VMware SQL with Postgres for Kubernetes Documentation

VMware SQL with Postgres for Kubernetes 1.7

You can find the most up-to-date technical documentation on the VMware by Broadcom website at:

https://docs.vmware.com/

VMware by Broadcom
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

VMware Tanzu™ SQL with Postgres for Kubernetes Documentation

Note: The name of the "Pivotal Postgres for Kubernetes" product has been changed to VMware Tanzu™ SQL with Postgres for Kubernetes.

This documentation describes how to deploy and use the Tanzu Postgres distribution.

Key topics in the Tanzu Postgres documentation include:

- Release Notes
- About Tanzu Postgres describes the VMware distribution of PostgreSQL and related components.
- Installing a Postgres Operator explains how to download and install the VMware Tanzu Postgres Operator.
- Creating a Postgres Instance describes how to use the Postgres Operator to deploy a Tanzu Postgres instance on a Kubernetes system.
- Backing up and Restoring discusses the usage of pgBackRest to backup and restore Postgres instances.
- Configuring High Availability shows you how to create a high available architecture using pg_auto_failover.
- Configuring TLS for Tanzu Postgres Instances describes how to enable TLS security for client connections to the Postgres server.
- Monitoring Postgres Instances explains how to set up a Postgres Metrics exporter, to allow you to collect and view Prometheus compatible Tanzu Postgres metrics.

VMware Tanzu™ SQL with Postgres for Kubernetes Release Notes

This document contains pertinent release information about VMware Tanzu SQL with Postgres for Kubernetes. Obtain the most recent version of the distribution from Broadcom Support Portal.

Release 1.7.3

Release Date: June 9th, 2022

Software Component Versions

VMware Postgres Version	Component	Component Version
1.7.3	PostgreSQL	14.3, 13.7, 12.11, and 11.16
	pgBackRest	2.38
	pg_auto_failover	1.6.4
	postGIS	3.2.1 (for 11, 12,13 and 14)
	Orafce	3.21
	pgAudit	1.6.1 (for 14), 1.5.1 (for 13), 1.4.1 (for 12), 1.3.1 (for 11), 1.2.1 (for 10)

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.11.x - 1.13.x
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x
- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Kubernetes version 1.19+

Additional Kubernetes environments, such as Minikube, can be used for testing or demonstration purposes.

IMPORTANT: VMware does not support customer deployments that have modified the packaged Docker images, or deployments that reference images other than the VMware Postgres Operator. VMware does not support changing the contents of the deployed containers and pods in any way.

Changes

- Release 1.7.3 updates the OCI artifact to version 1.0.2: `registry.tanzu.vmware.com/packages-for-vmware-tanzu-data-services/tds-packages:1.0.2`. For more information see Installing the Tanzu Operator using the Tanzu CLI.

Fixed Issues

- This release fixes an issue where user provided `process-max` and `log-level-console` values under the `additionalParameters` field in the PostgresBackupLocation resource was ignored.
- To prevent restore objects name collision in synchronized backups, the Postgres instance's backup object name now includes the instance's UUID.

Known Issues and Limitations

- When restoring a database to a different postgres instance on an IPv6-enabled Kubernetes cluster, client connections may fail. This issue does not occur when the source and target databases have the same name.
- The Postgres Operator does not support running backups concurrently on the same instance. If a backup is already running, starting another one will fail.
- The application user `appUser` field in the Postgres instance manifest cannot be updated after the Postgres instance deployment.
- Existing Tanzu Postgres instances cannot be upgraded to a new major version. The instances need to be deleted and recreated.
- The client application parameter `target_session_attrs` needs to be set to `target_session_attrs=read-write`, to ensure correct connection type handling for client applications.
- The High Availability configuration contains only one mirror.

Release 1.7.2

Release Date: May 19th, 2022

Software Component Versions

VMware Postgres Version	Component	Component Version
1.7.2	PostgreSQL	14.3, 13.7, 12.11, and 11.16

VMware Postgres Version	Component	Component Version
	pgBackRest	2.38
	pg_auto_failover	1.6.4
	postGIS	3.2.1 (for 11, 12,13 and 14)
	Orafce	3.21
	pgAudit	1.6.1 (for 14), 1.5.1 (for 13), 1.4.1 (for 12), 1.3.1 (for 11), 1.2.1 (for 10)

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.11.x - 1.13.x
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x
- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Kubernetes version 1.19+

Additional Kubernetes environments, such as Minikube, can be used for testing or demonstration purposes.

IMPORTANT: VMware does not support customer deployments that have modified the packaged Docker images, or deployments that reference images other than the VMware Postgres Operator. VMware does not support changing the contents of the deployed containers and pods in any way.

Changes

- This release supports VMware Postgres 14.3, 13.7, 12.11, and 11.16. For more information on VMware Postgres see VMware Postgres.
- Release 1.7.2 updates the OCI artifact to version 1.0.1: `registry.tanzu.vmware.com/packages-for-vmware-tanzu-data-services/tds-packages:1.0.1`. For more information see Installing the Tanzu Operator using the Tanzu CLI.

Known Issues and Limitations

- When restoring a database to a different postgres instance on an IPv6-enabled Kubernetes cluster, client connections may fail. This issue does not occur when the source and target databases have the same name.

- Running backups concurrently is not supported. If a backup is already running, starting another one will fail.
- The application user `appUser` field in the Postgres instance manifest cannot be updated after the Postgres instance deployment.
- Existing Tanzu Postgres instances cannot be upgraded to a new major version. The instances need to be deleted and recreated.
- The client application parameter `target_session_attrs` needs to be set to `target_session_attrs=read-write`, to ensure correct connection type handling for client applications.
- The High Availability configuration contains only one mirror.

Release 1.7.1

Release Date: April 25th, 2022

Software Component Versions

VMware Postgres Version	Component	Component Version
1.7.1	PostgreSQL	11.15, 12.10, 13.6, 14.2
	pgBackRest	2.37
	pg_auto_failover	1.6.3
	postGIS	2.5.5 (for 11.14), 3.2.0 (for 12,13 and 14)
	Orafce	3.17
	pgAudit	1.6.1

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.11.x - 1.13.x
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x
- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Kubernetes version 1.19+

Additional Kubernetes environments, such as Minikube, can be used for testing or demonstration purposes.

IMPORTANT: VMware does not support customer deployments that have modified the packaged Docker images, or deployments that reference images other than the

VMware Postgres Operator. VMware does not support changing the contents of the deployed containers and pods in any way.

Changes

- The Tanzu Postgres 1.7.1 release supports Carvel tools, and the 1.7.1. container images are packaged as an image bundle that is distributed as a new OCI artifact: `registry.tanzu.vmware.com/packages-for-vmware-tanzu-data-services/tds-packages:1.0.0`. For more information on image bundles see Resources in the Carvel documentation.
- The Tanzu Postgres Operator can now be installed using the Tanzu CLI and the new Carvel image bundle. Tanzu Application Platform (TAP) users can now use the same toolchain to manage both products. For details on the Tanzu CLI installation process see Installing a Tanzu Postgres Operator.

Fixed Issues

- Fixes an issue where backup synchronization to a different namespace or cluster was only supported for full backups. Release 1.7.1 now supports restoring incremental and differential backups to a different namespace or cluster.
- Release 1.7.1 updates the package golang.org/x/crypto to a later version which addresses the following CVE:
    - CVE-2022-27191

Known Issues and Limitations

- When restoring a database to a different postgres instance on an IPv6-enabled Kubernetes cluster, client connections may fail. This issue does not occur when the source and target databases have the same name.
- Running backups concurrently is not supported. If a backup is already running, starting another one will fail.
- The application user `appUser` field in the Postgres instance manifest cannot be updated after the Postgres instance deployment.
- Existing Tanzu Postgres instances cannot be upgraded to a new major version. The instances need to be deleted and recreated.
- The client application parameter `target_session_attrs` needs to be set to `target_session_attrs=read-write`, to ensure correct connection type handling for client applications.
- The High Availability configuration contains only one mirror.

Release 1.7.0

Release Date: April 15th, 2022

Software Component Versions

VMware Postgres Version	Component	Component Version
1.7.0	PostgreSQL	11.15, 12.10, 13.6, 14.2
	pgBackRest	2.37
	pg_auto_failover	1.6.3
	postGIS	2.5.5 (for 11.14), 3.2.0 (for 12,13 and 14)
	Orafce	3.17
	pgAudit	1.6.1

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.11.x - 1.13.x
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x
- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Kubernetes version 1.19+

Additional Kubernetes environments, such as Minikube, can be used for testing or demonstration purposes.

IMPORTANT: VMware does not support customer deployments that have modified the packaged Docker images, or deployments that reference images other than the VMware Postgres Operator. VMware does not support changing the contents of the deployed containers and pods in any way.

Features

- This release allows instance backups to be synchronized in two different namespaces or Kubernetes clusters. This feature allows users to restore to a new Tanzu Postgres instance from a previously created backup, even if the instance that originated the backup is not available, and even if the originating backup namespace is different from the restore namespace. For more information see Restoring Tanzu Postgres.
- The PostgresBackupLocation CRD now supports an additional field named `additionalParameters`. For more information see Backup and Restore Deployment Properties.
- Users can now provide service annotations for the Load Balancer service, to cater for internal and external load balancer configurations. The annotations provide more flexibility with Cloud provider environments. For more information see Internal Load Balancer.
- This release supports deploying Tanzu Postgres instances in Kubernetes clusters with IPv6 enabled.

Changes

- This release removes the deprecated property `repo-s3-verify-tls` from pgBackRest, and replaces it with `repo-storage-verify-tls`.
- Users are now prevented from using reserved system names like "postgres" as values in `pgConfig.adminUser`, `pgConfig.appUser`, and `pgConfig.dbName`. Creating an instance using reserved words now returns an error similar to: `pgconfig.dbname cannot be postgres`.
- Users cannot anymore alter the Custom Resource manifest of a backup or restore object that has been initiated, regardless if the backup or restore has succeeded or failed. If a user attempts to modify the Restore CR after applying it, the change will not be stored, and they will see a message similar to: "Forbidden: spec cannot be updated after Restore has been created."
- Tanzu Postgres pgBackRest configuration will not longer send log information to disk, at location `/pgsql/logs/<namespace>-<instance-name>-backup.log`. Instead it will send logs to stdout and stderr. Use a log collector of your choice to store the logs in a custom location.
- From this release, backups are not encrypted automatically. Customers are advised to use their S3 provider server-side encryption method and/or client-side encryption. For information on implementing client-side encryption, see Configure Client-side Encryption for Backups.

Fixed Issues

- Fixes an issue where successful or unsuccessful backups would show incorrect completion time.

Known Issues and Limitations

- Backup synchronization to a different namespace or cluster is only supported for full backups.
- When restoring a database to a different postgres instance on an IPv6-enabled Kubernetes cluster, client connections may fail. This issue does not occur when the source and target databases have the same name.
- Running backups concurrently is not supported. If a backup is already running, starting another one will fail.
- The application user `appUser` field in the Postgres instance manifest cannot be updated after the Postgres instance deployment.
- Existing Tanzu Postgres instances cannot be upgraded to a new major version. The instances need to be deleted and recreated.
- The client application parameter `target_session_attrs` needs to be set to `target_session_attrs=read-write`, to ensure correct connection type handling for client applications.
- The High Availability configuration contains only one mirror.

Upgrading to 1.7.0

To upgrade to Tanzu Postgres 1.7.0 review the Upgrading the Tanzu Postgres Operator and Instances page.

Release 1.6.2

Release Date: April 1st, 2022

Software Component Versions

VMware Postgres Version	Component	Component Version
1.6.2	PostgreSQL	11.15.1, 12.10.1, 13.6.1, 14.2.1
	psqlODBC	13.02.0000
	pgjdbc	42.3.2
	pgBackRest	2.38
	pg_auto_failover	1.6.3
	postGIS	2.5.5 (for 11.14), 3.2.0 (for 12,13 and 14)
	Orafce	3.17
	pgAudit	1.6.1

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.11.x - 1.13.x
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x
- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Kubernetes version 1.19+

Fixed Issues

- In release 1.6.2 the Prometheus metrics exporter is compiled with a newer version of golang that addresses the following CVEs:
  - CVE-2022-23772
  - CVE-2022-23806
  - CVE-2022-24921
  - CVE-2022-23773

Known Issues and Limitations

- Running backups concurrently is not supported. If a backup is already running, starting another one will fail.
- Recovering a backup to a new Postgres instance is restricted to the same namespace.
- The application user `appUser` field in the Postgres instance manifest cannot be updated after the Postgres instance deployment.
- Existing Tanzu Postgres instances cannot be upgraded to a new major version. The instances need to be deleted and recreated.
- The client application parameter `target_session_attrs` needs to be set to `target_session_attrs=read-write`, to ensure correct connection type handling for client applications.
- The High Availability configuration contains only one mirror.

Release 1.6.1

Release Date: March 16th, 2022

Software Component Versions

VMware Postgres Version	Component	Component Version
1.6.1	PostgreSQL	11.15.1, 12.10.1, 13.6.1, 14.2.1
	psqlODBC	13.02.0000
	pgjdbc	42.3.2
	pgBackRest	2.38
	pg_auto_failover	1.6.3
	postGIS	2.5.5 (for 11.14), 3.2.0 (for 12,13 and 14)
	Orafce	3.17
	pgAudit	1.6.1

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.11.x - 1.13.x
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x
- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Kubernetes version 1.19+

Changes

- Tanzu Postgres 1.6.1 updates pgBackRest to version 2.38. For information on the 2.38 release see v2.38: Minor Bug Fixes and Improvements.
- This release includes VMware Postgres 11.15.1, 12.10.1, 13.6.1, and 14.2.1.
- pgBackRest no longer writes logs to a default `/pgsql/logs/<namespace>-<instance-name>-backup.log` prescribed file location. Instead it outputs log information to stdout and stderr. Users who wish to store the logs, are advised to configure a custom log collector.
- Users are no longer able to modify an existing Backup object after the backup has been initiated. An attempt for such a change will result in an error similar to: "Forbidden: spec cannot be updated after Backup has been created.".

Fixed Issues

- From this release it is not possible for a user to scale the Monitor pod down to a value other than 1. The Tanzu Postgres Operator will now reconcile the Monitor component if a user scales it down.
- The output of `kubectl get postgresbackup` did not correctly display the time a backup had completed. The output of the command now displays the correct finish time of a backup.

Known Issues and Limitations

- Running backups concurrently is not supported. If a backup is already running, starting another one will fail.
- Recovering a backup to a new Postgres instance is restricted to the same namespace.
- The application user `appUser` field in the Postgres instance manifest cannot be updated after the Postgres instance deployment.
- Existing Tanzu Postgres instances cannot be upgraded to a new major version. The instances need to be deleted and recreated.
- The client application parameter `target_session_attrs` needs to be set to `target_session_attrs=read-write`, to ensure correct connection type handling for client applications.
- The High Availability configuration contains only one mirror.

Release 1.6.0

Release Date: February 28th, 2022

Software Component Versions

VMware Postgres Version	Component	Component Version
1.6.0	PostgreSQL	11.15, 12.10, 13.6, 14.2
	psqlODBC	13.02.0000
	pgjdbc	42.3.2
	pgBackRest	2.37

VMware Postgres Version	Component	Component Version
	pg_auto_failover	1.6.3
	postGIS	2.5.5 (for 11.14), 3.2.0 (for 12,13 and 14)
	Orafce	3.17
	pgAudit	1.6.1

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.11.x - 1.13.x
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x
- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Kubernetes version 1.19+

Additional Kubernetes environments, such as Minikube, can be used for testing or demonstration purposes.

> **IMPORTANT:** VMware does not support customer deployments that have modified the packaged Docker images, or deployments that reference images other than the VMware Postgres Operator. VMware does not support changing the contents of the deployed containers and pods in any way.

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.11.x - 1.13.x
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x
- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Kubernetes version 1.19+

Additional Kubernetes environments, such as Minikube, can be used for testing or demonstration purposes.

> **IMPORTANT:** VMware does not support customer deployments that have modified the packaged Docker images, or deployments that reference images other than the VMware Postgres Operator. VMware does not support changing the contents of the deployed containers and pods in any way.

Features

- Tanzu Postgres 1.6.0 supports PostgreSQL 14.2, 13.6, 12.10, and 11.15. When upgrading to Tanzu Operator 1.6.0, existing instances will be upgraded to the latest minor version.
- This release adds pgAudit 1.6.1 to the Tanzu Postgres component list. For more details, see pgAudit.
- Version 1.6.0 upgrades pgBackRest to 2.37.
- This release upgrades pgjdbc to 42.3.2, and postGIS to 3.2.0 (for PostgreSQL 12, 13, or 14).
- Release 1.6.0 supports Secure Compute Mode (seccomp) profiles. This release introduces a new Postgres CRD field `seccompProfile`. For more information, see seccompProfile in the Postgres API reference page.
- This release adds Azure Kubernetes Service (AKS), and Amazon Elastic Kubernetes Service (Amazon EKS) to the Supported Platforms list.
- Users can now specify a retention policy for full and differential backups. The PostgresBackupLocation CRD includes two new properties, `spec.retentionPolicy.fullRetention` and `spec.retentionPolicy.diffRetention`. For more details see Backup and Restore CRD API Reference.
- Tanzu Postgres 1.6.0 supports secret rotation for the Postgres Application user `appUser`. The Postgres database will automatically pickup up a refreshed secret.
- This release supports a new field `spec.imagePullSecret` in the Postgres instance CRD. Users can now replace the default `regsecret` to their own registry secret.

Changes

- This release removes the column `AGE` from the output of the command `kubectl get postgresversion`.

Fixed Issues

- **TSQL-2** - Fixes an issue where the restore process is unable to complete on Postgres instances that have gone through a timeline change.
- **TSQL-3** - This release fixes an issue where containers would use more CPU than expected, and `runc` process would restart repeatedly. Users can now set the `seccompProfile` to `Localhost` or `Unconfined`. For more details on `seccompProfile` see seccompProfile in the Postgres API reference page.
- This release fixes an issue where in a newly restored instance, the `appUser` secret for any application service-bindings would require manual secret reconfiguration.

Known Issues and Limitations

- A backup may still be running but show as finished in the output of `kubectl get postgresbackup`.
- Running backups concurrently is not supported. If a backup is already running, starting another one will fail.

- Recovering a backup to a new Postgres instance is restricted to the same namespace.
- The application user `appUser` field in the Postgres instance manifest cannot be updated after the Postgres instance deployment.
- Existing Tanzu Postgres instances cannot be upgraded to a new major version. The instances need to be deleted and recreated.
- The client application parameter `target_session_attrs` needs to be set to `target_session_attrs=read-write`, to ensure correct connection type handling for client applications.
- The High Availability configuration contains only one mirror.

Upgrading to 1.6.0

To upgrade to Tanzu Postgres 1.6.0 review the Upgrading the Tanzu Postgres Operator and Instances page.

Release 1.5.0

Release Date: January 7th, 2022

Software Component Versions

VMware Postgres Version	Component	Component Version
1.5.0	PostgreSQL	11.14, 12.9, 13.5, 14.1
	psqlODBC	13.2-0000
	pgjdbc	42.3.1
	pgBackRest	2.36
	pg_auto_failover	1.6.3
	postGIS	2.5.5 (for 11.14), 3.1.4 (for 12,13 and 14)
	Orafce	3.17

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.11.x - 1.13.x.
- VMware Tanzu Kubernetes Grid (TKGm) on AWS, version 1.2.x - 1.4.x.
- Google Kubernetes Engine (GKE)
- Kubernetes version 1.19+

Additional Kubernetes environments, such as Minikube, can be used for testing or demonstration purposes.

IMPORTANT: VMware does not support deployments that have been modified by adding layers to the packaged Docker images, or deployments that reference images other than the VMware Postgres Operator. VMware does not support changing the contents of the deployed containers and pods in any way.

Features

Tanzu Postgres 1.5.0 has the following new features:

- Release 1.5.0 supports pod Affinity and Tolerations for advanced scheduling and HA configurations. For more information, see Configuring Affinity and Tolerations at the *Deploying a Postgres Instance* page.
- Tanzu Postgres Operator 1.5.0 supports Tanzu Application Platform (TAP) with Service Binding. This release introduces a new Postgres CRD field `appUser` and an application secret `<pg-instance-name>-app-user-db-secret`. For more information, see Creating Service Bindings.

Changes

- This release removes the column `AGE` from the output of the command `kubectl get postgresversion`.

Fixed Issues

- An updated Postgres instance would still show state `Running` even if the monitor stateful set was restarting. Now the Tanzu Postgres Operator will correctly mark the Postgres instance as `Pending` while the monitor statefulset is not running.
- This release fixes an issue where a restore would not complete if the Postgres instance had gone through a timeline change.

Known Issues and Limitations

- The application user `appUser` field in the Postgres instance manifest cannot be updated after the Postgres instance deployment.
- Existing Tanzu Postgres instances cannot be upgraded to a new major version. The instances need to be deleted and recreated.
- The client application parameter `target_session_attrs` needs to be set to `target_session_attrs=read-write`, to ensure correct connection type handling for client applications.
- The High Availability configuration contains only one mirror.
- Recovering a backup to a new Postgres instance is restricted to the same namespace.

Upgrading to 1.5.0

To upgrade to Tanzu Postgres 1.5.0 review the Upgrading the Tanzu Postgres Operator and Instances page.

Release 1.4.1

Release Date: December 10th, 2021

Software Component Versions

VMware Postgres Version	Component	Component Version
1.4.1	PostgreSQL	11.14, 12.9, 13.5, 14.1
	psqlODBC	13.2-0000
	pgjdbc	42.3.1
	pgBackRest	2.36
	pg_auto_failover	1.6.3
	postGIS	2.5.5 (for 11.14), 3.1.4 (for 12,13 and 14)
	Orafce	3.17

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.9.x - 1.12.x.
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x.
- Google Kubernetes Engine (GKE)
- Kubernetes version 1.19+

Additional Kubernetes environments, such as Minikube, can be used for testing or demonstration purposes.

IMPORTANT: VMware does not support deployments that have been modified by adding layers to the packaged Docker images, or deployments that reference images other than the VMware Postgres Operator. VMware does not support changing the contents of the deployed containers and pods in any way.

Changes

- Release 1.4.1 has been updated to address a security CVE relating to Network Security Services (NSS). For details, see CVE-2021-43527

Known Issues and Limitations

- Existing Tanzu Postgres instances cannot be upgraded to a new major version. The instances need to be deleted and recreated.
- The client application parameter `target_session_attrs` needs to be set to `target_session_attrs=read-write`, to ensure correct connection type handling for client applications.

- Tanzu Postgres does not support Helm version 3.7.
- The High Availability configuration contains only one mirror.
- Recovering a backup to a new Postgres instance is restricted to the same namespace.
- Concurrent backup is not supported. An already running backup must finish before a second backup can successfully complete.

Upgrading to 1.4.1

To upgrade to Tanzu Postgres 1.4.1 review the Upgrading the Tanzu Postgres Operator and Instances page.

Existing Postgres instances will be associated with the `PostgresVersion` resource named `postgres-11`. If there are manifest files saved for those existing instances, please update the manifests to include `spec.postgresVersion.name` as `postgres-11`.

Release 1.4.0

Release Date: November 22nd, 2021

Software Component Versions

VMware Postgres Version	Component	Component Version
1.4.0	PostgreSQL	11.14, 12.9, 13.5, 14.1
	psqlODBC	13.2-0000
	pgjdbc	42.3.1
	pgBackRest	2.36
	pg_auto_failover	1.6.3
	postGIS	2.5.5 (for 11.14), 3.1.4 (for 12,13 and 14)
	Orafce	3.17

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.9.x - 1.12.x.
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x.
- Google Kubernetes Engine (GKE)
- Kubernetes version 1.19+

Additional Kubernetes environments, such as Minikube, can be used for testing or demonstration purposes.

IMPORTANT: VMware does not support deployments that have been modified by adding layers to the packaged Docker images, or deployments that reference images other than the VMware Postgres Operator. VMware does not support changing the contents of the deployed containers and pods in any way.

Features

Tanzu Postgres 1.4.0 has the following new features:

- Release 1.4.0 supports Postgres 11.14, 12.9, 13.5, and 14.1. The default version for new Tanzu Postgres instances is 14.1.
- This release updates the pgBackRest component to 2.36 and pg_auto_failover to 1.6.3.
- This release supports Orafce 3.17.0, and PostGIS 2.5.5 (for Postgres 11), or PostGIS 3.1.14 (for Postgres 12, 13, and 14).
- JDBC has been upgraded to 42.3.1, and ODBC to 13.02.00.
- The Postgres Operator 1.4.0 now supports multiple Postgres versions. To deploy a different version than the default Postgres 14.1, see Specifying the Tanzu Postgres Version.
- This release now supports scaling down from an HA configuration to a single Postgres instance. For more details, see Configuring High Availability in Tanzu Postgres.
- This release supports `ssl_min_protocol_version` and TLSv1.2 when users deploy 12, 13, or 14 Postgres instances. Users may use the psql client and the command `select setting from pg_settings where name='ssl_min_protocol_version'` to confirm their instances TLS version.

Changes

- This release removes PL/R and PL/Java from the 1.4.0 release components.
- The legacy Postgres CRD field `spec.backupLocationSecret` is now removed from the Tanzu Postgres manifest.
- The shutdown period for the Tanzu Postgres pods has been increased to allow for a more graceful shutdown period.

Fixed Issues

- This release fixes an issue with HA failover that occured when the demoted primary had a different secret than the promoted secondary.
- After an instance crash, certain files (for example unix socket files, lock files, and PID files) would prevent database restart. These files are now cleaned up.
- This release fixes an issue where wildcard matching by hostname in `pg_hba.conf` caused unsuccessful authentication between primary and mirrors. This issue has now been resolved by deploying wildcard authentication by subdomain.
- When customers were creating a `PostgresBackupSchedule` with incremental or differential backups, the backups were performed as full. This issue has now been fixed.
- Stale information stored in the Kubernetes client cache was causing the restore process to be unstable. The restore reliability has now been improved.

Known Issues and Limitations

- Existing Tanzu Postgres instances cannot be upgraded to a new major version. The instances need to be deleted and recreated.
- The client application parameter `target_session_attrs` needs to be set to `target_session_attrs=read-write`, to ensure correct connection type handling for client applications.
- Tanzu Postgres does not support Helm version 3.7.
- The High Availability configuration contains only one mirror.
- Recovering a backup to a new Postgres instance is restricted to the same namespace.
- Concurrent backup is not supported. An already running backup must finish before a second backup can successfully complete.

Upgrading to 1.4.0

To upgrade to Tanzu Postgres 1.4.0 review the Upgrading the Tanzu Postgres Operator and Instances page.

Existing Postgres instances will be associated with the `PostgresVersion` resource named `postgres-11`. If there are manifest files saved for those existing instances, please update the manifests to include `spec.postgresVersion.name` as `postgres-11`.

Release 1.3.0

Release Date: October 14th, 2021

Software Component Versions

VMware Postgres Version	Component	Component Version
1.3.0	PostgreSQL	11.13
	psqlODBC	11.0-0000
	pgjdbc	42.2.5
	pgBackRest	2.34
	pg_auto_failover	1.6.2
	postGIS	2.5.5
	Orafce	3.15
	PL/Java Beta	1.5.7

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.9.x - 1.12.x.
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x.

- Google Kubernetes Engine (GKE)
- Kubernetes version 1.19+

Additional Kubernetes environments, such as Minikube, can be used for testing or demonstration purposes.

IMPORTANT: VMware does not support deployments that have been modified by adding layers to the packaged Docker images, or deployments that reference images other than the VMware Postgres Operator. VMware does not support changing the contents of the deployed containers and pods in any way.

Features

Tanzu Postgres 1.3.0 has the following new features:

- Release 1.3.0 supports Postgres 11.13.
- This release updates the pgBackRest component to 2.34 and pg_auto_failover to 1.6.2.
- This release supports Guaranteed Quality of Service (Qos) for critical pods. When the resource limits are higher than the requests, the QoS class is Burstable.
- This release improves secrets handling, for Postgres instances and monitor.
- Customers can now create Postgres instances in an environment with restricted security policies. For details, see Prerequisites in the *Install Tanzu Operator* page.
- This release allows customers to configure the Postgres Operator pod resources. See Access the Resources in the *Install the Tanzu Operator* page.
- The Postgres Operator 1.3.0 now recreates any Postgres database or Postgres monitor secrets that are deleted by accident.
- The Tanzu Postgres administrator can now set the certificate issuer for the Postgres operator's certificate.

Backup and Restore

- This release deprecates the BackupLocationSecret object. Customers using a previous release of Tanzu Postgres, should migrate to the new backup and restore strategy. For details see Migrating to Tanzu Postgres 1.3.0 Backup and Restore.
- This release introduces four new Custom Resource Definitions for backup and restore: PostgresBackupLocation, PostgresBackup, PostgresBackupSchedule, and PostgresRestore. See Backing Up and Restoring Tanzu Postgres
- Users can filter existing backups based on Postgres instance name. For details, see Listing Backup Resources.
- Any changes to the backup location or backup secret are automatically applied to the related Postgres instances.
- Customers can now monitor real time the backup logs during a backup operation.

