Red Hat Enterprise Linux 9

Configuring and using database servers

Installing, configuring, backing up and migrating data on database servers

Last Updated: 2025-06-05

Red Hat Enterprise Linux 9 Configuring and using database servers

Installing, configuring, backing up and migrating data on database servers

Legal Notice

Abstract

Install the MariaDB, MySQL, or PostgreSQL database server on Red Hat Enterprise Linux 9. Configure the chosen database server, back up your data, and migrate your data to a later version of the database server.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. Let us know how we can improve it.

Submitting feedback through Jira (account required)

1. Log in to the Jira website.
2. Click **Create** in the top navigation bar
3. Enter a descriptive title in the **Summary** field.
4. Enter your suggestion for improvement in the **Description** field. Include links to the relevant parts of the documentation.
5. Click **Create** at the bottom of the dialogue.

CHAPTER 1. INTRODUCTION TO DATABASE SERVERS

A database server is a service that provides features of a database management system (DBMS). DBMS provides utilities for database administration and interacts with end users, applications, and databases.

Red Hat Enterprise Linux 9 provides the following database management systems:

- **MariaDB 10.5**
- **MariaDB 10.11** - available since RHEL 9.4
- **MySQL 8.0**
- **PostgreSQL 13**
- **PostgreSQL 15** - available since RHEL 9.2
- **PostgreSQL 16** - available since RHEL 9.4
- **Redis 6**

CHAPTER 2. USING MARIADB

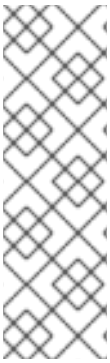
The **MariaDB** server is an open source fast and robust database server that is based on the **MySQL** technology. **MariaDB** is a relational database that converts data into structured information and provides an SQL interface for accessing data. It includes multiple storage engines and plugins, as well as geographic information system (GIS) and JavaScript Object Notation (JSON) features.

Learn how to install and configure **MariaDB** on a RHEL system, how to back up **MariaDB** data, how to migrate from an earlier **MariaDB** version, and how to replicate a database using the **MariaDB Galera Cluster**.

2.1. INSTALLING MARIADB

RHEL 9.0 provides **MariaDB 10.5** as the initial version of this Application Stream, which can be installed easily as an RPM package. Additional **MariaDB** versions are provided as modules with a shorter life cycle in minor releases of RHEL 9.

RHEL 9.4 introduced **MariaDB 10.11** as the **mariadb:10.11** module stream.



NOTE

By design, it is impossible to install more than one version (stream) of the same module in parallel. Therefore, you must choose only one of the available streams from the **mariadb** module. You can use different versions of the **MariaDB** database server in containers, see Running multiple MariaDB versions in containers.

The **MariaDB** and **MySQL** database servers cannot be installed in parallel in RHEL 9 due to conflicting RPM packages. You can use the **MariaDB** and **MySQL** database servers in parallel in containers, see Running multiple MySQL and MariaDB versions in containers.

To install **MariaDB**, use the following procedure.

Procedure

1. Install **MariaDB** server packages:
   a. For **MariaDB 10.5** from the RPM package:

      ```
      # dnf install mariadb-server
      ```

   b. For **MariaDB 10.11** by selecting stream (version) **11** from the **mariadb** module and specifying the server profile, for example:

      ```
      # dnf module install mariadb:10.11/server
      ```

2. Start the **mariadb** service:

   ```
   # systemctl start mariadb.service
   ```

3. Enable the **mariadb** service to start at boot:

   ```
   # systemctl enable mariadb.service
   ```

2.2. RUNNING MULTIPLE MARIADB VERSIONS IN CONTAINERS

To run different versions of **MariaDB** on the same host, run them in containers because you cannot install multiple versions (streams) of the same module in parallel.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Use your Red Hat Customer Portal account to authenticate to the **registry.redhat.io** registry:

```
# podman login registry.redhat.io
```

   Skip this step if you are already logged in to the container registry.

2. Run **MariaDB 10.5** in a container:

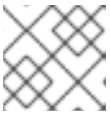
```
$ podman run -d --name <container_name> -e
MYSQL_ROOT_PASSWORD=<mariadb_root_password> -p <host_port_1>:3306
rhel9/mariadb-105
```

   For more information about the usage of this container image, see the Red Hat Ecosystem Catalog.

3. Run **MariaDB 10.11** in a container:

```
$ podman run -d --name <container_name> -e
MYSQL_ROOT_PASSWORD=<mariadb_root_password> -p <host_port_2>:3306
rhel9/mariadb-1011
```

   For more information about the usage of this container image, see the Red Hat Ecosystem Catalog.



NOTE

The container names and host ports of the two database servers must differ.

4. To ensure that clients can access the database server on the network, open the host ports in the firewall:

```
# firewall-cmd --permanent --add-port={<host_port_1>/tcp,<host_port_2>/tcp,...}
# firewall-cmd --reload
```

Verification

1. Display information about running containers:

```
$ podman ps
```

2. Connect to the database server and log in as root:

```
# mysql -u root -p -h localhost -P <host_port> --protocol tcp
```

Additional resources

- Building, running, and managing containers
- Browse containers in the Red Hat Ecosystem Catalog

2.3. CONFIGURING MARIADB

To configure the **MariaDB** server for networking, use the following procedure.

Procedure

1. Edit the **[mysqld]** section of the **/etc/my.cnf.d/mariadb-server.cnf** file. You can set the following configuration directives:
   - **bind-address** - is the address on which the server listens. Possible options are:
     - a host name
     - an IPv4 address
     - an IPv6 address
   - **skip-networking** - controls whether the server listens for TCP/IP connections. Possible values are:
     - 0 - to listen for all clients
     - 1 - to listen for local clients only
   - **port** - the port on which **MariaDB** listens for TCP/IP connections.
2. Restart the **mariadb** service:

```
# systemctl restart mariadb.service
```

2.4. SETTING UP TLS ENCRYPTION ON A MARIADB SERVER

By default, **MariaDB** uses unencrypted connections. For secure connections, enable TLS support on the **MariaDB** server and configure your clients to establish encrypted connections.

2.4.1. Placing the CA certificate, server certificate, and private key on the MariaDB server

Before you can enable TLS encryption in the **MariaDB** server, store the certificate authority (CA) certificate, the server certificate, and the private key on the **MariaDB** server.

Prerequisites

- The following files in Privacy Enhanced Mail (PEM) format have been copied to the server:
  - The private key of the server: **server.example.com.key.pem**

- The server certificate: **server.example.com.crt.pem**
- The Certificate Authority (CA) certificate: **ca.crt.pem**

For details about creating a private key and certificate signing request (CSR), as well as about requesting a certificate from a CA, see your CA's documentation.

Procedure

1. Store the CA and server certificates in the **/etc/pki/tls/certs/** directory:

```
# mv <path>/server.example.com.crt.pem /etc/pki/tls/certs/
# mv <path>/ca.crt.pem /etc/pki/tls/certs/
```

2. Set permissions on the CA and server certificate that enable the **MariaDB** server to read the files:

```
# chmod 644 /etc/pki/tls/certs/server.example.com.crt.pem /etc/pki/tls/certs/ca.crt.pem
```

Because certificates are part of the communication before a secure connection is established, any client can retrieve them without authentication. Therefore, you do not need to set strict permissions on the CA and server certificate files.

3. Store the server's private key in the **/etc/pki/tls/private/** directory:

```
# mv <path>/server.example.com.key.pem /etc/pki/tls/private/
```

4. Set secure permissions on the server's private key:

```
# chmod 640 /etc/pki/tls/private/server.example.com.key.pem
# chgrp mysql /etc/pki/tls/private/server.example.com.key.pem
```

If unauthorized users have access to the private key, connections to the **MariaDB** server are no longer secure.

5. Restore the SELinux context:

```
#  restorecon -Rv /etc/pki/tls/
```

2.4.2. Configuring TLS on a MariaDB server

To improve security, enable TLS support on the **MariaDB** server. As a result, clients can transmit data with the server using TLS encryption.

Prerequisites

- You installed the **MariaDB** server.
- The **mariadb** service is running.
- The following files in Privacy Enhanced Mail (PEM) format exist on the server and are readable by the **mysql** user:
  - The private key of the server: **/etc/pki/tls/private/server.example.com.key.pem**

- The server certificate: **/etc/pki/tls/certs/server.example.com.crt.pem**
- The Certificate Authority (CA) certificate **/etc/pki/tls/certs/ca.crt.pem**

- The subject distinguished name (DN) or the subject alternative name (SAN) field in the server certificate matches the server's host name.
- If the server runs RHEL 9.2 or later and the FIPS mode is enabled, clients must either support the Extended Master Secret (EMS) extension or use TLS 1.3. TLS 1.2 connections without EMS fail. For more information, see the Red Hat Knowledgebase solution TLS extension "Extended Master Secret" enforced enforced on RHEL 9.2 and later.

Procedure

1. Create the **/etc/my.cnf.d/mariadb-server-tls.cnf** file:
   a. Add the following content to configure the paths to the private key, server and CA certificate:

```
[mariadb]
ssl_key = /etc/pki/tls/private/server.example.com.key.pem
ssl_cert = /etc/pki/tls/certs/server.example.com.crt.pem
ssl_ca = /etc/pki/tls/certs/ca.crt.pem
```

   b. If you have a Certificate Revocation List (CRL), configure the **MariaDB** server to use it:

```
ssl_crl = /etc/pki/tls/certs/example.crl.pem
```

   c. Optional: Reject connection attempts without encryption. To enable this feature, append:

```
require_secure_transport = on
```

   d. Optional: Set the TLS versions the server should support. For example, to support TLS 1.2 and TLS 1.3, append:

```
tls_version = TLSv1.2,TLSv1.3
```

   By default, the server supports TLS 1.1, TLS 1.2, and TLS 1.3.

2. Restart the **mariadb** service:

```
# systemctl restart mariadb.service
```

Verification

To simplify troubleshooting, perform the following steps on the **MariaDB** server before you configure the local client to use TLS encryption:

1. Verify that **MariaDB** now has TLS encryption enabled:

```
# mysql -u root -p -e "SHOW GLOBAL VARIABLES LIKE 'have_ssl';"
+---------------+----------------+
| Variable_name | Value          |
```

```
+---------------+----------------+
| have_ssl      | YES            |
+---------------+----------------+
```

If the **have_ssl** variable is set to **yes**, TLS encryption is enabled.

2. If you configured the **MariaDB** service to only support specific TLS versions, display the **tls_version** variable:

```
# mysql -u root -p -e "SHOW GLOBAL VARIABLES LIKE 'tls_version';"
+---------------+----------------+
| Variable_name | Value          |
+---------------+----------------+
| tls_version   | TLSv1.2,TLSv1.3 |
+---------------+----------------+
```

Additional resources

- Placing the CA certificate, server certificate, and private key on the MariaDB server

2.4.3. Requiring TLS encrypted connections for specific user accounts

Users that have access to sensitive data should always use a TLS-encrypted connection to avoid sending data unencrypted over the network.

If you cannot configure on the server that a secure transport is required for all connections (**require_secure_transport = on**), configure individual user accounts to require TLS encryption.

Prerequisites

- The **MariaDB** server has TLS support enabled.
- The user you configure to require secure transport exists.

Procedure

1. Connect as an administrative user to the **MariaDB** server:

```
# mysql -u root -p -h server.example.com
```

If your administrative user has no permissions to access the server remotely, perform the command on the **MariaDB** server and connect to **localhost**.

2. Use the **REQUIRE SSL** clause to enforce that a user must connect using a TLS-encrypted connection:

```
MariaDB [(none)]> ALTER USER 'example'@'%' REQUIRE SSL;
```

Verification

1. Connect to the server as the **example** user using TLS encryption:

```
# mysql -u example -p -h server.example.com --ssl
...
MariaDB [(none)]>
```

If no error is shown and you have access to the interactive **MariaDB** console, the connection with TLS succeeds.

2. Attempt to connect as the **example** user with TLS disabled:

```
# mysql -u example -p -h server.example.com --skip-ssl
ERROR 1045 (28000): Access denied for user 'example'@'server.example.com' (using password: YES)
```

The server rejected the login attempt because TLS is required for this user but disabled (**--skip-ssl**).

Additional resources

- Setting up TLS encryption on a MariaDB server

2.5. GLOBALLY ENABLING TLS ENCRYPTION IN MARIADB CLIENTS

If your **MariaDB** server supports TLS encryption, configure your clients to establish only secure connections and to verify the server certificate. This procedure describes how to enable TLS support for all users on the server.

2.5.1. Configuring the MariaDB client to use TLS encryption by default

On RHEL, you can globally configure that the **MariaDB** client uses TLS encryption and verifies that the Common Name (CN) in the server certificate matches the hostname the user connects to. This prevents man-in-the-middle attacks.

Prerequisites

- The **MariaDB** server has TLS support enabled.
- If the certificate authority (CA) that issued the server's certificate is not trusted by RHEL, the CA certificate has been copied to the client.
- If the MariaDB server runs RHEL 9.2 or later and the FIPS mode is enabled, this client supports the Extended Master Secret (EMS) extension or uses TLS 1.3. TLS 1.2 connections without EMS fail. For more information, see the Red Hat Knowledgebase solution TLS extension "Extended Master Secret" enforced on RHEL 9.2 and later.

Procedure

1. If RHEL does not trust the CA that issued the server's certificate:
   a. Copy the CA certificate to the **/etc/pki/ca-trust/source/anchors/** directory:

```
# cp <path>/ca.crt.pem /etc/pki/ca-trust/source/anchors/
```

   b. Set permissions that enable all users to read the CA certificate file:

```
# chmod 644 /etc/pki/ca-trust/source/anchors/ca.crt.pem
```

c. Rebuild the CA trust database:

```
# update-ca-trust
```

2. Create the **/etc/my.cnf.d/mariadb-client-tls.cnf** file with the following content:

```
[client-mariadb]
ssl
ssl-verify-server-cert
```

These settings define that the **MariaDB** client uses TLS encryption (**ssl**) and that the client compares the hostname with the CN in the server certificate (**ssl-verify-server-cert**).

Verification

- Connect to the server using the hostname, and display the server status:

```
# mysql -u root -p -h server.example.com -e status
...
SSL:        Cipher in use is TLS_AES_256_GCM_SHA384
```

If the **SSL** entry contains **Cipher in use is…**, the connection is encrypted.

Note that the user you use in this command has permissions to authenticate remotely.

If the hostname you connect to does not match the hostname in the TLS certificate of the server, the **ssl-verify-server-cert** parameter causes the connection to fail. For example, if you connect to **localhost**:

```
# mysql -u root -p -h localhost -e status
ERROR 2026 (HY000): SSL connection error: Validation of SSL server certificate failed
```

Additional resources

- The **--ssl*** parameter descriptions in the **mysql(1)** man page on your system

2.6. BACKING UP MARIADB DATA

There are two main ways to back up data from a **MariaDB** database in Red Hat Enterprise Linux 9:

- Logical backup
- Physical backup

Logical backup consists of the SQL statements necessary to restore the data. This type of backup exports information and records in plain text files.

