	The Kubernetes Pod tolerations operator for the Arbiter nodes
Key	replsets.arbiter.tolerations.effect
Value Type	string
Example	`NoExecute`
Description	The Kubernetes Pod tolerations effect for the Arbiter nodes
Key	replsets.arbiter.tolerations.tolerationSeconds
Value Type	int
Example	`6000`
Description	The Kubernetes Pod tolerations time limit for the Arbiter nodes
Key	replsets.arbiter.priorityClassName
Value Type	string
Example	`high priority`
Description	The Kuberentes Pod priority class for the Arbiter nodes
Key	replsets.arbiter.annotations.iam.amazonaws.com/role
Value Type	string
Example	`role-arn`
Description	The AWS IAM role for the Arbiter nodes
Key	replsets.arbiter.labels
Value Type	label
Example	`rack:    rack-22`
Description	The Kubernetes affinity labels for the Arbiter nodes
Key	replsets.arbiter.nodeSelector
Value Type	label
Example	`disktype:    ssd`
Description	The Kubernetes nodeSelector affinity constraint for the Arbiter nodes
Key	replsets.resources.limits.cpu
Value Type	string
Example	`300m`
Description	Kubernetes CPU limit for MongoDB container

Table 12.1 – continued from previous page

Key	replsets.resources.limits.memory
Value Type	string
Example	`0.5G`
Description	Kubernetes Memory limit for MongoDB container

Key	replsets.resources.requests.cpu
Value Type	string
Example	
Description	The Kubernetes CPU requests for MongoDB container

Key	replsets.resources.requests.memory
Value Type	string
Example	
Description	The Kubernetes Memory requests for MongoDB container

Key	replsets.volumeSpec.emptyDir
Value Type	string
Example	`{}`
Description	The Kubernetes emptyDir volume, i.e. the directory which will be created on a node, and will be accessible to the MongoDB Pod containers

Key	replsets.volumeSpec.hostPath.path
Value Type	string
Example	`/data`
Description	Kubernetes hostPath volume, i.e. the file or directory of a node that will be accessible to the MongoDB Pod containers

Key	replsets.volumeSpec.hostPath.type
Value Type	string
Example	`Directory`
Description	The Kubernetes hostPath volume type

Key	replsets.volumeSpec.persistentVolumeClaim.storageClassName
Value Type	string
Example	`standard`
Description	The Kubernetes Storage Class to use with the MongoDB container Persistent Volume Claim. Use Storage Class with XFS as the default filesystem if possible, for better MongoDB performance

Key	replsets.volumeSpec.persistentVolumeClaim.accessModes
Value Type	array
Example	`[ "ReadWriteOnce" ]`
Description	The Kubernetes Persistent Volume access modes for the MongoDB container

Key	replsets.volumeSpec.persistentVolumeClaim.resources.requests.storage
Value Type	string
Example	`3Gi`
Description	The Kubernetes Persistent Volume size for the MongoDB container

PMM Section

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

Key	pmm.enabled
Value Type	boolean
Example	`false`
Description	Enables or disables monitoring Percona Server for MongoDB with PMM

Key	pmm.image
Value Type	string
Example	`perconalab/pmm-client:1.17.1`
Description	PMM Client docker image to use

Key	pmm.serverHost
Value Type	string
Example	`monitoring-service`
Description	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.

Key	mongod.net.port
Value Type	int
Example	`27017`
Description	Sets the MongoDB net.port option

Key	mongod.net.hostPort
Value Type	int
Example	`0`
Description	Sets the Kubernetes hostPort option

Key	mongod.security.redactClientLogData
Value Type	bool
Example	`false`
Description	Enables/disables PSMDB Log Redaction

Key	mongod.security.enableEncryption
Value Type	bool
Example	`true`
Description	Enables/disables PSMDB data at rest encryption

Key	mongod.security.encryptionKeySecret
Value Type	string
Example	`my-cluster-name-mongodb-encryption-key`
Description	Specifies a secret object with the encryption key

Key	mongod.security.encryptionCipherMode
Value Type	string

Continued on next page

Table 12.2 – continued from previous page

Example	`AES256-CBC`
Description	Sets PSMDB encryption cipher mode
Key	mongod.setParameter.ttlMonitorSleepSecs
Value Type	int
Example	`60`
Description	Sets the PSMDB *ttlMonitorSleepSecs* option
Key	setParameter.wiredTigerConcurrentReadTransactions
Value Type	int
Example	`128`
Description	Sets the wiredTigerConcurrentReadTransactions option
Key	mongod.setParameter.wiredTigerConcurrentWriteTransactions
Value Type	int
Example	`128`
Description	Sets the wiredTigerConcurrentWriteTransactions option
Key	mongod.storage.engine
Value Type	string
Example	`wiredTiger`
Description	Sets the storage.engine option
Key	mongod.storage.inMemory.engineConfig.inMemorySizeRatio
Value Type	float
Example	`0.9`
Description	The ratio used to compute the storage.engine.inMemory.inMemorySizeGb option
Key	mongod.storage.mmapv1.nsSize
Value Type	int
Example	`16`
Description	Sets the storage.mmapv1.nsSize option
Key	mongod.storage.mmapv1.smallfiles
Value Type	bool
Example	`false`
Description	Sets the storage.mmapv1.smallfiles option
Key	mongod.storage.wiredTiger.engineConfig.cacheSizeRatio
Value Type	float
Example	`0.5`
Description	The ratio used to compute the storage.wiredTiger.engineConfig.cacheSizeGB option
Key	mongod.storage.wiredTiger.engineConfig.directoryForIndexes
Value Type	bool
Example	`false`
Description	Sets the storage.wiredTiger.engineConfig.directoryForIndexes option
Key	mongod.storage.wiredTiger.engineConfig.journalCompressor
Value Type	string