- Postgres restore now supports recovering to a brand new instance, to help users with disaster recovery or debug scenarios. For details see Restore to a different instance.

Monitoring

- Tanzu Postgres 1.3.0 provides a Prometheus compatible endpoint for metrics collection. For details see Monitoring Postgres Instances.
- Customers can configure the resource limits and requests of the Postgres exporter. The metrics resources can be edited in the Postgres manifest file. For further details, see Configuring a Postgres Instance.
- This release supports TLS security for metrics collection.

Changes

- This release improves error messages.
- The Postgres Operator and Postgres Image tags for version and repository are combined into a single variable instead of two. The Operator `values.yaml` now contains just one variable for each, `operatorImage` and `postgresImage`.
- The Postgres instance `wal_keep_segments` value is now set to 0 by default. This provides greater flexibility and reduces space requirements for wal logs.

Limitations and Known Issues

- Tanzu Postgres does not support Helm version 3.7.
- The High Availability configuration contains only one mirror.
- Recovering a backup to a new Postgres instance is restricted to the same namespace.

Fixed Issues

- The `dockerRegistrySecretName` in the Operator `values.yaml` file was set to `regsecret` and could not be changed to an alternative name. This issue has been resolved and users can specify an alternative secret name in the overrides file.

Release 1.2.0

Release Date: July 14, 2021

Software Component Versions

VMware Postgres Version	Component	Component Version
1.2.0	PostgreSQL	11.12
	psqlODBC	11.0-0000
	pgjdbc	42.2.5
	pgBackRest	2.28

VMware Postgres Version	Component	Component Version
	pg_auto_failover	1.4.2
	postGIS	2.5.5
	Orafce	3.14
	PL/Java Beta	1.5.7

Supported Platforms

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.9.x.
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x.
- Google Kubernetes Engine (GKE)
- Kubernetes version 1.16+

Additional Kubernetes environments, such as Minikube, can be used for testing or demonstration purposes.

IMPORTANT: VMware does not support deployments that have been modified by adding layers to the packaged Docker images, or deployments that reference images other than the VMware Postgres Operator. VMware does not support changing the contents of the deployed containers and pods in any way.

Features

Tanzu Postgres 1.2.0 has the following new features:

Security Enhancements

- Tanzu Postgres 1.2.0 supports TLS security and user provided TLS certificates. See Creating a TLS Secret Manually.
- Support for custom TLS issuer. See Configuring TLS for Tanzu Postgres Instances.
- TLS certificates associated with cert-manager can be accidentally deleted and regenerated automatically.
- Kubernetes secrets associated with a cert-manager certificate can be deleted and recovered automatically. A new certificate will be generated, and the Postgres server will restart. Applications will need to reconnect.
- Tanzu Postgres instances are now created by default with service type `ClusterIP`, to enhance security.
- New Postgres instances now use a crypto library/algorithm for enhanced password generation.

Usability Enhancements

- The Tanzu Postgres Operator and instances are now available via the TanzuNet registry. See Installing a Postgres Operator for more information.

- This release supports VMware Tanzu Kubernetes Grid on AWS.
- PL/Java is bundled with the 1.2 release but currently provided as Beta.
- Tanzu Postgres now supports the Orafce extension. Users can now run Oracle queries like `SELECT months_between(date '1995-02-02', date '1995-01-01');`. See Installing Postgres Extensions.
- Release 1.2 now supports all Postgres contrib extensions, apart from `plpython3u`.
- Connections to Postgres instances are now writable by default. This allows applications that cannot use connection parameters such as `target_session_attrs`(for libpq) or `targetServertype`(for JDBC) to connect to a writable instance.
- Users can install the Tanzu Postgres Operator in a namespace of their choice.
- The new release supports enhanced labels. Users can search the Kubernetes resources created by the Tanzu Postgres Helm chart by using a label such as `app=postgres-operator`. See Installing a Tanzu Postgres Operator.
- Users can now set `logLevel: Debug` when creating an instance. Debug logs can be shared with VMware support for troubleshooting. See Configuring a Postgres Instance.
- The Postgres instance Monitor pod resources can now be manually altered, to support resource constrained environments such as Minikube on a client laptop. For more details see Updating the Monitor Resources.
- Release 1.2 now supports all Postgres contrib extensions, apart from `plpython3u`. For more information see Additional Supplied Modules in the Postgres documentation.

Fixed Issues

- (175768688) - Users can now create backups of similarly named instances in the same S3 bucket.
- (176064027) - When creating a backup operation, users do not need to specify the path `--pg1-path` on the command line.
- (175771699) - This release improves the error message when specifying below the minimum accepted disk space for the `StorageSize` field: `"pg-small-instance.yaml": admission webhook "vpostgres.kb.io" denied the request: The field(s) StorageSize field needs to be at least 250MB are incorrectly formatted and could not be parsed.`
- (177225289) - Users can now specify the `StorageSize` field using M, Mi, or MB.
- (176615909) - Non-admin users can now view Postgres objects in a specified namespace.
- (178402085) - Fixes a log display issue where, if the instance was older than one day, the logs stopped displaying to `stdout`.
- (177407650) - Fixes an issue where the database would be inaccessible for a period of time when scaling up from a single node to an HA configuration. The database is now accessible during the secondary node data copy period.
- (178528901) - In a HA configuration, when the user configured a S3 backup secret, he also had to manually create the backup stanza. This issue has been resolved, and the backup

stanza is created automatically when users apply the S3 secret.

Limitations

- The High Availability configuration contains only one mirror.
- The `dockerRegistrySecretName` in the Operator `values.yaml` file is set to `regsecret` and cannot be changed to an alternative name in the overrides file .

Release 1.1.0

Release Date: February 26, 2021

Software Components

VMware Postgres Version	Component	Component Version
1.1.0	PostgreSQL	11.10
	pgBackRest	2.31
	pg_auto_failover	1.4.0

Features

Tanzu Postgres 1.1.0 has the following features:

- Support for upgrading from Tanzu Postgres 1.0.0 to 1.1.0. See Upgrading the Tanzu Postgres Operator and Instances.
- Support for Postgres 11.10.
- Enhanced security by implementing Postgres cluster communications via SSL.
- Improved auto-healing, when instances or services are terminated abnormally or accidentally. The Postgres operator monitors and automatically restarts any deleted or stopped instances or agents.
- Postgres instances with the same name, in different namespaces, can now be backed up to the same S3 location.

Changed Features

- Updated the pgbackrest sample configuration file, from `pgbackrest.conf` to `pgbackrest.conf.template`.
- The `storageSize` parameter cannot be altered after Postgres instance creation. Any attempt to do so generates an error similar to: `storageSize cannot be reduced after the instance is created. No changes have been made to the running instance.`

Fixed Issues

- [166560384] - Tanzu Postgres backups to an S3 location, using the parameter `verifyTLS: true` and a well-known Certificate Authority, would fail with an error similar to: `2020-12-18 01:01:43.460 P00 DEBUG: common/io/http/request::httpRequestProcess: retry`

`CryptoError: unable to verify certificate presented by 's3.us-west-1.amazonaws.com:443': [20] unable to get local issuer certificate` This issue has been resolved.

- [175791284] - Fixed an issue where `storageSize` updates affected more than one instance, if the instances had matching names in separate namespaces.
- [175885808] - Updated the `s3-secret-example.yaml` file for S3 backups, and all parameters are now specified in double quotes.
- [175618701] - Resolved an issue with the s3 secret `yaml` file, where parameters marked as "optional" but not configured would cause the backup operation to fail.
- [175602831] - When the `archive_mode` flag was `on` (in the `postgresql.conf` file), but the `backupLocationSecret` was left unconfigured in the instance configuration `yaml` file, backups were still attempted. This issue has been resolved.
- [176061339] - Changed the detail level of the pgbackrest console output to `info`.

Release 1.0.0

Release Date: October 30, 2020

Tanzu Postgres 1.0.0 is the first release of VMware Tanzu Postgres on Kubernetes.

Software Components

VMware Postgres Version	Component	Component Version
1.0.0	PostgreSQL	11.9
	psqlODBC	11.0-0000
	pgjdbc	42.2.5
	pgBackRest	2.28
	pg_auto_failover	1.4.0
	postGIS	2.5.4

Features

Tanzu Postgres 1.0.0 has the following features:

- Support for backing up the Postgres instances to an S3 compatible storage location. See Backing Up and Restoring Tanzu Postgres.
- Support for creating a High Availability cluster configuration, with one primary and a mirror. See Configuring High Availability in Tanzu Postgres .
- Flexibility to update the Postgres images after deployment, and reconfigure CPU, memory, and persistent volume storage sizes. See Updating a Postgres Instance Configuration for more information.

Known Issues and Limitations

- Upgrades from the Beta program releases to the Tanzu Postgres 1.0.0 release are not supported. Download and install the latest version.
- The High Availability configuration contains only one mirror.
- The default storage size for the Postgres instance is too limited for long term running Postgres environments. Change the `storageSize` to 10G, and use an expandable storage class. See Deploying a Postgres Instance.
- During an upgrade from 1.0.0 to 1.1.0, in an HA scenario, the Postgres instances state does not show "Ready" until both the primary and the mirror nodes have restarted. This limitation stops the clients from connecting to a read-write instance during the upgrade.

About VMWare Tanzu SQL with Postgres for Kubernetes

VMware Tanzu™ SQL with Postgres for Kubernetes

Tanzu Postgres helps you quickly and reliably deploy Postgres instances on Kubernetes.

Tanzu Postgres packages a collection of 100% open source software, based on the PostgreSQL source code published at http://www.postgresql.org and other open source software from the PostgreSQL community. It also includes a Kubernetes Operator to help you deploy and manage one or more instances of the PostgreSQL database. It includes the following components:

- PostgreSQL – the core ORDBMS database engine.
- pgBackRest – reliable backup and restore for PostgreSQL.
- pg_auto_failover – simple and robust High Availability solution for PostgreSQL.
- PostgreSQL ODBC Driver (psqlODBC) - connectivity for Linux client applications.
- PostgreSQL JDBC Driver (pgjdbc) – connectivity for Java clients.

All components included in the Tanzu Postgres software distribution are intended for Enterprise deployments, and are supported by VMware. See the Support Lifecycle Policy and the Product Support Lifecycle Matrix for details about the duration of support for Tanzu Postgres.

IMPORTANT: VMware Postgres does not support any additional extension components or versions outside the following list. For component inquiries, please contact VMware Support.

VMware Postgres Version	Component	Component Version
1.7.3	PostgreSQL	14.3, 13.7, 12.11, and 11.16
	pgBackRest	2.38
	pg_auto_failover	1.6.4
	postGIS	3.2.1 (for 11, 12,13 and 14)
	Orafce	3.21
	pgAudit	1.6.1 (for 14), 1.5.1 (for 13), 1.4.1 (for 12), 1.3.1 (for 11), 1.2.1 (for 10)

PostgreSQL

PostgreSQL is a powerful, open source object-relational database system that has more than 15 years of active development. It offers a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. PostgreSQL is fully ACID compliant, has full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages). It includes most SQL:2008 data types, including `INTEGER`, `NUMERIC`, `BOOLEAN`, `CHAR`, `VARCHAR`, `DATE`, `INTERVAL`, and `TIMESTAMP`. It also supports storage of binary large objects, including pictures, sounds, or video. PostgreSQL has native programming interfaces for C/C++, Java, .Net, Perl, Python, Ruby, Tcl, ODBC, among others, and exceptional documentation.

Major features of PostgreSQL include:

- Multi-Version Concurrency Control (MVCC)
- Point in time recovery
- Tablespaces
- Asynchronous replication
- Nested transactions (savepoints)
- Online/hot backups
- A sophisticated query planner/optimizer
- Write ahead logging for fault tolerance
- International character sets
- Multibyte character encodings
- Unicode
- Locale-aware support for sorting case-sensitivity, and formatting

Platform Requirements

This version of Tanzu Postgres is supported on the following platforms:

- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI), version 1.11.x - 1.13.x.
    - **Note**: TKGI 1.13.0 and 1.13.1 clusters must be configured with the docker container runtime instead of the default containerd runtime due to an incompatibilty with the containerd provided. Existing clusters upgraded to TKGI 1.13 will continue to use the docker runtime. TKGI 1.13.2 is patched with version 1.5.9 of the containerd runtime to avoid the incompatibility issue.
- VMware Tanzu Kubernetes Grid (TKG) on AWS, version 1.2.x - 1.4.x.
- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Kubernetes version 1.19+

IMPORTANT:: Kubernetes is deprecating Docker as an underlying container runtime. If your environment uses containerd, avoid versions 1.5.6 and 1.5.7. There is a regression that blocks containers from being created if the image label key/value length is larger than 4096 characters.

You can check the container runtime for a Kubernetes cluster with:

```
kubectl get nodes -o wide
```

Installing a Tanzu Postgres Operator

This topic describes how to install Tanzu Postgres.

The primary method for installing Tanzu Postgres Operator is via Helm. For install instructions using the Tanzu Network Registry or a downloadable file, see Installing using Helm.

For Tanzu Application Platform (TAP) customers, the Tanzu Postgres Operator can be installed using the Tanzu CLI. For more details, see Installing using the Tanzu CLI.

Installing using Helm

Prerequisites

To run Tanzu Postgres you need:

- Access to Tanzu Network and Tanzu Network Registry. You can use the same credentials for both sites.
- Docker running and configured on your local computer, to access the Kubernetes cluster and Docker registry.
- A running Kubernetes cluster (check supported providers in Platform Requirements) - and the kubectl command-line tool, configured and authenticated to communicate with your Kubernetes cluster.
    - If you are using GKE, install the gcloud command-line tool.
    - If you are using TKG, install the tanzu command-line tool.
    - If you are using TKGI, install the tkgi command-line tool.
- The Helm v3 command-line tool installed. For more information, see Installing Helm from the Helm documentation.

  Note: Helm CLI 3.7.0 is not supported. Please use 3.7.1 and later.

- `cluster-admin` ClusterRole access to the Kubernetes cluster. For more information, see the Kubernetes documentation.
- review the Network Policies Configuration topic if you have any network plugins (for example Network Plugin) in your Kubernetes cluster.
- Cert Manager installed on the Kubernetes cluster.

  IMPORTANT: TKG users need to upgrade the TKG packaged cert-manager to a version above 1.0.

Install cert-manager by running these commands from your local client:

```
kubectl create namespace cert-manager
helm repo add jetstack https://charts.jetstack.io
helm repo update
helm install cert-manager jetstack/cert-manager --namespace cert-manager  --ver
sion <1.latest> --set installCRDs=true
```

where:

- `--namespace cert-manager` is the namespace used for cert manager in the Kubernetes cluster
- `--version <1.latest>` is the latest cert-manager version available (minimum above 1.0.2)
- `--set installCRDs=true` ensures cert manager installs all types necessary to create certificates

To verify the installation run:

```
kubectl get all --namespace=cert-manager
```

The output should be similar to:

```
NAME                                              READY   STATUS    RESTARTS    AGE
pod/cert-manager-57b65b7fc-x8vjt                  1/1     Running   5           4d1
9h
pod/cert-manager-cainjector-5f988f74c6-tgk25      1/1     Running   15          4d1
9h
pod/cert-manager-webhook-7cf554f879-b5ss9         1/1     Running   4           4d1
9h

NAME                           TYPE        CLUSTER-IP      EXTERNAL-IP    PORT
(S)     AGE
service/cert-manager           ClusterIP   10.106.253.7    <none>         9402/T
CP    4d19h
service/cert-manager-webhook   ClusterIP   10.108.17.113   <none>         443/TC
P     4d19h

NAME                                      READY   UP-TO-DATE   AVAILABLE    AGE
deployment.apps/cert-manager              1/1     1            1            4d19
h
deployment.apps/cert-manager-cainjector   1/1     1            1            4d19
h
deployment.apps/cert-manager-webhook      1/1     1            1            4d19
h

NAME                                                 DESIRED   CURRENT    READY
AGE
replicaset.apps/cert-manager-57b65b7fc               1         1          1
4d19h
replicaset.apps/cert-manager-cainjector-5f988f74c6   1         1          1
4d19h
replicaset.apps/cert-manager-webhook-7cf554f879      1         1          1
4d19h
```

For more advanced security scenarios, see Configuring TLS for Tanzu Postgres Instances.

Accessing the Resources

You can setup Tanzu Postgres using two different methods:

- Use Setup Tanzu Postgres Operator via Tanzu Network Registry for a faster installation process, and if your server hosts have access to the internet.
- Use Setup Tanzu Operator via Downloadable Archive File if your server hosts do not have access to the internet, or if you want to install from a private registry.

Setup the Tanzu Operator via the Tanzu Network Registry

1. Set the environment variable to enable Open Container Initiative (OCI) support in the Helm v3 client by running:

   ```
   export HELM_EXPERIMENTAL_OCI=1
   ```

   If you skip this step, the following error message might appear:

   ```
   Error: this feature has been marked as experimental and is not enabled by defau
   lt.
   ```

2. Use Helm to log in to the Tanzu Network Registry by running:

   ```
   helm registry login registry.tanzu.vmware.com \
          --username=<USERNAME> \
          --password=<PASSWORD>
   ```

   Follow the prompts to enter the email address and password for your Tanzu Network account.

3. Download the Helm chart from the Tanzu Distribution Registry, and export into a local `/tmp/` directory:

   If you're using Helm CLI 3.6 and earlier:

   ```
   helm chart pull registry.tanzu.vmware.com/tanzu-sql-postgres/postgres-operator-
   chart:v1.7.2
   helm chart export registry.tanzu.vmware.com/tanzu-sql-postgres/postgres-operato
   r-chart:v1.7.2  --destination=/tmp/
   ```

   If you're using Helm CLI 3.7.1 and later:

   ```
   helm pull oci://registry.tanzu.vmware.com/tanzu-sql-postgres/postgres-operator-
   chart --version v1.7.2 --untar --untardir /tmp
   ```

4. Follow the steps in Installing the Operator.

Setup the Tanzu Operator via a Downloaded Archive File

Choose this method if the installation destination (for example an air-gapped network) cannot access the VMware Tanzu Network, or you wish to load the Operator and instance images to private Docker registry.

1. Download the Tanzu Postgres distribution from Broadcom Support. The Tanzu Postgres download filename has the format: `postgres-for-kubernetes-v<version>.tar.gz`
2. Unpack the downloaded software:

   ```
   cd ~/Downloads
   tar xzf postgres-for-kubernetes-v<version>.tar.gz
   ```

   This command unpacks the distribution into a new directory named `postgres-for-kubernetes-v<version>`, for example `postgres-for-kubernetes-v1.7.2`.
3. Change to the new `postgres-for-kubernetes-v<version>` directory.

   ```
   cd ./postgres-for-kubernetes-v*
   ```

4. Load the Postgres instance image.

   ```
   docker load -i ./images/postgres-instance
   ```

   ```
   cc967c529ced: Loading layer [==================================================
   >]  65.57MB/65.57MB
   2c6ac8e5063e: Loading layer [==================================================
   >]  991.2kB/991.2kB
   6c01b5a53aac: Loading layer [==================================================
   >]  15.87kB/15.87kB
   e0b3afb09dc3: Loading layer [==================================================
   >]  3.072kB/3.072kB
   faee4b69eae8: Loading layer [==================================================
   >]  29.74MB/29.74MB
   6bc08b5f8a06: Loading layer [==================================================
   >]  4.096kB/4.096kB
   3bfb028071fa: Loading layer [==================================================
   >]  331.4MB/331.4MB
   6ef1a056590e: Loading layer [==================================================
   >]  57.86kB/57.86kB
   Loaded image: postgres-instance:v1.7.2
   ```

5. Load the Postgres operator image.

   ```
   docker load -i ./images/postgres-operator
   ```

   ```
   0d1435bd79e4: Loading layer [==================================================
   >]  3.062MB/3.062MB
   b50265a0f809: Loading layer [==================================================
   >]  40.87MB/40.87MB
   Loaded image: postgres-operator:v1.7.2
   ```

6. Verify that the two Docker images are now available.

   ```
   docker images "postgres-*"
   ```

   ```
   REPOSITORY          TAG       IMAGE ID       CREATED        SIZE
   postgres-operator   v1.7.2    063a6186109b   10 days ago    111MB
   postgres-instance   v1.7.2    cc6ca2396fda   10 days ago    1.72GB
   ```

7. Push the Tanzu Postgres Docker images to the container registry of your choice. Set each image's project and image repo name, tag the images, and then push them using the Docker command `docker push`.

   This example tags and pushes the images to the Google Cloud Registry, using the default (core) project name for the example Google Cloud account.

   ```
   gcloud auth configure-docker

   PROJECT=$(gcloud config list core/project --format='value(core.project)')
   REGISTRY="gcr.io/${PROJECT}"

   INSTANCE_IMAGE_NAME="${REGISTRY}/postgres-instance:$(cat ./images/postgres-inst
   ance-tag)"
   docker tag $(cat ./images/postgres-instance-id) ${INSTANCE_IMAGE_NAME}
   docker push ${INSTANCE_IMAGE_NAME}

   OPERATOR_IMAGE_NAME="${REGISTRY}/postgres-operator:$(cat ./images/postgres-oper
   ator-tag)"
   docker tag $(cat ./images/postgres-operator-id) ${OPERATOR_IMAGE_NAME}
   docker push ${OPERATOR_IMAGE_NAME}
   ```

8. Follow the steps in Installing the Operator.

Installing the Operator

Create a Kubernetes Access Secret

Create a `docker-registry` type secret to allow the Kubernetes cluster to authenticate with the private container registry, or the Tanzu Registry, so it can pull images. These examples create a secret named `regsecret`, in the current namespace (in this example it's the `default`), using VMware Tanzu Network, or Harbor.

IMPORTANT: Only pods created in the current `default` namespace can reference this secret. To create the instance in a different namespace, use the `--namespace` flag.

VMware Tanzu Network

```
kubectl create secret docker-registry regsecret \
    --docker-server=https://registry.tanzu.vmware.com/ \
    --docker-username='USERNAME' \
    --docker-password='PASSWD'
```

where `USERNAME` and password `PASSWD` are your access credentials to the VMware Tanzu Network. Surround both the `USERNAME` and the `PASSWD` by single quote marks to handle any special characters within those values.

Harbor

```
kubectl create secret docker-registry regsecret \
    --docker-server=${HARBOR_URL} \
    --docker-username=${HARBOR_USER} \
    --docker-password="${HARBOR_PASSWORD}"
```

The Postgres Operator will use this secret to allow the Kubernetes cluster to authenticate with the container registry to pull images.

Review the Operator Values

This step is optional. Go to the directory where you unpacked the Tanzu Postgres distribution. View the file `operator/values.yaml` in the Tanzu Postgres directory:

```
cat ./operator/values.yaml
```

The file specifies the location of the Postgres Operator and instance images. By default it contains the following values:

```
---
# specify the url for the docker image for the operator, e.g. gcr.io/<my_project>/post
gres-operator
operatorImage: registry.tanzu.vmware.com/tanzu-sql-postgres/postgres-operator:v1.7.2

# specify the docker image for postgres instance, e.g. gcr.io/<my_project>/postgres-in
stance
postgresImage: registry.tanzu.vmware.com/tanzu-sql-postgres/postgres-instance:v1.7.2

# specify the name of the docker-registry secret to allow the cluster to authenticate
with the container registry for pulling images
dockerRegistrySecretName: regsecret

# override the default self-signed cert-manager cluster issuer
certManagerClusterIssuerName: postgres-operator-ca-certificate-cluster-issuer

# set the resources for the postgres operator deployment
resources: {}
#   limits:
#     cpu: 100m
#     memory: 128Mi
#   requests:
#     cpu: 100m
#     memory: 128Mi
```

Determine which values in the `values.yaml` file need to be changed for your environment. Use the table below as a guide.

Key	Value Type	Description
`operatorImage`	URI	Reference to the Tanzu Postgres Operator image. Change this reference to show the URI of your private registry where you uploaded the Operator image.
`instanceImage`	URI	Reference to the Tanzu Postgres image. Change this reference to show the URI of your private registry where you uploaded the instance image.
`dockerRegistrySecretName`	String	Name of image secret. This value must match the name of the Kubernetes secret you created in Create a Kubernetes Access Secret above.

Key	Value Type	Description
`certManagerClusterIssuerName`	String	Name of TLS issuer. Change this field to match your custom CA issuer if you're using TLS. See Configuring TLS for Tanzu Postgres Instances.
`resources`	Object	Limits and requests for CPU and memory for the Operator. You can change these values to scale your resources.

To alter any of the default values create a `operator-values-overrides.yaml` (choose your own name) configuration file under the same location, and specify any custom values, for example a custom container registry, and a secret. For manual changes, you may also set individual parameters using the `--set` flag on the command line. See Helm Values Files in the Helm documentation for more information.

An example `values-overrides.yaml` file could contain the following lines, replacing `${REGISTRY}` with your private container registry name:

```
operatorImage: ${REGISTRY}/postgres-operator:v1.7.2
postgresImage: ${REGISTRY}/postgres-instance:v1.7.2
```

Deploy the Operator

1. Verify you don't have previously installed instance CRDs in your cluster:

   ```
   kubectl get crd postgres.sql.tanzu.vmware.com
   ```

   If this is a brand new Operator installation, the result should be similar to:

   ```
   Error from server (NotFound): customresourcedefinitions.apiextensions.k8s.io "p
   ostgres.sql.tanzu.vmware.com" not found
   ```

   If the result is similar to:

   ```
   NAME                             CREATED AT
   postgres.sql.tanzu.vmware.com    2021-06-09T06:04:45Z
   ```

   there are older instances running in the cluster, from a previous Operator deployment. When deploying the Operator, you need to refresh this CRD in order to apply the new updated Operator version (see step 5).

2. Install the Tanzu Postgres Operator by running one of the following:

   If you created a custom `operator-values-overrides.yaml` run the following helm command:

   ```
   helm install <OPERATOR_NAME> <PATH_TO_CHART> \
       --values=<PATH_TO_HELM_OVERRIDES_FILE> \
       --namespace=<OPERATOR_NAMESPACE> \
       --wait
   ```

   where

   - `OPERATOR_NAME` is a custom name for the helm release

  - `PATH_TO_CHART` is where you downloaded the helm chart based on the setup procedure
  - `--values` (optional) specifies the path where the helm override file resides
  - `--namespace` (optional) specifies the namespace you wish to deploy the Operator in, which must match the namespace for the secret created in Create a Kubernetes Access Secret
  - `--wait` flag waits for the Operator deployment to complete before any image installation starts

  If you did not create an `operator-values-overrides.yaml` configuration file run:

```
helm install my-postgres-operator /tmp/postgres-operator/ --wait
```

```
NAME: my-postgres-operator
LAST DEPLOYED: Wed Jun 16 13:28:05 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

```
helm install my-postgres-operator /tmp/postgres-operator/ \
  --namespace=${OPERATOR_NAMESPACE} \
  --create-namespace \
  --wait
```

  Note: The secret namespace in step Create a Kubernetes Access Secret must match the Operator namespace.

  Installing the Operator creates a new service account named `postgres-operator-service-account`. It is for internal use, but it is visible if you use the kubectl `get serviceaccount` command:

```
kubectl get serviceaccount
```

```
NAME                                 SECRETS    AGE
default                              1          12m
postgres-operator-service-account    1          8m56s
```

3. Use `watch kubectl get all` to monitor the progress of the deployment. The deployment is complete when the Postgres Operator pod status changes to `Running`. Use the label `app=postgres-operator` to search across resources created by the Postgres Operator Helm chart.

```
watch kubectl get all --selector app=postgres-operator
```

   If your namespace is different than the `default`, use the `-n <your-namaspace>` to specify your namespace.