The main advantage of logical backup over physical backup is portability and flexibility. The data can be restored on other hardware configurations, **MariaDB** versions or Database Management System (DBMS), which is not possible with physical backups.

Note that logical backup can be performed if the **mariadb.service** is running. Logical backup does not include log and configuration files.

Physical backup consists of copies of files and directories that store the content.

Physical backup has the following advantages compared to logical backup:

- Output is more compact.
- Backup is smaller in size.
- Backup and restore are faster.
- Backup includes log and configuration files.

Note that physical backup must be performed when the **mariadb.service** is not running or all tables in the database are locked to prevent changes during the backup.

You can use one of the following **MariaDB** backup approaches to back up data from a **MariaDB** database:

- Logical backup with **mariadb-dump**
- Physical online backup using the **Mariabackup** utility
- File system backup
- Replication as a backup solution

2.6.1. Performing logical backup with mariadb-dump

The **mariadb-dump** client is a backup utility, which can be used to dump a database or a collection of databases for the purpose of a backup or transfer to another database server. The output of **mariadb-dump** typically consists of SQL statements to re-create the server table structure, populate it with data, or both. **mariadb-dump** can also generate files in other formats, including XML and delimited text formats, such as CSV.

To perform the **mariadb-dump** backup, you can use one of the following options:

- Back up one or more selected databases
- Back up all databases
- Back up a subset of tables from one database

Procedure

- To dump a single database, run:

```
# mariadb-dump [options] --databases db_name > backup-file.sql
```

- To dump multiple databases at once, run:

```
# mariadb-dump [options] --databases db_name1 [db_name2 ...] > backup-file.sql
```

- To dump all databases, run:

```
# mariadb-dump [options] --all-databases > backup-file.sql
```

- To load one or more dumped full databases back into a server, run:

```
# mariadb < backup-file.sql
```

- To load a database to a remote **MariaDB** server, run:

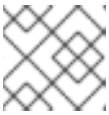
```
# mariadb --host=remote_host < backup-file.sql
```

- To dump a subset of tables from one database, add a list of the chosen tables at the end of the **mariadb-dump** command:

```
# mariadb-dump [options] db_name [tbl_name ...] > backup-file.sql
```

- To load a subset of tables dumped from one database, run:

```
# mariadb db_name < backup-file.sql
```



NOTE

The *db_name* database must exist at this point.

- To see a list of the options that **mariadb-dump** supports, run:

```
$ mariadb-dump --help
```

Additional resources

- MariaDB Documentation - mariadb-dump

2.6.2. Performing physical online backup using the Mariabackup utility

Mariabackup is a utility based on the Percona XtraBackup technology, which enables performing physical online backups of InnoDB, Aria, and MyISAM tables. This utility is provided by the **mariadb-backup** package from the AppStream repository.

Mariabackup supports full backup capability for **MariaDB** server, which includes encrypted and compressed data.

Prerequisites

- The **mariadb-backup** package is installed on the system:

```
# dnf install mariadb-backup
```

- You must provide **Mariabackup** with credentials for the user under which the backup will be run. You can provide the credentials either on the command line or by a configuration file.
- Users of **Mariabackup** must have the **RELOAD**, **LOCK TABLES**, and **REPLICATION CLIENT** privileges.

To create a backup of a database using **Mariabackup**, use the following procedure.

Procedure

- To create a backup while providing credentials on the command line, run:

  ```
  $ mariabackup --backup --target-dir <backup_directory> --user <backup_user> --password <backup_passwd>
  ```

  The **target-dir** option defines the directory where the backup files are stored. If you want to perform a full backup, the target directory must be empty or not exist.

  The **user** and **password** options allow you to configure the user name and the password.

- To create a backup with credentials set in a configuration file:

  1. Create a configuration file in the **/etc/my.cnf.d/** directory, for example, **/etc/my.cnf.d/mariabackup.cnf**.
  2. Add the following lines into the **[xtrabackup]** or **[mysqld]** section of the new file:

     ```
     [xtrabackup]
     user=myuser
     password=mypassword
     ```

  3. Perform the backup:

     ```
     $ mariabackup --backup --target-dir <backup_directory>
     ```

Additional resources

- Full Backup and Restore with Mariabackup

2.6.3. Restoring data using the Mariabackup utility

When the backup is complete, you can restore the data from the backup by using the **mariabackup** command with one of the following options:

- **--copy-back** allows you to keep the original backup files.
- **--move-back** moves the backup files to the data directory and removes the original backup files.

To restore data using the **Mariabackup** utility, use the following procedure.

Prerequisites

- Verify that the **mariadb** service is not running:

  ```
  # systemctl stop mariadb.service
  ```

- Verify that the data directory is empty.

- Users of **Mariabackup** must have the **RELOAD**, **LOCK TABLES**, and **REPLICATION CLIENT** privileges.

Procedure

1. Run the **mariabackup** command:
   - To restore data and keep the original backup files, use the **--copy-back** option:

     ```
     $ mariabackup --copy-back --target-dir=/var/mariadb/backup/
     ```

   - To restore data and remove the original backup files, use the **--move-back** option:

     ```
     $ mariabackup --move-back --target-dir=/var/mariadb/backup/
     ```

2. Fix the file permissions.
   When restoring a database, **Mariabackup** preserves the file and directory privileges of the backup. However, **Mariabackup** writes the files to disk as the user and group restoring the database. After restoring a backup, you may need to adjust the owner of the data directory to match the user and group for the **MariaDB** server, typically **mysql** for both.

   For example, to recursively change ownership of the files to the **mysql** user and group:

   ```
   # chown -R mysql:mysql /var/lib/mysql/
   ```

3. Start the **mariadb** service:

   ```
   # systemctl start mariadb.service
   ```

Additional resources

- Full Backup and Restore with Mariabackup

2.6.4. Performing file system backup

To create a file system backup of **MariaDB** data files, copy the content of the **MariaDB** data directory to your backup location.

To back up also your current configuration or the log files, use the optional steps of the following procedure.

Procedure

1. Stop the **mariadb** service:

   ```
   # systemctl stop mariadb.service
   ```

2. Copy the data files to the required location:

   ```
   # cp -r /var/lib/mysql /backup-location
   ```

3. Optional: Copy the configuration files to the required location:

```
# cp -r /etc/my.cnf /etc/my.cnf.d /backup-location/configuration
```

4. Optional: Copy the log files to the required location:

```
# cp /var/log/mariadb/* /backup-location/logs
```

5. Start the **mariadb** service:

```
# systemctl start mariadb.service
```

6. When loading the backed up data from the backup location to the **/var/lib/mysql** directory, ensure that **mysql:mysql** is an owner of all data in **/var/lib/mysql**:

```
# chown -R mysql:mysql /var/lib/mysql
```

2.6.5. Replication as a backup solution

Replication is an alternative backup solution for source servers. If a source server replicates to a replica server, backups can be run on the replica without any impact on the source. The source can still run while you shut down the replica and back the data up from the replica.

WARNING

Replication itself is not a sufficient backup solution. Replication protects source servers against hardware failures, but it does not ensure protection against data loss. It is recommended that you use any other backup solution on the replica together with this method.

Additional resources

- Replicating MariaDB with Galera
- Replication as a backup solution

2.7. MIGRATING TO MARIADB 10.5

In RHEL 8, the **MariaDB** server is available in versions 10.3, 10.5, and 10.11, each provided by a separate module stream. RHEL 9 provides **MariaDB 10.5**, **MariaDB 10.11**, and **MySQL 8.0.**

This part describes migration from a RHEL 8 version of **MariaDB 10.3** to RHEL 9 version of **MariaDB 10.5.**

2.7.1. Notable differences between MariaDB 10.3 and MariaDB 10.5

Significant changes between **MariaDB 10.3** and **MariaDB 10.5** include:

- **MariaDB** now uses the **unix_socket** authentication plugin by default. The plugin enables users to use operating system credentials when connecting to **MariaDB** through the local UNIX socket file.

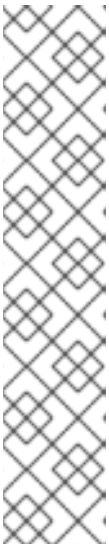
- **MariaDB** adds **mariadb-*** named binaries and **mysql*** symbolic links pointing to the **mariadb-*** binaries. For example, the **mysqladmin**, **mysqlaccess**, and **mysqlshow** symlinks point to the **mariadb-admin**, **mariadb-access**, and **mariadb-show** binaries, respectively.

- The **SUPER** privilege has been split into several privileges to better align with each user role. As a result, certain statements have changed required privileges.

- In parallel replication, the **slave_parallel_mode** now defaults to **optimistic**.

- In the **InnoDB** storage engine, defaults of the following variables have been changed: **innodb_adaptive_hash_index** to **OFF** and **innodb_checksum_algorithm** to **full_crc32**.

- **MariaDB** now uses the **libedit** implementation of the underlying software managing the **MariaDB** command history (the **.mysql_history** file) instead of the previously used **readline** library. This change impacts users working directly with the **.mysql_history** file. Note that **.mysql_history** is a file managed by the **MariaDB** or **MySQL** applications, and users should not work with the file directly. The human-readable appearance is coincidental.



NOTE

To increase security, you can consider not maintaining a history file. To disable the command history recording:

1. Remove the **.mysql_history** file if it exists.
2. Use either of the following approaches:
   - Set the **MYSQL_HISTFILE** variable to **/dev/null** and include this setting in any of your shell's startup files.
   - Change the **.mysql_history** file to a symbolic link to **/dev/null**:

     ```
     $ ln -s /dev/null $HOME/.mysql_history
     ```

MariaDB Galera Cluster has been upgraded to version 4 with the following notable changes:

- **Galera** adds a new streaming replication feature, which supports replicating transactions of unlimited size. During an execution of streaming replication, a cluster replicates a transaction in small fragments.

- **Galera** now fully supports Global Transaction ID (GTID).

- The default value for the **wsrep_on** option in the **/etc/my.cnf.d/galera.cnf** file has changed from **1** to **0** to prevent end users from starting **wsrep** replication without configuring required additional options.

Changes to the PAM plugin in **MariaDB 10.5** include:

- **MariaDB 10.5** adds a new version of the Pluggable Authentication Modules (PAM) plugin. The PAM plugin version 2.0 performs PAM authentication using a separate **setuid root** helper binary, which enables **MariaDB** to use additional PAM modules.

- The helper binary can be executed only by users in the **mysql** group. By default, the group contains only the **mysql** user. Red Hat recommends that administrators do not add more users to the **mysql** group to prevent password-guessing attacks without throttling or logging through this helper utility.
- In **MariaDB 10.5**, the Pluggable Authentication Modules (PAM) plugin and its related files have been moved to a new package, **mariadb-pam**. As a result, no new **setuid root** binary is introduced on systems that do not use PAM authentication for **MariaDB**.
- The **mariadb-pam** package contains both PAM plugin versions: version 2.0 is the default, and version 1.0 is available as the **auth_pam_v1** shared object library.
- The **mariadb-pam** package is not installed by default with the **MariaDB** server. To make the PAM authentication plugin available in **MariaDB 10.5**, install the **mariadb-pam** package manually.

2.7.2. Migrating from a RHEL 8 version of MariaDB 10.3 to a RHEL 9 version of MariaDB 10.5

This procedure describes migrating from the **MariaDB 10.3** to the **MariaDB 10.5** using the **mariadb-upgrade** utility.

The **mariadb-upgrade** utility is provided by the **mariadb-server-utils** subpackage, which is installed as a dependency of the **mariadb-server** package.

Prerequisites

- Before performing the upgrade, back up all your data stored in the **MariaDB** databases.

Procedure

1. Ensure that the **mariadb-server** package is installed on the RHEL 9 system:

```
# dnf install mariadb-server
```

2. Ensure that the **mariadb** service is not running on either of the source and target systems at the time of copying data:

```
# systemctl stop mariadb.service
```

3. Copy the data from the source location to the **/var/lib/mysql/** directory on the RHEL 9 target system.
4. Set the appropriate permissions and SELinux context for copied files on the target system:

```
# restorecon -vr /var/lib/mysql
```

5. Ensure that **mysql:mysql** is owner of all data in the **/var/lib/mysql** directory:

```
# chown -R mysql:mysql /var/lib/mysql
```

6. Adjust the configuration so that option files located in **/etc/my.cnf.d/** include only options valid for **MariaDB 10.5**. For details, see upstream documentation for MariaDB 10.4 and MariaDB 10.5.

7. Start the **MariaDB** server on the target system.
   - When upgrading a database running standalone:

     ```
     # systemctl start mariadb.service
     ```

   - When upgrading a **Galera** cluster node:

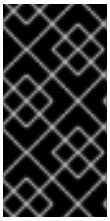
     ```
     # galera_new_cluster
     ```

     The **mariadb** service will be started automatically.
8. Execute the **mariadb-upgrade** utility to check and repair internal tables.
   - When upgrading a database running standalone:

     ```
     $ mariadb-upgrade
     ```

   - When upgrading a **Galera** cluster node:

     ```
     $ mariadb-upgrade --skip-write-binlog
     ```



IMPORTANT

There are certain risks and known problems related to an in-place upgrade. For example, some queries might not work or they will be run in a different order than before the upgrade. For more information about these risks and problems, and for general information about an in-place upgrade, see MariaDB 10.5 Release Notes .

2.8. UPGRADING FROM MARIADB 10.5 TO MARIADB 10.11

This part describes migration from **MariaDB 10.5** to **MariaDB 10.11** within RHEL 9.

2.8.1. Notable differences between MariaDB 10.5 and MariaDB 10.11

Significant changes between **MariaDB 10.5** and **MariaDB 10.11** include:

- A new **sys_schema** feature is a collection of views, functions, and procedures to provide information about database usage.
- The **CREATE TABLE**, **ALTER TABLE**, **RENAME TABLE**, **DROP TABLE**, **DROP DATABASE**, and related Data Definition Language (DDL) statements are now atomic. The statement must be fully completed, otherwise the changes are reverted. Note that when deleting multiple tables with **DROP TABLE**, only each individual drop is atomic, not the full list of tables.
- A new **GRANT ... TO PUBLIC** privilege is available.
- The **SUPER** and **READ ONLY ADMIN** privileges are now separate.
- You can now store universally unique identifiers in the new **UUID** database data type.
- **MariaDB** now supports the Secure Socket Layer (SSL) protocol version 3.