Continued on next page

Table 12.2 – continued from previous page

Example	`snappy`
Description	Sets the storage.wiredTiger.engineConfig.journalCompressor option
Key	mongod.storage.wiredTiger.collectionConfig.blockCompressor
Value Type	string
Example	`snappy`
Description	Sets the storage.wiredTiger.collectionConfig.blockCompressor option
Key	mongod.storage.wiredTiger.indexConfig.prefixCompression
Value Type	bool
Example	`true`
Description	Sets the storage.wiredTiger.indexConfig.prefixCompression option
Key	mongod.operationProfiling.mode
Value Type	string
Example	`slowOp`
Description	Sets the operationProfiling.mode option
Key	mongod.operationProfiling.slowOpThresholdMs
Value Type	int
Example	`100`
Description	Sets the operationProfiling.slowOpThresholdMs option
Key	mongod.operationProfiling.rateLimit
Value Type	int
Example	`1`
Description	Sets the operationProfiling.rateLimit option
Key	mongod.auditLog.destination
Value Type	string
Example	
Description	Sets the auditLog.destination option
Key	mongod.auditLog.format
Value Type	string
Example	`BSON`
Description	Sets the auditLog.format option
Key	mongod.auditLog.filter
Value Type	string
Example	`{}`
Description	Sets the auditLog.filter option

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	backup.enabled

Table 12.3 – continued from previous page

Value Type	boolean
Example	`true`
Description	Enables or disables making backups
Key	backup.debug
Value Type	boolean
Example	`true`
Description	Enables or disables debug mode for bakups
Key	backup.restartOnFailure
Value Type	boolean
Example	`true`
Description	Enables or disables restarting the previously failed backup process
Key	backup.image
Value Type	string
Example	`percona/percona-server-mongodb-operator:1.4.0-backup`
Description	The Percona Server for MongoDB Docker image to use for the backup
Key	backup.serviceAccountName
Value Type	string
Example	`percona-server-mongodb-operator`
Description	Nname of the separate privileged service account used by the Operator
Key	backup.coordinator.enableClientsLogging
Value Type	boolean
Example	`true`
Description	Enables or disables backups-related client logging
Key	backup.coordinator.resources.limits.cpu
Value Type	string
Example	`100m`
Description	Kubernetes CPU limit for the MongoDB Coordinator container
Key	backup.coordinator.resources.limits.memory
Value Type	string
Example	`0.2G`
Description	Kubernetes Memory limit for the MongoDB Coordinator container
Key	backup.coordinator.resources.requests.cpu
Value Type	string
Example	`100m`
Description	The Kubernetes CPU requests for the MongoDB Coordinator container
Key	backup.coordinator.resources.requests.memory
Value Type	string
Example	`0.1G`
Description	The Kubernetes Memory requests for the MongoDB Coordinator container
Key	backup.coordinator.resources.requests.storage

Continued on next page

Table 12.3 – continued from previous page

Value Type	string
Example	`1Gi`
Description	The Kubernetes Persistent Volume size for the MongoDB Coordinator container
Key	backup.coordinator.storageClass
Value Type	string
Example	`aws-gp2`
Description	Sets the Kubernetes Storage Class to use with the MongoDB Coordinator container
Key	backup.coordinator.afinity.antiAffinityTopologyKey
Value Type	string
Example	`kubernetes.io/hostname`
Description	The Kubernetes topologyKey node affinity constraint for the backups
Key	backup.coordinator.affinity.advanced
Value Type	sudbdoc
Example	`0`
Description	In cases where the pods require complex tuning the *advanced* option turns off the `topologykey` effect. This setting allows the standard Kubernetes affinity constraints of any complexity to be used
Key	backup.coordinator.tolerations.key
Value Type	string
Example	`node.alpha.kubernetes.io/unreachable`
Description	The Kubernetes Pod tolerations key for the backups nodes
Key	backup.coordinator.tolerations.operator
Value Type	string
Example	`Exists`
Description	The Kubernetes Pod tolerations operator for the backups nodes
Key	backup.coordinator.tolerations.effect
Value Type	string
Example	`NoExecute`
Description	The Kubernetes Pod tolerations effect for the backups nodes
Key	backup.coordinator.tolerations.tolerationSeconds
Value Type	int
Example	`6000`
Description	The Kubernetes Pod tolerations time limit for the backups nodes
Key	backup.coordinator.priorityClassName
Value Type	string
Example	`high priority`
Description	The Kuberentes Pod priority class for the backups nodes
Key	backup.coordinator.annotations.iam.amazonaws.com/role
Value Type	string
Example	`role-arn`
Description	The AWS IAM role for the backup storage nodes