```
Every 2.0s: kubectl get all
```

```
NAME                                      READY   STATUS    RESTARTS   AGE
pod/postgres-operator-6754b58976-24zwx    1/1     Running   0          5m15s

NAME                                         TYPE        CLUSTER-IP       EXTERN
AL-IP   PORT(S)   AGE
service/postgres-operator-webhook-service    ClusterIP   10.101.230.150   <none>
443/TCP   5m15s

NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/postgres-operator       1/1     1            1           5m15s

NAME                                             DESIRED   CURRENT   READY   AGE
replicaset.apps/postgres-operator-6754b58976     1         1         1       5m15
s
```

You may also check the logs to confirm the Operator is running properly:

```
kubectl logs -l app=postgres-operator
```

To view all the Operator resources run:

```
kubectl api-resources --api-group=sql.tanzu.vmware.com
```

```
NAME                          SHORTNAMES   APIVERSION                 NAMESPACED   K
IND
postgres                      pg           sql.tanzu.vmware.com/v1    true         P
ostgres
postgresbackuplocations                    sql.tanzu.vmware.com/v1    true         P
ostgresBackupLocation
postgresbackups                            sql.tanzu.vmware.com/v1    true         P
ostgresBackup
postgresbackupschedules                    sql.tanzu.vmware.com/v1    true         P
ostgresBackupSchedule
postgresrestores                           sql.tanzu.vmware.com/v1    true         P
ostgresRestore
postgresversions                           sql.tanzu.vmware.com/v1    false        P
ostgresVersion
```

4. If you have existing Postgres instances running from a previous Operator deployment, go to the location you setup your Operator:

```
cd /<your-path>/postgres-for-kubernetes-v<your-version>/
```

and re-apply the instance CRD, using a command similar to:

```
kubectl apply -f operator/crds/
```

Installing using the Tanzu CLI

Prerequisites

- A Tanzu Network account to access images from the VMware Tanzu Network registry.

- Before using the Tanzu CLI, certain prerequisites (kapp-controller and secretgen-controller) must be installed on the Kubernetes cluster. For details on these requirements review Accepting Tanzu Application Platform EULAs, installing Cluster Essentials and the Tanzu CLI in the TAP documentation.
- Cert Manager installed on the Kubernetes cluster.

Relocate Images to a Private Registry

Relocate the images from VMware Tanzu Network registry to a private registry before attempting installation. The VMware Tanzu Network registry does not offer uptime guarantees for installations. Skipping image relocation should only occur when configuring an evaluation, testing, or proof-of-concept environment.

To relocate images from the VMware Tanzu Network registry to a private registry:

1. Log in to your image registry by running:

   ```
   docker login <MY-REGISTRY>
   ```

   where:

   - `MY-REGISTRY` is your own image registry

2. Log in to the VMware Tanzu Network registry with your VMware Tanzu Network credentials by running:

   ```
   docker login registry.tanzu.vmware.com
   ```

3. Relocate the images with the Carvel tool `imgpkg` by running:

```
imgpkg copy -b registry.tanzu.vmware.com/packages-for-vmware-tanzu-data-services/tds-p
ackages:<TDS-VERSION> --to-repo <MY-REGISTRY>/<TARGET-REPOSITORY>/tds-packages
```

where:

- `MY-REGISTRY` is your own image registry
- `TARGET-REPOSITORY` is your target repository
- `TDS-VERSION` is the tag for the image bundle (e.g `1.0.0`)

Create a Kubernetes Secret

Verify the existing secrets in your environment:

```
tanzu secret registry list
```

The output would be similar to:

```
NAME                  REGISTRY                        EXPORTED             AGE
test-registry         my-registry                     to all namespaces    47h
tanzu-registry        registry.tanzu.vmware.com       to all namespaces    47h
```

Verify there is an exported secret for your custom image registry. If there is no associated secret, create a secret and export the secret to all namespaces:

```
tanzu secret registry add <SECRET-NAME> \
    --username <MY-REGISTRY-USERNAME> \
    --password <MY-REGISTRY-PASSWORD> \
    --server <MY-REGISTRY> \
    --export-to-all-namespaces --yes
```

where:

- `SECRET-NAME`: is the name of the Kubernetes secret that will be created
- `MY-REGISTRY` is your own image registry
- `MY-REGISTRY-USERNAME` is the username for your own container registry
- `MY-REGISTRY-PASSWORD` is the password for your own container registry

Add the Package Repository

Add the package repository for VMware Tanzu Data Services:

```
tanzu package repository add tanzu-data-services-repository --url <MY-REGISTRY>/<TARGE
T-REPOSITORY>/tds-packages
```

where:

- `MY-REGISTRY` is your own image registry
- `TARGET-REPOSITORY` is your target repository

List the available packages to confirm the addition:

```
tanzu package available list
```

```
- Retrieving available packages...
  NAME                                      DISPLAY-NAME
SHORT-DESCRIPTION                    LATEST-VERSION
  postgres-operator.sql.tanzu.vmware.com    VMware Tanzu SQL with Postgres for Kuberne
tes  Kubernetes Operator for PostgreSQL  1.7.2
```

Check the values for the Postgres Operator package:

```
tanzu package available get postgres-operator.sql.tanzu.vmware.com/1.7.2 --values-sche
ma
```

```
- Retrieving package details for postgres-operator.sql.tanzu.vmware.com/1.7.2...
  KEY                            DEFAULT                                            TYPE
DESCRIPTION
  certManagerClusterIssuerName  postgres-operator-ca-certificate-cluster-issuer  strin
g  A cert-manager based clusterissuer used to sign postgres certificates using a custo
m certificate authority
  dockerRegistrySecretName      regsecret                                          strin
g  The name of the docker-registry secret to allow the cluster to authenticate with th
```

```
e container registry for pulling images
  resources                     map[]                                          objec
t  Resources describes the CPU and Memory compute resource requirements for Postgres o
perator pod
```

Consider overriding the Operator values in a separate YAML file, if the defaults do not suit your deployment environment. A sample overrides YAML could be:

```
certManagerClusterIssuerName: custom-issuer
dockerRegistrySecretName: custom-secret
resources:
  limits:
    cpu: 500m
    memory: 300Mi
  requests:
    cpu: 500m
    memory: 300Mi
```

Installing the Operator

1. **Install the operator package**

   Install the Postgres operator package, using the overrides file you created:

   ```
   tanzu package install <PACKAGE-NAME> --package-name postgres-operator.sql.tanz
   u.vmware.com --version 1.7.2 -f <YOUR-OVERRIDES-FILE-PATH>
   ```

   where:

   - `PACKAGE-NAME` is the name you choose for the package installation.
   - `YOUR-OVERRIDES-FILE-PATH` is your custom overrides path and file, for example `overrides.yaml`.

   The output is similar to:

   ```
   / Installing package 'postgres-operator.sql.tanzu.vmware.com'
   | Getting package metadata for 'my-postgres-operator.sql.tanzu.vmware.com'
   | Creating service account 'my-postgres-operator-default-sa'
   | Creating cluster admin role 'my-postgres-operator-default-cluster-role'
   | Creating cluster role binding 'my-postgres-operator-default-cluster-rolebindi
   ng'
   | Creating secret 'my-postgres-operator-default-values'
   | Creating package resource
   - Waiting for 'PackageInstall' reconciliation for 'my-postgres-operator'
   \ 'PackageInstall' resource install status: Reconciling


   Added installed package 'my-postgres-operator'
   ```

2. **Verify PackageInstall has been created**

   ```
   tanzu package installed list
   ```

```
- Retrieving installed packages...
  NAME                  PACKAGE-NAME                             PACKAGE-VERSION
STATUS
  my-postgres-operator  postgres-operator.sql.tanzu.vmware.com  1.7.2
Reconcile succeeded
```

A service account is created so that the `kapp-controller` can create cluster-scope objects such as CustomResourceDefinitions, and so it will have permissions to create objects on any namespace. This service account is different than the service account for the Postgres operator to manage other Kubernetes resources (statefulsets, secrets, etc...)

To check the service accounts run:

```
kubectl get serviceaccount
```

```
NAME                                    SECRETS   AGE
default                                 1         4d4h
my-postgress-operator-default-sa        1         12m
my-postgres-operator-service-account    1         12m
```

3. **Verify the Operator Deployment**

Use `watch kubectl get all` to monitor the progress of the deployment. The deployment is complete when the Postgres Operator pod status changes to `Running`. Use the label `app=postgres-operator` to search across resources created by the Postgres Operator.

```
watch kubectl get all --selector app=postgres-operator
```

If your namespace is different than the `default`, use `-n <your-namaspace>` to specify your namespace.

```
Every 2.0s: kubectl get all

NAME                                     READY   STATUS    RESTARTS   AGE
pod/postgres-operator-6754b58976-24zwx   1/1     Running   0          5m15s

NAME                                        TYPE        CLUSTER-IP       EXTERN
AL-IP   PORT(S)   AGE
service/postgres-operator-webhook-service   ClusterIP   10.101.230.150   <none>
443/TCP   5m15s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/postgres-operator   1/1     1            1           5m15s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/postgres-operator-6754b58976   1         1         1       5m15
s
```

You may also check the logs to confirm the Operator is running properly:

```
kubectl logs -l app=postgres-operator
```

To view all the Operator resources run:

```
kubectl api-resources --api-group=sql.tanzu.vmware.com
```

```
NAME                        SHORTNAMES   APIVERSION               NAMESPACED   K
IND
postgres                    pg           sql.tanzu.vmware.com/v1  true         P
ostgres
postgresbackuplocations                  sql.tanzu.vmware.com/v1  true         P
ostgresBackupLocation
postgresbackups                          sql.tanzu.vmware.com/v1  true         P
ostgresBackup
postgresbackupschedules                  sql.tanzu.vmware.com/v1  true         P
ostgresBackupSchedule
postgresrestores                         sql.tanzu.vmware.com/v1  true         P
ostgresRestore
postgresversions                         sql.tanzu.vmware.com/v1  false        P
ostgresVersion
``
```

Next steps

After you install the Postgres Operator, you can use it to deploy and manage Postgres instances. To interact with the Postgres Operator, you place a set of instructions into a YAML-formatted configuration file (a Kubernetes manifest) and then use the `kubectl` utility to send the file instructions to the Operator. The Postgres Operator is then responsible for following the instructions that you provide, and also for maintaining the state of the Postgres instance according to the properties that you defined.

For more details, see:

- Deploying a New Postgres Instance
- Updating a Postgres Instance Configuration
- Deleting a Postgres Instance
- Upgrading the Operator or the Postgres Instances

Deploying a Postgres Instance

This section describes how to deploy a Postgres instance to your Kubernetes cluster, using the Postgres operator. Use these instructions either to deploy a brand new instance (by provisioning a new empty Persistent Volume Claims in Kubernetes), or to update an instance by re-using existing Persistent Volumes (PVC) if available.

Prerequisites

1. Ensure you have installed the Tanzu Postgres docker images and created the Postgres operator in your Kubernetes cluster. See Installing a Postgres Operator for instructions.

   Verify that the Postgres operator is installed and running in your system:

   ```
   helm list
   ```

   ```
   NAME                        REVISION        UPDATED                                 STATUS
   CHART                        APP VERSION       NAMESPACE
   postgres-operator           1               Fri Feb 25 16:03:19 2022                DEPLOYE
   D       postgres-operator-1.7.2  v1.7.2                 default
   ```

2. Request an expandable storage volume for your Postgres instance, to be able to resize the volume online. For more information, see Allow Volume Expansion.

   Ensure that the storage class `VOLUMEBINDINGMODE` field is set to `volumeBindingMode=WaitForFirstConsumer`, to avoid Postgres pods and Persistent Volumes (PV) scheduling issues. For more details on the Kubernetes storage class binding modes see Volume Binding Mode.

   To verify the `ALLOWVOLUMEEXPANSION` and `VOLUMEBINDINGMODE` fields use:

   ```
   kubectl get storageclasses
   ```

   The output would be similar to:

   ```
   NAME                     PROVISIONER                  RECLAIMPOLICY   VOLUMEBINDINGMO
   DE          ALLOWVOLUMEEXPANSION   AGE
   standard (default)    k8s.io/minikube-hostpath    Delete          WaitForFirstCon
   sumer       true                    4h25m
   ```

3. If you're planning to bind a TAP application workload to the Postgres database, and wish to change the default `pgappuser` application user name, edit your instance yaml before deployment. For details see Custom Database Name and User Account.

Configuring a Postgres Instance

1. Target the namespace where you want to create the Postgres instance:

   ```
   kubectl config set-context --current --namespace=<POSTGRES-NAMESPACE>
   ```

   where `POSTGRES-NAMESPACE` is the namespace you want to deploy the Postgres instance.

2. From this namespace, create a secret that Kubernetes will use to access the registry that stores the Tanzu Postgres images:

   ```
   kubectl create secret --namespace=POSTGRES-NAMESPACE docker-registry regsecret
   \
      --docker-server=https://registry.tanzu.vmware.com
      --docker-username=`USERNAME`
      --docker-password=`PASSWD`
   ```

   Use `my-postgres-secret` for the field `imagePullSecret` in your custom Postgres yaml file, that you create in the next steps.

3. Locate the sample Postgres manifest `postgres.yaml` in the `./samples` directory of the location where you unpacked the Tanzu Postgres distribution.

   ```
   cd ./postgres-for-kubernetes-v*
   ```

4. Copy the example `postgres.yaml` to a new file, and customize the values according to your needs. The values in the sample file are only examples, and they include:

```
---
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: postgres-sample
spec:
  #
  # Global features
  #
  pgConfig:
    dbname: postgres-sample
    username: pgadmin
    appUser: pgappuser
  postgresVersion:
    name: postgres-14 # View available versions with `kubectl get postgresversion`
  serviceType: ClusterIP
#  serviceAnnotations:
  seccompProfile:
    type: RuntimeDefault
  imagePullSecret:
    name: regsecret
  #  highAvailability:
  #    enabled: true
  #  logLevel: Debug
  #  backupLocation:
  #    name: backuplocation-sample
  #  certificateSecretName:
```

```
  #
  # Data Pod features
  #
  storageClassName: standard
  storageSize: 800M
  cpu: "0.8"
  memory: 800Mi
  dataPodConfig:
#    tolerations:
#      - key:
#        operator:
#        value:
#        effect:
    affinity:
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                  - key: type
                    operator: In
                    values:
                      - data
                      - monitor
                  - key: postgres-instance
                    operator: In
                    values:
                      - postgres-sample
              topologyKey: kubernetes.io/hostname
            weight: 100


  #
  # Monitor Pod features
  #
  monitorStorageClassName: standard
  monitorStorageSize: 1G
  monitorPodConfig:
#    tolerations:
#      - key:
#        operator:
#        value:
#        effect:
    affinity:
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                  - key: type
                    operator: In
                    values:
                      - data
                      - monitor
                  - key: postgres-instance
                    operator: In
                    values:
                      - postgres-sample
              topologyKey: kubernetes.io/hostname
          weight: 100

  #
  # Resources
  #
  resources:
    monitor:
      limits:
        cpu: 800m
        memory: 800Mi
      requests:
        cpu: 800m
        memory: 800Mi
    metrics:
      limits:
        cpu: 100m
        memory: 100Mi
      requests:
        cpu: 100m
        memory: 100Mi
```

```
For details on the Postgres CR values see the [Postgres Deployment Properties](postgre
s-crd-reference.html) page.

**IMPORTANT**: The default values for `spec.memory`, `spec.CPU`, `spec.storageClassNam
e`, and `spec.storageSize` specify a very small Postgres instance that may be too limi
ted for your use case.

To review the defaults for your instance use a commands similar to:

```
kubectl get postgres <your-instance-name> -o yaml
```
```

Specifying the Tanzu Postgres Version

The Tanzu Postgres Operator by default deploys the latest Postgres version (for Tanzu Operator 1.7.2, the Postgres version is 14.3). To view the available Tanzu Postgres versions for your Operator, run the command:

```
kubectl get postgresversion
```

The command displays:

```
NAME           DB VERSION
postgres-11    11.16
postgres-12    12.11
postgres-13    13.7
postgres-14    14.3
```

where:

- `NAME` denotes the `postgresVersion` CR name. Each postgres major version has one CR. This value can be used in the Postgres manifest file to choose the postgres version.
- `DB VERSION` displayes the minor version supported for that particular Postgres major version.

Use the values under the column `NAME` to specify the `spec.postgresVersion.name` field in the Postgres instance manifest if you require a specific version of Postgres, for example:

```
...
postgresVersion:
  name: postgres-13
...
```

See Deploying a Postgres Instance on how to deploy your Postgres instance.

When upgrading the Tanzu Postgres Operator, the Operator will ensure that existing Postgres instances have the appropriate version reference added to the object. Specifically, the Operator will set `spec.postgresVersion.name` to `postgres-11`. If you are tracking manifest files in source control, update those manifests files to reflect the change.

IMPORTANT: Existing Postgres instances cannot be upgraded to a different major version.

Specifying Namespaces

The sample configuration manifest omits a namespace, so the Postgres object will be created in whatever namespace is set in the kubectl context. If you wish to create objects in a different namespace, ensure that you have created your registry secrets in the new namespace and defined the `namespace` field nested under the `metadata` field. For example, to create a postgres instance `postgres-sample` in the `postgres-databases` namespace, edit the file accordingly:

```
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: postgres-sample
  namespace: postgres-databases
spec:
  imagePullSecret:
    name: postgres-databases-registry-secret
......
```

where `spec.imagePullSecret.name` is the registry secret you defined during the Postgres Operator deployment, see Create a Kubernetes Access Secret.

You may create multiple Postgres instances with the same YAML file, separating the configurations with three `---`:

```
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: postgres-ha-sample
  namespace: postgres-databases
spec:
  memory: 800Mi
  cpu: "0.8"
  storageClassName: standard
  storageSize: 800M
  serviceType: LoadBalancer
  highAvailability:
    enabled: true
```

```
---
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: pg-mypostgres
  namespace: postgres-databases
spec:
  memory: 800Mi
  cpu: "0.8"
  storageClassName: standard
  storageSize: 10G
  highAvailability:
    enabled: false
```

Custom Database Name and User Account

When creating a Postgres instance, the default database name matches the instance name, as described in step 2 in Configuring the Postgres Instance Manifest file.

To create a custom database name and account username, configure the `pgConfig` field values in the manifest file. The following example creates a Postgres instance called `postgres-sample`, with a database named `postgres-sample` and a user called `pgadmin`.

```
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: postgres-sample
spec:
  memory: 800Mi
  cpu: "0.8"
  storageClassName: standard
  storageSize: 10G
  pgConfig:
    dbname: postgres-sample
    username: pgadmin
    appUser: pgappuser
```

Where:

- `dbname` (optional) is the name of the default database created when the Postgres instance is initiated. The `dbname` string must be less than 63 characters, and can contain any characters and capitalization. If the `dbanme` field is left empty, the database name defaults to the instance name.
- `username` (optional) is the database username account for the specified database. By default this user inherits all Read/Write permissions to all databases in the instance. If left empty, the default username is `pgadmin`.
- `appUser` (optional) specifies the name of the Postgres user with read-write privileges. It will be used to bind an application with the Postgres instance. The default Service Binding application user is `pgappuser`. You may change during instance deployment to a value of your choice.

Updating the Monitor Resources

When the Operator creates a Postgres instance, it also creates a monitor pod that holds the state information for the instance environment.

To view the default values use:

```
kubectl get postgres/postgres-sample -o yaml
```

```
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"sql.tanzu.vmware.com/v1","kind":"Postgres","metadata":{"annotatio
ns":{},"name":"postgres-sample","namespace":"default"},"spec":{"cpu":"0.8","memory":"8
00Mi","monitorStorageClassName":"standard","monitorStorageSize":"1G","pgConfig":{"dbna
me":"postgres-sample","username":"pgadmin"},"postgresVersion":{"name":"postgres-1
4"},"resources":{"metrics":{"limits":{"cpu":"100m","memory":"100Mi"},"requests":{"cp
u":"100m","memory":"100Mi"}},"monitor":{"limits":{"cpu":"800m","memory":"800Mi"},"requ
ests":{"cpu":"800m","memory":"800Mi"}}},"serviceType":"ClusterIP","storageClassNam
e":"standard","storageSize":"800M"}}
  creationTimestamp: "2021-11-19T18:51:40Z"
  generation: 3
  labels:
    app: postgres
    postgres-instance: postgres-sample
  name: postgres-sample
  namespace: default
  resourceVersion: "12427"
  uid: 1ccb90a2-9990-4a2c-8cf9-a3b9d5e1e53e
spec:
  backupLocation: {}
  cpu: 800m
  highAvailability: {}
  memory: 800Mi
  monitorStorageClassName: standard
  monitorStorageSize: 1G
  pgConfig:
    dbname: postgres-sample
    username: pgadmin
  postgresVersion:
    name: postgres-14
  resources:
    metrics:
      limits:
        cpu: 100m
        memory: 100Mi
      requests:
        cpu: 100m
        memory: 100Mi
    monitor:
      limits:
        cpu: 800m
        memory: 800Mi
      requests:
        cpu: 800m
        memory: 800Mi
  serviceType: ClusterIP
  storageClassName: standard
  storageSize: 800M
status:
  currentState: Running
  dbVersion: "14.1"
```

Alter the monitor resources in the instance `yaml` to reflect your requirements. For example, change the CPU limit from 800m to 900m:

```
.....
  resources:
    monitor:
      limits:
        cpu: 900m
        memory: 800Mi
      requests:
        cpu: 800m
        memory: 800Mi
....
```

Apply the changes:

```
kubectl apply -f postgres.yaml
```

The monitor will restart and the new values will take effect. Verify the changes using the `describe` command:

```
kubectl describe pod/postgres-sample-monitor-0
```

The output includes the new updates:

```
...
Containers:
  monitor:
    Container ID:  docker://9dc1f58fbe8042497d05004e1d084f8976996d2d92c1dad474cb6996ee
d2319b
    Image:         postgres-instance:latest
    Image ID:      docker://sha256:f493b6e8139a9728663034914b4a8e5c3416fca0f548d49f61a
52e4ed2ec3be3
    Port:          <none>
    Host Port:     <none>
    Args:
      /usr/local/apps/start_monitor
    State:          Running
      Started:      Thu, 24 Jun 2021 12:36:15 -0700
    Ready:          False
    Restart Count:  0
    Limits:
      cpu:     900m
      memory:  800Mi
    Requests:
      cpu:      800m
      memory:   800Mi
...
```

For details on resource `requests` and `limits` see Managing Resources for Containers in the Kubernetes documentation.

Configuring Node Affinity and Tolerations

Tanzu Postgres Operator supports the affinity/anti-affinity and tolerations feature, that introduces advanced scheduling for pods. Affinity rules help schedule pods with mission critical workloads on specific high performant and resilient nodes. This feature also allows pod scheduling based on the failover strategy, or low latency goals.

The Tanzu Postgres Operator adds a monitor and data pod affinity and tolerations section to the Postgres manifest. The `monitorPodConfig` includes the following fields:

```
  monitorPodConfig:
#    tolerations:
#      - key:
#        operator:
#        value:
#        effect:
    affinity:
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                  - key: type
                    operator: In
                    values:
                      - data
                      - monitor
                  - key: postgres-instance
                    operator: In
                    values:
                      - postgres-sample
              topologyKey: kubernetes.io/hostname
            weight: 100
```

Edit the tolerations section to customize the values based on your environment. By default, the Tanzu Postgres Operator does not apply any tolerations.

The default `affinity` rule is a pod `preferred anti-affinity` rule that tries to avoid scheduling the monitor and data pods of the same instance (key: postgres-instance) on the same node, based on the standard `topologyKey:kubernetes.io/hostname` node label. The `matchExpressions` use the operator `In`. You may create your own custom rules using operators like `NotIn`, `Exists`, `DoesNotExist`, `Gt`, `Lt`, `NotIn`, or `DoesNotExist`. See Inter-pod affinity and anti-affinity for more information on the pod affinity or anti-affinity rules.

The `dataPodConfig` includes the following:

```
dataPodConfig:
#   tolerations:
#      - key:
#        operator:
#        value:
#        effect:
    affinity:
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                  - key: type
                    operator: In
                    values:
                      - data
                      - monitor
                  - key: postgres-instance
                    operator: In
                    values:
                      - postgres-sample
              topologyKey: kubernetes.io/hostname
            weight: 100
```

For details on the `tolerations` sub-key, refer to the Taints and Tolerations topic in the Kubernetes documentation.

For further examples on Tanzu Postgres affinity and tolerations, see Postgres Deployment Properties.

Quality of Service

To implement a **Guaranteed Quality of Service (QoS)** for any of the resources (for example, primary, mirror, metrics or monitor), set the limits equal to the requests. When the limits are higher than the request, the QoS is Burstable. By default, the monitor, primary, and the mirror have a Guaranteed QoS. To check the `status.qosClass` of your instance, use:

```
kubectl describe pod/postgres-sample-0 | grep "QoS Class:"
```

Security Profile

To enable a **security profile (seccomp)** for the instance, edit the field `seccompProfile:type: RuntimeDefault`. The default `RuntimeDefault` is the most restrictive. For further details on the field, see the Postgres Deployment Properties page, and also Restrict a Container's Syscalls with seccomp in the Kubernetes documentation.

Internal Load Balancer

When deploying instances in a public cloud, you can enable cloud-specific behaviour on the load balancer service. Edit the Postgres manifest file and change the default `serviceType` to `LoadBalancer`, and edit the field `serviceAnnotations` with the values required for your cloud environment. For example, for Azure, AWS, or Google, you could use similar to:

```
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: postgres-sample
spec:
  serviceType: LoadBalancer
  serviceAnnotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true",
    service.beta.kubernetes.io/azure-load-balancer-internal-subnet: "apps-subnet"
    cloud.google.com/load-balancer-type: "Internal"
    service.beta.kubernetes.io/aws-load-balancer-internal: "true"
```

For more information, see Internal Load Balancer in the Kubernetes documentation.

Deploying a Postgres Instance

1. Request a Postgres instance using your manifest file.

   ```
   kubectl apply -f postgres.yaml
   ```

   ```
   postgres.sql.tanzu.vmware.com/postgres-sample created
   ```

   The Postgres operator deploys the resources according to your specification, and also initializes the Postgres instance. If there are no existing Persistent Volume Claims (PVC) for the instance, new PVCs are created and used for the deployment. If a PVC for the instance already exists, it is used as-is with the available data.

2. Check the status of the instance to verify that it was created successfully:

   ```
   kubectl get postgres/postgres-sample
   ```

   You should see output similar to:

   ```
   NAME              STATUS    DB VERSION   BACKUP LOCATION   AGE
   postgres-sample   Running   14.1                           4m29s
   ```

   where `DB VERSION` displays the corresponding major/minor version associated with the `spec.postgresVersion.name` field in the instance manifest file. If left at the default value, it defaults to `postgres-11` and the `DB VERSION` column displays 14.1.

Using the Postgres Instance

If you are in an HA configuration (for details see Configuring High Availability in Tanzu Postgres), ensure you are connecting to the primary pod. To confirm which pod is primary or secondary, use a command similar to:

```
kubectl exec -ti pod/postgres-sample-1 -- pg_autoctl show state


Name |  Node |                                                          Host:Port
|       TLI: LSN |   Connection |       Current State |      Assigned State
-------+-------+-----------------------------------------------------------------
--+----------------+--------------+--------------------+--------------------
node_1 |      1 | postgres-sample-0.postgres-sample-agent.default.svc.cluster.local:543
2 |    2: 0/3002690 |    read-only |            secondary |            secondary
node_2 |      2 | postgres-sample-1.postgres-sample-agent.default.svc.cluster.local:543
2 |    2: 0/3002690 |   read-write |              primary |              primary
```

Use the locally installed `kubectl` tool (pre-authenticated to securely access the Kubernetes cluster) to run the `psql` utility on the `postgres-sample-0` pod:

```
kubectl exec -it postgres-sample-0 -- bash -c "psql"
```

```
psql (11.13 (VMware Postgres 11.13.1))
Type "help" for help.

postgres=# \l
                                 List of databases
Name                 |  Owner    | Encoding   | Collate | Ctype |   Access privileges
----------------------+----------+-----------+---------+-------+----------------------
-
postgres              | postgres | SQL_ASCII | C       | C     |
template0             | postgres | SQL_ASCII | C       | C     | =c/postgres          +
                      |          |           |         |       | postgres=CTc/postgres
template1             | postgres | SQL_ASCII | C       | C     | =c/postgres          +
                      |          |           |         |       | postgres=CTc/postgres
pg-instance-example   | postgres | SQL_ASCII | C       | C     |
(4 rows)

(Enter \q to exit the `psql` utility.)
```

The newly created database uses UTF-8 encoding. To verify the encoding run:

```
postgres=# show server_encoding;
server_encoding
-----------------
 UTF8
(1 row)
```

See also Accessing a Postgres Instance in Kubernetes.

Installing Tanzu Postgres Extensions

This topic covers installation steps for the extensions packaged with Tanzu Postgres 1.2 and later. The extensions include:

- Orafce
- PostGIS
- pgAudit

Refer to the linked extension documentation for instructions on using the extensions.

pgAudit

pgAudit is packaged with the Postgres instance images. It needs to be manually installed on the primary data pod:

1. Connect to the primary pod via `psql` and use:

   ```
   CREATE EXTENSION pgaudit;
   ```

   ```
   CREATE EXTENSION
   ```

2. Verify the installation using a command similar to:

   ```
   select count(*) from pg_extension where extname = 'pgaudit';
   ```

   ```
   count
   -------
   1
   (1 row)
   ```

For further details on Session and Object logging, see Session Audit logging and Object Session logging in the VMware Postgres documentation.

Orafce

Orafce is packaged with the Postgres instance images. It needs to be manually installed on the primary data pod:

1. Connect to the primary pod via `psql` and use:

   ```
   CREATE EXTENSION orafce;
   ```

2. To verify the installation:

```
SELECT months_between(date '1995-02-02', date '1995-01-01');
```

The output should be similar to:

```
months_between
------------------
1.03225806451613
(1 row)
```

For more information, refer to the Orafce documention.

PostGIS

PostGIS is packaged with the Postgres instance images. It needs to be manually installed on the primary data pod:

1. Connect to the primary pod via `psql` and use:

```
CREATE EXTENSION postgis;
```

2. To verify the installation:

```
SELECT st_pointfromtext('POINT(1 1)');
```

The output should be similar to:

```
st_pointfromtext
--------------------------------------------
0101000000000000000000F03F000000000000F03F
(1 row)
```

For more information refer to the PostGIS documentation.

Address Standardizer

1. Connect to the primary pod via `psql` and use:

```
CREATE EXTENSION address_standardizer;
```

2. To verify the installation:

```
SELECT num, street, city, state, zip FROM parse_address('1 Devonshire Place PH3
01, Boston, MA 02109');
```

You should see output similar to:

```
num |         street         |  city  | state |  zip
-----+------------------------+--------+-------+-------
1   | Devonshire Place PH301 | Boston | MA    | 02109
(1 row)
```

For more information refer to the Address Standardizer documentation.

Upgrading the Tanzu Postgres Operator and Instances

This topic provides steps to upgrade the Tanzu Postgres Operator and Tanzu Postgres instances to a newer version. Upgrading the Postgres Operator also upgrades the existing Postgres instances.

Important Upgrade Notes:

- customers upgrading from Tanzu Operator 1.5.0 and earlier, need to update their regsecret `dockerRegistrySecretName` from `registry.pivotal.io` to the updated `registry.tanzu.vmware.com`.
- customers upgrading to 1.7.0 from 1.6.0 and earlier, who are using "postgres" as a value for `pgConfig.dbName`, `pgConfig.appUser`, or `pgConfig.username`, need to restore to another instance before upgrading. For information on the restore process, see Restore to Another Instance.

The topic covers two upgrade scenarios:

- Existing customers who want to upgrade to release 1.7.2 using the VMware Tanzu Registry. See Upgrading the Operator using the Tanzu Registry.
- Existing customers who want to upgrade to release 1.7.2 using the downloadable files from Vmware Tanzu Network. See Upgrading the Operator using the Tanzu Network download.

Upgrading the Operator using the Tanzu Registry

Ensure you have access to Broadcom Support and Tanzu Network Registry. You can use the same credentials for both sites.

Note: Helm CLI 3.7.0 is not supported. Please use 3.7.1 and later.

1. Set the environment variable to enable Open Container Initiative (OCI) support in the Helm v3 client by running:

   ```
   export HELM_EXPERIMENTAL_OCI=1
   ```

   If you skip this step, the following error message might appear:

   ```
   Error: this feature has been marked as experimental and is not enabled by defau
   lt.
   ```

2. Use Helm to log in to the Tanzu Network Registry by running:

```
helm registry login registry.tanzu.vmware.com \
       --username=<USERNAME> \
       --password=<PASSWORD>
```

Follow the prompts to enter the email address and password for your Tanzu Network account.

3. Download the Helm chart from the Tanzu Distribution Registry into a local `/tmp/` directory:

   With helm CLI 3.6 and earlier,

   ```
   helm chart pull registry.tanzu.vmware.com/tanzu-sql-postgres/postgres-operator-
   chart:v1.7.2
   helm chart export registry.tanzu.vmware.com/tanzu-sql-postgres/postgres-operato
   r-chart:v1.7.2  --destination=/tmp/
   ```

   With helm CLI 3.7.1 and later,

   ```
   helm pull oci://registry.tanzu.vmware.com/tanzu-sql-postgres/postgres-operator-
   chart --version v1.7.2 --untar --untardir /tmp
   ```

4. Update the existing Postgres instance Custom Resource Definition (CRD) with the new values:

   ```
   cd /tmp/postgres-operator/
   ```

   ```
   kubectl apply -f crds/
   ```

5. If you do not have an existing overrides yaml file, perform the Helm upgrade using:

   ```
   helm upgrade postgres-operator /tmp/postgres-operator/ --wait
   ```

   The output is similar to:

   ```
   Release "postgres-operator" has been upgraded. Happy Helming!
   NAME: postgres-operator
   LAST DEPLOYED: Fri Jan 7 15:31:43 2022
   NAMESPACE: default
   STATUS: deployed
   REVISION: 4
   TEST SUITE: None
   ```

   where `REVISION` is a counter for the number of Operator you have performed. If you have upgraded from 1.0 to 1.1, and from 1.1 to 1.2, the REVISION number would be 3.

   If you have an existing overrides file, and you are upgrading from a Tanzu Operator before 1.4.0, you must make updates to the structure of the overrides file.

   Before Tanzu Operator 1.4.0, `operatorImageRepository` and `operatorImageTag` were separate keys used to describe the operator image. Similarly, `postgresImageRepository` and `postgresImageTag` were separate keys used to describe the postgres image. Now the values are combined into new keys named `operatorImage` and `postgresImage` respectively.

   For example, if the overrides file contained:

```
---
operatorImageRepository: my-custom-registry/postgres-operator
operatorImageTag: v1.3.0

postgresImageRepository: my-custom-registry/postgres-instance
postgresImageTag: v1.3.0
```

then the new overrides file would look like:

```
---
operatorImage: my-custom-registry/postgres-operator:v1.7.2

postgresImage: my-custom-registry/postgres-instance:v1.7.2
```

Then, upgrade using:

```
helm upgrade postgres-operator /tmp/postgres-operator/ -f /<path-to-your-file>/
operator_overrides_values.yaml --wait
```

where you substitute `operator_overrides_values.yaml` with your custom name and file location.

6. Wait for the Operator, Monitor, and Postgres instances to restart. Verify the new Postgres Operator version. The Postgres Operator is updated across all namespaces, including the default.

```
helm ls
```

```
NAME                        NAMESPACE       REVISION        UPDATED
STATUS          CHART                           APP VERSION
postgres-operator           default         4               2022-01-06 13:28:05.704
226 -0500 CDT   deployed        postgres-operator-v1.7.2   v1.7.2
```

To verify the new Postgres instance version, use a command similar to:

```
kubectl exec -i pod/postgres-sample-0 -- bash -c "psql -c 'select version()'"
```

Upgrading the Operator using the Tanzu Network download

1. Download the latest VMware Tanzu Postgres version from VMware Tanzu Network. Load the new Postgres Operator and Postgres instances images into your container registry following the steps in Setup the Tanzu Operator via a Downloaded Archive File.

2. Create an `operator-overrides.yaml` file at a location of your choice or update an existing overrides file. Enter the parameters below replacing `operatorImage`, `postgresImage`, and `dockerRegistrySecretName` with your values:

   Note: Prior to Tanzu Operator 1.5.0, `operatorImageRepository` and `operatorImageTag` were separate keys used to describe the operator image. Similarly, `postgresImageRepository` and `postgresImageTag` were separate keys used to describe the postgres image. Now the values are combined into new keys named `operatorImage` and `postgresImage` respectively.

```
# specify the url for the docker image for the Operator, e.g. gcr.io/<my_projec
t>/postgres-operator
operatorImage: gcr.io/data-pcf-db/postgres-operator:v1.7.2

# specify the docker image for postgres instance, e.g. gcr.io/<my_project>/post
gres-instance
postgresImage: gcr.io/data-pcf-db/postgres-instance:v1.7.2

# specify the name of the docker-registry secret to allow the cluster to authen
ticate with the container registry for pulling images
dockerRegistrySecretName: regsecret
```

3. Update the Postgres instance Custom Resource Definition (CRD) with the new values:

   ```
   cd <your-download-location>/postgres-for-kubernetes-<postgres-version>/
   ```

   ```
   kubectl apply -f operator/crds/
   ```

4. Upgrade the Postgres Operator and instances with the `helm upgrade` command, specifying your location of the `operator-overrides.yaml` file:

   ```
   helm upgrade -f ./operator-overrides.yaml postgres-operator operator/
   ```

   The output is similar to:

   ```
   Release "postgres-operator" has been upgraded. Happy Helming!
   NAME: postgres-operator
   LAST DEPLOYED: Mon Nov 22 15:31:43 2021
   NAMESPACE: default
   STATUS: deployed
   REVISION: 4
   TEST SUITE: None
   ```

5. Verify the new Postgres Operator version. The Postgres Operator is updated across all namespaces, including the default.

   ```
   helm ls
   ```

   ```
   NAME                    NAMESPACE       REVISION        UPDATED
   STATUS          CHART                     APP VERSION
   postgres-operator       default         4               2021-02-16 13:28:05.704
   226 -0500 CDT   deployed        postgres-operator-v1.7.2  v1.7.2
   ```

   Verify the image version matches the new Operator version:

   ```
   cat images/postgres-*-tag
   ```

   ```
   v1.7.2
   v1.7.2
   ```

   Confirm that the images version has also been upgraded:

```
kubectl describe statefulset.apps/<your-database-name> | grep Image | uniq
```

```
Image:      gcr.io/data-pcf-db/postgres-instance:v1.7.2
```

If you have a High Availability configuration, verify the upgrade using:

```
kubectl describe statefulset.apps/<your-ha-database> | grep Image | uniq
```

```
Image:      gcr.io/data-pcf-db/postgres-instance:v1.7.2
```

To verify the new Postgres instance version, use a command similar to:

```
kubectl exec -i pod/postgres-sample-0 -- bash -c "psql -c 'select version()'"
```

Updating a Postgres Instance Configuration

This topic describes how to update CPU, memory, and storage configuration of an existing Postgres instance.

To update an existing Postgres instance for high availability or backup, see Configuring High Availability in Tanzu Postgres, and Backing Up and Restoring Tanzu Postgres.

Prerequisites

The steps in this topic require:

- the `kubectl` command line tool installed on a client that accesses the Kubernetes cluster.
- appropriate access permissions to the Kubernetes cluster project and namespace where the Postgres instances reside.
- access permissions to the running Postgres instances to be updated.
- access permissions to the Kubernetes `storageclass`.

Modifying Memory and CPU

The memory and CPU allocation are specified in the instance `yaml` manifest file created during instance deployment. Edit the file to make the required changes. Before increasing any values, ensure that the Kubernetes cluster does not have any limiting resource quotas. See the Kubernetes Resource Quotas documentation for more information.

1. Move to the Tanzu Postgres workspace directory with the Postgres instance Kubernetes manifest file.

   ```
   cd ./postgres-for-kubernetes-v<version>
   ```

2. Edit the manifest `yaml` file you used to deploy the instance; in this example the file is called `postgres.yaml`. Set new values for the `memory` and `cpu` attributes.

   For example:

   ```
   apiVersion: sql.tanzu.vmware.com/v1
   kind: Postgres
   metadata:
     name: postgres-sample
   spec:
     storageClassName: standard
     storageSize: 800M
   ```

```
  cpu: "1.5"
  memory: 2G
  monitorStorageClassName: standard
  monitorStorageSize: 1G
  resources:
    monitor:
      limits:
        cpu: 800m
        memory: 800Mi
      requests:
        cpu: 800m
        memory: 800Mi
    metrics:
      limits:
        cpu: 100m
        memory: 100Mi
      requests:
        cpu: 100m
        memory: 100Mi
  pgConfig:
    dbname: postgres-sample
    username: pgadmin
  serviceType: ClusterIP
```

Note: You cannot alter the `name`, `storageClassName`, `monitorStorageClassName`, `dbname`, or `username` of an existing instance.

3. Execute the `kubectl apply` command, specifying the manifest file you edited. For example:

```
kubectl apply -f ./postgres.yaml --wait=false
```

```
postgres.sql.tanzu.vmware.com "postgres-sample" configured
```

If the manifest file contains any incorrectly formatted values or unrecognized field names, an error message is displayed identifying the issue. Edit the manifest to correct the error and run the command again.

4. Verify the updated configuration by specifying the memory and cpu fields of the instance object.

```
kubectl get postgres/postgres-sample -o jsonpath='{.spec.memory}'
```

```
2G
```

```
kubectl get postgresinstance/postgres-sample -o jsonpath='{.spec.cpu}'
```

```
1.5
```

5. Similarly, you can update monitor CPU and memory limits and requests too by updating cpu/memory fields in `resources.monitor.limits` and `resources.monitor.requests`

Modifying Storage Volume Size

To expand a Postgres instance storage volume, verify that the storage volume is expandable and then update the instance `yaml` file.

The Postgres operator sets the sizes of the Postgres data volume at instance initialization. The actual size of the expanded volumes may be greater than your specified value if the storage manager allocates space in fixed increments.

Note: Kubernetes does not support shrinking a volume size.

Verifying Volume Expansion

To expand the PV volumes, review the storage class object and check if the `allowVolumeExpansion` field is set to `true`. If the attribute does not exist, you may add it to the storage class, and then expand the storage volumes. The following steps describe this process.

Note: Minikube does not support volume expansion. If you set `allowVolumeExpansion` to `true` in Minikube and request a larger volume size, it fails with an error message.

1. Show the current Persistent Volume(s) (PVs).

   ```
   kubectl get pv
   ```

   ```
   NAME                                       CAPACITY   ACCESS MODES   RECLAIM PO
   LICY   STATUS   CLAIM                                                      STO
   RAGECLASS   REASON   AGE
   pvc-00d130f6-1fb6-49e0-b145-c58e152f76ab   5Gi        RWO            Delete
   Bound    default/postgres-sample-pgdata-postgres-sample-0           standard
   14s
   pvc-33371405-6853-49f7-b704-f49ae8771b5b   1Gi        RWO            Delete
   Bound    default/postgres-sample-monitor-postgres-sample-monitor-0   standard
   34s
   ```

2. Check the `standard` storage class's `allowVolumeExpansion` attribute value:

   ```
   kubectl get storageclass standard
   ```

   ```
   NAME                 PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMO
   DE   ALLOWVOLUMEEXPANSION   AGE
   standard (default)   k8s.io/minikube-hostpath   Delete          Immediate
   false                  20h
   ```

3. Modify the storage class after checking the permissions:

   ```
   kubectl auth can-i update storageclass
   ```

   ```
   yes
   ```

4. Amend the storage class configuration after saving it to a local `yaml` file:

   ```
   kubectl get storageclass standard -o yaml > storagesize.yaml
   ```

5. Edit the saved file and change the `allowVolumeExpansion` attribute to `true`, or add the attribute if it is not already present.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: standard
provisioner: kubernetes.io/aws-ebs # AWS specific
reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate
```

For more information about expanding volumes, see the Kubernetes Allow Volume Expansion documentation.

6. Apply the change.

```
kubectl apply -f storagesize.yaml
```

7. Verify the change.

```
kubectl get storageclass standard --output=jsonpath='{.allowVolumeExpansion}'
```

```
true
```

See the Kubernetes Documentation for more information about storage classes and persistent volumes.

Increasing Volume Size

1. Edit the manifest file that was used to deploy the instance. Set the value for the `storageSize` attribute to the desired size:

```
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: postgres-sample
spec:
  storageClassName: standard
  storageSize: 2G
  cpu: "1.5"
  memory: 2G
  monitorStorageClassName: standard
  monitorStorageSize: 1G
  resources:
    monitor:
      limits:
        cpu: 800m
        memory: 800Mi
      requests:
        cpu: 800m
        memory: 800Mi
    metrics:
      limits:
        cpu: 100m
        memory: 100Mi
      requests:
        cpu: 100m
        memory: 100Mi
  pgConfig:
    dbname: postgres-sample
    username: pgadmin
  serviceType: ClusterIP
```

2. Apply the edited manifest to the Postgres instance.

```
kubectl apply -f postgres.yaml
```

```
postgres.sql.tanzu.vmware.com/postgres-sample configured
```

If the manifest file contains any incorrectly formatted values or unrecognized field names, an error message is displayed identifying the issue. Edit the manifest to correct the error and run the command again.

3. Verify that the persistent volume size has increased.

```
watch kubectl get pv
```

```
NAME                                       CAPACITY   ACCESS MODES   RECLAIM PO
LICY   STATUS   CLAIM                                                       STO
RAGECLASS   REASON   AGE
pvc-00d130f6-1fb6-49e0-b145-c58e152f76ab   2Gi        RWO            Delete
Bound    default/postgres-sample-pgdata-postgres-sample-0            standard
14s
```

If the storage class does not have the `allowVolumeExpansion` attribute set to `true`, the persistent volumes will not be expanded. No message is immediately displayed, but errors are written to the Postgres operator logs. View the logs with commands like the following.

```
kubectl get pods
```

```
NAME                                 READY   STATUS    RESTARTS   AGE
postgres-sample-0                    1/1     Running   0          15m
postgres-operator-54fb679bc5-p8lps   1/1     Running   0          30m
```

```
kubectl logs postgres-operator-54fb679bc5-p8lps
```

```
INFO    controllers.PersistentVolumeClaims Reconciler    Error updating PVC res
ources: persistentvolumeclaims "postgres-sample-pgbackrest-postgres-sample-0" i
s forbidden: only dynamically provisioned pvc can be resized and the storagecla
ss that provisions the pvc must support resize
ERROR    controllers.PostgresInstance    found error reconciling backup persist
ent volume claim    {"error": "persistentvolumeclaims \"postgres-sample-pgbackr
est-postgres-sample-0\" is forbidden: only dynamically provisioned pvc can be r
esized and the storageclass that provisions the pvc must support resize"}
```

If the persistent volumes could not be resized, edit the `StorageSize` attribute in the manifest to match the actual size.

4. Similarly, you can expand monitor volume size by updating `spec.monitorStorageSize` if the monitor storage class has `allowVolumeExpansion` attribute set to `true`.

Accessing a Postgres Instance in Kubernetes

After you deploy a Postgres instance, you can access the databases either by executing Postgres utilities from within Kubernetes, or by using a locally-installed tool, such as `psql`.

Accessing a Pod with Kubectl

Use the `kubectl` tool to run utilities directly in a Postgres pod. This `psql` command connects to the default Postgres database, `postgres`.

```
kubectl exec -it postgres-sample-0 -- psql
```

```
psql (11.13 (VMware Postgres 11.13.1))
Type "help" for help.

postgres=#
```

You can also simply execute a bash shell in the pod and then execute Postgres utilities as needed. For example:

```
kubectl exec -it postgres-sample-0 -- /bin/bash
```

```
postgres@postgres-sample-0:/$ createdb mydb
```

```
postgres@postgres-sample-0:/$ psql mydb
```

```
psql (11.13 (VMware Postgres 11.13.1))
Type "help" for help.

mydb=# create role user1 login;
```

Accessing Postgres with External Clients

If you have installed `psql`, or another Postgres client application outside of Kubernetes (for example, on your local client machine), you can connect to a Tanzu Postgres database using Postgres connection parameters passed as command-line options, or in a connection string. For example:

```
PGPASSWORD=$password psql -h $host -p $port -d $dbname -U $username
```

```
psql (11.13 (VMware Postgres 11.13.1))
Type "help" for help.

postgres-sample=#
```

where `$password`, `$host`, `$port`, `$dbname`, `$username` are the connection parameters required to access the database. To acquire those, depending on your environment, you may use the following methods.

For the sample database name, database role, and password from a Kubernetes secret use:

```
dbname=$(kubectl get secret postgres-sample-db-secret -o go-template='{{.data.dbname |
base64decode}}')
username=$(kubectl get secret postgres-sample-db-secret -o go-template='{{.data.userna
me | base64decode}}')
password=$(kubectl get secret postgres-sample-db-secret -o go-template='{{.data.passwo
rd | base64decode}}')
```

To get the application user for a Service Binding use a command similar to:

```
dbname=$(kubectl get secrets postgres-sample-app-user-db-secret -o jsonpath='{.data.da
tabase}' | base64 -D)
username=$(kubectl get secrets postgres-sample-app-user-db-secret -o jsonpath='{.data.
username}' | base64 -D)
password=$(kubectl get secrets postgres-sample-app-user-db-secret -o jsonpath='{.data.
password}' | base64 -D)
```

For a remote Kubernetes environment, get the external host address and port from the Postgres load balancer:

```
host=$(kubectl get service postgres-sample -o jsonpath='{.status.loadBalancer.ingress
[0].ip}')
port=$(kubectl get service postgres-sample -o jsonpath='{.spec.ports[0].port}')
```

For a local Minikube Kubernetes environment, the Postgres load balancer is not used. Get the external host address and port using:

```
host=$(minikube ip)
port=$(kubectl get service postgres-sample -o jsonpath='{.spec.ports[0].nodePort}')
```

Deleting a Postgres Instance from Kubernetes

This section describes how to delete Postgres pods and other resources that are created when you deploy a Postgres instance to Kubernetes. When deleting the resources, everything related to that instance is deleted, including the persistent volume claims (PVCs) and the access secrets. Any application bound to the deleted instance will have to rebind with new credentials.

Warning: Do not delete the Postgres operator when there are existing Postgres instances. If the Postgres operator is deleted, you will get an error if you try to reinstall it. See Cannot Reinstall Operator after Deleting for a workaround to this problem.

Deleting Postgres Pods and Resources

Follow these steps to delete a Postgres instance from Kubernetes.

1. Change to the Tanzu Postgres workspace directory with the Kubernetes manifest you used to deploy the Postgres instance.

   ```
   cd ./postgres-v<postgres-version>+<vmware-version>
   ```

2. Execute the `kubectl delete` command, specifying the manifest you used to deploy the instance. For example:

   ```
   kubectl delete -f ./postgres.yaml --wait=false
   ```

   ```
   postgres.sql.tanzu.vmware.com "postgres-sample" deleted
   ```

   Note: The optional `wait=false` flag returns immediately without waiting for the deletion to complete.

   `kubectl` stops the Postgres instance and deletes the Kubernetes resources for the deployment.

3. If for any reason stopping the Postgres instance fails, use `kubectl` to display the `postgres-operator` log.

   ```
   kubectl logs -l app=postgres-operator
   ```

   The Postgres operator should remain for future deployments:

```
kubectl get all
```

```
NAME                                        READY   STATUS    RESTARTS   AGE
pod/postgres-operator-546ffd888b-dc6zk    1/1     Running   0          26h

NAME                                          TYPE          CLUSTER-IP       EXT
ERNAL-IP      PORT(S)          AGE
service/kubernetes                            ClusterIP     10.96.0.1        <no
ne>           443/TCP          40h
service/postgres-operator-webhook-service   ClusterIP     10.98.126.247    <no
ne>           443/TCP          26h

NAME                                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/postgres-operator    1/1     1            1           26h

NAME                                              DESIRED   CURRENT   READY   AGE
replicaset.apps/postgres-operator-546ffd888b   1         1         1       26h
```

Deleting the Postgres Operator

If you want to remove the Postgres operator, follow the instructions in Uninstalling VMware Tanzu SQL with Postgres for Kubernetes.

Creating Service Bindings

This topic describes how to enable Tanzu Application Platform (TAP) or Tanzu Application Services (TAS) applications to connect and utilize Tanzu Postgres database services.

Binding an Application to a Postgres Instance using TAP workflow

The Tanzu Postgres Operator 1.5.0 supports Service Binding with Tanzu Application Platform (TAP). This feature eliminates the manual management of the configuration steps needed to securely and successfuly bind a TAP application to a Tanzu Postgres deployment. For more information on Service Binding, see Service Binding Specification for Kubernetes in the Kubernetes documentation.

This feature introduces the field `appUser` in the Postgres instance CRD, and an application user secret `postgres-sample-app-user-db-secret`.

For more information on how to create or update TAP workloads, see Create a workload in the TAP documentation.

Prerequisites

- TAP v1.0.0 must be installed. For TAP installation instructions, see Installing Tanzu Application Platform.
- Tanzu Postgres Operator must be installed on a Kubernetes cluster. See Installing a Postgres Operator for instructions.
- Postgres instance needs to be deployed. See Deploying a Postgres Instance for instructions.
- TAP services-toolkit controller manager needs the required permissions to access Postgres instances. Create a `ClusterRole` with the Role-based access control (RBAC) required by the services-toolkit controller-manager for the Postgres resource, and save it in a file like `resource-claims-postgres.yaml`:

  ```
  ---
  apiVersion: rbac.authorization.k8s.io/v1
  kind: ClusterRole
  metadata:
    name: resource-claims-postgres
    labels:
      resourceclaims.services.apps.tanzu.vmware.com/controller: "true"
  rules:
  - apiGroups: ["sql.tanzu.vmware.com"]
  ```

```
  resources: ["postgres"]
  verbs: ["get", "list", "watch", "update"]
```

and then run:

```
kubectl apply -f resource-claims-postgres.yaml
```

For more information on RBAC and ClusterRole, view Role and ClusterRole in the Kubernetes documentation.

Bind a new TAP workload

To bind a new TAP workload, use the TAP command `tanzu apps workload create`. Name the workload, specify a source code location to create the workload from, and reference the postgres instance name that you have deployed:

```
tanzu apps workload create WORKLOAD-NAME [create flags] --service-ref "db=sql.tanzu.vm
ware.com/v1:Postgres:POSTGRES-INSTANCE-NAME"
```

where,

- `WORKLOAD-NAME` is the name that will be given to the workload.
- [create flags] are the appropriate flags to build the workload from source control, registry image, or local file.
- `--service-ref` is the reference to the service using the format `{name}={apiVersion}:{kind}:{name}`.
  `POSTGRES-INSTANCE-NAME` is the the postgres instance CR name that you have deployed.

See Tanzu apps workload create for more information on the different flags and options available.

Example Workload

You can also create a Workload yaml file where you can specify environment variables, build parameters, etc. that are required by your application to be built and run successfully. The example workload yaml below shows a spring-petclinic application that binds to a postgres instance `postgres-sample` in the same namespace:

```
---
apiVersion: carto.run/v1alpha1
kind: Workload
metadata:
  name: pet-clinic
  labels:
    apps.tanzu.vmware.com/workload-type: web
    app.kubernetes.io/part-of: pet-clinic
spec:
  build:
    env:
    - name: BP_MAVEN_POM_FILE
      value: skip-pom.xml
  env:
  - name: SPRING_PROFILES_ACTIVE
    value: postgres
  params:
  - name: live-update
    value: "true"
  serviceClaims:
  - name: db
    ref:
      apiVersion: sql.tanzu.vmware.com/v1
      kind: Postgres
      name: postgres-sample
  source:
    git:
      ref:
        branch: main
      url: https://github.com/spring-projects/spring-petclinic
```

where,

- `postgres-sample` is the example instance name.
- `BP_MAVEN_POM_FILE` needs to be set to some non-existent value (in this example, `skip-pom.xml`) in order to build the upstream spring-petclinic app correctly. Refer to this open issue for more details.
- `SPRING_PROFILES_ACTIVE` is set to `postgres` to override the default application configuration
- `live-update` is set to `true` to keep the workload pods up and avoid the auto-scale down done by the TAP components. This can help with debugging the workload.

Save the above content in `workload.yaml` file and then run `kubectl apply -f workload.yaml`

IMPORTANT: This example creates a workload with the sample Spring application spring-petclinic that is available on GitHub. The sample application is subject to change and may face issues during build and run steps in the TAP workflow. Consider specifying a commit sha to use an older state of the sample application as needed for demo and test purposes.

Bind an existing TAP workload

To update an existing TAP workload to connect and utilize a Postgres database instance, run a command similar to:

```
tanzu apps workload update WORKLOAD-NAME [update flags] --service-ref "db=sql.tanzu.vm
ware.com/v1:Postgres:POSTGRES-INSTANCE-NAME"
```

where,

- `WORKLOAD-NAME` is the name of the workload.
- [update flags] are the appropriate flags to build the workload from source control, registry image, or local file.
- `--service-ref` is the reference to the service using the format `{name}={apiVersion}:{kind}:{name}`.
  `POSTGRES-INSTANCE-NAME` is the the postgres instance CR name that you have deployed.

For example:

```
tanzu apps workload update pet-clinic --service-ref "db=sql.tanzu.vmware.com/v1:Postgr
```

```
es:postgres-sample"
```

See Tanzu apps workload update for more information on the different flags available.

Bind a TAP workload in a different namespace

- To enable cross-namespace binding, create a `ResourceClaimPolicy` resource in the namespace where the Postgres instance is deployed. In this example, the `postgres-sample` is deployed in the `default` namespace.

  ```
  ---
    apiVersion: services.apps.tanzu.vmware.com/v1alpha1
    kind: ResourceClaimPolicy
    metadata:
      name: postgres-cross-namespace
    spec:
      consumingNamespaces:
      - '*'
      subject:
         group: sql.tanzu.vmware.com
         kind: Postgres
  ```

  Where * indicates this policy permits any namespace to claim a Postgres resource from the `default` namespace.