- The **MariaDB** server now requires correctly configured SSL to start. Previously, **MariaDB** silently disabled SSL and used insecure connections in case of misconfigured SSL.
- **MariaDB** now supports the natural sort order through the **natural_sort_key()** function.
- A new **SFORMAT** function is now available for arbitrary text formatting.
- The **utf8** character set (and related collations) is now by default an alias for **utf8mb3**.
- **MariaDB** supports the Unicode Collation Algorithm (UCA) 14 collations.
- **systemd** socket activation files for **MariaDB** are now available in the **/usr/share/** directory. Note that they are not a part of the default configuration in RHEL as opposed to upstream.
- Error messages now contain the **MariaDB** string instead of **MySQL**.
- Error messages are now available in the Chinese language.
- The default logrotate file has changed significantly. Review your configuration before migrating to **MariaDB 10.11**.
- For **MariaDB** and **MySQL** clients, the connection property specified on the command line (for example, **--port=3306**), now forces the protocol type of communication between the client and the server, such as **tcp**, **socket**, **pipe**, or **memory**. Previously, for example, the specified port was ignored if a **MariaDB** client connected through a UNIX socket.

2.8.2. Upgrading from a RHEL 9 version of MariaDB 10.5 to MariaDB 10.11

This procedure describes upgrading from **MariaDB 10.5** provided by the **mariadb-server** RPM package to the **mariadb:10.11** module stream using the **dnf** and **mariadb-upgrade** utilities.

The **mariadb-upgrade** utility is provided by the **mariadb-server-utils** subpackage, which is installed as a dependency of the **mariadb-server** package.

Prerequisites

- Before performing the upgrade, back up all your data stored in the **MariaDB** databases.

Procedure

1. Stop the **MariaDB** server:

```
# systemctl stop mariadb.service
```

2. Switch from the non-modular **MariaDB 10.5** to modular **MariaDB 10.11**:

```
# dnf module switch-to mariadb:10.11
```

3. Adjust the configuration so that option files located in **/etc/my.cnf.d/** include only options valid for **MariaDB 10.11**. For details, see upstream documentation for MariaDB 10.6 and MariaDB 10.11.
4. Start the **MariaDB** server.
   - When upgrading a database running standalone:

```
# systemctl start mariadb.service
```

- When upgrading a **Galera** cluster node:

```
# galera_new_cluster
```

The **mariadb** service will be started automatically.

5. Execute the **mariadb-upgrade** utility to check and repair internal tables.
   - When upgrading a database running standalone:

```
# mariadb-upgrade
```

   - When upgrading a **Galera** cluster node:

```
# mariadb-upgrade --skip-write-binlog
```

IMPORTANT

There are certain risks and known problems related to an in-place upgrade. For example, some queries might not work or they will be run in a different order than before the upgrade. For more information about these risks and problems, and for general information about an in-place upgrade, see MariaDB 10.11 Release Notes .

2.9. REPLICATING MARIADB WITH GALERA

You can replicate a MariaDB database by using the Galera solution on Red Hat Enterprise Linux 9.

2.9.1. Introduction to MariaDB Galera Cluster

Galera replication is based on the creation of a synchronous multi-source **MariaDB Galera Cluster** consisting of multiple MariaDB servers. Unlike the traditional primary/replica setup where replicas are usually read-only, nodes in the MariaDB Galera Cluster can be all writable.

The interface between Galera replication and a **MariaDB** database is defined by the write set replication API (**wsrep API**).

The main features of **MariaDB Galera Cluster** are:

- Synchronous replication
- Active-active multi-source topology
- Read and write to any cluster node
- Automatic membership control, failed nodes drop from the cluster
- Automatic node joining
- Parallel replication on row level
- Direct client connections: users can log on to the cluster nodes, and work with the nodes directly while the replication runs

Synchronous replication means that a server replicates a transaction at commit time by broadcasting the write set associated with the transaction to every node in the cluster. The client (user application) connects directly to the Database Management System (DBMS), and experiences behavior that is similar to native **MariaDB**.

Synchronous replication guarantees that a change that happened on one node in the cluster happens on other nodes in the cluster at the same time.

Therefore, synchronous replication has the following advantages over asynchronous replication:

- No delay in propagation of the changes between particular cluster nodes
- All cluster nodes are always consistent
- The latest changes are not lost if one of the cluster nodes crashes
- Transactions on all cluster nodes are executed in parallel
- Causality across the whole cluster

Additional resources

- About Galera replication
- What is MariaDB Galera Cluster
- Getting started with MariaDB Galera Cluster

2.9.2. Components to build MariaDB Galera Cluster

To build **MariaDB Galera Cluster**, you must install the following packages on your system:

- **mariadb-server-galera** - contains support files and scripts for **MariaDB Galera Cluster**.
- **mariadb-server** - is patched by **MariaDB** upstream to include the write set replication API (**wsrep API**). This API provides the interface between **Galera** replication and **MariaDB**.
- **galera** - is patched by **MariaDB** upstream to add full support for **MariaDB**. The **galera** package contains the following:
  - **Galera Replication Library** provides the whole replication functionality.
  - The **Galera Arbitrator** utility can be used as a cluster member that participates in voting in split-brain scenarios. However, **Galera Arbitrator** cannot participate in the actual replication.
  - **Galera Systemd service** and **Galera wrapper script** which are used for deploying the Galera Arbitrator utility. RHEL 9 provides the upstream version of these files, located at **/usr/lib/systemd/system/garbd.service** and **/usr/sbin/garb-systemd**.

Additional resources

- Galera Replication Library
- Galera Arbitrator

2.9.3. Deploying MariaDB Galera Cluster

You can deploy the **MariaDB Galera Cluster** packages and update the configuration. To form a new cluster, you must bootstrap the first node of the cluster.

Prerequisites

- All of the nodes in the cluster have TLS set up.
- All certificates on all nodes must have the **Extended Key Usage** field set to:

```
TLS Web Server Authentication, TLS Web Client Authentication
```

Procedure

1. Install the **MariaDB Galera Cluster** packages:

```
dnf install mariadb-server-galera
```

   As a result, the following packages are installed together with their dependencies:

   - **mariadb-server-galera**
   - **mariadb-server**
   - **galera**
     For more information about which packages you need to install to build **MariaDB Galera Cluster**, see Components to build MariaDB Cluster .

2. Update the **MariaDB** server replication configuration before the system is added to a cluster for the first time. The default configuration is distributed in the **/etc/my.cnf.d/galera.cnf** file. Before deploying **MariaDB Galera Cluster**, set the **wsrep_cluster_address** option in the **/etc/my.cnf.d/galera.cnf** file on all nodes to start with the following string:

```
gcomm://
```

   - For the initial node, it is possible to set **wsrep_cluster_address** as an empty list:

```
wsrep_cluster_address="gcomm://"
```

   - For all other nodes, set **wsrep_cluster_address** to include an address to any node which is already a part of the running cluster. For example:

```
wsrep_cluster_address="gcomm://10.0.0.10"
```

     For more information about how to set Galera Cluster address, see Galera Cluster Address .

3. Enable the **wsrep** API on every node by setting the **wsrep_on=1** option in the **/etc/my.cnf.d/galera.cnf** configuration file.

4. Add the **wsrep_provider_options** variable to the Galera configuration file with the TLS keys and certificates. For example:

```
wsrep_provider_options="socket.ssl_cert=/etc/pki/tls/certs/source.crt;socket.ssl_key=/etc/pki/tls/
rivate/source.key;socket.ssl_ca=/etc/pki/tls/certs/ca.crt"
```

5. Bootstrap a first node of a new cluster by running the following wrapper on that node:

```
# galera_new_cluster
```

This wrapper ensures that the **MariaDB** server daemon (**mysqld**) runs with the **--wsrep-new-cluster** option. This option provides the information that there is no existing cluster to connect to. Therefore, the node creates a new UUID to identify the new cluster.

NOTE

The **mariadb** service supports a systemd method for interacting with multiple **MariaDB** server processes. Therefore, in cases with multiple running **MariaDB** servers, you can bootstrap a specific instance by specifying the instance name as a suffix:

```
# galera_new_cluster mariadb@node1
```

6. Connect other nodes to the cluster by running the following command on each of the nodes:

```
# systemctl start mariadb.service
```

As a result, the node connects to the cluster, and synchronizes itself with the state of the cluster.

Verification

- See Checking the status of a MariaDB Galera cluster .

Additional resources

- Getting started with MariaDB Galera Cluster

2.9.4. Checking the status of a MariaDB Galera cluster

It is important to monitor and ensure the health, performance, and synchronization of a MariaDB Galera cluster. For that, you can query status variables on each node to monitor the node and the cluster.

To check the status of a MariaDB Galera cluster, you can use the following queries:

- Display the number of nodes in the cluster:

```
# mysql -u root -p -e 'show status like "wsrep_cluster_size";'
+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| wsrep_cluster_size | 4     |
+--------------------+-------+
```

- Display the node's cluster component status:

```
# mysql -u root -p -e 'show status like "wsrep_cluster_status";'
+----------------------+---------+
| Variable_name        | Value   |
+----------------------+---------+
| wsrep_cluster_status | Primary |
+----------------------+---------+
```

The value of the **wsrep_cluster_status** variable indicates the status of the cluster component the current node belongs to. Possible values are:

- **Primary**: The cluster is functioning normally. A quorum is present. In a healthy cluster, all nodes report **Primary**.
- **Non-primary**: The node has lost the connection to the primary component of the cluster and is no longer part of the active cluster. However, the node still can serve read queries but cannot process write operations.
- **Disconnected**: The node is not connected to any cluster component. Consequently, it cannot accept queries and is not replicating any data.

- Display the node's status:

```
# mysql -u root -p -e 'show status like "wsrep_local_state_comment";'
+---------------------------+--------+
| Variable_name             | Value  |
+---------------------------+--------+
| wsrep_local_state_comment | Synced |
+---------------------------+--------+
```

The following are frequent values of the **wsrep_local_state_comment** variable:

- **Synced**: The node is fully synchronized within the cluster and actively participating in replication.
- **Desynced**: The node is still part of the cluster but it is primarily busy with the state transfer.
- **Joining**: The node is in the process of joining a cluster.
- **Joined**: The node has successfully joined a cluster. It can receive and apply write sets from the cluster.
- **Donor**: The node currently provides a State Snapshot Transfer (SST) to a joining node. When a new node joins and requires a full state transfer, the cluster selects an existing node to send the necessary data.

- Check whether the node accepts write sets from the cluster:

```
# mysql -u root -p -e 'show status like "wsrep_ready";'
+--------------+-------+
| Variable_name | Value |
+--------------+-------+
| wsrep_ready   | ON    |
+--------------+-------+
```

When the **wsrep_ready** variable is **ON**, the node has successfully initialized its components and is connected to a cluster. Additionally, the node is synchronized or has reached a state where it can serve queries.

- Check whether the node has network connectivity with other hosts:

```
# mysql -u root -p -e 'show status like "wsrep_connected";'
+----------------+-------+
| Variable_name  | Value |
+----------------+-------+
| wsrep_connected | ON    |
+----------------+-------+
```

The **ON** value means that node has connectivity to at least one member in the cluster.

- Display the average size of the local received queue for write sets since the last **FLUSH STATUS** command or since the server started:

```
# mysql -u root -p -e 'show status like "wsrep_local_recv_queue_avg";'
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| wsrep_local_recv_queue_avg | 0.012 |
+---------------------------+-------+
```

A value near 0 is the ideal state and indicates that the node continues applying write sets as they are received. A persistently high or growing value can be an indicator of performance bottlenecks, such as slow disk I/O.

- Display the flow control status:

```
# mysql -u root -p -e 'show status like "wsrep_flow_control_paused";'
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| wsrep_flow_control_paused | 0     |
+--------------------------+-------+
```

This variable represents the fraction of time a node has been paused and is unable to process new incoming transactions because its local receive queue was too full, triggering flow control. A value close to 0 indicates the node continues with the replication workload efficiently. A value approaching 1.0 means that the node frequently or constantly encounters difficulty in applying write sets and can be a bottleneck for the cluster.

If the node is frequently pausing, you can adjust the **wsrep_slave_threads** parameter in the **/etc/my.cnf.d/galera.cnf** file.

- Display the average distance between the lowest and highest sequence numbers the node can apply in parallel:

```
# mysql -u root -p -e 'show status like "wsrep_cert_deps_distance";'
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| wsrep_cert_deps_distance | 1     |
+-------------------------+-------+
```

-

A higher value indicates a greater degree of parallelism. It is the optimal value you can use in the **wsrep_slave_threads** parameter in the **/etc/my.cnf.d/galera.cnf** file.

2.9.5. Adding a new node to MariaDB Galera Cluster

To add a new node to **MariaDB Galera Cluster**, use the following procedure.

Note that you can also use this procedure to reconnect an already existing node.

Procedure

- On the particular node, provide an address to one or more existing cluster members in the **wsrep_cluster_address** option within the **[mariadb]** section of the **/etc/my.cnf.d/galera.cnf** configuration file :

```
[mariadb]
wsrep_cluster_address="gcomm://192.168.0.1"
```

When a new node connects to one of the existing cluster nodes, it is able to see all nodes in the cluster.

However, preferably list all nodes of the cluster in **wsrep_cluster_address**.

As a result, any node can join a cluster by connecting to any other cluster node, even if one or more cluster nodes are down. When all members agree on the membership, the cluster's state is changed. If the new node's state is different from the state of the cluster, the new node requests either an Incremental State Transfer (IST) or a State Snapshot Transfer (SST) to ensure consistency with the other nodes.

Additional resources

- Getting started with MariaDB Galera Cluster
- Introduction to State Snapshot Transfers
- Securing Communications in Galera Cluster

2.9.6. Restarting MariaDB Galera Cluster

If you shut down all nodes at the same time, you stop the cluster, and the running cluster no longer exists. However, the cluster's data still exist.

To restart the cluster, bootstrap a first node as described in Configuring MariaDB Galera Cluster .

WARNING

If the cluster is not bootstrapped, and **mariadb** on the first node is started with only the **systemctl start mariadb.service** command, the node tries to connect to at least one of the nodes listed in the **wsrep_cluster_address** option in the **/etc/my.cnf.d/galera.cnf** file. If no nodes are currently running, the restart fails.

Additional resources

- Getting started with MariaDB Galera Cluster

2.10. DEVELOPING MARIADB CLIENT APPLICATIONS

Red Hat recommends developing your **MariaDB** client applications against the **MariaDB** client library.

The development files and programs necessary to build applications against the **MariaDB** client library are provided by the **mariadb-connector-c-devel** package.

Instead of using a direct library name, use the **mariadb_config** program, which is distributed in the **mariadb-connector-c-devel** package. This program ensures that the correct build flags are returned.

CHAPTER 3. USING MYSQL

The **MySQL** server is an open source fast and robust database server. **MySQL** is a relational database that converts data into structured information and provides an SQL interface for accessing data. It includes multiple storage engines and plugins, as well as geographic information system (GIS) and JavaScript Object Notation (JSON) features.

Learn how to install and configure **MySQL** on a RHEL system, how to back up **MySQL** data, how to migrate from an earlier **MySQL** version, and how to replicate a **MySQL**.

3.1. INSTALLING MYSQL

RHEL 9.0 provides **MySQL 8.0** as the initial version of this Application Stream, which you can install easily as an RPM package.