Continued on next page

Table 12.3 – continued from previous page

Key	backup.coordinator.labels
Value Type	label
Example	`rack: rack-22`
Description	The Kubernetes affinity labels for the backup storage nodes

Key	backup.coordinator.nodeSelector
Value Type	label
Example	`disktype: ssd`
Description	The Kubernetes nodeSelector affinity constraint for the backup storage nodes

Key	backup.tasks.name
Value Type	string
Example	`sat-night-backup`
Description	The backup name

Key	backup.tasks.enabled
Value Type	boolean
Example	`true`
Description	Enables or disables this exact backup

Key	backup.tasks.schedule
Value Type	int
Example	`0 0 * * 6`
Description	The scheduled time to make a backup, specified in the crontab format

Key	backup.tasks.storageName
Value Type	string
Example	`st-us-west`
Description	The name of the S3-compatible storage for backups, configured in the *storages* subsection

Key	backup.tasks.compressionType
Value Type	string
Example	`gzip`
Description	The backup compression format

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
```

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
```

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.

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.

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
```

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.

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 several ways. By default, the Operator generates certificates automatically if there are no certificate secrets available. Other options are the following ones:

- The Operator can use a specifically installed *cert-manager* for the automatic certificates generation,
- Certificates can be generated manually.

You can also use pre-generated certificates available in the `deploy/ssl-secrets.yaml` file for test purposes, but we strongly recommend

avoiding their usage on any production system!

The following subsections explain how to configure TLS security with the Operator yourself, as well as 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 clusteris-suer 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.

To disable TLS protocol (e.g. for demonstration purposes) edit the `cr.yaml/spec/allowUnstafeConfigurations` setting to `true` and make sure that there are no certificate secrets available.

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

TWENTYONE

KUBERNETES FOR PERCONA SERVER FOR MONGODB 1.4.0 RELEASE NOTES

Percona Kubernetes Operator for PSMDB 1.4.0

Date March 31, 2020

Installation Installing Percona Kubernetes Operator for PSMDB

New Features

- K8SPSMDB-89: Amazon Elastic Container Service for Kubernetes (EKS) was added to the list of the officially supported platforms
- K8SPSMDB-113: Percona Server for MongoDB 4.2 is now supported
- OpenShift Container Platform 4.3 is now supported

Improvements

- K8SPSMDB-79: The health check algorithm improvements have increased the overall stability of the Operator
- K8SPSMDB-176: The Operator was updated to use Percona Backup for MongoDB version 1.2
- K8SPSMDB-153: Now the user can adjust securityContext, replacing the automatically generated securityContext with the customized one
- K8SPSMDB-175: Operator now updates observedGeneration status message to allow better monitoring of the cluster rollout or backups/restore process

Bugs Fixed

- K8SPSMDB-182: Setting the `updateStrategy: OnDelete` didn't work if was not specified from scratch in CR
- K8SPSMDB-174: The inability to update or delete existing CRD was possible because of too large records in etcd, resulting in "request is too large" errors. Only 20 last status changes are now stored in etcd to avoid this problem.

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 1.3.0

Percona announces the *Percona Kubernetes Operator for Percona Server for MongoDB* 1.3.0 release on December 11, 2019. This release is now the current GA release in the 1.3 series. Install the Kubernetes Operator for Percona Server for MongoDB by following the instructions.

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:

- CLOUD-415: Non-default cluster domain can now be specified with the new `ClusterServiceDNSSuffix` Operator option.

- CLOUD-395: The Percona Server for MongoDB images size decrease by 42% was achieved by removing unnecessary dependencies and modules to reduce the cluster deployment time.

- CLOUD-390: Helm chart for Percona Monitoring and Management (PMM) 2.0 have been provided.

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 1.2.0

Percona announces the *Percona Kubernetes Operator for Percona Server for MongoDB* 1.2.0 release on September 20, 2019. This release is now the current GA release in the 1.2 series. Install the Kubernetes Operator for Percona Server for MongoDB by following the instructions.

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:

- A Service Broker was implemented for the Operator, allowing a user to deploy Percona XtraDB Cluster on the OpenShift Platform, configuring it with a standard GUI, following the Open Service Broker API.

- Now the Operator supports Percona Monitoring and Management 2, which means being able to detect and register to PMM Server of both 1.x and 2.0 versions.

- Data-at-rest encryption is now enabled by default unless `EnableEncryption=false` is explicitly specified in the `deploy/cr.yaml` configuration file.

- Now it is possible to set the `schedulerName` option in the operator parameters. This allows using storage which depends on a custom scheduler, or a cloud provider which optimizes scheduling to run workloads in a cost-effective way.

- The resource constraint values were refined for all containers to eliminate the possibility of an out of memory error.

Fixed bugs:

- Oscillations of the cluster status between "initializing" and "ready" took place after an update.

- The Operator was removing other cron jobs in case of the enabled backups without defined tasks (contributed by Marcel Heers).

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 1.1.0

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.

Symbols