- Save the above yaml content in the `resource-claim-policy.yaml` file and apply it:

  ```
  kubectl -n default apply -f resource-claim-policy.yaml
  ```

- If you have a TAP workload deployed in a separate namespace, bind the workload to the Postgres service instance by specifying the namespace using the `n` flag. The command would be similar to:

  ```
  tanzu apps workload update WORKLOAD-NAME [update flags] -n WORKLOAD-OTHER-NAMES
  PACE-NAME --service-ref "db=sql.tanzu.vmware.com/v1:Postgres:default:POSTGRES-I
  NSTANCE-NAME"
  ```

  Where `service-ref` specifies the service instance's namespace. In this example, it is `default` and WORKLOAD-OTHER-NAMESPACE-NAME is the different namespace where the workload is deployed.

- If you are creating a new workload using the workload yaml in `app` namespace, then you will need to add an annotation in the metadata section that would specify the postgres instance's namespace:

  ```
  ---
  apiVersion: carto.run/v1alpha1
  kind: Workload
  metadata:
    name: pet-clinic
    namespace: app
    annotations:
      serviceclaims.supplychain.apps.x-tanzu.vmware.com/extensions: '{"kind":"Ser
  viceClaimsExtension","apiVersion":"supplychain.apps.x-tanzu.vmware.com/v1alpha
  1", "spec":   {"serviceClaims":{"db":{"namespace":"default"}}}}'
  ```

```
  labels:
    apps.tanzu.vmware.com/workload-type: web
    app.kubernetes.io/part-of: pet-clinic
spec:
  build:
    env:
    - name: BP_MAVEN_POM_FILE
      value: skip-pom.xml
  env:
  - name: SPRING_PROFILES_ACTIVE
    value: postgres
  params:
  - name: live-update
    value: "true"
  serviceClaims:
  - name: db
    ref:
      apiVersion: sql.tanzu.vmware.com/v1
      kind: Postgres
      name: postgres-sample
  source:
    git:
      ref:
        branch: main
      url: https://github.com/spring-projects/spring-petclinic
```

Save the above content in `workload.yaml` file and then run `kubectl apply -f workload.yaml`

Verify a TAP Workload Service Binding

To check if the application has been successfully bound to the Tanzu Postgres instance:

- Check that the corresponding ResourceClaim object is ready. The ResourceClaim name follows the format <metadata.name>-<spec.serviceClaim.name> as given in the workload yaml as shown above. In this example, it would be `pet-clinic-db`

  ```
  kubectl get resourceclaims pet-clinic-db
  NAME                                                          READY   REASON
  resourceclaim.services.apps.tanzu.vmware.com/pet-clinic-db   True
  ```

- Query the database using the Postgres application user `spec.pgConfig.appUser`. The corresponding credentials and connection details can be fetched from the secret referenced in status.binding.name field in the instance CR. In this example, the secret would be `postgres-sample-app-user-db-secret` and the data can be verified in the `postgres-sample` database.

  ```
  dbname=$(kubectl get secrets postgres-sample-app-user-db-secret -o jsonpath='{.
  data.database}' | base64 -D)
  username=$(kubectl get secrets postgres-sample-app-user-db-secret -o jsonpath
  ='{.data.username}' | base64 -D)
  password=$(kubectl get secrets postgres-sample-app-user-db-secret -o jsonpath
  ='{.data.password}' | base64 -D)

  host=$(kubectl get secrets postgres-sample-app-user-db-secret -o jsonpath='{.da
  ta.host}' | base64 -D)
  ```

```
port=$(kubectl get secrets postgres-sample-app-user-db-secret -o jsonpath='{.da
ta.port}' | base64 -D)
```

For example, for the spring-petclinic app, use a `psql` query similar to:

```
PGPASSWORD=$password psql -h $host -p $port -d $dbname -U $username
```

```
psql (14.2 (VMware Postgres 14.2.0))
Type "help" for help.

my-postgres=# SELECT COUNT(1) from vets;
 count
-------
     6
(1 row)
```

Binding Tanzu Postgres to a TAS Application

Prerequisites

- Tanzu Postgres has been successfully installed on a Kubernetes cluster.
- The external public ingress service type of the instance is set to `LoadBalancer`. See Deploying a Postgres Instance.
- You have deployed an application to TAS, and the application is able to reach the Kubernetes cluster ingress points.

Binding an Application

1. Export the environment variables `$dbname`, `$username`, `$password`, `$host`, and `$port`, described in Accessing Postgres with External Clients.
2. (Optional) From your computer validate that the Postgres instance is reachable from the location where TAS is deployed, by using a command similar to:

   ```
   # create a CF SSH tunnel, this will hang and you need to leave it open until yo
   u're done with this step
   cf ssh <appname> -L 54320:$host:$port
   ```

   which returns similar to:

   ```
   vcap@f00949bd-6601-4731-6f7e-e859:~$
   ```

   where `f00949bd-6601-4731-6f7e-e859` is an example GUID. In a new window or tab, use `psql` to test the connection:

   ```
   PGPASSWORD=$password psql -h 127.0.0.1 -p 54320 -U $username -d $dbname
   ```

   ```
   psql (11.13 (VMware Postgres 11.13.1))
   Type "help" for help.
   ```

```
postgres-sample=#
```

3. Choose a name and create the user provided service:

   ```
   cf create-user-provided-service my-postgres-instance -p "{\"uri\":\"postgres://
   $username:$password@$host:$port/$dbname?sslmode=require\"}"
   ```

   For more details, see User-Provided Service Instances.

4. Bind, restage, and use the service as you would normally with a "cf marketplace" provisioned SQL instance. See Bind a Service Instance to an App for an example of this process, and Restage Your App.

   Like any data service, the application will automatically connect to the database instance depending on the buildpack. If autoconnect is not supported, the application will have to manually make use of the `uri` service credential to make a connection.

   Bound service credentials are available at runtime to TAS applications via the `DATABASE_URL` and `VCAP_SERVICES` environment variables. For further details see DATABASE_URL in *TAS for VMs Environment Variables* documentation.

Configuring TLS for Tanzu Postgres Instances

This topic describes how to configure TLS for a Postgres instance.

Overview

Tanzu Postgres (from version 1.2) supports Transport Layer Security (TLS) encrypted connections to the Postgres server from clients and applications. The Postgres server by default requires cert-manager self-signed certificates (see Prerequisites in the *Installing Tanzu Postgres* page) for internal Kubernetes communications. From release 1.2 clients can connect to the Postgres server and verify the connection using user provided certificates or certificates provided by a corporate Certificate Authority (CA).

The Tanzu Postgres Operator uses a Kubernetes Secret to manage TLS. There are several ways to create the Secret and this topic describes two supported methods:

- create the secret using a cert-manager. See Creating the TLS Secret using cert-manager.
- create the secret manually, using the Kubernetes Command Line Interface (kubectl). See Create the TLS Secret Manually.

For general information about Kubernetes and TLS secrets, see TLS Secrets in the Kubernetes documentation.

Prerequisites

Before you configure TLS for a Postgres instance, you must have:

- The Kubernetes Command Line Interface (kubectl) installed. For more information see Install Tools in the Kubernetes documentation.
- The name of your Postgres instance. Refer to the Configuring a Postgres Instance for an overview of this information. You do not need to create a Postgres instance before configuring TLS.
- The namespace of your Postgres instance.
- The Postgres instance Service type, internal (ClusterIP) or external (LoadBalancer). The instance Service type determines the server hostname that is used during certificate generation.
- Access and admin permissions to the Postgres instance.
- If you're using cert-manager to configure TLS, obtain your custom or corporate CA public key, and the private/public key certificate pair.

Creating the TLS Secret Using cert-manager

This procedure describes how to configure a custom Certificate Authority and custom certificates using cert-manager. To create the TLS Secret through the kubectl interface instead, see Create the TLS Secret Manually above.

1. Verify that cert-manager was configured during Installing a Postgres Operator prerequisites:

   ```
   kubectl get all --namespace=cert-manager
   ```

2. For the CA certificate, create a Kubernetes Secret. For example `my-CA-secret.yaml`, with values similar to:

   ```
   kind: Secret
   metadata:
       name: my-ca-certificate
       namespace: cert-manager-namespace
   data:
      tls.crt: this is CA public key
      tls.key: this is the CA private key
   ```

   and apply with:

   ```
   kubectl apply -f my-CA-secret.yaml
   ```

3. Create a CA issuer in cert-manager using a `ClusterIssuer` resource and associate it with the CA Secret created in step 2. For information about the cert-manager `Issuer` types, see the cert-manager documentation.

   Note: The Postgres instance TLS secret requires the `ca.crt` key, therefore VMware does not recommend using the ACME Issuer.

   Create a `ClusterIssuer` resource using a `yaml` file, for example `my-cluster-issuer.yaml`, similar to:

   ```
   apiVersion: cert-manager.io/v1
   kind: ClusterIssuer
   metadata:
       name: sample-postgres-ca-certificate-clusterissuer
   spec:
       ca:
        secretName: my-ca-certificate
   ```

4. Apply the Secret using:

   ```
   kubectl apply -f my-cluster-issuer.yaml
   ```

   For certificate creation troubleshooting see the cert-manager documentation.

5. For new Tanzu Postgres customers, create the Postgres Operator and set it to use the custom TLS issuer by using a command similar to:

```
helm install postgres-operator --set=certManagerClusterIssuerName=sample-postgr
es-ca-certificate-clusterissuer  operator/
```

For existing Tanzu Postgres customers, update the Operator using a command similar to:

```
helm upgrade postgres-operator --set=certManagerClusterIssuerName=sample-postgr
es-ca-certificate-clusterissuer  operator/
```

Note: If you need to use more than one CA issuer, create another Postgres Operator in a different Kubernetes cluster.

To verify the TLS security setup see Verifying TLS Security for an example using `psql`.

Creating a TLS Secret Manually

This procedure describes how to create the TLS Secret manually, using kubectl. To create the TLS Secret using cert-manager instead, see Creating the TLS Secret with cert-manager below.

Note: Installing cert-manager when you install the Tanzu Postgres prerequisites does not prevent the manual TLS configuration. Cert-manager is required for internal Postgres Operator communications.

1. Generate a public/private key pair certificate using a certificate generation tool such as OpenSSL, certstrap, or Let's Encrypt. For an example using OpenSSL, see Creating Certificates in the PostgreSQL documentation.
   During the certificate request, ensure that you supply the correct server domain name as the common name or the subject alternative name (SAN) for which the certificate is valid for. The server domain name is the DNS name used to connect to the database, and it depends on the Service type deployment scenario:

    - If you configure `ClusterIP` as Service type (`spec.serviceType`) in the Postgres instance `yaml` (`ClusterIP` is the default Service type in Tanzu Postgres release 1.2), your database is deployed in the same Kubernetes cluster as your instance, is not exposed externally, and the hostname has the format `*.[instance-name]-agent.[namespace].svc.cluster.local`. For example `*.postgres-sample-agent.default.svc.cluster.local`.
    - If you configure `LoadBalancer` in the Postgres instance `yaml` (`LoadBalancer` is the default Service type in Tanzu Postgres releases 1.0 and 1.1), your database is accessible outside the Kubernetes cluster, and the hostname is the external DNS name or the IP address of the load balancer. To find the load balancer IP, use a command similar to:

      ```
      kubectl get service postgres-sample
      ```

      which returns:

      ```
      NAME              TYPE           CLUSTER-IP       EXTERNAL-IP      PORT(S)
      AGE
      postgres-sample   LoadBalancer   10.107.136.143   192.168.64.101   5432:31
      958/TCP    66s
      ```

The load balancer IP is located under the header `EXTERNAL-IP`.

Save the required certificates files locally.

2. Create the TLS Secret by running:

```
kubectl create secret generic <some-tls-secret> \
     --type kubernetes.io/tls \
     --from-file=ca.crt=/path/to/ca.crt \
     --from-file=tls.crt=/path/to/tls.crt \
     --from-file=tls.key=/path/to/tls.key \
     --namespace <postgres-instance-namespace>
```

Where:

- `<some-tls-secret>` is the TLS Secret.
- `/path/to/ca.crt` is the file path to the CA's public key that is used by clients or applications to verify that the certificate used to communicate with the Postgres Server was issued using the specified Certificate Authority.
- `/path/to/tls.crt` is the file path to the public key of the certificate created in step 1 above.
- `/path/to/tls.key` is the file path to the private key of the certificate created in step 1 above.
- `<instance-namespace>` is the namespace for the Postgres instance.

For example:

```
kubectl create secret generic postgres-tls-secret \
      --type kubernetes.io/tls \
      --from-file=tls.crt=/path/server.crt \
      --from-file=tls.key=/path/server.key \
      --from-file=ca.crt=/path/server_ca.crt \
      --namespace sample-postgres-namespace
```

Use the TLS Secret in the instance yaml file during instance creation, or apply the TLS secret after the instance creation by updating the instance `yaml`:

1. Edit the `postgres.yaml` file for the Postgres instance, and add `postgres-tls-secret` as the name of the TLS Secret created in the `sample-postgres-namespace` namespace. For example:

```
spec:
   certificateSecretName: postgres-tls-secret
```

2. Update the Postgres instance:

```
kubectl apply -f postgres.yaml -n sample-postgres-namespace
```

3. Restart the instance and the monitor:

```
 kubectl rollout restart statefulset/postgres-sample
 kubectl rollout restart statefulset/postgres-sample-monitor
```

After the Postgres instance restarts, client connections use TLS security. See Verifying TLS Security for an example scenario on how to verify the TLS setup.

Verifying TLS Security Using psql

To verify a successful TLS implementation, use `psql` as an example client to confirm the certificate usage. The example scenario uses the `jq` command line tool, which you need to install on your local computer.

The credential values required for the `psql` command are base64 encoded. For example, for the instance `postgres-sample` the values would look similar to:

```
kubectl get secret  postgres-sample-db-secret -o json | jq '.data | map_values(@base64
d)'
```

```
    {
       "dbname": "postgres-sample",
       "instancename": "postgres-sample",
       "namespace": "default",
       "password": "e0AJi1xiRr51m6gZKqb5Cwj0a6i53L",
       "username": "pgadmin"
    }
```

or for the application user:

```
kubectl get secret/postgres-sample-app-user-db-secret -o json | jq '.data | map_values
(@base64d)'
```

```
{
  "database": "postgres-sample",
  "host": "postgres-sample.default",
  "password": "w6J0U600C7jbpfSQl677Cjnid8Dcjz",
  "port": "5432",
  "provider": "vmware",
  "type": "postgresql",
  "uri": "postgresql://pgappuser:w6J0U600C7jbpfSQl677Cjnid8Dcjz@postgres-sample.defaul
t:5432/postgres-sample",
  "username": "pgappuser"
}
```

Acquire the Service IP address using a command similar to:

```
kubectl get service postgres-sample -o jsonpath='{.status.loadBalancer.ingress[0].ip}'
```

```
192.168.64.100
```

First test the `psql` connection with the acquired credentials but without using TLS validation:

```
psql "host=192.168.64.100 \
      port=5432 \
      dbname=postgres-sample \
      user=pgadmin \
      password=e0AJi1xiRr51m6gZKqb5Cwj0a6i53L \
      target_session_attrs=read-write"
```

Get your Certificate Authority public key and place it in a file like `/tmp/ca.crt`:

```
kubectl exec -q -ti postgres-sample-0 -- bash -c 'cat /etc/postgres_ssl/ca.crt' > /tm
p/ca.crt
```

Connect using TLS with the acquired credentials, and validate the signing CA (sslmode=verify-ca):

```
psql "host=192.168.64.100 \
      port=5432 \
      dbname=postgres-sample \
      user=pgadmin \
      password=e0AJi1xiRr51m6gZKqb5Cwj0a6i53L \
      target_session_attrs=read-write \
      sslmode=verify-ca \
      sslrootcert=/tmp/ca.crt"
```

Connect using TLS with the acquired credentials, validate the signing CA, and validate that the certificate matches the correct hostname (verify-full):

```
psql "host=192.168.64.100 port=5432 dbname=postgres-sample password=e0AJi1xiRr51m6gZKq
b5Cwj0a6i53L user=pgadmin target_session_attrs=read-write sslmode=verify-full sslrootc
ert=/tmp/ca.crt"
```

which returns similar to:

```
psql (11.13 (VMware Postgres 11.13.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compress
ion: off)
Type "help" for help.

postgres-sample=#
```

To test the connection within the cluster, use the DNS of the Service which resolves to the `ClusterIP`, similar to:

```
kubectl exec -ti pod/postgres-sample-monitor-0 -- bash
```

```
postgres@postgres-sample-monitor-0:/psql "host=postgres-sample.default.svc.cluster.loc
al dbname=postgres-sample user=pgadmin password=e0AJi1xiRr51m6gZKqb5Cwj0a6i53L port=54
32"
```

which returns similar to:

```
psql (11.13 (VMware Postgres 11.13.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compress
ion: off)
Type "help" for help.

postgres-sample=#
```

Backing Up and Restoring Tanzu Postgres

This topic describes how to back up and restore Tanzu Postgres.

Overview

Tanzu Postgres allows you to backup instances on-demand, schedule automated backups, restore in-place, and restore from a backup to new Postgres instances.

The supported locations for uploading and retrieving backup artifacts are Amazon S3, or other S3-compatible data stores like Minio.

Tanzu Postgres backup and restore uses four Custom Resource Definitions (CRDs):

- **PostgresBackup:** References a Postgres backup artifact that exists in an external blobstore such as S3 or Minio. Every time you generate an on-demand or scheduled backup, Tanzu Postgres creates a new PostgresBackup resource.
- **PostgresBackupLocation**: References an external blobstore and the necessary credentials for blobstore access.
- **PostgresBackupSchedule**: Represents a CronJob schedule specifying when to perform backups.
- **PostgresRestore**: References a Postgres restore artifact that receives a PostgresBackup resource and restores the data from the backup to a new Postgres instance or to the same postgres instance (an in-place restore).

For detailed information about the CRDs, see Backup and Restore CRD API Reference.

Prerequisites

Before creating a Tanzu Postgres backup you need:

- the `kubectl` command line tool installed on your local client, with access permissions to the Kubernetes cluster.
- access permissions to a preconfigured S3 bucket where the `pgdata` persistent volume (PV) backups will be stored.
- the access credentials that will populate the `accessKeyId` and `secretAccessKey`of the S3 backup secret.
- the instance namespace, if the Postgres instance is already created. Use `kubectl get namespaces` for a list of available namespaces.
- (optional) a pre-agreed backup schedule to be used to configure scheduled backups.

Backing Up Tanzu Postgres

Create on-demand or scheduled backups by configuring the PostgresBackupLocation CRD, which specifies the details of the location and access credentials to the external S3 blobstore.

Configure the Backup Location

To take a backup to an external S3 location, create a PostgresBackupLocation resource:

1. Locate the `backuplocation.yaml` deployment yaml in the `./samples` directory of your downloaded release, and create a copy with a unique name. For example:

```
cp ~/Downloads/postgres-for-kubernetes-v1.3.0/samples/backuplocation.yaml testb
ackuplocation.yaml
```

2. Edit the file using the configuration details of your external S3 bucket. The same file contains the properties of your backup credentials secret. For example:

```
---
apiVersion: sql.tanzu.vmware.com/v1
kind: PostgresBackupLocation
metadata:
  name: backuplocation-sample
spec:
  retentionPolicy:
    fullRetention:
      type: count
      number: 9999999
    diffRetention:
      number: 9999999
  storage:
    s3:
      bucket: "name-of-bucket"
      bucketPath: "/my-bucket-path"
      region: "us-east-1"
      endpoint: "custom-endpoint"
      forcePathStyle: false
      enableSSL: true
      secret:
        name: backuplocation-creds-sample
  additionalParameters: {}
---
apiVersion: v1
kind: Secret
metadata:
  name: backuplocation-creds-sample
type: generic
stringData:
  # Credentials
  accessKeyId: "my-access-key-id"
  secretAccessKey: "my-secret-access-key"
```

For details on the various properties see Backup and Restore CRD API Reference and Secret Properties.

3. Create the PostgresBackupLocation resource in the Postgres instance namespace:

```
kubectl apply -f FILENAME -n DEVELOPMENT-NAMESPACE
```

where:

- `FILENAME` is the name of the configuration file you created in Step 2 above.
- `DEVELOPMENT-NAMESPACE` is the namespace for the Postgres instance you intend to backup.

For example:

```
kubectl apply -f testbackuplocation.yaml -n my-namespace
```

```
postgresbackuplocation.sql.tanzu.vmware.com/backuplocation-sample created
secret/backuplocation-creds-sample configured
```

4. View the created PostgresBackupLocation by running:

```
kubectl get postgresbackuplocation <BACKUP-LOCATION-NAME> \
-o jsonpath={.spec} -n DEVELOPMENT-NAMESPACE
```

For example:

```
kubectl get postgresbackuplocation backuplocation-sample -o jsonpath={.spec} -n
my-namespace
```

which returns similar to:

```
{
    "storage":{
        "s3":{
            "bucket":"name-of-bucket",
            "bucketPath":"/my-bucket-path",
            "enableSSL":true,
            "endpoint":"custom-endpoint",
            "forcePathStyle":false,
            "region":"us-east-1",
            "secret":{
                "name":"backuplocation-creds-sample"
            }
        }
    }
}
```

5. Update the Postgres instance manifest with the PostgresBackupLocation field. Go to the location where you have stored the Tanzu Postgres instance manifest file. For example:

```
cd ./postgres-for-kubernetes-v<version>
```

6. Edit the manifest yaml file you used to deploy the instance; in this example the file is called `postgres.yaml`. Provide a value for the `backupLocation` attribute.

   For example:

```
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: postgres-sample
spec:
  storageClassName: standard
  storageSize: 800M
  cpu: "0.8"
  memory: 800Mi
  monitorStorageClassName: standard
  monitorStorageSize: 1G
  pgConfig:
     dbname: postgres-sample
     username: pgadmin
  serviceType: ClusterIP
  highAvailability:
     enabled: false
  backupLocation:
     name: backuplocation-sample
```

7. Execute the `kubectl apply` command, specifying the manifest file you edited. For example:

   ```
   kubectl apply -f ./postgres.yaml --wait=false
   ```

   ```
   postgres.sql.tanzu.vmware.com "postgres-sample" configured
   ```

   If the manifest file contains any incorrectly formatted values or unrecognized field names, an error message is displayed identifying the issue. Edit the manifest to correct the error and run the command again.

8. Verify the updated configuration by viewing the backupLocation fields of the instance object:

   ```
   kubectl get postgres/postgres-sample -o jsonpath='{.spec.backupLocation}'
   ```

   ```
   {"name":"backuplocation-sample"}
   ```

[Optional] Configure Client-side Encryption for Backups

From Tanzu Postgres 1.7.0 backups are not automatically encrypted. VMware recommends configuring server-side encryption, using the tools offered by your S3 provider. Users can optionally configure client-side encryption using the steps in this topic.

Note: The backups performed using the current backup location can still be restored as long as the backuplocation resource is not deleted from Kubernetes.

1. Identify the backuplocation attached to the instance you'll like to set up encryption for:

   ```
   kubectl get postgres postgres-sample -n <namespace>
   ```

```
NAME               STATUS     DB VERSION    BACKUP LOCATION          AGE
postgres-sample    Running    14.2          backuplocation-sample    17h
```

where `BACKUP LOCATION` lists the name of the desired backup location.

2. Output the contents of the backuplocation to a temporary file:

```
kubectl get postgresbackuplocation backuplocation-sample -n default -o yaml > /
tmp/backuplocation-to-encrypt.yaml
```

3. Edit the `/tmp/backuplocation-to-encrypt.yaml` file to reflect a new unique metadata name, a new bucket or bucket path, and a cipher. In the example below, we're using `/sample-backup-path` as the new bucket path, `my-simple-cipher` as the encryption key, and `encrypted-backup-location` as the unique backuplocation name:

```
---
apiVersion: sql.tanzu.vmware.com/v1
kind: PostgresBackupLocation
metadata:
  name: encrypted-backup-location
spec:
  storage:
    s3:
      bucket: "name-of-bucket"
      bucketPath: "/sample-backup-path"
      region: "us-east-1"
      endpoint: "custom-endpoint"
      forcePathStyle: false
      enableSSL: true
      secret:
        name: backuplocation-creds-sample
  additionalParameters:
    repo1-cipher-pass: "my-simple-cipher"
    repo1-cipher-type: "aes-256-cbc"
```

4. Execute the `kubectl apply` command, specifying the manifest file you edited. For example:

```
kubectl apply -f /tmp/backuplocation-to-encrypt.yaml
```

```
postgresbackuplocation.sql.tanzu.vmware.com/encrypted-backup-location created
```

5. Edit the `backupLocation` attribute in the manifest yaml file you used to deploy the instance to reflect the recently created backuplocation.

   For example:

```
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: postgres-sample
spec:
  storageClassName: standard
  storageSize: 800M
  cpu: "0.8"
  memory: 800Mi
  monitorStorageClassName: standard
  monitorStorageSize: 1G
  pgConfig:
     dbname: postgres-sample
     username: pgadmin
  serviceType: ClusterIP
  highAvailability:
     enabled: false
  backupLocation:
     name: encrypted-backup-location
```

6. Execute the `kubectl apply` command, specifying the manifest file you edited. For example:

   ```
   kubectl apply -f ./postgres.yaml --wait=false
   ```

   ```
   postgres.sql.tanzu.vmware.com "postgres-sample" configured
   ```

   If the manifest file contains any incorrectly formatted values or unrecognized field names, an error message is displayed identifying the issue. Edit the manifest to correct the error and run the command again.

7. Verify the updated configuration by viewing the backupLocation fields of the instance object:

   ```
   kubectl get postgres/postgres-sample -o jsonpath='{.spec.backupLocation}'
   ```

   ```
   {"name":"encrypted-backup-location"}
   ```

8. The data pod will then be restarted to pick up the new changes to the backuplocation. Once the pods are up and running, use the following command to confirm that cipher is now in use

   ```
   kubectl exec -t postgres-sample-0 -- bash -c 'pgbackrest info --stanza=$BACKUP_
   STANZA_NAME | grep cipher'
   ```

   The output looks similar to:

   ```
   Defaulted container "pg-container" out of: pg-container, instance-logging, reco
   nfigure-instance, postgres-metrics-exporter, postgres-sidecar
       cipher: aes-256-cbc
   ```

Perform an On-Demand Backup

To take a backup:

1. Locate the `backup.yaml` deployment template located in the `./samples` directory of the downloaded release file.

2. Create a copy of the `backup.yaml` file and give it a unique name. For example:

   ```
   cp ~/Downloads/postgres-for-kubernetes-v1.3.0/backup.yaml testbackup.yaml
   ```

3. Edit the file according to your environment. For details on the properties of the PostgresBackup resource, see Properties for the PostgresBackup Resource.

4. Trigger the backup by creating the PostgresBackup resource in the instance namespace, by running:

   ```
   kubectl apply -f FILENAME -n DEVELOPMENT-NAMESPACE
   ```

   where `FILENAME` is the name of the configuration file you created in Step 3 above.

   For example:

   ```
   kubectl apply -f testbackup.yaml -n my-namespace
   ```

   ```
   postgresbackup.sql.tanzu.vmware.com/backup-sample created
   ```

5. Verify that the backup has been generated, and track its progress by using:

   ```
   kubectl get postgresbackup backup-sample -n DEVELOPMENT-NAMESPACE
   ```

   For example:

   ```
   kubectl get postgresbackup backup-sample -n my-namespace
   ```

   ```
   NAME             STATUS      SOURCE INSTANCE     TYPE   TIME STARTED           T
   IME COMPLETED
   backup-sample    Succeeded   postgres-sample     full   2021-08-31T14:29:14Z   2
   021-08-31T14:29:14Z
   ```

   For further details on the above output, see List Existing PostgresBackup Resources below.

Create Scheduled Backups

To create scheduled backups, create a PostgresBackupSchedule resource:

1. Locate the `backupschedule.yaml` template in the `./samples` directory of the release download, and copy to a new file. For example:

   ```
   cp ~/Downloads/postgres-for-kubernetes-v1.3.0/samples/backupschedule.yaml testb
   ackupschedule.yaml
   ```

2. Edit the file with the name of the Postgres instance you want to backup. For example:

   ```
   apiVersion: sql.tanzu.vmware.com/v1
   kind: PostgresBackupSchedule
   metadata:
     name: backupschedule-sample
   spec:
     backupTemplate:
       spec:
          sourceInstance:
             name: postgres-sample
   ```

```
        type: full
  schedule: "0 0 * * SAT"
```

where:

- `postgres-sample` is the instance you're planning to backup.
- `type` is full.
- `schedule` is a cron job schedule; in this example it is planned for every Saturday at 00:00:00.

For an explanation of the PostgresBackupSchedule properties, see Properties for the PostgresBackupSchedule Resource.

3. Create the PostgresBackupSchedule resource in the same namespace of the Postgres instance that you referenced in the PostgresBackupSchedule manifest file.

```
kubectl apply -f FILENAME -n DEVELOPMENT-NAMESPACE`
```

where:

- `FILENAME` is the name of the configuration file you created in Step 1.
- `DEVELOPMENT-NAMESPACE` is the namespace for the Postgres instance you intend to backup.