NOTE

The **MySQL** and **MariaDB** database servers cannot be installed in parallel in RHEL 9 due to conflicting RPM packages. You can use the **MySQL** and **MariaDB** database servers in parallel in containers, see Running multiple MySQL and MariaDB versions in containers .

To install **MySQL**, use the following procedure.

Procedure

1. Install **MySQL** server packages:
   - For **MySQL 8.0** from the RPM package, enter:

     ```
     # dnf install mysql-server
     ```

   - For **MySQL 8.4** by selecting the **8.4** stream from the **mysql** module and specifying the server profile:

     ```
     # dnf module install mysql:8.4/server
     ```

2. Start the **mysqld** service:

   ```
   # systemctl start mysqld.service
   ```

3. Enable the **mysqld** service to start at boot:

   ```
   # systemctl enable mysqld.service
   ```

4. *Recommended:* To improve security when installing **MySQL**, run the following command:

   ```
   $ mysql_secure_installation
   ```

   The command launches a fully interactive script, which prompts for each step in the process. The script enables you to improve security in the following ways:

   - Setting a password for root accounts

- Removing anonymous users
- Disallowing remote root logins (outside the local host)

3.2. RUNNING MULTIPLE MYSQL AND MARIADB VERSIONS IN CONTAINERS

To run both **MySQL** and **MariaDB** on the same host, run them in containers because you cannot install these database servers in parallel due to conflicting RPM packages.

This procedure includes **MySQL 8.0** and **MariaDB 10.5** as examples but you can use any **MySQL** or **MariaDB** container version available in the Red Hat Ecosystem Catalog.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Use your Red Hat Customer Portal account to authenticate to the **registry.redhat.io** registry:

```
# podman login registry.redhat.io
```

   Skip this step if you are already logged in to the container registry.

2. Run **MySQL 8.0** in a container:

```
$ podman run -d --name <container_name> -e
MYSQL_ROOT_PASSWORD=<mysql_root_password> -p <host_port_1>:3306
rhel9/mysql-80
```

   For more information about the usage of this container image, see the Red Hat Ecosystem Catalog.

3. Run **MySQL 8.4** in a container:

```
$ podman run -d --name <container_name> -e
MYSQL_ROOT_PASSWORD=<mysql_root_password> -p <host_port_2>:3306
rhel9/mysql-84
```

   For more information about the usage of this container image, see the Red Hat Ecosystem Catalog.

4. Run **MariaDB 10.5** in a container:

```
$ podman run -d --name <container_name> -e
MYSQL_ROOT_PASSWORD=<mariadb_root_password> -p <host_port_3>:3306
rhel9/mariadb-105
```

   For more information about the usage of this container image, see the Red Hat Ecosystem Catalog.

5. Run **MariaDB 10.11** in a container:

```
$ podman run -d --name <container_name> -e
MYSQL_ROOT_PASSWORD=<mariadb_root_password> -p <host_port_4>:3306
rhel9/mariadb-1011
```

For more information about the usage of this container image, see the Red Hat Ecosystem Catalog.

NOTE

The container names and host ports of the two database servers must differ.

6. To ensure that clients can access the database server on the network, open the host ports in the firewall:

```
# firewall-cmd --permanent --add-port=
{<host_port_1>/tcp,<host_port_2>/tcp,<host_port_3>/tcp,<host_port_4>/tcp,...}
# firewall-cmd --reload
```

Verification

1. Display information about running containers:

```
$ podman ps
```

2. Connect to the database server and log in as root:

```
# mysql -u root -p -h localhost -P <host_port> --protocol tcp
```

Additional resources

- Building, running, and managing containers
- Browse containers in the Red Hat Ecosystem Catalog

3.3. CONFIGURING MYSQL

To configure the **MySQL** server for networking, use the following procedure.

Procedure

1. Edit the **[mysqld]** section of the **/etc/my.cnf.d/mysql-server.cnf** file. You can set the following configuration directives:
   - **bind-address** - is the address on which the server listens. Possible options are:
     - a host name
     - an IPv4 address
     - an IPv6 address
   - **skip-networking** - controls whether the server listens for TCP/IP connections. Possible values are:

- 0 - to listen for all clients
- 1 - to listen for local clients only

- **port** - the port on which **MySQL** listens for TCP/IP connections.

2. Restart the **mysqld** service:

```
# systemctl restart mysqld.service
```

3.4. SETTING UP TLS ENCRYPTION ON A MYSQL SERVER

By default, **MySQL** uses unencrypted connections. For secure connections, enable TLS support on the **MySQL** server and configure your clients to establish encrypted connections.

3.4.1. Placing the CA certificate, server certificate, and private key on the MySQL server

Before you can enable TLS encryption on the **MySQL** server, store the certificate authority (CA) certificate, the server certificate, and the private key on the **MySQL** server.

Prerequisites

- The following files in Privacy Enhanced Mail (PEM) format have been copied to the server:
  - The private key of the server: **server.example.com.key.pem**
  - The server certificate: **server.example.com.crt.pem**
  - The Certificate Authority (CA) certificate: **ca.crt.pem**

  For details about creating a private key and certificate signing request (CSR), as well as about requesting a certificate from a CA, see your CA's documentation.

Procedure

1. Store the CA and server certificates in the **/etc/pki/tls/certs/** directory:

```
# mv <path>/server.example.com.crt.pem /etc/pki/tls/certs/
# mv <path>/ca.crt.pem /etc/pki/tls/certs/
```

2. Set permissions on the CA and server certificate that enable the **MySQL** server to read the files:

```
# chmod 644 /etc/pki/tls/certs/server.example.com.crt.pem /etc/pki/tls/certs/ca.crt.pem
```

   Because certificates are part of the communication before a secure connection is established, any client can retrieve them without authentication. Therefore, you do not need to set strict permissions on the CA and server certificate files.

3. Store the server's private key in the **/etc/pki/tls/private/** directory:

```
# mv <path>/server.example.com.key.pem /etc/pki/tls/private/
```

4. Set secure permissions on the server's private key:

```
# chmod 640 /etc/pki/tls/private/server.example.com.key.pem
# chgrp mysql /etc/pki/tls/private/server.example.com.key.pem
```

   If unauthorized users have access to the private key, connections to the **MySQL** server are no longer secure.

5. Restore the SELinux context:

```
# restorecon -Rv /etc/pki/tls/
```

3.4.2. Configuring TLS on a MySQL server

To improve security, enable TLS support on the **MySQL** server. As a result, clients can transmit data with the server using TLS encryption.

Prerequisites

- You installed the **MySQL** server.
- The **mysqld** service is running.
- The following files in Privacy Enhanced Mail (PEM) format exist on the server and are readable by the **mysql** user:
  - The private key of the server: **/etc/pki/tls/private/server.example.com.key.pem**
  - The server certificate: **/etc/pki/tls/certs/server.example.com.crt.pem**
  - The Certificate Authority (CA) certificate **/etc/pki/tls/certs/ca.crt.pem**
- The subject distinguished name (DN) or the subject alternative name (SAN) field in the server certificate matches the server's host name.

Procedure

1. Create the **/etc/my.cnf.d/mysql-server-tls.cnf** file:
   a. Add the following content to configure the paths to the private key, server and CA certificate:

```
[mysqld]
ssl_key = /etc/pki/tls/private/server.example.com.key.pem
ssl_cert = /etc/pki/tls/certs/server.example.com.crt.pem
ssl_ca = /etc/pki/tls/certs/ca.crt.pem
```

   b. If you have a Certificate Revocation List (CRL), configure the **MySQL** server to use it:

```
ssl_crl = /etc/pki/tls/certs/example.crl.pem
```

   c. Optional: Reject connection attempts without encryption. To enable this feature, append:

```
require_secure_transport = on
```

d. Optional: Set the TLS versions the server should support. For example, to support only TLS 1.3, append:

```
tls_version = TLSv1.3
```

By default, the server supports TLS 1.2 and TLS 1.3.

2. Restart the **mysqld** service:

```
# systemctl restart mysqld.service
```

Verification

To simplify troubleshooting, perform the following steps on the **MySQL** server before you configure the local client to use TLS encryption:

1. Verify that **MySQL** now has TLS encryption enabled:

```
# mysql -u root -p -h <MySQL_server_hostname> -e "SHOW session status LIKE 'Ssl_cipher';"
+---------------+-----------------------+
| Variable_name | Value                 |
+---------------+-----------------------+
| Ssl_cipher    | TLS_AES_256_GCM_SHA384 |
+---------------+-----------------------+
```

2. If you configured the **MySQL** server to only support specific TLS versions, display the **tls_version** variable:

```
# mysql -u root -p -e "SHOW GLOBAL VARIABLES LIKE 'tls_version';"
+---------------+---------+
| Variable_name | Value   |
+---------------+---------+
| tls_version   | TLSv1.3 |
+---------------+---------+
```

3. Verify that the server uses the correct CA certificate, server certificate, and private key files:

```
# mysql -u root -e "SHOW GLOBAL VARIABLES WHERE Variable_name REGEXP '^ssl_ca|^ssl_cert|^ssl_key';"
+-----------------+-------------------------------------------------+
| Variable_name   | Value                                           |
+-----------------+-------------------------------------------------+
| ssl_ca          | /etc/pki/tls/certs/ca.crt.pem                   |
| ssl_capath      |                                                 |
| ssl_cert        | /etc/pki/tls/certs/server.example.com.crt.pem   |
| ssl_key         | /etc/pki/tls/private/server.example.com.key.pem |
+-----------------+-------------------------------------------------+
```

Additional resources

- Placing the CA certificate, server certificate, and private key on the MySQL server

3.4.3. Requiring TLS encrypted connections for specific user accounts

Users that have access to sensitive data should always use a TLS-encrypted connection to avoid sending data unencrypted over the network.

If you cannot configure on the server that a secure transport is required for all connections (**require_secure_transport = on**), configure individual user accounts to require TLS encryption.

Prerequisites

- The **MySQL** server has TLS support enabled.
- The user you configure to require secure transport exists.
- The CA certificate is stored on the client.

Procedure

1. Connect as an administrative user to the **MySQL** server:

```
# mysql -u root -p -h server.example.com
```

   If your administrative user has no permissions to access the server remotely, perform the command on the **MySQL** server and connect to **localhost**.

2. Use the **REQUIRE SSL** clause to enforce that a user must connect using a TLS-encrypted connection:

```
MySQL [(none)]> ALTER USER 'example'@'%' REQUIRE SSL;
```

Verification

1. Connect to the server as the **example** user using TLS encryption:

```
# mysql -u example -p -h server.example.com
...
MySQL [(none)]>
```

   If no error is shown and you have access to the interactive **MySQL** console, the connection with TLS succeeds.

   By default, the client automatically uses TLS encryption if the server provides it. Therefore, the **--ssl-ca=ca.crt.pem** and **--ssl-mode=VERIFY_IDENTITY** options are not required, but improve the security because, with these options, the client verifies the identity of the server.

2. Attempt to connect as the **example** user with TLS disabled:

```
# mysql -u example -p -h server.example.com --ssl-mode=DISABLED
ERROR 1045 (28000): Access denied for user 'example'@'server.example.com' (using password: YES)
```

   The server rejected the login attempt because TLS is required for this user but disabled (**--ssl-mode=DISABLED**).

Additional resources

- Configuring TLS on a MySQL server

3.5. GLOBALLY ENABLING TLS ENCRYPTION WITH CA CERTIFICATE VALIDATION IN MYSQL CLIENTS

If your **MySQL** server supports TLS encryption, configure your clients to establish only secure connections and to verify the server certificate. This procedure describes how to enable TLS support for all users on the server.

3.5.1. Configuring the MySQL client to use TLS encryption by default

On RHEL, you can globally configure that the **MySQL** client uses TLS encryption and verifies that the Common Name (CN) in the server certificate matches the hostname the user connects to. This prevents man-in-the-middle attacks.

Prerequisites

- The **MySQL** server has TLS support enabled.
- The CA certificate is stored in the **/etc/pki/tls/certs/ca.crt.pem** file on the client.

Procedure

- Create the **/etc/my.cnf.d/mysql-client-tls.cnf** file with the following content:

```
[client]
ssl-mode=VERIFY_IDENTITY
ssl-ca=/etc/pki/tls/certs/ca.crt.pem
```

These settings define that the **MySQL** client uses TLS encryption and that the client compares the hostname with the CN in the server certificate (**ssl-mode=VERIFY_IDENTITY**). Additionally, it specifies the path to the CA certificate (**ssl-ca**).

Verification

- Connect to the server using the hostname, and display the server status:

```
# mysql -u root -p -h server.example.com -e status
...
SSL:        Cipher in use is TLS_AES_256_GCM_SHA384
```

If the **SSL** entry contains **Cipher in use is…**, the connection is encrypted.

Note that the user you use in this command has permissions to authenticate remotely.

If the hostname you connect to does not match the hostname in the TLS certificate of the server, the **ssl-mode=VERIFY_IDENTITY** parameter causes the connection to fail. For example, if you connect to **localhost**:

```
# mysql -u root -p -h localhost -e status
ERROR 2026 (HY000): SSL connection error: error:0A000086:SSL routines::certificate verify failed
```

Additional resources

- The **--ssl*** parameter descriptions in the **mysql(1)** man page on your system

3.6. BACKING UP MYSQL DATA

There are two main ways to back up data from a **MySQL** database:

Logical backup

Logical backup consists of the SQL statements necessary to restore the data. This type of backup exports information and records in plain text files.
The main advantage of logical backup over physical backup is portability and flexibility. The data can be restored on other hardware configurations, **MySQL** versions or Database Management System (DBMS), which is not possible with physical backups.

Note that logical backup can only be performed if the **mysqld.service** is running. Logical backup does not include log and configuration files.

Physical backup

Physical backup consists of copies of files and directories that store the content.
Physical backup has the following advantages compared to logical backup:

- Output is more compact.
- Backup is smaller in size.
- Backup and restore are faster.
- Backup includes log and configuration files.
  Note that physical backup must be performed when the **mysqld.service** is not running or all tables in the database are locked to prevent changes during the backup.

You can use one of the following **MySQL** backup approaches to back up data from a **MySQL** database:

- Logical backup with **mysqldump**
- File system backup
- Replication as a backup solution

3.6.1. Performing logical backup with mysqldump

The **mysqldump** client is a backup utility, which can be used to dump a database or a collection of databases for the purpose of a backup or transfer to another database server. The output of **mysqldump** typically consists of SQL statements to re-create the server table structure, populate it with data, or both. **mysqldump** can also generate files in other formats, including XML and delimited text formats, such as CSV.

To perform the **mysqldump** backup, you can use one of the following options:

- Back up one or more selected databases
- Back up all databases

- Back up a subset of tables from one database

Procedure

- To dump a single database, run:

```
# mysqldump [options] --databases db_name > backup-file.sql
```

- To dump multiple databases at once, run:

```
# mysqldump [options] --databases db_name1 [db_name2 ...] > backup-file.sql
```

- To dump all databases, run:

```
# mysqldump [options] --all-databases > backup-file.sql
```

- To load one or more dumped full databases back into a server, run:

```
# mysql < backup-file.sql
```

- To load a database to a remote **MySQL** server, run:

```
# mysql --host=remote_host < backup-file.sql
```

- To dump a literal,subset of tables from one database, add a list of the chosen tables at the end of the **mysqldump** command:

```
# mysqldump [options] db_name [tbl_name ...] > backup-file.sql
```

- To load a literal,subset of tables dumped from one database, run:

```
# mysql db_name < backup-file.sql
```

NOTE

The *db_name* database must exist at this point.

- To see a list of the options that **mysqldump** supports, run:

```
$ mysqldump --help
```

Additional resources

- Logical backup with mysqldump

3.6.2. Performing file system backup

To create a file system backup of **MySQL** data files, copy the content of the **MySQL** data directory to your backup location.

To back up also your current configuration or the log files, use the optional steps of the following procedure.

Procedure

1. Stop the **mysqld** service:

```
# systemctl stop mysqld.service
```

2. Copy the data files to the required location:

```
# cp -r /var/lib/mysql /backup-location
```

3. Optional: Copy the configuration files to the required location:

```
# cp -r /etc/my.cnf /etc/my.cnf.d /backup-location/configuration
```

4. Optional: Copy the log files to the required location:

```
# cp /var/log/mysql/* /backup-location/logs
```

5. Start the **mysqld** service:

```
# systemctl start mysqld.service
```

6. When loading the backed up data from the backup location to the **/var/lib/mysql** directory, ensure that **mysql:mysql** is an owner of all data in **/var/lib/mysql**:

```
# chown -R mysql:mysql /var/lib/mysql
```

3.6.3. Replication as a backup solution

Replication is an alternative backup solution for source servers. If a source server replicates to a replica server, backups can be run on the replica without any impact on the source. The source can still run while you shut down the replica and back the data up from the replica.

For instructions on how to replicate a **MySQL** database, see Replicating MySQL.

WARNING

Replication itself is not a sufficient backup solution. Replication protects source servers against hardware failures, but it does not ensure protection against data loss. It is recommended that you use any other backup solution on the replica together with this method.

Additional resources

- MySQL replication documentation

3.7. MIGRATING TO A RHEL 9 VERSION OF MYSQL 8.0

RHEL 8 contains the **MySQL 8.0**, **MariaDB 10.3**, and **MariaDB 10.5** implementations of a server from the MySQL databases family. RHEL 9 provides **MySQL 8.0** and **MariaDB 10.5**.

This procedure describes migration from a RHEL 8 version of **MySQL 8.0** to a RHEL 9 version of **MySQL 8.0** using the **mysql_upgrade** utility. The **mysql_upgrade** utility is provided by the **mysql-server** package.

Prerequisites

- Before performing the upgrade, back up all your data stored in the **MySQL** databases. See Backing up MySQL data.

Procedure

1. Ensure that the **mysql-server** package is installed on the RHEL 9 system:

```
# dnf install mysql-server
```

2. Ensure that the **mysqld** service is not running on either of the source and target systems at the time of copying data:

```
# systemctl stop mysqld.service
```

3. Copy the data from the source location to the **/var/lib/mysql/** directory on the RHEL 9 target system.

4. Set the appropriate permissions and SELinux context for copied files on the target system:

```
# restorecon -vr /var/lib/mysql
```

5. Ensure that **mysql:mysql** is an owner of all data in the **/var/lib/mysql** directory:

```
# chown -R mysql:mysql /var/lib/mysql
```

6. Start the **MySQL** server on the target system:

```
# systemctl start mysqld.service
```

Note: In earlier versions of **MySQL**, the **mysql_upgrade** command was needed to check and repair internal tables. This is now done automatically when you start the server.

3.8. UPGRADING FROM MYSQL 8.0 TO MYSQL 8.4

If you currently use MySQL 8.0 on Red Hat Enterprise Linux 9 and require features that are only available in a newer version, you can upgrade to MySQL 8.4. RHEL 9.6 and later provides MySQL 8.4 as an alternative Application Stream.

3.8.1. Notable differences between MySQL 8.0 and MySQL 8.4

Significant changes between **MySQL 8.0** and **MySQL 8.4** include:

- Enhancements to password management: Administrators can now enforce password expiration, lengths, strength, reuse policy, and other password-related settings.
- Authentication: The **caching_sha2_password** plugin is now the default and replaces the **mysql_native_password** plugin to increase the security.
- Backup Compatibility: The **mysqldump** utility now provides an **--output-as-version** option which enables logical backups to be compatible with older MySQL versions.
- **EXPLAIN**: This statement can now display results in JSON format.
- Deprecation and removal: The following features, which were previously deprecated have been removed:
  - The **mysqlpump** utility
  - The **mysql_native_password** authentication plugin
  - The **mysql_upgrade** utility

3.8.2. Upgrading from a RHEL 9 version of MySQL 8.0 to MySQL 8.4

RHEL 9 contains **MySQL 8.0** provided by the **mysql-server** RPM package. If you want to upgrade to **MySQL 8.4**, switch to the **mysql:8.4** module stream.

Prerequisites

- You run **MySQL 8.0** on RHEL 9.
- You created a backup of the MySQL databases.

Procedure

1. Stop the **mysqld** service:

```
# systemctl stop mysqld.service
```

2. Switch from the non-modular **MySQL 8.0** to the modular **MySQL 8.4** stream:

```
# dnf module switch-to mysql:8.4
```

3. Adjust the configuration so that files located in the **/etc/my.cnf.d/** directory include only settings valid for MySQL 8.4. For details, see upstream documentation.
4. Start the **mysqld** service:

```
# systemctl start mysqld.service
```

When the service starts, **MySQL** automatically checks, repairs, and updates internal tables.

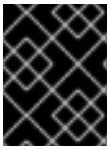
3.9. REPLICATING MYSQL WITH TLS ENCRYPTION

MySQL provides various configuration options for replication, ranging from basic to advanced. This section describes a transaction-based way to replicate in **MySQL** on freshly installed **MySQL** servers

using global transaction identifiers (GTIDs). Using GTIDs simplifies transaction identification and consistency verification.

To set up replication in **MySQL**, you must:

- Configure a source server
- Configure a replica server
- Create a replication user on the source server
- Connect the replica server to the source server



IMPORTANT

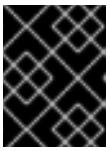
If you want to use existing **MySQL** servers for replication, you must first synchronize data. See the upstream documentation for more information.

3.9.1. Configuring a MySQL source server

You can set configuration options required for a **MySQL** source server to properly run and replicate all changes made on the database server through the TLS protocol.

Prerequisites

- The source server is installed.
- The source server has TLS set up.



IMPORTANT

The source and replica certificates must be signed by the same certificate authority.

Procedure

1. Include the following options in the **/etc/my.cnf.d/mysql-server.cnf** file under the **[mysqld]** section:
   - **bind-address=*source_ip_adress***
     This option is required for connections made from replicas to the source.
   - **server-id=*id***
     The *id* must be unique.
   - **log_bin=*path_to_source_server_log***
     This option defines a path to the binary log file of the **MySQL** source server. For example: **log_bin=/var/log/mysql/mysql-bin.log**.
   - **gtid_mode=ON**
     This option enables global transaction identifiers (GTIDs) on the server.
   - **enforce-gtid-consistency=ON**
     The server enforces GTID consistency by allowing execution of only statements that can be safely logged using a GTID.

- *Optional:* **binlog_do_db=*db_name***
  Use this option if you want to replicate only selected databases. To replicate more than one selected database, specify each of the databases separately:

  ```
  binlog_do_db=db_name1
  binlog_do_db=db_name2
  binlog_do_db=db_name3
  ```

- *Optional:* **binlog_ignore_db=*db_name***
  Use this option to exclude a specific database from replication.

2. Restart the **mysqld** service:

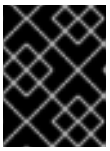
   ```
   # systemctl restart mysqld.service
   ```

3.9.2. Configuring a MySQL replica server

You can set configuration options required for a **MySQL** replica server to ensure a successful replication.

Prerequisites

- The replica server is installed.
- The replica server has TLS set up.



IMPORTANT

The source and replica certificates must be signed by the same certificate authority.

Procedure

1. Include the following options in the **/etc/my.cnf.d/mysql-server.cnf** file under the **[mysqld]** section:

   - **server-id=*id***
     The *id* must be unique.
   - **relay-log=*path_to_replica_server_log***
     The relay log is a set of log files created by the **MySQL** replica server during replication.
   - **log_bin=*path_to_replica_sever_log***
     This option defines a path to the binary log file of the **MySQL** replica server. For example: **log_bin=/var/log/mysql/mysql-bin.log**.

     This option is not required in a replica but strongly recommended.
   - **gtid_mode=ON**
     This option enables global transaction identifiers (GTIDs) on the server.
   - **enforce-gtid-consistency=ON**
     The server enforces GTID consistency by allowing execution of only statements that can be safely logged using a GTID.

- **log-replica-updates=ON**
  This option ensures that updates received from the source server are logged in the replica's binary log.

- **skip-replica-start=ON**
  This option ensures that the replica server does not start the replication threads when the replica server starts.

- *Optional:* **binlog_do_db=*db_name***
  Use this option if you want to replicate only certain databases. To replicate more than one database, specify each of the databases separately:

  ```
  binlog_do_db=db_name1
  binlog_do_db=db_name2
  binlog_do_db=db_name3
  ```

- *Optional:* **binlog_ignore_db=*db_name***
  Use this option to exclude a specific database from replication.

2. Restart the **mysqld** service:

```
# systemctl restart mysqld.service
```

3.9.3. Creating a replication user on the MySQL source server

You must create a replication user and grant this user permissions required for replication traffic. This procedure shows how to create a replication user with appropriate permissions. Execute these steps only on the source server.

Prerequisites

- The source server is installed and configured as described in Configuring a MySQL source server.

Procedure

1. Create a replication user:

```
mysql> CREATE USER 'replication_user'@'replica_server_hostname' IDENTIFIED
WITH mysql_native_password BY 'password';
```

2. Grant the user replication permissions:

```
mysql> GRANT REPLICATION SLAVE ON *.* TO
'replication_user'@'replica_server_hostname';
```

3. Reload the grant tables in the **MySQL** database:

```
mysql> FLUSH PRIVILEGES;
```

4. Set the source server to read-only state:

```
mysql> SET @@GLOBAL.read_only = ON;
```

-

3.9.4. Connecting the replica server to the source server

On the **MySQL** replica server, you must configure credentials and the address of the source server. Use the following procedure to implement the replica server.

Prerequisites

- The source server is installed and configured as described in Configuring a MySQL source server.
- The replica server is installed and configured as described in Configuring a MySQL replica server.
- You have created a replication user. See Creating a replication user on the MySQL source server.

Procedure

1. Set the replica server to read-only state:

```
mysql> SET @@GLOBAL.read_only = ON;
```

2. Configure the replication source:

```
mysql> CHANGE REPLICATION SOURCE TO
    SOURCE_HOST='source_hostname',
    SOURCE_USER='replication_user',
    SOURCE_PASSWORD='password',
    SOURCE_AUTO_POSITION=1,
    SOURCE_SSL=1,
    SOURCE_SSL_CA='path_to_ca_on_source',
    SOURCE_SSL_CAPATH='path_to_directory_with_certificates',
    SOURCE_SSL_CERT='path_to_source_certificate',
    SOURCE_SSL_KEY='path_to_source_key';
```

3. Start the replica thread in the **MySQL** replica server:

```
mysql> START REPLICA;
```

4. Unset the read-only state on both the source and replica servers:

```
mysql> SET @@GLOBAL.read_only = OFF;
```

5. Optional: Inspect the status of the replica server for debugging purposes:

```
mysql> SHOW REPLICA STATUS\G;
```

NOTE

If the replica server fails to start or connect, you can skip a certain number of events following the binary log file position displayed in the output of the **SHOW MASTER STATUS** command. For example, skip the first event from the defined position:

```
mysql> SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1;
```

and try to start the replica server again.

6. Optional: Stop the replica thread in the replica server:

```
mysql> STOP REPLICA;
```

3.9.5. Verifying replication on a MySQL server

After you configured replication among multiple MySQL servers, you should verify that that it works.

Procedure

1. Create an example database on the source server:

```
mysql> CREATE DATABASE test_db_name;
```

2. Verify that the ***test_db_name*** database replicates on the replica server.

3. Display status information about the binary log files of the **MySQL** server by executing the following command on either of the source or replica servers:

```
mysql> SHOW MASTER STATUS;
```

The **Executed_Gtid_Set** column, which shows a set of GTIDs for transactions executed on the source, must not be empty.

NOTE

The same set of GTIDs is displayed in the **Executed_Gtid_Set** row when you use the **SHOW REPLICA STATUS** on the replica server.

3.9.5.1. Additional resources

- MySQL Replication documentation
- Replication with Global Transaction Identifiers

3.10. DEVELOPING MYSQL CLIENT APPLICATIONS

Red Hat recommends developing your **MySQL** client applications against the **MariaDB** client library. The communication protocol between client and server is compatible between **MariaDB** and **MySQL**. The **MariaDB** client library works for most common **MySQL** scenarios, with the exception of a limited number of features specific to the **MySQL** implementation.

The development files and programs necessary to build applications against the MariaDB client library are provided by the **mariadb-connector-c-devel** package.

Instead of using a direct library name, use the **mariadb_config** program, which is distributed in the **mariadb-connector-c-devel** package. This program ensures that the correct build flags are returned.

CHAPTER 4. USING POSTGRESQL

The **PostgreSQL** server is an open source robust and highly-extensible database server based on the SQL language. The **PostgreSQL** server provides an object-relational database system that can manage extensive datasets and a high number of concurrent users. For these reasons, **PostgreSQL** servers can be used in clusters to manage high amounts of data.

The **PostgreSQL** server includes features for ensuring data integrity, building fault-tolerant environments and applications. With the **PostgreSQL** server, you can extend a database with your own data types, custom functions, or code from different programming languages without the need to recompile the database.

Learn how to install and configure **PostgreSQL** on a RHEL system, how to back up **PostgreSQL** data, and how to migrate from an earlier **PostgreSQL** version.

4.1. INSTALLING POSTGRESQL

RHEL 9 provides **PostgreSQL 13** as the initial version of this Application Stream, which you can install easily as an RPM package.

Additional **PostgreSQL** versions are provided as modules with a shorter life cycle in minor releases of RHEL 9:

- RHEL 9.2 introduced **PostgreSQL 15** as the **postgresql:15** module stream
- RHEL 9.4 introduced **PostgreSQL 16** as the **postgresql:16** module stream

To install **PostgreSQL**, use the following procedure.

NOTE

By design, it is impossible to install more than one version (stream) of the same module in parallel. Therefore, you must choose only one of the available streams from the **postgresql** module. You can use different versions of the **PostgreSQL** database server in containers, see Running multiple PostgreSQL versions in containers.