For example:

```
kubectl apply -f testbackupschedule.yaml -n my-namespace
```

```
postgresbackupschedule.sql.tanzu.vmware.com/backupschedule-sample created
```

4. Verify that the PostgresBackupSchedule has been created by running:

```
kubectl get postgresbackupschedule backupschedule-sample -o jsonpath={.spec} -n
DEVELOPMENT-NAMESPACE
```

For example:

```
kubectl get postgresbackupschedule backupschedule-sample -o jsonpath={.spec} -n
my-namespace
```

```
{
    "backupTemplate": {
    "spec": {
    "sourceInstance": {
      "name": "postgres-sample"
    },
    "type": "full"
    }
    },
    "schedule": "@daily"
}
```

After configuring the `PostgresBackupLocation` resource and the `PostgresBackupSchedule` resource for an existing Postgres instance, backups will be generated and uploaded to the external blobstore at the scheduled time.

The PostgresBackupSchedule generates PostgresBackup resources that have a name format like: `SCHEDULE-NAME-TIMESTAMP`. For example, if the PostgresBackup resource on the Kubernetes cluster is named `pgbackupschedule-sample`, and a backup was taken on Thursday, December 10, 2020 at 8:51:03 PM GMT, the PostgresBackup resource name is `pgbackupschedule-sample-20201210-205103`.

Backup Schedule Status

Check the `status.message` field in PostgresBackupSchedule CR to understand any issues with PostgresBackupSchedule spec, backups scheduling, invalid cron schedule syntax, or errors like the `backuplocation.name` not configured in the `sourceInstance`.

For example:

```
kubectl get postgresbackupschedule backupschedule-sample -o jsonpath={.status.message}
```

could return an output similar to:

```
Instance my-postgres-1 does not exist in the default namespace
```

if the user inserted the wrong Postgres instance name, `my-postgres-1` instead of `postgres-sample`, in PostgresBackupSchedule's `spec.backupTemplate.spec.sourceInstance.name`.

If the backup was successfully scheduled but the backup itself failed, then to troubleshoot, see Troubleshoot Backup and Restore.

Listing Backup Resources

You might want to list existing PostgresBackup resources for various reasons, for example:

- To select a backup to restore. For steps to restore a backup, see Restoring Tanzu Postgres.
- To see the last successful backup.
- To verify that scheduled backups are running as expected.
- To find old backups that need to be cleaned up. For steps to delete backups, see Deleting Old Backups.

List existing PostgresBackup resources by running:

```
kubectl get postgresbackup
```

```
NAME            STATUS      SOURCE INSTANCE    TYPE    TIME STARTED          TIME COMPL
ETED
backup-sample   Succeeded   postgres-sample    full    2021-08-31T14:29:14Z  2021-08-31
T14:29:14Z
```

Where:

- `STATUS` Represents the current status of the backup. Allowed values are:
  - Pending: The backup has been received but not scheduled on a Postgres Pod.
  - Running: The backup is being generated and streamed to the external blobstore.
  - Succeeded: The backup has completed successfully.
  - Failed: The backup has failed to complete. To troubleshoot a failed backup, see Troubleshoot Backup and Restore.
- `SOURCE INSTANCE` is the Postgres instance the backup was taken from.
- `TYPE` is the type of Postgres backup that was executed.
- `TIME STARTED` is the time that the backup process started.
- `TIME COMPLETED` is the time that the backup process finished. If the backup fails, this value is empty.

Note: Users with version 1.2.0 backups can still list the previous backup information, along with the new 1.3.0 backups. Use the `pgbackrest` command directly on the primary pod to review all existing backups, independent of version. For example, login into the pod and run:

```
postgres@postgres-sample-0:/$ pgbackrest info --stanza=${BACKUP_STANZA_NAME}
```

If the `BACKUP_STANZA_NAME` is `default-postgres-sample`, the output would be similar to:

```
stanza: default-postgres-sample
    status: ok
    cipher: aes-256-cbc

    db (current)
        wal archive min/max (11): 000000010000000000000004/000000010000000000000009

        full backup: 20210915-140558F
            timestamp start/stop: 2021-09-15 14:05:58 / 2021-09-15 14:06:04
            wal start/stop: 000000010000000000000004 / 000000010000000000000004
            database size: 31.0MB, database backup size: 31.0MB
            repo1: backup set size: 3.7MB, backup size: 3.7MB

        full backup: 20210916-143321F
            timestamp start/stop: 2021-09-16 14:33:21 / 2021-09-16 14:33:41
            wal start/stop: 000000010000000000000009 / 000000010000000000000009
            database size: 31MB, database backup size: 31MB
            repo1: backup set size: 3.7MB, backup size: 3.7MB
```

To list backups related to a specific Postgres instance in the cluster, use:

```
kubectl get postgresbackups -l postgres-instance=postgres-sample
```

with output similar to:

```
NAME                 STATUS       SOURCE INSTANCE    TYPE           TIME STARTED
TIME COMPLETED
backup-sample        Succeeded    postgres-sample    full           2021-10-05T21:17:34Z
2021-10-05T21:17:41Z
backup-sample-1      Succeeded    postgres-sample    full           2021-10-05T21:28:46Z
2021-10-05T21:28:54Z
```

```
backup-sample-2   Succeeded   postgres-sample   full           2021-10-05T21:29:44Z
2021-10-05T21:29:51Z
backup-sample-3   Succeeded   postgres-sample   differential   2021-10-05T21:36:43Z
2021-10-05T21:36:49Z
backup-sample-4   Succeeded   postgres-sample   differential   2021-10-05T21:37:20Z
2021-10-05T21:37:26Z
backup-sample-5   Succeeded   postgres-sample   differential   2021-10-05T21:37:39Z
2021-10-05T21:37:45Z
backup-sample-6   Succeeded   postgres-sample   full           2021-10-05T21:43:35Z
2021-10-05T21:43:42Z
backup-sample-7   Succeeded   postgres-sample   full           2021-10-05T21:49:33Z
2021-10-05T21:49:41Z
backup-sample-8   Succeeded   postgres-sample   full           2021-10-05T22:07:43Z
2021-10-05T22:07:50Z
```

Deleting Old Backups

Removing Backup Artifacts from an S3 location

Tanzu Postgres for Kubernetes does not natively support retention policies for backup artifacts. To delete backups from an S3 location use the `pgbackrest expire` command that deletes the backup information from `pgbackrest info`, and also removes it from the blob store.

NOTE: You can only expire a full backup if another full backup exists. If you only have a single full backup, take another full backup before you expire the first one.

- Use `kubectl exec` to get access into a pod:

  ```
  kubectl exec -ti pod/my-postgres-ha-0 -- bash
  ```

- Use `pg_autoctl show state` to ensure you're on the primary pod:

  ```
  pg_autoctl show state
  ```

  ```
  Name |  Node |                                                       Host:
  Port |        TLI: LSN |   Connection |    Reported State |    Assigned State
  -------+-------+----------------------------------------------------------------
  -------+-----------------+--------------+-----------------+------------------
  node_1 |     1 | my-postgres-ha-0.my-postgres-ha-agent.default.svc.cluster.loca
  l:5432 |   1: 0/15000148 |   read-write |         primary |           primary
  node_2 |     2 | my-postgres-ha-1.my-postgres-ha-agent.default.svc.cluster.loca
  l:5432 |   1: 0/15000148 |    read-only |       secondary |         secondary
  ```

- Use the `pgbackrest info` to view all of the existing backups.

  ```
  pgbackrest info --stanza=$BACKUP_STANZA_NAME
  ```

  ```
   stanza: default-my-postgres-ha
   status: ok
   cipher: aes-256-cbc

   db (current)
      wal archive min/max (14): 000000010000000000000001/000000010000000000000014
  ```

```
    full backup: 20220119-190307F
        timestamp start/stop: 2022-01-19 19:03:07 / 2022-01-19 19:03:22
        wal start/stop: 000000010000000000000011 / 000000010000000000000011
        database size: 34.3MB, database backup size: 34.3MB
        repo1: backup set size: 4.3MB, backup size: 4.3MB

    incr backup: 20220119-190307F_20220119-190339I
        timestamp start/stop: 2022-01-19 19:03:39 / 2022-01-19 19:03:41
        wal start/stop: 000000010000000000000013 / 000000010000000000000013
        database size: 34.3MB, database backup size: 50.0KB
        repo1: backup set size: 4.3MB, backup size: 3.5KB
        backup reference list: 20220119-190307F

    incr backup: 20220119-190307F_20220119-190345I
        timestamp start/stop: 2022-01-19 19:03:45 / 2022-01-19 19:03:47
        wal start/stop: 000000010000000000000014 / 000000010000000000000014
        database size: 34.3MB, database backup size: 52.0KB
        repo1: backup set size: 4.3MB, backup size: 3.6KB
        backup reference list: 20220119-190307F

    full backup: 20220119-191742F
        timestamp start/stop: 2022-01-19 19:17:42 / 2022-01-19 19:17:53
        wal start/stop: 000000010000000000000016 / 000000010000000000000016
        database size: 34.3MB, database backup size: 34.3MB
        repo1: backup set size: 4.3MB, backup size: 4.3MB
```

- Use `pgbackrest expire --stanza=$BACKUP_STANZA_NAME` to expire a full backup and remove it from the s3 blobstore. The command removes the backup and the related incremental backups from the stanza and from s3. If it fails, ensure you have at least two full backups, so you can delete one.

```
pgbackrest expire --stanza=default-postgres-sample --set=20220302-075533F
```

```
2022-03-16 19:03:12.408 P00   INFO: expire command begin 2.38: --config=/etc/pg
backrest/pgbackrest.conf --exec-id=11799-7d0900ea --log-level-console=info --lo
g-level-file=off --repo1-cipher-pass=<redacted> --repo1-cipher-type=aes-256-cbc
--repo1-path=/my-bucket-path2 --repo1-retention-diff=5 --repo1-retention-full=3
--repo1-retention-full-type=count --repo1-s3-bucket=postgresql-backups --repo1-
s3-endpoint=minio.minio.svc.cluster.local:9000 --repo1-s3-key=<redacted> --repo
1-s3-key-secret=<redacted> --repo1-s3-region=us-east-1 --repo1-s3-uri-style=pat
h --no-repo1-storage-verify-tls --repo1-type=s3 --set=20220302-075533F --stanza
=default-postgres-sample
2022-01-19 19:17:57.355 P00   INFO: repo1: expire adhoc backup set 20220119-190
307F, 20220119-190307F_20220119-190339I, 20220119-190307F_20220119-190345I
2022-01-19 19:17:57.392 P00   INFO: repo1: remove expired backup 20220119-19030
7F_20220119-190345I
2022-01-19 19:17:57.395 P00   INFO: repo1: remove expired backup 20220119-19030
7F_20220119-190339I
2022-01-19 19:17:57.398 P00   INFO: repo1: remove expired backup 20220119-19030
7F
2022-01-19 19:17:57.689 P00   INFO: expire command end: completed successfully
(377ms)
```

- Use `pgbackrest info` to see the new state of the stanza:

```
pgbackrest info --stanza=$BACKUP_STANZA_NAME
```

```
stanza: default-my-postgres-ha
status: ok
cipher: aes-256-cbc

db (current)
   wal archive min/max (14): 000000010000000000000001/000000010000000000000016

   full backup: 20220119-191742F
       timestamp start/stop: 2022-01-19 19:17:42 / 2022-01-19 19:17:53
       wal start/stop: 000000010000000000000016 / 000000010000000000000016
       database size: 34.3MB, database backup size: 34.3MB
       repo1: backup set size: 4.3MB, backup size: 4.3MB
```

Deleting a Backup

Tanzu Postgres does not automatically manage backup removal. To delete a backup and the S3 backup artifacts associated with it, follow two steps:

- Use the steps in Removing Backup Artifacts from an S3 location to remove all the specific $BACKUP_STANZA_NAME backups from the S3 location.
- Delete the associated PostgresBackup resources in the Kubernetes cluster by running:

  ```
  kubectl delete postgresbackup BACKUP-NAME -n DEVELOPMENT-NAMESPACE
  ```

  For example:

  ```
  kubectl delete postgresbackup backup-sample -n my-namespace
  ```

Restoring Tanzu Postgres

Tanzu Postgres allows you to perform three types of data restores:

- Restoring from a full backup to an existing instance, overriding existing data. See Restore In-place to the same instance.
- Restoring a full backup to a different instance in the same namespace. See Restore to a different instance.
- Restoring a backup to a different namespace or cluster. See Restore to a different namespace or cluster.

Customers on Tanzu Operator 1.7.x that wish to restore backups from prior versions, see Restoring Backups taken prior to Postgres Operator 1.7.0.

Restore In-place

In this scenario, you can use a previous backup to override data in an existing instance.

Prerequisites

Before you restore from a backup, you must have:

- An existing PostgresBackup in your current namespace. To list the existing PostgresBackup resources, see Listing Backup Resources. You can restore from any kind of backup (full, differential, incremental) provided it follows the pgbackrest guidelines.
- A PostgresBackupLocation that represents the bucket where the existing backup artifact is stored. See Configure the Backup Location above.

Procedure

To restore from a full backup:

1. Locate the `restore.yaml` deployment yaml in the `./samples` directory of your downloaded release, and create a copy with a unique name. For example:

```
cp ~/Downloads/postgres-for-kubernetes-v1.3.0/samples/restore.yaml testrestore.
yaml
```

2. Locate all backups of your existing instance. For example, for all the backups executed against a Postgres instance called 'postgres-sample' use:

```
kubectl get postgresbackups -n NAMESPACE -l postgres-instance=postgres-sample
```

```
NAME                 STATUS      SOURCE INSTANCE   TYPE   TIME STARTED           T
IME COMPLETED
backup-sample-4      Succeeded   postgres-sample   full   2021-09-24T19:40:17Z   2
021-09-24T19:40:24Z
backup-sample-5      Succeeded   postgres-sample   full   2021-09-25T16:54:55Z   2
021-09-25T16:55:02Z
backup-sample-6      Succeeded   postgres-sample   full   2021-09-24T19:48:41Z   2
021-09-24T19:48:48Z
backup-sample-7      Succeeded   postgres-sample   full   2021-09-27T23:04:06Z   2
021-09-27T23:04:13Z
backup-sample-8      Succeeded   postgres-sample   full   2021-09-27T23:19:51Z   2
021-09-27T23:19:58Z
```

3. Locate the backup you'll like to restore, and edit the 'restore.yaml' with your information. For information about the PostgresRestore resource properties, see Backup and Restore CRD API Reference.

```
  apiVersion: sql.tanzu.vmware.com/v1
  kind: PostgresRestore
  metadata:
    name: restore-sample
  spec:
    sourceBackup:
      name: backup-sample
    targetInstance:
      name: postgres-sample
```

4. For in-place restore, ensure that the `sourceBackup` was performed on the `targetInstance`. Refer to step 2 for validation.

5. Trigger the restore by creating the PostgresRestore resource in the same namespace as the PostgresBackup and PostgresBackupLocation. Run:

   ```
   kubectl apply -f FILENAME -n DEVELOPMENT-NAMESPACE
   ```

   where `FILENAME` is the name of the configuration file you created in Step 2 above.

   For example:

   ```
   kubectl apply -f testrestore.yaml -n my-namespace
   ```

   ```
   postgresrestores.sql.tanzu.vmware.com/restore-sample created
   ```

6. Verify that a restore has been triggered and track the progress of your restore by running:

   ```
   kubectl get postgresrestore restore-sample -n DEVELOPMENT-NAMESPACE
   ```

   For example:

   ```
   kubectl get postgresrestore restore-sample -n my-namespace
   ```

   ```
   NAME             STATUS      SOURCE BACKUP     TARGET INSTANCE    TIME STARTED
   TIME COMPLETED
   restore-sample   Succeeded   backup-sample     postgres-sample    2021-09-27T23:
   34:13Z   2021-09-27T23:34:26Z
   ```

   To understand the output, see the table below:

Column Name	Meaning
`STATUS`	Represents the current status of the restore process. Allowed values are: ◦ Running: The restore is in progress. ◦ RecreatingNodes: The postgres nodes are being restarted as part of the restore workflow. ◦ RecreatingPrimary: In case of HA target instance, the primary pod is being restarted ◦ WaitForPrimary: Wait for the primary pod to be up and running ◦ RecreatingSecondary: In case of HA target instance, the secondary pod is being restarted ◦ Finalizing: The restore is nearly complete, waiting for the restart to be done and target instance to be up. ◦ Succeeded: The restore has completed successfully. ◦ Failed: The restore failed. To troubleshoot, see Troubleshooting Backup and Restore.
`SOURCE BACKUP`	The name of the backup being restored.
`TARGET INSTANCE`	The name of the source postgres instance to be restored with the backup contents.
`TIME STARTED`	The time that the restore process started.

`TIME COMPLETED`	The time that the restore process finished.

Restore to a Different Instance

Prerequisites

- Ensure the new targeted instance exists in the same namespace as the Postgres instance you're restoring from.

Procedure

1. Create a new Postgres instance, if your target instance doesn't already exist.
2. Locate the `restore.yaml` deployment yaml in the `./samples` directory of your downloaded release, and create a copy with a unique name. For example:

   ```
   cp ~/Downloads/postgres-for-kubernetes-v1.3.0/samples/restore.yaml testrestore.
   yaml
   ```

3. Locate all the backups for the existing instance. For example, to list all backups executed against a Postgres instance called 'postgres-sample' use a command similar to:

   ```
   kubectl get postgresbackups -n NAMESPACE -l postgres-instance=postgres-sample
   ```

   ```
   NAME               STATUS      SOURCE INSTANCE   TYPE   TIME STARTED          T
   IME COMPLETED
   backup-sample-4    Succeeded   postgres-sample   full   2021-09-24T19:40:17Z   2
   021-09-24T19:40:24Z
   backup-sample-5    Succeeded   postgres-sample   full   2021-09-25T16:54:55Z   2
   021-09-25T16:55:02Z
   backup-sample-6    Succeeded   postgres-sample   full   2021-09-24T19:48:41Z   2
   021-09-24T19:48:48Z
   backup-sample-7    Succeeded   postgres-sample   full   2021-09-27T23:04:06Z   2
   021-09-27T23:04:13Z
   backup-sample-8    Succeeded   postgres-sample   full   2021-09-27T23:19:51Z   2
   021-09-27T23:19:58Z
   ```

4. Once you've located the backup you'll like to restore to, edit the 'restore.yaml' For information about the properties that you can set for the PostgresRestore resource, see Backup and Restore CRD API Reference.

   ```
     apiVersion: sql.tanzu.vmware.com/v1
     kind: PostgresRestore
     metadata:
       name: restore-sample
     spec:
       sourceBackup:
         name: backup-sample
       targetInstance:
         name: postgres-sample
   ```

5. For restore to a new instance, make sure that the sourceBackup was NOT performed on the target instance, refer to step 3 for validation.

6. Trigger the restore by creating the PostgresRestore resource in the same namespace as the PostgresBackup and PostgresBackupLocation by running:

   ```
   kubectl apply -f FILENAME -n DEVELOPMENT-NAMESPACE
   ```

   Where `FILENAME` is the name of the configuration file you created in Step 2 above.

   For example:

   ```
   kubectl apply -f testrestore.yaml -n my-namespace
   ```

   ```
   postgresrestores.sql.tanzu.vmware.com/restore-sample created
   ```

7. Verify that a restore has been triggered and track the progress of your restore by running:

   ```
   kubectl get postgresrestore restore-sample -n DEVELOPMENT-NAMESPACE
   ```

   For example:

   ```
   kubectl get postgresrestore restore-sample -n my-namespace
   ```

   ```
   NAME             STATUS      SOURCE BACKUP   TARGET INSTANCE   TIME STARTED
   TIME COMPLETED
   restore-sample   Succeeded   backup-sample   postgres-sample   2021-09-27T23:3
   4:13Z   2021-09-27T23:34:26Z
   ```

8. To understand the output, see the table below:

Column Name	Meaning
`STATUS`	Represents the current status of the restore process. Allowed values are: ◦ Running: The restore is in progress. ◦ RecreatingNodes: The postgres nodes are being restarted as part of the restore workflow. ◦ RecreatingPrimary: In case of HA target instance, the primary pod is being restarted ◦ WaitForPrimary: Wait for the primary pod to be up and running ◦ RecreatingSecondary: In case of HA target instance, the secondary pod is being restarted ◦ Finalizing: The restore is nearly complete, waiting for the restart to be done and target instance to be up. ◦ Succeeded: The restore has completed successfully. ◦ Failed: The restore failed. To troubleshoot, see Troubleshoot Backup and Restore below.
`SOURCE BACKUP`	The name of the backup being restored.
`TARGET INSTANCE`	The name of the new Postgres instance to be restored with the backup contents.
`TIME STARTED`	The time that the restore process started.

`TIME COMPLETED`	The time that the restore process finished.

Restore to a different namespace or cluster

This scenario allows users to restore a Tanzu Postgres instance to a different namespace than the namespace it was backed up in. Using this feature, users can perform the restore even if the original instance has been accidentally deleted.

This feature synchronizes the backupLocation and backups of a single instance across different namespaces.

Important: This feature was introduced in Tanzu Postgres 1.7.0. Customers on previous releases, with existing backups, cannot utilize this feature without upgrading. Customers on Tanzu Postgres 1.6.0 and earlier, are advised to upgrade their instances to 1.7.0, and perform a full backup after the upgrade. This full backup and any future backups will be able to be used for restores to a different namespace.

Prerequisites

In this section, the namespace where the backup was created is referred to as "origin" namespace. The namespace for the restore is referred to as "target" namespace.

Ensure that:

- You know the S3 backup location details of the given instance in the origin namespace.
- You have the name of a Tanzu Postgres instance on version 1.7.0 and above, that exists or existed in the origin namespace.
- You have an existing PostgresBackup object in the origin namespace. To list any existing PostgresBackup resources, see Listing Backup Resources.
- You have a PostgresBackupLocation that represents the bucket where the existing backup artifact is stored.

Procedure

1. On the origin namespace, identify the backupLocation name of the backup you wish to restore to the target namespace:

   ```
   kubectl get postgresbackup backup-sample -n <namespace-origin> -o jsonpath='{.status.backupLocation}'
   ```

   where `backup-sample` is the name of the backup resource created for the instance backup in origin namespace. The output is similar to:

   ```
   my-backup-location
   ```

2. On the targe namespace, create a `backuplocation.yaml`, using as content the output you receive from:

```
kubectl get postgresbackuplocation <name of backup location> -n <namespace-orig
in> -o yaml
```

Edit the namespace field to match the target namespace before applying the CRD.

At this point any backups taken by Tanzu Operator 1.7.0 (or later), for this specific `postgresbackuplocation` resource, will start to synchronize.

3. Create the secret that is attached to that `postgresbackupLocation` CRD. To retrieve the secret name, issue a command similar to:

   ```
   kubectl get postgresbackuplocation <name-of-backup-location> -n <namespace-orig
   in> -o jsonpath='{.spec.storage.s3.secret.name}'
   ```

   which returns an output similar to:

   ```
   my-backuplocation-secret
   ```

   Create the secret on the target namespace, using the content you receive from the output of a command similar to:

   ```
   kubectl get secret <name-of-secret-retrieved-above> -n <namepsace-origin> -o ya
   ml
   ```

   Edit the namespace field to match the target namespace before applying the CRD.

4. List the synchronized backups by using a command similar to:

   ```
   kubectl get postgresbackup -n <namespace-target> -l sql.tanzu.vmware.com/recove
   red-from-backuplocation=true
   ```

   The output would be similar to:

   ```
   NAMESPACE               NAME                               STATUS      SOURCE INS
   TANCE    TYPE   TIME STARTED          TIME COMPLETED
   target-namespace        sync-20220414-143642f-2aff7501     Succeeded   my-postgre
   s        full   2022-04-14T14:36:42Z  2022-04-14T14:37:01Z
   ```

5. Select which backup you wish to restore.

6. On the target namespace, create a postgres instance that you'll like to restore the backup to. If an instance already exists, skip this step.

   ```
   kubectl apply -f postgres.yaml
   ```

7. Wait until the instance is up and running:

   ```
   kubectl wait postgres postgres-sample -n <namespace-target> --for=jsonpath={.st
   atus.currentState}=Running
   ```

8. Edit the target namespace instance restore yaml, and edit it to reflect the backup you'll like to restore, and the instance you'll like to restore that backup, Create the `PostgresRestore yaml`. A sample restore yaml is shown below:

```
apiVersion: sql.tanzu.vmware.com/v1
kind: PostgresRestore
metadata:
   name: <provide-name-for-restore>
spec:
  sourceBackup:
    name: <provide-the-backup-name>
  targetInstance:
    name: <provide-the-instance-name>
```

9. Perform the restore by using:

```
kubectl apply -f restore.yaml
```

10. Validate that the restore succeeded by using a command similar to:

```
kubectl get postgresrestore.sql.tanzu.vmware.com/restore-sample -n <namespace-t
arget>
```

which will show an output similar to:

```
NAME                  STATUS      SOURCE BACKUP                       TARGET INSTA
NCE   TIME STARTED           TIME COMPLETED
restore-sample        Succeeded   sync-20220415-201444f-2aff7501   my-postgres
2022-04-15T20:44:50Z   2022-04-15T20:45:31Z
```

Where the `SOURCE BACKUP` uniquely identifies the synchronized backup object by using the date (for example 20220415), time (for example 201444 represents 20:14:44), type of backup (f - full, d - differential, i - incremental), and part of the instance UUID number.

11. Validate that the Tanzu Postgres instance in your target namespace is running, using:

```
kubectl get postgres.sql.tanzu.vmware.com/postgres-sample -n <namespace-target>
```

Restore Backups taken prior to Postgres Operator 1.7.0

1. Select which backups you'll like to restore:

```
kubectl get postgresbackup -n <namespace>
```

```
NAME             STATUS      SOURCE INSTANCE      TYPE   TIME STARTED
TIME COMPLETED
backup-sample    Succeeded   postgres-sample      full   2022-04-15T15:06:35Z
2022-04-15T15:07:10Z
```

2. Use kubectl edit-status to edit the status attribute of the selected PostgresBackup object to contain the following information:

```
kubectl edit-status postgresbackup backup-sample -n <namespace>
```

Attribute	Command to get the value

`backupLocation`	<ul><li>Find backupLocation attached to the instance kubectl get postgres postgres-sample -o jsonpath={.spec.backupLocation.name}</li><li>Get the cipher that'll be used for restore kubectl get secrets postgres-sample-pgbackrest-secret -o jsonpath={.data.cipher} | base64 -d</li><li>Follow the steps here to create a client-side encrypted backupLocation and provide the value retrieved above as the cipher value</li></ul>
`dbName`	kubectl get postgres postgres-sample -o jsonpath={.spec.pgConfig.dbname}
`stanzaName`	kubectl exec -t pod/my-postgres-0 -c pg-container -- bash -c 'echo $OLD_BACKUP_STANZA_NAME'

3. Confirm that the status fields are correctly populated:

```
kubectl describe postgresbackup postgres-sample -n <namespace>
```

```
Status:
  Status:
  Backup Location:  encrypted-backup-location
  Db Name:          postgres-sample
  Phase:            Succeeded
  Restore Label:    20220415-150641F
  Stanza Name:      default-my-postgres
  Time Completed:   2022-04-15T15:07:10Z
  Time Started:     2022-04-15T15:06:35Z
```

4. Once you've updated the backup you'll like to restore, edit the `restore.yaml` to reflect the said backup and the instance you'll like to restore to:

```
  apiVersion: sql.tanzu.vmware.com/v1
  kind: PostgresRestore
  metadata:
    name: restore-sample
  spec:
    sourceBackup:
      name: backup-sample
    targetInstance:
      name: postgres-sample
```

5. Execute the `kubectl apply` command, specifying the manifest file you edited. For example:

```
kubectl apply -f ./restore.yaml --wait=false
```

Validating a Successful Restore

Validate that the Postgres instance has a status of Running using a command like:

```
kubectl get postgres.sql.tanzu.vmware.com/postgres-sample
```

which should show an output similar to:

```
NAME             STATUS    BACKUP LOCATION         AGE
postgres-sample  Running   backuplocation-sample   43h
```

Migrating to Tanzu Postgres 1.3.0 Backup and Restore

This topic applies to customers that already use Tanzu Postgres 1.2.0 and earlier.

Tanzu Postgres 1.3.0 deprecates the `backupLocationSecret` field on the Postgres instance spec which was used in versions 1.2.0 and earlier. Version 1.3.0 introduces three new backup and restore CRDs. Users can still view and restore from earlier backups, but Tanzu Postgres 1.3.0 requires migration to the new backup strategy. Perform the following steps to migrate:

1. Confirm that the existing Postgres instance references a backupLocationSecret:

   ```
   kubectl get postgres.sql.tanzu.vmware.com/postgres-sample -o jsonpath='{.spec.b
   ackupLocationSecret.name}'
   ```

   ```
   my-postgres-s3-secret
   ```

   where `my-postgres-s3-secret` is an example of a secret referenced in the postgres-sample manifest file.