Procedure

1. Install the **PostgreSQL** server packages:
   - For **PostgreSQL 13** from the RPM package:

     ```
     # dnf install postgresql-server
     ```

   - For **PostgreSQL 15** or **PostgreSQL 16** by selecting stream (version) 15 or 16 from the **postgresql** module and specifying the **server** profile, for example:

     ```
     # dnf module install postgresql:16/server
     ```

     The **postgres** superuser is created automatically.
2. Initialize the database cluster:

   ```
   # postgresql-setup --initdb
   ```

Red Hat recommends storing the data in the default **/var/lib/pgsql/data** directory.

3. Start the **postgresql** service:

```
# systemctl start postgresql.service
```

4. Enable the **postgresql** service to start at boot:

```
# systemctl enable postgresql.service
```

IMPORTANT

If you want to upgrade from an earlier **postgresql** stream within RHEL 9, follow both procedures described in Switching to a later stream and in Migrating to a RHEL 9 version of PostgreSQL.

4.2. RUNNING MULTIPLE POSTGRESQL VERSIONS IN CONTAINERS

To run different versions of **PostgreSQL** on the same host, run them in containers because you cannot install multiple versions (streams) of the same module in parallel.

This procedure includes **PostgreSQL 13** and **PostgreSQL 15** as examples but you can use any **PostgreSQL** container version available in the Red Hat Ecosystem Catalog.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Use your Red Hat Customer Portal account to authenticate to the **registry.redhat.io** registry:

```
# podman login registry.redhat.io
```

Skip this step if you are already logged in to the container registry.

2. Run **PostgreSQL 13** in a container:

```
$ podman run -d --name <container_name> -e POSTGRESQL_USER=<user_name> -e
POSTGRESQL_PASSWORD=<password> -e
POSTGRESQL_DATABASE=<database_name> -p <host_port_1>:5432
rhel9/postgresql-13
```

For more information about the usage of this container image, see the Red Hat Ecosystem Catalog.

3. Run **PostgreSQL 15** in a container:

```
$ podman run -d --name <container_name> -e POSTGRESQL_USER=<user_name> -e
POSTGRESQL_PASSWORD=<password> -e
POSTGRESQL_DATABASE=<database_name> -p <host_port_2>:5432
rhel9/postgresql-15
```

For more information about the usage of this container image, see the Red Hat Ecosystem Catalog.

4. Run **PostgreSQL 16** in a container:

```
$ podman run -d --name <container_name> -e POSTGRESQL_USER=<user_name> -e
POSTGRESQL_PASSWORD=<password> -e
POSTGRESQL_DATABASE=<database_name> -p <host_port_3>:5432
rhel9/postgresql-16
```

For more information about the usage of this container image, see the Red Hat Ecosystem Catalog.

NOTE

The container names and host ports of the two database servers must differ.

5. To ensure that clients can access the database server on the network, open the host ports in the firewall:

```
# firewall-cmd --permanent --add-port=
{<host_port_1>/tcp,<host_port_2>/tcp,<host_port_3>/tcp,...}
# firewall-cmd --reload
```

Verification

1. Display information about running containers:

```
$ podman ps
```

2. Connect to the database server and log in as root:

```
# psql -u postgres -p -h localhost -P <host_port> --protocol tcp
```

Additional resources

- Building, running, and managing containers
- Browse containers in the Red Hat Ecosystem Catalog

4.3. CREATING POSTGRESQL USERS

PostgreSQL users are of the following types:

- The **postgres** UNIX system user - should be used only to run the **PostgreSQL** server and client applications, such as **pg_dump**. Do not use the **postgres** system user for any interactive work on **PostgreSQL** administration, such as database creation and user management.
- A database superuser - the default **postgres PostgreSQL** superuser is not related to the **postgres** system user. You can limit access of the **postgres** superuser in the **pg_hba.conf** file, otherwise no other permission limitations exist. You can also create other database superusers.
- A role with specific database access permissions:

- A database user - has a permission to log in by default
- A group of users - enables managing permissions for the group as a whole

Roles can own database objects (for example, tables and functions) and can assign object privileges to other roles using SQL commands.

Standard database management privileges include **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **TRUNCATE**, **REFERENCES**, **TRIGGER**, **CREATE**, **CONNECT**, **TEMPORARY**, **EXECUTE**, and **USAGE**.

Role attributes are special privileges, such as **LOGIN**, **SUPERUSER**, **CREATEDB**, and **CREATEROLE**.

IMPORTANT

Red Hat recommends performing most tasks as a role that is not a superuser. A common practice is to create a role that has the **CREATEDB** and **CREATEROLE** privileges and use this role for all routine management of databases and roles.

Prerequisites

- The **PostgreSQL** server is installed.
- The database cluster is initialized.

Procedure

- To create a user, set a password for the user, and assign the user the **CREATEROLE** and **CREATEDB** permissions:

```
postgres=# CREATE USER mydbuser WITH PASSWORD 'mypasswd' CREATEROLE CREATEDB;
```

  Replace *mydbuser* with the username and *mypasswd* with the user's password.

Additional resources

- PostgreSQL database roles
- PostgreSQL privileges
- Configuring PostgreSQL

Example 4.1. Initializing, creating, and connecting to a PostgreSQL database

This example demonstrates how to initialize a PostgreSQL database, create a database user with routine database management privileges, and how to create a database that is accessible from any system account through the database user with management privileges.

1. Install the PosgreSQL server:

```
# dnf install postgresql-server
```

2. Initialize the database cluster:

```
# postgresql-setup --initdb
* Initializing database in '/var/lib/pgsql/data'
* Initialized, logs are in /var/lib/pgsql/initdb_postgresql.log
```

3. Set the password hashing algorithm to **scram-sha-256**.

   a. In the **/var/lib/pgsql/data/postgresql.conf** file, change the following line:

   ```
   #password_encryption = md5              # md5 or scram-sha-256
   ```

   to:

   ```
   password_encryption = scram-sha-256
   ```

   b. In the **/var/lib/pgsql/data/pg_hba.conf** file, change the following line for the IPv4 local connections:

   ```
   host    all             all             127.0.0.1/32            ident
   ```

   to:

   ```
   host    all             all             127.0.0.1/32            scram-sha-256
   ```

4. Start the postgresql service:

   ```
   # systemctl start postgresql.service
   ```

5. Log in as the system user named **postgres**:

   ```
   # su - postgres
   ```

6. Start the **PostgreSQL** interactive terminal:

   ```
   $ psql
   psql (13.7)
   Type "help" for help.

   postgres=#
   ```

7. Optional: Obtain information about the current database connection:

   ```
   postgres=# \conninfo
   You are connected to database "postgres" as user "postgres" via socket in
   "/var/run/postgresql" at port "5432".
   ```

8. Create a user named **mydbuser**, set a password for **mydbuser**, and assign **mydbuser** the **CREATEROLE** and **CREATEDB** permissions:

   ```
   postgres=# CREATE USER mydbuser WITH PASSWORD 'mypasswd' CREATEROLE
   CREATEDB;
   CREATE ROLE
   ```

The **mydbuser** user now can perform routine database management operations: create databases and manage user indexes.

9. Log out of the interactive terminal by using the **\q** meta command:

```
postgres=# \q
```

10. Log out of the **postgres** user session:

```
$ logout
```

11. Log in to the **PostgreSQL** terminal as **mydbuser**, specify the hostname, and connect to the default **postgres** database, which was created during initialization:

```
# psql -U mydbuser -h 127.0.0.1 -d postgres
Password for user mydbuser:
Type the password.
psql (13.7)
Type "help" for help.

postgres=>
```

12. Create a database named **mydatabase**:

```
postgres=> CREATE DATABASE mydatabase;
CREATE DATABASE
postgres=>
```

13. Log out of the session:

```
postgres=# \q
```

14. Connect to mydatabase as **mydbuser**:

```
# psql -U mydbuser -h 127.0.0.1 -d mydatabase
Password for user mydbuser:
psql (13.7)
Type "help" for help.
mydatabase=>
```

15. Optional: Obtain information about the current database connection:

```
mydatabase=> \conninfo
You are connected to database "mydatabase" as user "mydbuser" on host "127.0.0.1" at
port "5432".
```

4.4. CONFIGURING POSTGRESQL

In a **PostgreSQL** database, all data and configuration files are stored in a single directory called a database cluster. Red Hat recommends storing all data, including configuration files, in the default **/var/lib/pgsql/data/** directory.

PostgreSQL configuration consists of the following files:

- **postgresql.conf** - is used for setting the database cluster parameters.
- **postgresql.auto.conf** - holds basic **PostgreSQL** settings similarly to **postgresql.conf**. However, this file is under the server control. It is edited by the **ALTER SYSTEM** queries, and cannot be edited manually.
- **pg_ident.conf** - is used for mapping user identities from external authentication mechanisms into the **PostgreSQL** user identities.
- **pg_hba.conf** - is used for configuring client authentication for **PostgreSQL** databases.

To change the **PostgreSQL** configuration, use the following procedure.

Procedure

1. Edit the respective configuration file, for example, **/var/lib/pgsql/data/postgresql.conf**.
2. Restart the **postgresql** service so that the changes become effective:

```
# systemctl restart postgresql.service
```

Example 4.2. Configuring PostgreSQL database cluster parameters

This example shows basic settings of the database cluster parameters in the **/var/lib/pgsql/data/postgresql.conf** file.

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
password_encryption = scram-sha-256
```

Example 4.3. Setting client authentication in PostgreSQL

This example demonstrates how to set client authentication in the **/var/lib/pgsql/data/pg_hba.conf** file.

```
# TYPE    DATABASE       USER        ADDRESS             METHOD
local     all            all                             trust
host      postgres       all         192.168.93.0/24     ident
host      all            all         .example.com        scram-sha-256
```

4.5. CONFIGURING TLS ENCRYPTION ON A POSTGRESQL SERVER

By default, **PostgreSQL** uses unencrypted connections. For more secure connections, you can enable Transport Layer Security (TLS) support on the **PostgreSQL** server and configure your clients to establish encrypted connections.

Prerequisites

Prerequisites

- The **PostgreSQL** server is installed.
- The database cluster is initialized.
- If the server runs RHEL 9.2 or later and the FIPS mode is enabled, clients must either support the Extended Master Secret (EMS) extension or use TLS 1.3. TLS 1.2 connections without EMS fail. For more information, see the Red Hat Knowledgebase solution TLS extension "Extended Master Secret" enforced on RHEL 9.2 and later.

Procedure

1. Install the OpenSSL library:

```
# dnf install openssl
```

2. Generate a TLS certificate and a key:

```
# openssl req -new -x509 -days 365 -nodes -text -out server.crt \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

   Replace *dbhost.yourdomain.com with* your database host and domain name.

3. Copy your signed certificate and your private key to the required locations on the database server:

```
# cp server.{key,crt} /var/lib/pgsql/data/.
```

4. Change the owner and group ownership of the signed certificate and your private key to the **postgres** user:

```
# chown postgres:postgres /var/lib/pgsql/data/server.{key,crt}
```

5. Restrict the permissions for your private key so that it is readable only by the owner:

```
# chmod 0400 /var/lib/pgsql/data/server.key
```

6. Set the password hashing algorithm to **scram-sha-256** by changing the following line in the **/var/lib/pgsql/data/postgresql.conf** file:

```
#password_encryption = md5              # md5 or scram-sha-256
```

   to:

```
password_encryption = scram-sha-256
```

7. Configure PostgreSQL to use SSL/TLS by changing the following line in the **/var/lib/pgsql/data/postgresql.conf** file:

```
#ssl = off
```

   to:

```

```

```
ssl=on
```

8. Restrict access to all databases to accept only connections from clients using TLS by changing the following line for the IPv4 local connections in the **/var/lib/pgsql/data/pg_hba.conf** file:

```
host  all  all  127.0.0.1/32  ident
```

to:

```
hostssl  all  all  127.0.0.1/32  scram-sha-256
```

Alternatively, you can restrict access for a single database and a user by adding the following new line:

```
hostssl mydatabase mydbuser 127.0.0.1/32  scram-sha-256
```

Replace *mydatabase* with the database name and *mydbuser* with the username.

9. Make the changes effective by restarting the **postgresql** service:

```
# systemctl restart postgresql.service
```

Verification

- To manually verify that the connection is encrypted:

1. Connect to the **PostgreSQL** database as the *mydbuser* user, specify the hostname and the database name:

```
$ psql -U mydbuser -h 127.0.0.1 -d mydatabase
Password for user mydbuser:
```

Replace *mydatabase* with the database name and *mydbuser* with the username.

2. Obtain information about the current database connection:

```
mydbuser=> \conninfo
You are connected to database "mydatabase" as user "mydbuser" on host "127.0.0.1" at port "5432".
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
```

- You can write a simple application that verifies whether a connection to **PostgreSQL** is encrypted. This example demonstrates such an application written in C that uses the **libpq** client library, which is provided by the **libpq-devel** package:

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>

int main(int argc, char* argv[])
{
//Create connection
```

```
PGconn* connection = PQconnectdb("hostaddr=127.0.0.1 password=mypassword port=5432
dbname=mydatabase user=mydbuser");

if (PQstatus(connection) ==CONNECTION_BAD)
    {
    printf("Connection error\n");
    PQfinish(connection);
    return -1; //Execution of the program will stop here
    }
    printf("Connection ok\n");
    //Verify TLS
    if (PQsslInUse(connection)){
     printf("TLS in use\n");
     printf("%s\n", PQsslAttribute(connection,"protocol"));
    }
    //End connection
    PQfinish(connection);
    printf("Disconnected\n");
    return 0;
}
```

Replace *mypassword* with the password, *mydatabase* with the database name, and *mydbuser* with the username.

NOTE

You must load the **pq** libraries for compilation by using the **-lpq** option. For example, to compile the application by using the GCC compiler:

```
$ gcc source_file.c -lpq -o myapplication
```

where the *source_file.c* contains the example code above, and *myapplication* is the name of your application for verifying secured **PostgreSQL** connection.

Example 4.4. Initializing, creating, and connecting to a PostgreSQL database using TLS encryption

This example demonstrates how to initialize a PostgreSQL database, create a database user and a database, and how to connect to the database using a secured connection.

1. Install the PosgreSQL server:

```
# dnf install postgresql-server
```

2. Initialize the database cluster:

```
# postgresql-setup --initdb
* Initializing database in '/var/lib/pgsql/data'
* Initialized, logs are in /var/lib/pgsql/initdb_postgresql.log
```

3. Install the OpenSSL library:

```
# dnf install openssl
```

4. Generate a TLS certificate and a key:

```
# openssl req -new -x509 -days 365 -nodes -text -out server.crt \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

Replace *dbhost.yourdomain.com with* your database host and domain name.