2. Remove the backupLocationSecret from the instance spec:

   ```
   kubectl patch postgres.sql.tanzu.vmware.com/postgres-sample --type='json' -p
   ='[{"op": "remove", "path":"/spec/backupLocationSecret"}]'
   ```

   ```
   postgres.sql.tanzu.vmware.com/postgres-sample patched
   ```

3. Wait until all the changes have been successfully applied

   ```
   kubectl rollout status statefulset.apps/postgres-sample
   ```

   ```
   partitioned roll out complete: 1 new pods have been updated...
   ```

4. Upgrade the Tanzu Postgres Operator. For details see Upgrading the Tanzu Postgres Operator and Instances.

5. Create a new backup resource and reference it in the instance spec. See Configure the Backup Location.

6. Confirm that all the instances previously using backupLocationSecret have been updated to use the PostgresBackupLocation CR.

7. Delete the secret that is no longer necessary.

8. Perform an on demand backup or create a schedule for scheduled backups. See Perform an On-Demand Backup, and Create Scheduled Backups.

Troubleshooting Backup and Restore

To troubleshoot any issues, review the resource status, and read any messages associated with the resource events. Monitor the `STATUS` column of any Postgres custom resource using `kubectl get postgresbackup`, to confirm if status is `Failed`, or is stuck in `Pending`, `Scheduled`, or `Running`. Then try to investigate:

- Misconfiguration issues
- Problems with the external blobstore
- Issues with the Postgres Operator

FAILED status for PostgresBackup Resource

Check the `STATUS` column in the output of `kubectl get postgresbackup` command to view if a backup has failed:

```
kubectl get postgresbackup
```

```
NAME            STATUS   SOURCE INSTANCE   TYPE   TIME STARTED           TIME COMPLETE
D
backup-sample   Failed   postgres-sample   full   2021-08-31T14:29:14Z   2021-08-31T1
4:29:14Z
```

Diagnose the issue by inspecting the Kubernetes events for the `postgresbackup` resource, using `kubectl describe`, and check the `Events` section.

Note: By default, Kubernetes events are stored in etcd for a limited amount of time, so there may not be any events if the failure occurred several hours ago.

Below are a couple of failure events and their corresponding resolution:

Failed due to missing pgbackrest.conf file

Error:

```
kubectl describe postgresbackup backup-sample
```

```
Spec:
  Source Instance:
    Name:  postgres-sample
  Type:    full
Status:
  Phase:           Failed
  Time Completed:  2022-02-25T19:37:07Z
  Time Started:    2022-02-25T19:37:07Z
Events:
  Type     Reason  Age                  From                          Message
  ----     ------  ----                 ----                          -------
  Warning  Failed  22s (x2 over 27s)   postgres-backup-controller  WARN: environment co
ntains invalid option 'config-version'ERROR: [055]: unable to open missing file '/etc/
pgbackrest/pgbackrest.conf' for read
```

Resolution:

In the example above, the `backup-sample` expected a file called `/etc/pgbackrest/pgbackrest.conf` to exist. Fix this problem by creating and attaching a PostgresBackupLocation CR to the Postgres instance.

FAILED due to S3 server certificate validation error

Error:

```
kubectl describe postgresbackup backup-sample
```

```
Spec:
  Source Instance:
    Name:  postgres-sample
  Type:    full
Status:
  Phase:           Failed
  Time Completed:  2022-02-25T19:37:07Z
  Time Started:    2022-02-25T19:37:07Z
Events:
  Type     Reason  Age   From                        Message
  ----     ------  ----  ----                        -------
  Warning  Failed  20s   postgres-backup-controller  WARN: environment contains invali
d option 'config-version'ERROR: [095]: unable to load info file '/my-bucket-path/archi
ve/default-postgres-sample/archive.info' or '/my-bucket-path/archive/default-postgres-
sample/archive.info.copy':       CryptoError: unable to verify certificate presented b
y 'minio.minio.svc.cluster.local:9000': [20] unable to get local issuer certificate
HINT: is or was the repo encrypted?       CryptoError: unable to verify certificate pr
esented by 'minio.minio.svc.cluster.local:9000': [20] unable to get local issuer certi
ficate       HINT: is or was the repo encrypted?       HINT: archive.info cannot be op
ened but is required to push/get WAL segments.       HINT: is archive_command configur
ed correctly in postgresql.conf?       HINT: has a stanza-create been performed?
HINT: use --no-archive-check to disable archive checks during backup if you have an al
ternate archiving scheme.
```

Resolution:

In this example, the PostgresBackupLocation associated with the source instance was configured with `enableSSL: true` but the S3 server TLS is not properly configured (for e.g. it might be using a self-signed certificate). To resolve this issue, set the S3 server TLS appropriately. If this is a testing/demo scenario, you can set `enableSSL` to `false` in the PostgresBackupLocation, wait for the instance to restart, and then create a PostgresBackup again.

FAILED status for PostgresRestore

In this example, the `kubectl get` command outputs a `Failed` status:

```
kubectl get postgresrestore
```

```
NAME            STATUS      SOURCE BACKUP          TARGET INSTANCE   TIME STARTED
TIME COMPLETED
restore-test    Failed      sample-source-backup   postgres-sample   2021-09-29T19:0
0:26Z   2021-09-29T19:00:26Z
```

Diagnose the issue by inspecting the status message on the resource. For example:

```
kubectl describe postgresrestore restore-test
```

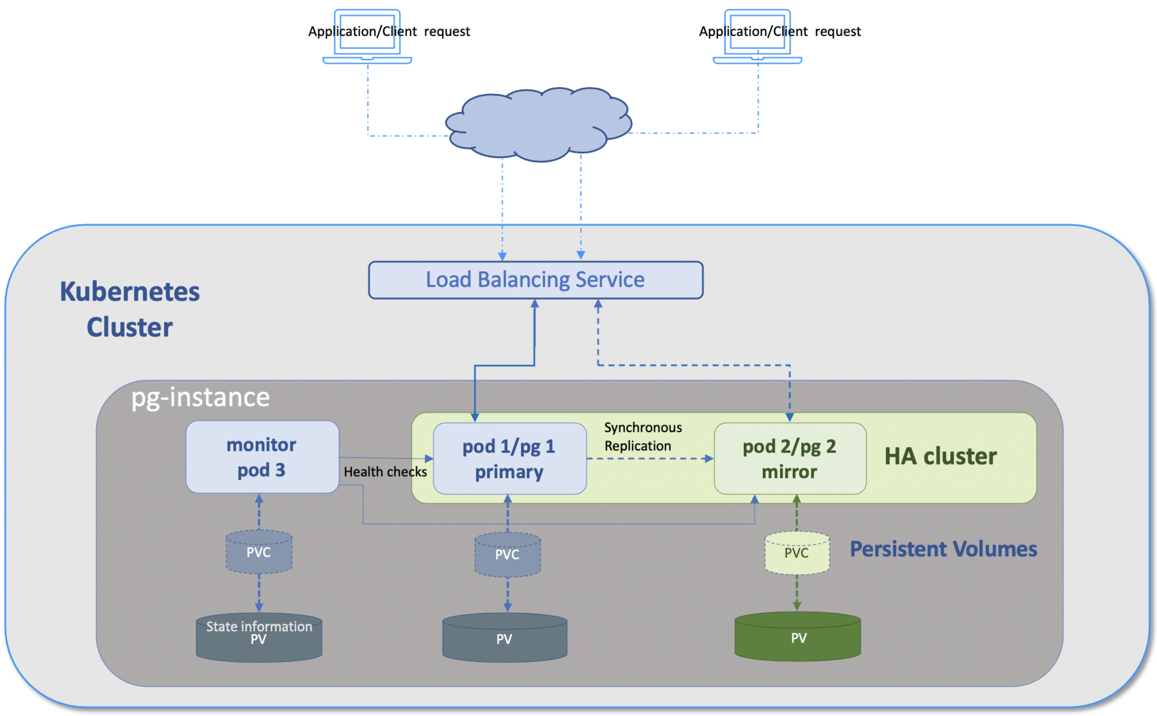
```
Spec:
  Source Backup:
    Name:  sample-source-backup
  Target Instance:
    Name:  my-postgres
Status:
  Message:         Backup sample-source-backup does not exist in namespace default
  Phase:           Failed
  Time Completed:  2022-02-25T19:27:27Z
  Time Started:    2022-02-25T19:27:27Z
Events:
  Type     Reason                        Age    From                         Message
  ----     ------                        ----   ----                         -------
  Warning  ReferencedBackupDoesNotExist  2m56s  postgres-restore-controller  PostgresB
ackup.sql.tanzu.vmware.com "sample-source-backup" not found
```

In the example above, the restore failed because the backup specified in the sourceInstance field does not exist.

Configuring High Availability in Tanzu Postgres

This topic describes how to enable High Availability (HA) for Tanzu Postgres. HA offers automatic failover ensuring that any application requests operate continuously and without downtime.

Tanzu Postgres uses the pg_auto_failover extension to provide a highly available Tanzu Postgres cluster on Kubernetes. For detailed information about pg_auto_failover features, see the pg_auto_failover documentation.



In the Tanzu Postgres HA cluster configuration, the topology consists of three pods: one monitor, one primary and one hot standby mirror. pg_auto_failover ensures that the data is synchronously replicated from the primary to the mirror node. If the primary node is unresponsive, the application requests are re-directed to the mirror node, which gets promoted to the primary. All application requests continue to the promoted primary, while a new postgres instance is started which becomes the new mirror. If the monitor pod fails, operations continue as normal. The Postgres operator redeploys a monitor pod, and when ready it resumes monitoring of the primary and secondary.

Configuring High Availability

Ensure that you have completed the Installing a Postgres Operator procedures before proceeding. Also review Deploying a New Postgres Instance.

1. To enable Tanzu Postgres high availability (cluster mode), edit your copy of the instance `yaml` file you created during Deploying a New Postgres Instance. In the `yaml`, alter the `highAvailability` field:

   ```
   apiVersion: sql.tanzu.vmware.com/v1
   kind: Postgres
   metadata:
     name: postgres-sample
   spec:
     memory: 800Mi
     cpu: "0.8"
     storageClassName: standard
     storageSize: 10G
     serviceType: CluterIP
     pgConfig:
      dbname:
      username:
     highAvailability:
      enabled: true
   ```

   `highAvailability` values can be `enabled: <true|false>`. If this field is left empty, the postgres instance is by default a single node configuration.

2. Execute this command to deploy or redeploy the cluster with the new `highAvailability` setting:

   ```
   kubectl apply -f postgres.yaml
   ```

   where `postgres.yaml`is the Kubernetes manifest created for this instance.

   The command output is similar to:

   ```
   postgres.sql.tanzu.vmware.com/postgres-sample created
   ```

   where *postgres-sample* is the Postgres instance `name` defined in the `yaml` file.

   At this point, the Postgres operator deploys the three Postgres instance pods: the monitor, the primary, and the mirror.

Verifying the HA Configuration

To confirm your HA configuration is ready for access, use `kubectl get` to review the STATUS field and confirm that it shows "Running". Initially STATUS will show Created, until all artifacts are deployed. Use Ctr-C to escape the watch command.

```
watch kubectl get postgres/postgres-sample
```

```
NAME               STATUS    BACKUP LOCATION          AGE
postgres-sample    Pending   backuplocation-sample    17m
```

To view the created pods, use:

```
kubectl get pods
```

```
NAME                                    READY   STATUS      RESTARTS      AGE
pod/postgres-sample-0                   1/1     Running        0          11m
pod/postgres-sample-monitor             1/1     Running        0          12m
pod/postgres-sample-1                   1/1     Running        0          4m28s
```

You can now log into the primary pod using `kubectl exec -it <pod-name> -- bash`:

```
kubectl exec -it postgres-sample-0 -- bash
```

You can log into any pod with `kubectl exec` and use the `pg_autoctl` tool to inspect the state of the cluster. Run `pg_autoctl show state` to see which pod is currently the primary:

```
kubectl exec -ti pod/postgres-sample-1 -- pg_autoctl show state


 Name |  Node |                                                          Host:Port
|       TLI: LSN |   Connection |       Current State |      Assigned State
-------+-------+------------------------------------------------------------------
--+----------------+--------------+---------------------+--------------------
node_1 |      1 | postgres-sample-0.postgres-sample-agent.default.svc.cluster.local:543
2 |    2: 0/3002690 |     read-only |             secondary |             secondary
node_2 |      2 | postgres-sample-1.postgres-sample-agent.default.svc.cluster.local:543
2 |    2: 0/3002690 |    read-write |               primary |               primary
```

The `pg_autoctl` set of commands manage the pg_autofailover services. For further information, refer to the pg_autoctl reference documentation.

Note: VMware supports a limited range of `pg_autoctl` commands, involving inspecting the nodes and performing a manual failover.

If the primary is unreachable, during the primary to mirror failover, the `Current State` and `Assigned State` status columns toggle between demoted, catching_up, wait_primary, secondary, and primary. You can monitor the states using `pg_autoctl`:

```
watch pg_autoctl show state
```

```
Name |  Node |                                                           Host:Port
|       TLI: LSN |   Connection |       Current State |      Assigned State
-------+-------+------------------------------------------------------------------
--+----------------+--------------+---------------------+--------------------
node_1 |      1 | postgres-sample-0.postgres-sample-agent.default.svc.cluster.local:543
2 |    2: 0/3002690 |     read-only |               demoted |          catching_up
node_2 |      2 | postgres-sample-1.postgres-sample-agent.default.svc.cluster.local:543
2 |    2: 0/3002690 |    read-write |               primary |               primary
```

```
Name |  Node |                                                           Host:Port
|       TLI: LSN |   Connection |       Current State |      Assigned State
-------+-------+------------------------------------------------------------------
--+----------------+--------------+---------------------+--------------------
node_1 |      1 | postgres-sample-0.postgres-sample-agent.default.svc.cluster.local:543
2 |    2: 0/3002690 |     read-only |             secondary |             secondary
node_2 |      2 | postgres-sample-1.postgres-sample-agent.default.svc.cluster.local:543
2 |    2: 0/3002690 |    read-write |               primary |               primary
```

Scaling down the HA Configuration

To alter an HA cluster to a single node configuration, alter the Postgres instance YAML and change the HA the field from:

```
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: ha-postgres-sample
spec:
  highAvailability:
     enabled: true
```

to:

```
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: ha-postgres-sample
spec:
  highAvailability:
     enabled: false
```

After this step has been completed the mirror no longer exists. You may verify that there is no replication happening by using accessing the pod:

```
kubectl exec -it  pod/ha-postgres-sample -- bash
```

and running:

```
pg_autoctl show state
```

where the output should show `single` under the column `Current State`.

Monitoring Postgres Instances

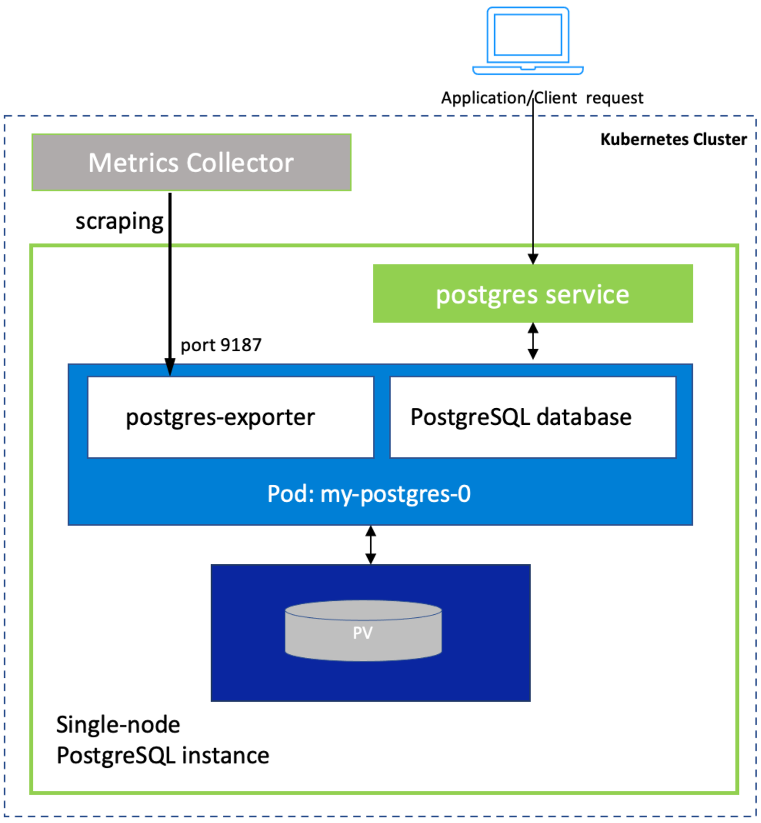
This topic describes how to collect metrics and monitor Tanzu Postgres instances in a Kubernetes cluster.

Overview

Tanzu Postgres uses the Postgres Exporter, a Prometheus exporter for Postgres server metrics. The Prometheus exporter provides an endpoint for Prometheus to scrape metrics from different application services. The Postgres Server Exporter shares metrics about the Postgres instances.

Upon initialization, each Postgres pod adds a Postgres server exporter container. Prometheus sends HTTPS requests to the exporter. The exporter queries the Postgres database and provides metrics in the Prometheus format on a `/metrics` https endpoint (port 9187) on the pod, conforming to the Prometheus HTTP API.

The diagram below shows the architecture of a single-node Postgres instance with Postgres server exporter, where the metrics are exported on port 9187:



Click here to view a larger version of this diagram

Prometheus could be your primary consumer of the metrics, but any monitoring tool can take advantage of the `/metrics` endpoint.

Prerequisites

To take advantage of the metrics endpoint, ensure your environments has a metrics collector like Prometheus, or Wavefront. For an example installation of Prometheus, see Using Prometheus Operator to Scrape the Tanzu Postgres Metrics.

Verifying Postgres Metrics

The Tanzu Postgres pods include the exporter that emits the built-in Postgres metrics. To test that the metrics are being emitted, you may use port forwarding (for more details see Use Port Forwarding to Access Applications in a Cluster in the Kubernetes documentation):

```
kubectl port-forward pod/<postgres-instance-pod-name> 9187:9187
```

And then in another shell window, use a tool like `curl` to run:

```
curl -k https://localhost:9187/metrics
```

A successful output would show metrics emitted by the exporter, similar to (this example is a small extract):

```
# HELP pg_stat_database_xact_rollback Number of transactions in this database that hav
e been rolled back
# TYPE pg_stat_database_xact_rollback counter
pg_stat_database_xact_rollback{datid="13737",datname="postgres",server="localhost:543
2"} 3553
```

where `xact_rollback` is part of the `pg_stat_database` metrics map, as described in Postgres Exporter. For a list of the Postgres relations that are used to retrieve the default Tanzu Postgres metrics see Postgres Exporter Default Metrics.

Using Prometheus Operator to Scrape the Tanzu Postgres Metrics

This section demonstrates how to scrape the Postgres metrics using the Prometheus Operator.

The Prometheus Operator defines and manages monitoring instances as Kubernetes resources. This section provides an example installation of the Prometheus Operator, and an example Prometheus `PodMonitor` CRD (Custom Resource Definition) that will be used demonstrate how to scrape the metrics. The `PodMonitor` defines the configuration details for the Tanzu Postgres pod monitoring.

1. Install the Prometheus Operator using Helm:

   For e.g.:

   ```
   helm install prometheus prometheus-community/kube-prometheus-stack \
      --create-namespace \
   ```

```
   --namespace=prometheus \
   --set prometheus.service.port=80 \
   --set prometheus.service.type=LoadBalancer \
   --set grafana.enabled=false,alertmanager.enabled=false,nodeExporter.enabled=
false \
   --set prometheus.prometheusSpec.podMonitorSelectorNilUsesHelmValues=false \
   --wait
```

Note: In the above prometheus installation example, `prometheus.prometheusSpec.podMonitorSelectorNilUsesHelmValues` is set to `false`. It is configured to avoid the need to add a `release` label on each PodMonitor CR that you create.

If you already have a prometheus installation with `prometheus.prometheusSpec.podMonitorSelectorNilUsesHelmValues` set to `true` (default value), then, in order for your PodMonitor CRs to be discoverable by the Prometheus Operator, you will need to add a label `release: <prometheus-release-name>` to the metadata section of each PodMonitor CR that you create in the cluster. For e.g. if the helm release name is `prometheus`, you will add a label `release: prometheus` in the PodMonitor's `metadata` section.

2. Confirm the `PodMonitor` CRD exists using:

```
kubectl get customresourcedefinitions.apiextensions.k8s.io podmonitors.monitori
ng.coreos.com
```

3. Create a `PodMonitor` that scrapes all Tanzu Postgres instances every 10 seconds:

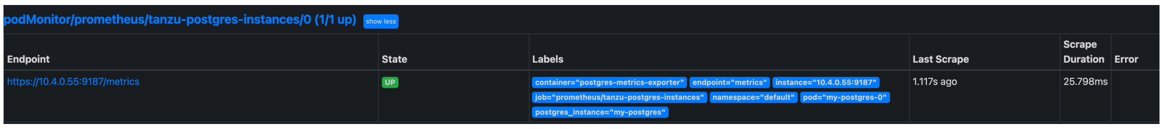
```
cat <<EOF | kubectl apply -f -
---
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: tanzu-postgres-instances
  namespace: prometheus
spec:
  namespaceSelector:
    any: true
  selector:
    matchLabels:
      type: data
      app: postgres
  podTargetLabels:
  - postgres-instance
  podMetricsEndpoints:
  - port: "metrics"
    interval: "10s"
    scheme: https
    tlsConfig:
      insecureSkipVerify: true
EOF
```

where `tlsConfig:insecureSkipVerify:true` skips TLS verification.

4. Check if Prometheus is successfully monitoring the instances by opening the Prometheus UI in the browser and visit the `/targets` URI to check the status of the `PodMonitor` under

podMonitor/prometheus/tanzu-postgres-instances/0 (1/1 up).

You should see something similar to:



podMonitor/prometheus/tanzu-postgres-instances/0 (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
https://10.4.0.55:9187/metrics	UP	container="postgres-metrics-exporter" endpoint="metrics" instance="10.4.0.55:9187" job="prometheus/tanzu-postgres-instances" namespace="default" pod="my-postgres-0" postgres_instance="my-postgres"	1.117s ago	25.798ms	

Click here to view a larger version of this diagram

For details on the PodMonitor API see PodMonitor in the Prometheus Operator documentation.

Using TLS for the Metrics Endpoint

The Tanzu Postgres Operator creates a metrics related TLS certificate during the Tanzu Postgres initialization. The TLS credentials are stored in a Secret named after the Postgres instance name: if the instance name is postgres-sample, the metrics Secret name is postgres-sample-metrics-tls-secret.

In order to use TLS for a metrics endpoint, configure Prometheus to use the CA certificate from the Secret. Use the following command to fetch the CA certificate:

```
kubectl get secret <POSTGRES-INSTANCE-NAME>-metrics-tls-secret -o 'go-template={{index
.data "tls.crt" | base64decode}}'
```

where <POSTGRES-INSTANCE-NAME> is the name of the Postgres instance.

For details on how to configure Prometheus with TLS, see tls_config in the Prometheus Configuration documentation.

To enable the Prometheus Operator to scrape metrics from the Postgres instance with TLS verification enabled, create a PodMonitor in the Postgres instance namespace and provide the metrics TLS configuration. An example PodMonitor for a Postgres instance named as postgres-sample is shown below:

```
cat <<EOF | kubectl apply -f -
---
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: postgres-sample
  namespace: postgres-sample-namespace
spec:
  namespaceSelector:
    matchNames:
      - postgres-sample-namespace
  selector:
    matchLabels:
      type: data
      app: postgres
      postgres-instance: postgres-sample
  podTargetLabels:
```

```
  - postgres-instance
  podMetricsEndpoints:
  - port: "metrics"
    interval: "10s"
    scheme: https
    tlsConfig:
      serverName: "postgres-sample.metrics.default"
      ca:
        secret:
          key: tls.crt
          name: postgres-sample-metrics-tls-secret
EOF
```

where you should replace `postgres-sample` and `postgres-sample-namespace` with your Postgres instance name and namespace.

Collecting Metrics in a Secure Namespace

Prometheus will not be able to scrape metrics from Postgres instances that have a strict NetworkPolicy configuration on their namespace. See Network Policy Configuration for a detailed explanation about NetworkPolicy configuration.

Create a NetworkPolicy manifest in the instance namespace to allow traffic flow from the metrics port `9187`:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-metrics-access
  namespace: <INSTANCE_NAMESPACE>
spec:
  podSelector:
    matchLabels:
      app: postgres
      type: data
  policyTypes:
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              networking/namespace: <PROMETHEUS_NAMESPACE>
    - ports:
        - port: 9187
          protocol: TCP
```

Label the Prometheus namespace to easily use the `namespaceSelector` section of the NetworkPolicy spec, for example:

```
kubectl label namespace prometheus networking/namespace=prometheus
namespace/prometheus labeled
```

Save the sample yaml to a file, and apply to the cluster:

```
kubectl apply -n INSTANCE-NAMESPACE -f metrics-network-policy-sample.yaml
```

```
networkpolicy.networking.k8s.io/allow-metrics-access created
```

Postgres Exporter Default Metrics

The built-in metrics that are supported by the Postgres Exporter are listed below:

- pg_stat_bgwriter
- pg_stat_database
- pg_stat_database_conflicts
- pg_locks
- pg_stat_replication
- pg_replication_slots
- pg_stat_archiver
- pg_stat_activity
- pg_settings

For more details, see Postgres Exporter Metrics Map.

Troubleshooting Common Problems

This topic provides information that can help troubleshoot problems you may encounter using Postgres for Kubernetes.

Monitor Deployment Progress

Use watch `kubectl get all` to monitor the progress of the Postgres operator deployment. The deployment is complete when the postgres operator pod is in the `Running` state. For example:

```
watch kubectl get all
```

```
NAME                                              READY       STATUS       RESTARTS
AGE
pod/postgres-operator-567dbc67b9-nrq5t            1/1         Running      0
57s
NAME                                              TYPE        CLUSTER-IP   EXTERNAL-IP
PORT(S)    AGE
service/kubernetes                                ClusterIP   10.96.0.1    <none>
443/TCP    2d4h
NAME                                              READY       UP-TO-DATE   AVAILABLE
AGE
deployment.apps/postgres-operator                 1/1         1            1
57s
NAME                                              DESIRED     CURRENT      READY
AGE
replicaset.apps/postgres-operator-567dbc67b9      1           1            1
57s
```

Viewing Postgres Operator Logs

Check the logs of the operator to ensure that it is running properly.

```
kubectl logs -l app=postgres-operator
```

```
2019-08-05T17:24:16.182Z        INFO    controller-runtime.controller   Starting Event
Source{"controller": "postgres", "source": "kind source: /, Kind="}
2019-08-05T17:24:16.182Z        INFO    setup   starting manager
2019-08-05T17:24:16.285Z        INFO    controller-runtime.controller   Starting Contr
oller   {"controller": "postgres"}
2019-08-05T17:24:16.386Z        INFO    controller-runtime.controller
Starting workers        {"controller": "postgres", "worker count": 1}
```

List All Postgres Instances in the Cluster

When you create Postgres instances, each instance is created in its own namespace. To see all Postgres instances in the cluster, add the `-all-namespaces` option to the `kubectl get` command.

```
kubectl get postgres --all-namespaces
```

```
NAMESPACE   NAME               STATUS    AGE
default     postgres-sample    Running   19d
default     postgres-sample2   Running   15d
test        my-postgres        Failed    15d
test        my-postgres3       Failed    15d
```

Find the Versions of the Deployed Postgres Operator and Instances

To find the currently deployed version of the Postgres operator, use the `helm` command:

```
helm ls
```

```
NAME                    NAMESPACE       REVISION        UPDATED
STATUS          CHART                           APP VERSION
postgres-operator       default         1               2021-10-11 13:26:00.769535 -05
00 CDT  deployed        postgres-operator-v1.3.0         v1.3.0
```

The version is in the chart name and the `APP VERSION` column.

To find the version of a Postgres instance, use the `kubectl` command to describe the instance's pod.