5. Copy your signed certificate and your private key to the required locations on the database server:

```
# cp server.{key,crt} /var/lib/pgsql/data/.
```

6. Change the owner and group ownership of the signed certificate and your private key to the **postgres** user:

```
# chown postgres:postgres /var/lib/pgsql/data/server.{key,crt}
```

7. Restrict the permissions for your private key so that it is readable only by the owner:

```
# chmod 0400 /var/lib/pgsql/data/server.key
```

8. Set the password hashing algorithm to **scram-sha-256**. In the **/var/lib/pgsql/data/postgresql.conf** file, change the following line:

```
#password_encryption = md5              # md5 or scram-sha-256
```

to:

```
password_encryption = scram-sha-256
```

9. Configure PostgreSQL to use SSL/TLS. In the **/var/lib/pgsql/data/postgresql.conf** file, change the following line:

```
#ssl = off
```

to:

```
ssl=on
```

10. Start the **postgresql** service:

```
# systemctl start postgresql.service
```

11. Log in as the system user named **postgres**:

```
# su - postgres
```

12. Start the **PostgreSQL** interactive terminal as the **postgres** user:

```
$ psql -U postgres
psql (13.7)
Type "help" for help.
```

```
postgres=#
```

13. Create a user named **mydbuser** and set a password for **mydbuser**:

```
postgres=# CREATE USER mydbuser WITH PASSWORD 'mypasswd';
CREATE ROLE
postgres=#
```

14. Create a database named **mydatabase**:

```
postgres=# CREATE DATABASE mydatabase;
CREATE DATABASE
postgres=#
```

15. Grant all permissions to the **mydbuser** user:

```
postgres=# GRANT ALL PRIVILEGES ON DATABASE mydatabase TO mydbuser;
GRANT
postgres=#
```

16. Log out of the interactive terminal:

```
postgres=# \q
```

17. Log out of the **postgres** user session:

```
$ logout
```

18. Restrict access to all databases to accept only connections from clients using TLS by changing the following line for the IPv4 local connections in the **/var/lib/pgsql/data/pg_hba.conf** file:

```
host  all  all  127.0.0.1/32  ident
```

to:

```
hostssl  all  all  127.0.0.1/32  scram-sha-256
```

19. Make the changes effective by restarting the **postgresql** service:

```
# systemctl restart postgresql.service
```

20. Connect to the **PostgreSQL** database as the **mydbuser** user, specify the hostname and the database name:

```
$ psql -U mydbuser -h 127.0.0.1 -d mydatabase
Password for user mydbuser:
psql (13.7)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
compression: off)
```

```
Type "help" for help.

mydatabase=>
```

4.6. BACKING UP POSTGRESQL DATA

To back up **PostgreSQL** data, use one of the following approaches:

- SQL dump
- File system level backup
- Continuous archiving

4.6.1. Backing up PostgreSQL data with an SQL dump

The SQL dump method is based on generating a dump file with SQL commands. When a dump is uploaded back to the database server, it recreates the database in the same state as it was at the time of the dump.

The SQL dump is ensured by the following **PostgreSQL** client applications:

- **pg_dump** dumps a single database without cluster-wide information about roles or tablespaces
- **pg_dumpall** dumps each database in a given cluster and preserves cluster-wide data, such as role and tablespace definitions.

By default, the **pg_dump** and **pg_dumpall** commands write their results into the standard output. To store the dump in a file, redirect the output to an SQL file. The resulting SQL file can be either in a text format or in other formats that allow for parallelism and for more detailed control of object restoration.

You can perform the SQL dump from any remote host that has access to the database.

4.6.1.1. Advantages and disadvantages of an SQL dump

An SQL dump has the following advantages compared to other **PostgreSQL** backup methods:

- An SQL dump is the only **PostgreSQL** backup method that is not server version-specific. The output of the **pg_dump** utility can be reloaded into later versions of **PostgreSQL**, which is not possible for file system level backups or continuous archiving.
- An SQL dump is the only method that works when transferring a database to a different machine architecture, such as going from a 32-bit to a 64-bit server.
- An SQL dump provides internally consistent dumps. A dump represents a snapshot of the database at the time **pg_dump** began running.
- The **pg_dump** utility does not block other operations on the database when it is running.

A disadvantage of an SQL dump is that it takes more time compared to file system level backup.

4.6.1.2. Performing an SQL dump using pg_dump

To dump a single database without cluster-wide information, use the **pg_dump** utility.

Prerequisites

- You must have read access to all tables that you want to dump. To dump the entire database, you must run the commands as the **postgres** superuser or a user with database administrator privileges.

Procedure

- Dump a database without cluster-wide information:

```
$ pg_dump dbname > dumpfile
```

  To specify which database server **pg_dump** will contact, use the following command-line options:

  - The **-h** option to define the host.
    The default host is either the local host or what is specified by the **PGHOST** environment variable.
  - The **-p** option to define the port.
    The default port is indicated by the **PGPORT** environment variable or the compiled-in default.

4.6.1.3. Performing an SQL dump using pg_dumpall

To dump each database in a given database cluster and to preserve cluster-wide data, use the **pg_dumpall** utility.

Prerequisites

- You must run the commands as the **postgres** superuser or a user with database administrator privileges.

Procedure

- Dump all databases in the database cluster and preserve cluster-wide data:

```
$ pg_dumpall > dumpfile
```

To specify which database server **pg_dumpall** will contact, use the following command-line options:

- The **-h** option to define the host.
  The default host is either the local host or what is specified by the **PGHOST** environment variable.
- The **-p** option to define the port.
  The default port is indicated by the **PGPORT** environment variable or the compiled-in default.
- The **-l** option to define the default database.
  This option enables you to choose a default database different from the **postgres** database created automatically during initialization.

4.6.1.4. Restoring a database dumped using pg_dump

To restore a database from an SQL dump that you dumped using the **pg_dump** utility, follow the steps below.

Prerequisites

- You must run the commands as the **postgres** superuser or a user with database administrator privileges.

Procedure

1. Create a new database:

```
$ createdb dbname
```

2. Verify that all users who own objects or were granted permissions on objects in the dumped database already exist. If such users do not exist, the restore fails to recreate the objects with the original ownership and permissions.

3. Run the **psql** utility to restore a text file dump created by the **pg_dump** utility:

```
$ psql dbname < dumpfile
```

where **dumpfile** is the output of the **pg_dump** command. To restore a non-text file dump, use the **pg_restore** utility instead:

```
$ pg_restore non-plain-text-file
```

4.6.1.5. Restoring databases dumped using pg_dumpall

To restore data from a database cluster that you dumped using the **pg_dumpall** utility, follow the steps below.

Prerequisites

- You must run the commands as the **postgres** superuser or a user with database administrator privileges.

Procedure

1. Ensure that all users who own objects or were granted permissions on objects in the dumped databases already exist. If such users do not exist, the restore fails to recreate the objects with the original ownership and permissions.

2. Run the **psql** utility to restore a text file dump created by the **pg_dumpall** utility:

```
$ psql < dumpfile
```

where **dumpfile** is the output of the **pg_dumpall** command.

4.6.1.6. Performing an SQL dump of a database on another server

Dumping a database directly from one server to another is possible because **pg_dump** and **psql** can write to and read from pipes.

Procedure

- To dump a database from one server to another, run:

```
$ pg_dump -h host1 dbname | psql -h host2 dbname
```

4.6.1.7. Handling SQL errors during restore

By default, **psql** continues to execute if an SQL error occurs, causing the database to restore only partially.

To change the default behavior, use one of the following approaches when restoring a dump.

Prerequisites

- You must run the commands as the **postgres** superuser or a user with database administrator privileges.

Procedure

- Make **psql** exit with an exit status of 3 if an SQL error occurs by setting the **ON_ERROR_STOP** variable:

```
$ psql --set ON_ERROR_STOP=on dbname < dumpfile
```

- Specify that the whole dump is restored as a single transaction so that the restore is either fully completed or canceled.
  - When restoring a text file dump using the **psql** utility:

```
$ psql -1
```

  - When restoring a non-text file dump using the **pg_restore** utility:

```
$ pg_restore -e
```

    Note that when using this approach, even a minor error can cancel a restore operation that has already run for many hours.

Additional resources

- PostgreSQL Documentation - SQL dump

4.6.2. Backing up PostgreSQL data with a file system level backup

To create a file system level backup, copy **PostgreSQL** database files to another location. For example, you can use any of the following approaches:

- Create an archive file using the **tar** utility.
- Copy the files to a different location using the **rsync** utility.
- Create a consistent snapshot of the data directory.

4.6.2.1. Advantages and limitations of file system backing up

File system level backing up has the following advantage compared to other **PostgreSQL** backup methods:

- File system level backing up is usually faster than an SQL dump.

File system level backing up has the following limitations compared to other **PostgreSQL** backup methods:

- This backing up method is not suitable when you want to upgrade from RHEL 8 to RHEL 9 and migrate your data to the upgraded system. File system level backup is specific to an architecture and a RHEL major version. You can restore your data on your RHEL 8 system if the upgrade is not successful but you cannot restore the data on a RHEL 9 system.
- The database server must be shut down before backing up and restoring data.
- Backing up and restoring certain individual files or tables is impossible. Backing up a file system works only for complete backing up and restoring of an entire database cluster.

4.6.2.2. Performing file system level backing up

To perform file system level backing up, use the following procedure.

Procedure

1. Choose the location of a database cluster and initialize this cluster:

```
# postgresql-setup --initdb
```

2. Stop the postgresql service:

```
# systemctl stop postgresql.service
```

3. Use any method to create a file system backup, for example a **tar** archive:

```
$ tar -cf backup.tar /var/lib/pgsql/data/
```

4. Start the postgresql service:

```
# systemctl start postgresql.service
```

Additional resources

- PostgreSQL Documentation - file system level backup

4.6.3. Backing up PostgreSQL data by continuous archiving

PostgreSQL records every change made to the database's data files into a write ahead log (WAL) file that is available in the **pg_wal/** subdirectory of the cluster's data directory. This log is intended primarily for a crash recovery. After a crash, the log entries made since the last checkpoint can be used for restoring the database to a consistency.

The continuous archiving method, also known as an online backup, combines the WAL files with a copy of the database cluster in the form of a base backup performed on a running server or a file system level backup.

If a database recovery is needed, you can restore the database from the copy of the database cluster and then replay log from the backed up WAL files to bring the system to the current state.

With the continuous archiving method, you must keep a continuous sequence of all archived WAL files that extends at minimum back to the start time of your last base backup. Therefore the ideal frequency of base backups depends on:

- The storage volume available for archived WAL files.
- The maximum possible duration of data recovery in situations when recovery is necessary. In cases with a long period since the last backup, the system replays more WAL segments, and the recovery therefore takes more time.

NOTE

You cannot use **pg_dump** and **pg_dumpall** SQL dumps as a part of a continuous archiving backup solution. SQL dumps produce logical backups and do not contain enough information to be used by a WAL replay.

4.6.3.1. Advantages and disadvantages of continuous archiving

Continuous archiving has the following advantages compared to other **PostgreSQL** backup methods:

- With the continuous backup method, it is possible to use a base backup that is not entirely consistent because any internal inconsistency in the backup is corrected by the log replay. Therefore you can perform a base backup on a running **PostgreSQL** server.
- A file system snapshot is not needed; **tar** or a similar archiving utility is sufficient.
- Continuous backup can be achieved by continuing to archive the WAL files because the sequence of WAL files for the log replay can be indefinitely long. This is particularly valuable for large databases.
- Continuous backup supports point-in-time recovery. It is not necessary to replay the WAL entries to the end. The replay can be stopped at any point and the database can be restored to its state at any time since the base backup was taken.
- If the series of WAL files are continuously available to another machine that has been loaded with the same base backup file, it is possible to restore the other machine with a nearly-current copy of the database at any point.

Continuous archiving has the following disadvantages compared to other **PostgreSQL** backup methods:

- Continuous backup method supports only restoration of an entire database cluster, not a subset.
- Continuous backup requires extensive archival storage.

4.6.3.2. Setting up WAL archiving

A running **PostgreSQL** server produces a sequence of write ahead log (WAL) records. The server physically divides this sequence into WAL segment files, which are given numeric names that reflect

their position in the WAL sequence. Without WAL archiving, the segment files are reused and renamed to higher segment numbers.

When archiving WAL data, the contents of each segment file are captured and saved at a new location before the segment file is reused. You have multiple options where to save the content, such as an NFS-mounted directory on another machine, a tape drive, or a CD.

Note that WAL records do not include changes to configuration files.

To enable WAL archiving, use the following procedure.

Procedure

1. In the **/var/lib/pgsql/data/postgresql.conf** file:
   a. Set the **wal_level** configuration parameter to **replica** or higher.
   b. Set the **archive_mode** parameter to **on**.
   c. Specify the shell command in the **archive_command** configuration parameter. You can use the **cp** command, another command, or a shell script.

NOTE

The archive command is executed only on completed WAL segments. A server that generates little WAL traffic can have a substantial delay between the completion of a transaction and its safe recording in archive storage. To limit how old unarchived data can be, you can:

- Set the **archive_timeout** parameter to force the server to switch to a new WAL segment file with a given frequency.
- Use the **pg_switch_wal** parameter to force a segment switch to ensure that a transaction is archived immediately after it finishes.

Example 4.5. Shell command for archiving WAL segments

This example shows a simple shell command you can set in the **archive_command** configuration parameter.