```
kubectl get pods
```

```
kubectl get pods
NAME                                   READY   STATUS    RESTARTS   AGE
postgres-sample-0                      1/1     Running   0          9s
postgres-operator-85f777b9db-wbj9b     1/1     Running   0          4m15s
```

```
```

```
Name:             postgres-sample-0
Namespace:        default
Priority:         0
Node:             minikube/192.168.64.32
Start Time:       Mon, 11 Oct 2021 14:10:38 -0500
Labels:           app=postgres
                  controller-revision-hash=postgres-sample-5fc8fb8b4b
                  headless-service=postgres-sample
                  postgres-instance=postgres-sample
                  role=read
                  statefulset.kubernetes.io/pod-name=postgres-sample-0
                  type=data
Annotations:      <none>
Status:           Running
IP:               172.17.0.8
Controlled By:    StatefulSet/my-postgres
```

```
Containers:
  pg-container:
    Container ID:  docker://6c651d690a6fdb6d1c0d3644ad8225037d31da1c33fd3f88f1625bdfd4
5cea3a
    Image:         postgres-instance:v1.3.0
    Image ID:      docker://sha256:00359ca344dd96eb05f2bd430430c97a6d46a40996c395fca44
c209cb954a6e7
    Port:          5432/TCP
    Host Port:     0/TCP
```

The Tanzu Postgres version can be found in the image name of the `pg-container` entry.

Cannot Reduce Instance Data Size After Deployment

When deploying an instance using a specific storage size in the instance `yaml` deployment file, you cannot reduce the instance data storage size at a later stage. For example, after creating an instance and setting the storage size to 100M:

```
kubectl create -f postgres.yaml
```

Verify the storage size using a command similar to:

```
kubectl get postgres.sql.tanzu.vmware.com/postgres-sample -o jsonpath='{.spec.storageS
ize}'
```

```
100M
```

If you later patch the instance to decrease the storage size from 100M to 2M:

```
kubectl patch postgres.sql.tanzu.vmware.com/postgres-sample --type merge -p '{"spec":
{"storageSize": "2M"}}'
```

the operation returns an error similar to:

```
Error from server (spec.storageSize: Invalid Value: "2M" spec.storageSize cannot be re
duced for an existing instance
spec.storageSize: Invalid Value: "2M" spec.storageSize needs to be at least 250M): adm
ission webhook "vpostgres.kb.io" denied the request: spec.storageSize: Invalid Value:
"2M" spec.storageSize cannot be reduced for an existing instance
spec.storageSize: Invalid Value: "2M" spec.storageSize needs to be at least 250M
```

To reduce the instance data size, create a new instance and migrate the source data over. Ensure that the source data fits in the reduced data size allocation of the newly created instance.

Errors during Backup of two Different Instances on the Same Bucket

This scenario occurs when you have two separate Kubernetes clusters with matching instance and namespace names. This scenario requires the following conditions:

- Each cluster has a matching namespace name; for example cluster 1 has a namespace called `my-namespace`, and cluster 2 has a namespace called `my-namespace`.

- Each cluster has a Postgres instance with the same name, for example `my-instance`.
- Both clusters share the same S3 bucket for backups.

During backup, the first Postgres instance creates a backup stanza using the format `my-instance-my-namespace`. That stanza is encrypted with a randomly-generated backup cipher. During backup configuration for the second instance, the instance detects that a backup stanza with the same name already exists in the bucket. However, the second instance cannot decrypt the backup information because it uses a different cipher. The error is similar to: :

```
ERROR: [043]: WAL segment to get required 2021-09-02 15:55:35.615 P00 INFO: archive-ge
t command end: aborted with exception [043] command terminated with exit code 43 or Fo
rmatError: key/value found outside of section at line 1: ▒▒▒H▒t=Ő@▒Y▒.
```

Workaround: Use different instance names, or different namespace names, or different buckets for backups.

Postgres CRD API Reference

This topic describes the available fields of the Postgres Custom Resource Definition.

Synopsis

```
apiVersion: sql.tanzu.vmware.com/v1
kind: Postgres
metadata:
  name: postgres-sample
spec:
  #
  # Global features
  #
  imagePullSecret:
    name: regsecret
  pgConfig:
    dbname: postgres-sample
    username: pgadmin
    appUser: pgappuser
  postgresVersion:
    name: postgres-14 # View available versions with `kubectl get postgresversion`
  serviceType: ClusterIP
# serviceAnnotations:
  seccompProfile:
     type: RuntimeDefault
  #  highAvailability:
  #    enabled: true
  #  logLevel: Debug
  #  backupLocation:
  #    name: backuplocation-sample
  #  certificateSecretName:

#
# Data Pod features
#
storageClassName: standard
storageSize: 800M
cpu: "0.8"
memory: 800Mi
dataPodConfig:
#    tolerations:
#      - key:
#        operator:
#        value:
#        effect:
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: type
                operator: In
                values:
                  - data
                  - monitor
              - key: postgres-instance
                operator: In
                values:
                  - postgres-sample
          topologyKey: kubernetes.io/hostname
          weight: 100
 #
 # Monitor Pod features
 #
 monitorStorageClassName: standard
 monitorStorageSize: 1G
 monitorPodConfig:
 #    tolerations:
 #      - key:
 #        operator:
 #        value:
 #        effect:
 affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
             matchExpressions:
               - key: type
                 operator: In
                 values:
                   - data
                   - monitor
               - key: postgres-instance
                 operator: In
                 values:
                  - postgres-sample
        topologyKey: kubernetes.io/hostname
    weight: 100

#
# Resources
#
resources:
  monitor:
    limits:
    cpu: 800m
    memory: 800Mi
  requests:
    cpu: 800m
    memory: 800Mi
  metrics:
     limits:
       cpu: 100m
       memory: 100Mi
  requests:
       cpu: 100m
       memory: 100Mi
```

Description

Applying this resource causes the Kubernetes Operator to create a StatefulSet with a single Pod and three containers. One container runs the Postgres database software, one runs the components to support backups, and the third runs the postgresd_exporter (for monitoring). The Postgres Pod mounts a persistent volume claim (PVC) which holds the Postgres data.

You specify Postgres instance configuration properties to the Postgres operator with a YAML-formatted manifest file. A sample manifest file is provided with the release, in `postgres.yaml`. See also Deploying a New Postgres Instance for information about deploying a new Postgres instance using a manifest file.

Metadata

The metadata follows standard Kubernetes conventions. For more details, refer to Metadata in the Kubernetes documentation.

The metadata sets the name, namespace, labels, annotations, and more for the Postgres object.

name

Type: string
Required
Default: n/a
Sets the name of the Postgres instance. The Kubernetes operator will append an index like `-0` or `-1` to the end of the name when it creates the pods, for example `postgres-sample-0`. By default, a newly created Postgres object will include a default database with the same name as the object unless you change the default database name with `pgConfig.dbname`.

Spec

The spec describes the desired state for the Postgres object.

imagePullSecret

Type: Object
Optional
Default: name: regsecret
The secret value defaults to the `dockerRegistrySecretName` in the Operator's values.yaml. If your namespace's docker-registry secret uses a different secret than the Operator's helm chart secret, alter the secret name accordingly.
Note: Recreate the secret within your pod's namespace in order to be accessed by the pod.
An existing Kubernetes docker-registry secret that can access the registry containing the Postgres image.

pgConfig

Type: Object
Optional
This collection of fields describes the Postgres database table and user that is created at database initialization. See Custom Database Name and User Account for more information.

- dbname
  Type: string
  Optional
  Default: Postgres metadata name
  The name of the default Postgres database. By default, the Postgres instance name is used as the default database name. See Custom Database Name and User Account.
- username
  Type: string
  Optional
  Default: pgadmin
  The name of the default Postgres user. See Custom Database Name and User Account.
- appUser
  Type: string
  Optional
  Default: pgappuser
  Specifies the name of the Postgres user with read-write privileges. It will be used to bind your application with the Postgres instance. See Creating Service Bindings

postgresVersion

Type: string
Optional
Default: <latest_version>
This string must be a reference to an existing `PostgresVersion` object. If omitted, the most up-to-date PostgresVersion is chosen (e.g. postgres-14). For more information see Specifying the Tanzu Postgres Version.

serviceType

Type: string
Optional
Default: ClusterIP
The Kubernetes publishing service used for the Postgres instance. Options are `LoadBalancer` or `ClusterIP`. The default `ClusterIP` exposes the Postgres service internally and uses cluster-internal IP address instead of a load balancer. See Publishing Services (ServiceTypes) in the Kubernetes documentation for more information.

serviceAnnotations

Type: (map[string]string)
Optional
Default: n/a
Used mostly for instances with `serviceType` set to `LoadBalancer`, where the instances are deployed

in public clouds, and require cloud-specific behavior. Can also be used to set custom annotations.
Example:

```
spec:
  serviceType: LoadBalancer
  serviceAnnotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true",
    service.beta.kubernetes.io/azure-load-balancer-internal-subnet: "apps-subnet"
    cloud.google.com/load-balancer-type: "Internal"
    service.beta.kubernetes.io/aws-load-balancer-internal: "true"
```

For more information, see Internal Load Balancer, and Annotations in the Kubernetes documentation.

seccompProfile

Type: corev1.SeccompProfile
Optional
Default: RuntimeDefault
Enables the use of Secure Compute Mode (seccomp) profiles for the instances. The default profile `RuntimeDefault` is the most restrictive, with a strong set of security defaults for container syscalls. Set to `Unconfined` to disable seccomp profiles. Set to `Localhost` to indicates the path of a pre-configured profile on the node. For more details, see Restrict a Container's Syscalls with seccomp in the Kubernetes documentation.
For further information on the `corev1.SeccompProfile` type, see SeccompProfile v1 core.

highAvailability

Type: Object
Optional
Default: enabled: false
Specifies whether the Postgres instance is created in a single or cluster mode configuration. The default, `false`, creates a single node cluster. See Configuring High Availability for more information about clustered Postgres deployments.

```
highAvailability:
  enabled: true
```

loglevel

Type: string
Optional
Default: n/a
Sets the level of information detail displayed in the logs. By default this field is not in the instance `yaml`, and the log level is non-verbose. Set to `Debug` for verbose logs.

backupLocation

Type: LocalObjectReference
Optional
Default: n/a

When using a S3-compatible storage location for backups, this value specifies the PostgresBackupLocation CRD holding the configuration for the S3 backup location. For more details on configurating backup and restore, see Backing Up and Restoring.

```
backupLocation:
  name: "custom-s3-location"
```

certificateSecretName

Type: string
Optional
Default: n/a
When using TLS security, this value specifies the name of a secret created to enable TLS connections in the Postgres cluster. See Configuring TLS for Tanzu Postgres Instances.

storageClassName

Type: string
Optional
Default: standard
The Storage Class name to use for dynamically provisioning Persistent Volumes (PVs) for a Postgres instance pod. If the PVs already exist, either from a previous deployment or because you manually provisioned the PVs, then the Operator uses the existing PVs. You can configure the Storage Class according to your performance needs. To understand the different configuration options see Storage Classes in the Kubernetes documentation.
IMPORTANT: Edit the default `standard` storage class with your storage class name.

storageSize

Type: Quantity
Optional
Default: 800M
The storage size of the Persistent Volume Claim (PVC) for a Postgres instance pod. Specify a suffix for the units (for example: `100G`, `1T`).

cpu

Type: Quantity
Optional
Default: 0.8
The amount of CPU resources allocated to a Postgres instance pod, specified as a Kubernetes CPU unit (for example, `cpu: "1.2"`). If left empty, the pod has no upper bound on the CPU resource it can use or inherits the default limit if one is specified in its deployed namespace. See Assign CPU Resources to Containers and Pods in the Kubernetes documentation for more information.

memory

Type: Quantity
Optional
Default: 800Mi

The amount of memory allocated to a Postgres instance pod. This value defines a memory limit; if a pod tries to exceed the limit it is removed and replaced by a new pod. You can specify a suffix to define the memory units (for example, `4.5Gi`.). If left empty, the default for the Postgres instance is 800 mebibytes, or about 800 megabytes. See Assign Memory Resources to Containers and Pods in the Kubernetes documentation for more information.

dataPodConfig

Type: Object
Optional

- tolerations
  Type: array of corev1.Toleration
  Optional
  Default: []
  Defines the data pod tolerations that match specific node taints, using `corev1.Toleration` notation. The default is no tolerations. For details on the Toleration values, see Toleration v1 core in the Kubernetes API documentation.

  Example:
  To ensure data pods are scheduled on less optimized "admin" nodes, first create the taint on a node. This command adds the label `nodetype=admin` and the effect `NoSchedule` to the `my-admin-node`:

  ```
  kubectl taint nodes my-admin-node nodetype=admin:NoSchedule
  ```

  ```
  node/my-admin-node tainted
  ```

  Node `my-admin-node` now repels all pods that do not have the toleration `nodetype=admin`. Now add the toleration of the taint to the data pod by editing the instance CRD:

  ```
  ......
  dataPodConfig:
    tolerations:
      - key: nodetype
        operator: Equal
        value: admin
        effect: NoSchedule
  .....
  ```

  Note that a matching toleration gives permission for the scheduling of pod to tainted nodes, but does not guarantee it. Kubernetes uses node affinity to actually determine where to schedule the Pods.

  For further details, see Taints and Tolerations in the Kubernetes documenation.

- affinity
  Type: corev1.Affinity
  Optional
  Default: podAntiAffinity object with preferred scheduling. See above sample YAML
  Defines the data pod anti-affinity rules, using `corev1.Affinity` notation. By default the pods

of a single Postgres instance prefer to be scheduled on separate Kubernetes nodes. For details on the affinity values, see Affinity v1 core in the Kubernetes API documentation.

Example:
To ensure data pods are schedule on separate zones, we can set podAntiAffinity and require that the Kubernetes scheduler follow the configuration as specified below.

```
......
dataPodConfig:
  podAntiAffinity:
     requiredDuringSchedulingIgnoredDuringExecution:
     - labelSelector:
         matchExpressions:
         - key: type
           operator: In
           values:
           - data
       topologyKey: "failure-domain.beta.kubernetes.io/zone"
.....
```

monitorStorageClassName

Type: string
Optional
Default: standard
The Storage Class name to use for dynamically provisioning Persistent Volumes (PVs) for the Postgres monitor pod. By default it is set to `standard`. The default value can be changed at the time of the Postgres instance initialization but the `monitorStorageClassName` must match the Postgres instance `storageClassName`.

monitorStorageSize

Type: Quantity
Optional
Default: 1G
The storage size of the Persistent Volume Claim (PVC) for a Postgres instance monitor pod. Specify a suffix for the units (for example: `100G`, `1T`). The default value is 1G.

monitorPodConfig

Type: Object
Optional

- tolerations
  Type: array of corev1.Toleration
  Optional
  Default: []
  Defines the monitor pod tolerations that match specific node taints, using `corev1.Toleration` notation. The default is no tolerations. For details on the Toleration values, see Toleration v1 core in the Kubernetes API documentation.

Example:
To ensure monitor pods are scheduled on less optimized "admin" nodes, first create the taint on a node. This command adds the label `nodetype=admin` and the effect `NoSchedule` to the `my-admin-node`:

```
kubectl taint nodes my-admin-node nodetype=admin:NoSchedule
```

```
node/my-admin-node tainted
```

Node `my-admin-node` now repels all pods that do not have the toleration `nodetype=admin`. Now add the toleration of the taint to the monitor pod by editing the instance CRD:

```
......
monitorPodConfig:
  tolerations:
    - key: nodetype
      operator: Equal
      value: admin
      effect: NoSchedule
.....
```

Note that a matching toleration gives permission for the scheduling of pod to tainted nodes, but does not guarantee it. Kubernetes uses node affinity to actually determine where to schedule the Pods.

For further details, see Taints and Tolerations in the Kubernetes documenation.

- affinity
  Type: corev1.Affinity
  Optional
  Default: podAntiAffinity object with preferred scheduling. See above sample YAML
  Defines the monitor pod anti-affinity rules, using `corev1.Affinity` notation. By default the pods of a single Postgres instance will prefer to be scheduled on separate Kubernetes nodes. For details on the affinity values, see Affinity v1 core in the Kubernetes API documentation.

resources

Type: Object
Optional
Defaults: monitor.limits.cpu: 0.8, monitor.limits.memory: 800Mi
Defaults: metrics.limits.cpu: 0.1, metrics.limits.memory: 100Mi
This object is a mapping of strings to `ResourceRequirements`. The supported keys are `monitor` and `metrics`, which are containers in the data and monitor pod respectively.
A `ResourceRequirements` object describes the compute resource requirements (requests and limits of cpu and memory).

```
monitor:
  limits:
    cpu: 1
    memory: 800Mi
  requests:
    cpu: 0.8
    memory: 400Mi
metrics:
  limits:
    cpu: 0.8
    memory: 500Mi
  requests:
    cpu: 0.2
    memory: 100Mi
```

Status

The status fields show the most recently observed status of the Postgres object. This information is generated by the Kubernetes operator as it reconciles the object in the cluster.

currentPgbackrestConfigResourceVersion

Type: string
This field reflects the revision number of the associated Kubernetes secret holding pgbackrest configuration.

currentState

Type: string
This field shows the status of the Postgres object. Possible values are `Created`, `Pending`, and `Running`.

binding

Type: LocalObjectReference
This field shows the name of the Secret for service bindings. For more information, see service binding spec.

dbVersion

Type: string
This field shows the major and minor version of the Postgres database used for this instance.

Backup and Restore CRD API Reference

The Tanzu Postgres Backup and Restore uses four CRDs. Refer to each property reference page for details on the resource fields for each CRD.

PostgresBackupLocation Resource
PostgresBackupSchedule Resource
PostgresBackup Resource
PostgresRestore Resource

For more information relating to Tanzu Postgres backup and restore, see Backing Up and Restoring.

Backup and Restore CRD API Reference - PostgresBackupLocation Resource

PostgresBackupLocation Synopsis

```
apiVersion: sql.tanzu.vmware.com/v1
kind: PostgresBackupLocation
metadata:
  name: backuplocation-sample
spec:
  retentionPolicy:
    fullRetention:
      type: count
      number: 9999999
    diffRetention:
      number: 9999999
  storage:
    s3:
      bucket: "name-of-bucket"
      bucketPath: "/my-bucket-path"
      region: "us-east-1"
      endpoint: "custom-endpoint"
      forcePathStyle: false
      enableSSL: true
      secret:
        name: backuplocation-creds-sample
  additionalParameters: {}
```

The list below explains the properties that can be set for the PostgresBackupLocation resource.

Metadata

The metadata sets the name, namespace, labels, annotations, and more for the PostgresBackupLocation object.
The metadata follows standard Kubernetes conventions. See more at the Kubernetes API structure - Metadata documentation.

name

Type: String
Required
Default: n/a
The name of the PostgresBackupLocation. Must be unique within a namespace.
Example:
`backuplocation-sample`

Spec

The spec describes the desired state for the Postgres object.

retentionPolicy

Type: Object
Optional
Default: n/a
This collection of fields describes the Postgres database backup retention plans. For more details, see the topic Retention in the pgBackRest User Guide.

- fullRetention
  Type: Object
  Optional
  Default: 9999999
  This field describes the retention period of the full backups for this instance.
  The object `fullRetention` has two fields, `type` and `number`:
  `type` is a string of either `count` or `time`; default is `count` with value `9999999` (the maximum value allowed by pgbackrest).
  `number` is an integer.
  If `type: time` then `number` indicates the number of days backups are retained before expiring.
  If `type: count` then it indicates the number of backups that are retained.
  A `fullRetention` value of 1 retains one full backup; older backups will be deleted when a new backup is taken.
  NOTE: Scheduled and adhoc backups affect the retention count. Users should be aware of retention count when executing adhoc backup operations.
  Example:

  ```
  retentionPolicy:
      fullRetention:
          type: count
          number: 2
  ```

creates a retention policy of 2 full backups before an older third backup can be expired.
Example:

```
retentionPolicy:
    fullRetention:
        type: time
        number: 20
```

creates a retention policy of 20 days before a backup can be expired.

- diffRetention
  Type: Object
  Optional
  Default: 9999999
  This field describes the retention period of the differential backups for this instance.
  The object `diffRetention` has one field, `number`, an integer.
  A `diffRetention` value of 1 retains one differential backup; older backups will be deleted when a new backup is taken.
  The differential backup retention does not support the `type` field. Differential retention does not support deleting backups based on time, only count.
  Example:

  ```
  retentionPolicy:
      diffRetention:
          number: 2
  ```

  creates a retention policy of 2 differential backups before an older third differential backup can be expired.

storage

Type: Object
Optional
Default: n/a
This collection of fields describes the S3 bucket characteristics.

- s3.bucket
  Type: String
  Required
  Default: n/a
  The name of an existing S3-compatible bucket for this backup location. A bucket of this name should already exist in s3.
  Example
  `s3-bucket-sample`
- s3.bucketPath
  Type: String
  Optional
  Default: /
  The name of the path where backup artifacts will be uploaded. If a folder in the path does not already exist, it is created automatically. The trailing slash in the path is required.

Example
`s3-sample-path/sample-subpath/`

- s3.region
  Type: String
  Optional
  Default: us-east-1
  The geographic region of the bucket. Some non-AWS S3 implementations do not require this value.
  Example
  `us-west-1`
- s3.endpoint
  Type: String
  Required
  **Default: **
  The endpoint URL for the configured S3-compatible provider.
  Example
  `http://minio.default:9000`
- s3.forcePathStyle
  Type: Boolean
  Optional
  Default: false
  A value of `true` forces the use of path-style S3 URLs for compatibility. May be required for some non-AWS S3 providers.
  A value of `false` uses virtual hosted-style S3 URLs.
  Path-style URLs look like the following: https://bucket-endpoint.example.com/bucket
  Virtual hosted-style URLs look like the following: https://bucket.bucket-endpoint.example.com For information about AWS S3 Path Deprecation, see the Amazon S3 Path Deprecation Plan blog post.
- s3.enableSSL
  Type: Boolean
  Required
  Default: true
  `true` enables SSL for S3 server validation.
  `false` disables SSL.
- s3.secret.name
  Type: String
  Required
  Default: n/a
  The name of the Kubernetes secret that contains the credentials for connecting to S3.
  Example
  `backuplocation-sample-creds`

additionalParameters

Type: Object
Optional

Default: n/a
Advanced users may pass additional parameters to PgBackrest. Review the various parameters in pgBackRest Configuration Reference and use them without the leading `--` dashes.
Example:

```
buffer-size: "4MB"
archive-timeout: "100"
process-max: "4"
```

Status

The status fields show the observed status of the PostgresBackupLocation object and its values are populated by the Kubernetes Operator

currentSecretResourceVersion

Type: string
This field shows the resource version of the backup secret described below.

Backup Secret Synopsis

```
apiVersion: v1
kind: Secret
metadata:
  name: backuplocation-credentials
type: generic
stringData:
  accessKeyId: "my-access-key-id"
  secretAccessKey: "my-secret-access-key"
```

The list below explains the properties that can be set in the `secret` for the PostgresBackupLocation resource.

metadata

name

Type: String
Required
Default: n/a
The name of the Secret. Must match `spec.storage.s3.secret.name` in a BackupLocation.
Must be unique within a namespace.
Example:
`backuplocation-sample-creds`

stringData

accessKeyID

Type: String
Required
Default: n/a
The Access Key ID for an AWS IAM user that has permissions to read/write from the S3 bucket.
Example:
`AKIAIOSFODNN7EXAMPLE`

secretAccessKey

Type: String
Required
Default: n/a
The Secret Access Key ID for an AWS IAM user that has permissions to read/write from the S3 bucket.
Example:
`wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`

Backup and Restore CRD API Reference - PostgresBackupSchedule Resource

PostgresBackupSchedule Synopsis

```
apiVersion: sql.tanzu.vmware.com/v1
kind: PostgresBackupSchedule
metadata:
  name: backupschedule-sample
spec:
  backupTemplate:
    spec:
      sourceInstance:
        name: postgres-sample
      type: full
  schedule: "0 0 * * *"
```

The list below explains the properties that can be set for the PostgresBackupSchedule resource.

Metadata

name

Type: String
Required
Default: n/a
The name of the PostgresBackupSchedule. Must be unique within a namespace.
Example:
`backupschedule-sample`

Spec

The spec describes the desired state for the Postgres object.

backupTemplate

Type: Object
Optional
Default: n/a
This collection of fields describes the Postgres database backup template.

- spec.sourceInstance.name
  Type: String
  Required
  Default: n/a
  The name of the Postgres instance on which you want scheduled backups for.
  Example

  ```
  postgres-sample
  ```

- spec.sourceInstance.type
  Type: String
  Optional
  Default: n/a
  The type of the Postgres Backup you want to take at a scheduled interval.
  It can be one of three values, full,incremental, or differential.

schedule

Type: String (cron schedule)
Required
Default: n/a
The cron schedule for backups. Must be a valid cron schedule.
Example (every Saturday at 11PM)
`"0 23 * * 6"`

Status

The status fields show the observed status of the PostgresBackupSchedule object

message

Type: String
Optional
Default: n/a
Success/failure status message for PostgresBackupSchedule
Example
`Instance my-2-postgres does not exist in the namespace default`

Backup and Restore CRD API Reference - PostgresBackup Resource

PostgresBackup Synopsis

```
apiVersion: sql.tanzu.vmware.com/v1
kind: PostgresBackup
metadata:
  name: backup-sample
spec:
  sourceInstance:
    name: postgres-sample
  type: full
```

The sections below explain the properties that can be set for the PostgresBackup resource.

Metadata

name

Type: String
Required
Default: n/a
The name of the PostgresBackup. Must be unique within a namespace.
Example:

```
backup-sample
```

Spec

The spec describes the desired state for the Postgres backup object.

sourceInstance

Type: Object
Required
Default: n/a
The fields that describe the Postgres instance backup are `name` and `type`:

- name
  Type: String
  Required
  Default: n/a
  The name of the Postgres instance on which you want to perform the on-demand backup.
  Example

  ```
  my-postgres-sample
  ```

- type
  Type: String
  Optional
  Default: full
  The type of the Postgres Backup you want to take. The possible values full, incremental, or differential.

Example

`incremental`

Status

The status fields show the observed status of the PostgresBackup object and these fields are populated by the controller that processes backups.

phase

Type: string
This field reflects the current state of the PostgresBackup resource. It can be empty or have the following values: Running, Failed, or Succeeded.

restoreLabel

Type: string
This field denotes the backup label. For e.g. 20220306-155026F_20220306-155048F. Backup labels for full backups ends with F, differential ends with D, and incremental ends with I.

timeStarted

Type: time
This field denotes the time when the backup started

timeCompleted

Type: time
This field indicates the time when the backup got completed. This field is populated once the PostgresBackup CR reaches Failed or Succeeded state. It is empty when the backup is running.

backupLocation

Type: string
This field contains the name of the backupLocation that's configured on the instance for which this backup is being taken.

dbName

Type: string
This field denotes the database name associated with the instance. It is same as the instance's `spec.PgConfig.Dbname`

stanzaName

Type: string
This field shows the backup stanza name of the instance for which the backup was taken. It has the format of --

conditions

Type: Object
This field shows the current state of the PostgresBackup resource. It is useful for debugging. If the backup fails, this field would show the failure and the corresponding error message. For e.g.:

```
Conditions:
    Last Transition Time:  2022-03-21T21:33:14Z
    Message:               pgbackrest failed with error: WARN: environment contains in
valid option 'config-version'ERROR: [050]: unable to acquire lock on file '/tmp/pgback
rest/default-postgres-sample2-backup.lock': Resource temporarily unavailable       HIN
T: is another pgBackRest process running?
    Reason:                PgbackrestFailure
    Status:                True
    Type:                  BackupFailed
```

Backup and Restore CRD API Reference - PostgresRestore Resource

PostgresRestore Synopsis

```
apiVersion: sql.tanzu.vmware.com/v1
kind: PostgresRestore
metadata:
  name: restore-sample
spec:
  sourceBackup:
    name: backup-sample
  targetInstance:
    name: postgres-sample
```

The list below explains the properties that can be set for the PostgresRestore resource.

Metadata

name

Type: String
Required
Default: n/a
The name of the PostgresRestore. Must be unique within a namespace.
Example:
`restore-sample`

Spec

The spec describes the desired state for the Postgres backup object.

sourceBackup

Type: Object
Required

Default: n/a

- name
  Type: String
  Required
  Default: n/a
  The name of the PostgresBackup that represents the backup artifact to restore.
  Must be in the same namespace as the PostgresRestore.
  Example
  `backup-sample`

targetInstance

Type: Object
Required
Default: n/a

- name
  Type: String
  Required
  Default: n/a
  The name of the target instance where the sourceBackup should be restored.
  In case of in-place restore, this should be the source instance where the sourceBackup was taken
  Example
  `my-target-instance`

Status

The status fields show the observed status of the PostgresRestore object and the values are updated by the controller that handles Backup/Restore in the instance.

phase

Type: string
This field shows the current state of a PostgresRestore CR. It can be either empty or have any of the following values: Running, RecreatingNodes, RecreatingPrimary, RecreatingSecondary, WaitForPrimary, Finalizing, Succeeded, or Failed. All these phases are related to some internal processing that the operator does to achieve a restore. The terminal phasees are `Failed` or `Succeeded`.

timeStarted

Type: time
This field denotes the time when the restore started

timeCompleted

Type: time
This field indicates the time when the restore got completed. This field is populated once the

PostgresRestore CR reaches Failed or Succeeded state.

Message

Type: time
This field shows some error/useful messages for why a restore is stuck or pending. **Example**:

```
Instance postgres-sample does not specify spec.backupLocation.name
```