The following command copies a completed segment file to the required location:

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p
/mnt/server/archivedir/%f'
```

where the **%p** parameter is replaced by the relative path to the file to archive and the **%f** parameter is replaced by the file name.

This command copies archivable WAL segments to the **/mnt/server/archivedir/** directory. After replacing the **%p** and **%f** parameters, the executed command looks as follows:

```
test ! -f /mnt/server/archivedir/000000010000000A900000065 && cp
pg_wal/000000010000000A900000065
/mnt/server/archivedir/000000010000000A900000065
```

A similar command is generated for each new file that is archived.

2. Restart the **postgresql** service to enable the changes:

```
# systemctl restart postgresql.service
```

3. Test your archive command and ensure it does not overwrite an existing file and that it returns a nonzero exit status if it fails.

4. To protect your data, ensure that the segment files are archived into a directory that does not have group or world read access.

Additional resources

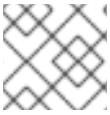
- PostgreSQL 16 Documentation

4.6.3.3. Making a base backup

You can create a base backup in several ways. The simplest way of performing a base backup is using the **pg_basebackup** utility on a running **PostgreSQL** server.

The base backup process creates a backup history file that is stored into the WAL archive area and is named after the first WAL segment file that you need for the base backup.

The backup history file is a small text file containing the starting and ending times, and WAL segments of the backup. If you used the label string to identify the associated dump file, you can use the backup history file to determine which dump file to restore.



NOTE

Consider keeping several backup sets to be certain that you can recover your data.

Prerequisites

- You must run the commands as the **postgres** superuser, a user with database administrator privileges, or another user with at least **REPLICATION** permissions.
- You must keep all the WAL segment files generated during and after the base backup.

Procedure

1. Use the **pg_basebackup** utility to perform the base backup.
   - To create a base backup as individual files (plain format):

```
$ pg_basebackup -D backup_directory -Fp
```

     Replace *backup_directory* with your chosen backup location.

     If you use tablespaces and perform the base backup on the same host as the server, you must also use the **--tablespace-mapping** option, otherwise the backup will fail upon an attempt to write the backup to the same location.

   - To create a base backup as a **tar** archive (**tar** and compressed format):

```
$ pg_basebackup -D backup_directory -Ft -z
```

Replace *backup_directory* with your chosen backup location.

To restore such data, you must manually extract the files in the correct locations.

To specify which database server **pg_basebackup** will contact, use the following command-line options:

- The **-h** option to define the host.
  The default host is either the local host or a host specified by the **PGHOST** environment variable.
- The **-p** option to define the port.
  The default port is indicated by the **PGPORT** environment variable or the compiled-in default.

2. After the base backup process is complete, safely archive the copy of the database cluster and the WAL segment files used during the backup, which are specified in the backup history file.
3. Delete WAL segments numerically lower than the WAL segment files used in the base backup because these are older than the base backup and no longer needed for a restore.

Additional resources

- PostgreSQL Documentation - base backup
- PostgreSQL Documentation - pg_basebackup utility

4.6.3.4. Restoring the database using a continuous archive backup

To restore a database using a continuous backup, use the following procedure.

Procedure

1. Stop the server:

```
# systemctl stop postgresql.service
```

2. Copy the necessary data to a temporary location.
   Preferably, copy the whole cluster data directory and any tablespaces. Note that this requires enough free space on your system to hold two copies of your existing database.

   If you do not have enough space, save the contents of the cluster's **pg_wal** directory, which can contain logs that were not archived before the system went down.
3. Remove all existing files and subdirectories under the cluster data directory and under the root directories of any tablespaces you are using.
4. Restore the database files from your base backup.
   Ensure that:

   - The files are restored with the correct ownership (the database system user, not **root**).
   - The files are restored with the correct permissions.

- The symbolic links in the **pg_tblspc/** subdirectory are restored correctly.

5. Remove any files present in the **pg_wal/** subdirectory.
   These files resulted from the base backup and are therefore obsolete. If you did not archive **pg_wal/**, recreate it with proper permissions.

6. Copy any unarchived WAL segment files that you saved in step 2 into **pg_wal/**.

7. Create the **recovery.conf** recovery command file in the cluster data directory and specify the shell command in the **restore_command** configuration parameter. You can use the **cp** command, another command, or a shell script. For example:

   ```
   restore_command = 'cp /mnt/server/archivedir/%f "%p"'
   ```

8. Start the server:

   ```
   # systemctl start postgresql.service
   ```

   The server will enter the recovery mode and proceed to read through the archived WAL files that it needs.

   If the recovery is terminated due to an external error, the server can be restarted and it will continue the recovery. When the recovery process is completed, the server renames **recovery.conf** to **recovery.done**. This prevents the server from accidental re-entering the recovery mode after it starts normal database operations.

9. Check the contents of the database to verify that the database has recovered into the required state.
   If the database has not recovered into the required state, return to step 1. If the database has recovered into the required state, allow the users to connect by restoring the client authentication configuration in the **pg_hba.conf** file.

4.6.3.4.1. Additional resources

- Continuous archiving method

4.7. MIGRATING TO A RHEL 9 VERSION OF POSTGRESQL

Red Hat Enterprise Linux 8 provides **PostgreSQL** in multiple module streams: **PostgreSQL 10** (the default postgresql stream), **PostgreSQL 9.6**, **PostgreSQL 12**, **PostgreSQL 13**, **PostgreSQL 15**, and **PostgreSQL 16**.

In RHEL 9, **PostgreSQL 13**, **PostgreSQL 15**, and **PostgreSQL 16** are available.

On RHEL, you can use two **PostgreSQL** migration paths for the database files:

- Fast upgrade using the pg_upgrade utility
- Dump and restore upgrade

The fast upgrade method is quicker than the dump and restore process. However, in certain cases, the fast upgrade does not work, and you can only use the dump and restore process, for example in case of cross-architecture upgrades.

As a prerequisite for migration to a later version of **PostgreSQL**, back up all your **PostgreSQL** databases.

Dumping the databases and performing backup of the SQL files is required for the dump and restore process and recommended for the fast upgrade method.

Before migrating to a later version of **PostgreSQL**, see the upstream compatibility notes for the version of **PostgreSQL** to which you want to migrate, and for all skipped **PostgreSQL** versions between the one you are migrating from and the target version.

4.7.1. Notable differences between PostgreSQL 15 and PostgreSQL 16

PostgreSQL 16 introduced the following notable changes.

The postmasters binary is no longer available

PostgreSQL is no longer distributed with the **postmaster** binary. Users who start the **postgresql** server by using the provided **systemd** unit file (the **systemctl start postgres.service** command) are not affected by this change. If you previously started the **postgresql** server directly through the **postmaster** binary, you must now use the **postgres** binary instead.

Documentation is no longer packaged

PostgreSQL no longer provides documentation in PDF format within the package. Use the online documentation instead.

4.7.2. Notable differences between PostgreSQL 13 and PostgreSQL 15

PostgreSQL 15 introduced the following backwards incompatible changes.

Default permissions of the public schema

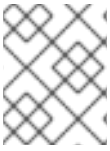
The default permissions of the public schema have been modified in **PostgreSQL 15**. Newly created users need to grant permission explicitly by using the **GRANT ALL ON SCHEMA public TO myuser;** command.

The following example works in **PostgreSQL 13** and earlier:

```
postgres=# CREATE USER mydbuser;
postgres=# \c postgres mydbuser
postgres=$ CREATE TABLE mytable (id int);
```

The following example works in **PostgreSQL 15** and later:

```
postgres=# CREATE USER mydbuser;
postgres=# GRANT ALL ON SCHEMA public TO mydbuser;
postgres=# \c postgres mydbuser
postgres=$ CREATE TABLE mytable (id int);
```



NOTE

Ensure that the **mydbuser** access is configured appropriately in the **pg_hba.conf** file. See Creating PostgreSQL users for more information.

PQsendQuery() no longer supported in pipeline mode

Since **PostgreSQL 15**, the **libpq** library **PQsendQuery()** function is no longer supported in pipeline mode. Modify affected applications to use the **PQsendQueryParams()** function instead.

4.7.3. Fast upgrade using the pg_upgrade utility

As a system administrator, you can upgrade to the most recent version of **PostgreSQL** by using the fast upgrade method. To perform a fast upgrade, copy binary data files to the **/var/lib/pgsql/data/** directory and use the **pg_upgrade** utility.

You can use this method for migrating data:

- From the RHEL 8 version of **PostgreSQL 12** to a RHEL version of **PostgreSQL 13**
- From a RHEL 8 or 9 version of **PostgreSQL 13** to a RHEL version of **PostgreSQL 15**
- From a RHEL 8 or 9 version of **PostgreSQL 15** to a RHEL version of **PostgreSQL 16**

The following procedure describes migration from the RHEL 8 version of **PostgreSQL 12** to the RHEL 9 version of **PostgreSQL 13** using the fast upgrade method. For migration from **postgresql** streams other than **12**, use one of the following approaches:

- Update your **PostgreSQL** server to version 12 on RHEL 8 and then use the **pg_upgrade** utility to perform the fast upgrade to RHEL 9 version of **PostgreSQL 13**.
- Use the dump and restore upgrade directly between any RHEL 8 version of **PostgreSQL** and an equal or later **PostgreSQL** version in RHEL 9.

Prerequisites

- Before performing the upgrade, back up all your data stored in the **PostgreSQL** databases. By default, all data is stored in the **/var/lib/pgsql/data/** directory on both the RHEL 8 and RHEL 9 systems.

Procedure

1. On the RHEL 9 system, install the **postgresql-server** and **postgresql-upgrade** packages:

   ```
   # dnf install postgresql-server postgresql-upgrade
   ```

   Optionally, if you used any **PostgreSQL** server modules on RHEL 8, install them also on the RHEL 9 system in two versions, compiled both against **PostgreSQL 12** (installed as the **postgresql-upgrade** package) and the target version of **PostgreSQL 13** (installed as the **postgresql-server** package). If you need to compile a third-party **PostgreSQL** server module, build it both against the **postgresql-devel** and **postgresql-upgrade-devel** packages.

2. Check the following items:

   - Basic configuration: On the RHEL 9 system, check whether your server uses the default **/var/lib/pgsql/data** directory and the database is correctly initialized and enabled. In addition, the data files must be stored in the same path as mentioned in the **/usr/lib/systemd/system/postgresql.service** file.
   - **PostgreSQL** servers: Your system can run multiple **PostgreSQL** servers. Ensure that the data directories for all these servers are handled independently.

- **PostgreSQL** server modules: Ensure that the **PostgreSQL** server modules that you used on RHEL 8 are installed on your RHEL 9 system as well. Note that plugins are installed in the **/usr/lib64/pgsql/** directory.

3. Ensure that the **postgresql** service is not running on either of the source and target systems at the time of copying data.

```
# systemctl stop postgresql.service
```

4. Copy the database files from the source location to the **/var/lib/pgsql/data/** directory on the RHEL 9 system.

5. Perform the upgrade process by running the following command as the **PostgreSQL** user:

```
# postgresql-setup --upgrade
```

This launches the **pg_upgrade** process in the background.

In case of failure, **postgresql-setup** provides an informative error message.

6. Copy the prior configuration from **/var/lib/pgsql/data-old** to the new cluster.
Note that the fast upgrade does not reuse the prior configuration in the newer data stack and the configuration is generated from scratch. If you want to combine the old and new configurations manually, use the *.conf files in the data directories.

7. Start the new **PostgreSQL** server:

```
# systemctl start postgresql.service
```

8. Analyze the new database cluster.

- For **PostgreSQL 13**:

```
su postgres -c '~/analyze_new_cluster.sh'
```

- For **PostgreSQL 15** or later:

```
su postgres -c 'vacuumdb --all --analyze-in-stages'
```

NOTE

You may need to use **ALTER COLLATION name REFRESH VERSION**, see the upstream documentation for details.

9. If you want the new **PostgreSQL** server to be automatically started on boot, run:

```
# systemctl enable postgresql.service
```

4.7.4. Dump and restore upgrade

When using the dump and restore upgrade, you must dump all databases contents into an SQL file dump file. Note that the dump and restore upgrade is slower than the fast upgrade method and it may require some manual fixing in the generated SQL file.

You can use this method for migrating data from any RHEL 8 version of **PostgreSQL** to any equal or later version of PostgreSQL in RHEL 9.

On RHEL 8 and RHEL 9 systems, **PostgreSQL** data is stored in the **/var/lib/pgsql/data/** directory by default.

To perform the dump and restore upgrade, change the user to **root**.

The following procedure describes migration from the RHEL 8 default version of **PostgreSQL 10** to the RHEL 9 version of **PostgreSQL 13**.

Procedure

1. On your RHEL 8 system, start the **PostgreSQL 10** server:

```
# systemctl start postgresql.service
```

2. On the RHEL 8 system, dump all databases contents into the **pgdump_file.sql** file:

```
su - postgres -c "pg_dumpall > ~/pgdump_file.sql"
```

3. Ensure that the databases were dumped correctly:

```
su - postgres -c 'less "$HOME/pgdump_file.sql"'
```

   As a result, the path to the dumped sql file is displayed: **/var/lib/pgsql/pgdump_file.sql**.

4. On the RHEL 9 system, install the **postgresql-server** package:

```
# dnf install postgresql-server
```

   Optionally, if you used any **PostgreSQL** server modules on RHEL 8, install them also on the RHEL 9 system. If you need to compile a third-party **PostgreSQL** server module, build it against the **postgresql-devel** package.

5. On the RHEL 9 system, initialize the data directory for the new **PostgreSQL** server:

```
# postgresql-setup --initdb
```

6. On the RHEL 9 system, copy the **pgdump_file.sql** into the **PostgreSQL** home directory, and check that the file was copied correctly:

```
su - postgres -c 'test -e "$HOME/pgdump_file.sql" && echo exists'
```

7. Copy the configuration files from the RHEL 8 system:

```
su - postgres -c 'ls -1 $PGDATA/*.conf'
```

   The configuration files to be copied are:

   - **/var/lib/pgsql/data/pg_hba.conf**
   - **/var/lib/pgsql/data/pg_ident.conf**

- **/var/lib/pgsql/data/postgresql.conf**

8. On the RHEL 9 system, start the new **PostgreSQL** server:

```
# systemctl start postgresql.service
```

9. On the RHEL 9 system, import data from the dumped sql file:

```
su - postgres -c 'psql -f ~/pgdump_file.sql postgres'
```

4.8. INSTALLING AND CONFIGURING A POSTGRESQL DATABASE SERVER BY USING RHEL SYSTEM ROLES

You can use the **postgresql** RHEL system role to automate the installation and management of the PostgreSQL database server. By default, this role also optimizes PostgreSQL by automatically configuring performance-related settings in the PostgreSQL service configuration files.

4.8.1. Configuring PostgreSQL with an existing TLS certificate by using the postgresql RHEL system role

If your application requires a PostgreSQL database server, you can configure this service with TLS encryption to enable secure communication between the application and the database. By using the **postgresql** RHEL system role, you can automate this process and remotely install and configure PostgreSQL with TLS encryption. In the playbook, you can use an existing private key and a TLS certificate that was issued by a certificate authority (CA).

NOTE

The **postgresql** role cannot open ports in the **firewalld** service. To allow remote access to the PostgreSQL server, add a task that uses the **firewall** RHEL system role to your playbook.

Prerequisites

- You have prepared the control node and the managed nodes
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions on them.
- Both the private key of the managed node and the certificate are stored on the control node in the following files:
  - Private key: **~/*<FQDN_of_the_managed_node>*.key**
  - Certificate: **~/*<FQDN_of_the_managed_node>*.crt**

Procedure

1. Store your sensitive variables in an encrypted file:
   a. Create the vault:

```
$ ansible-vault create ~/vault.yml
New Vault password: <vault_password>
Confirm New Vault password: <vault_password>
```

b. After the **ansible-vault create** command opens an editor, enter the sensitive data in the ***<key>: <value>*** format:

```
pwd: <password>
```

c. Save the changes, and close the editor. Ansible encrypts the data in the vault.

2. Create a playbook file, for example ~/**playbook.yml**, with the following content:

```
---
- name: Installing and configuring PostgreSQL
  hosts: managed-node-01.example.com
  vars_files:
    - ~/vault.yml
  tasks:
    - name: Create directory for TLS certificate and key
      ansible.builtin.file:
        path: /etc/postgresql/
        state: directory
        mode: 755

    - name: Copy CA certificate
      ansible.builtin.copy:
        src: "~/{{ inventory_hostname }}.crt"
        dest: "/etc/postgresql/server.crt"

    - name: Copy private key
      ansible.builtin.copy:
        src: "~/{{ inventory_hostname }}.key"
        dest: "/etc/postgresql/server.key"
        mode: 0600

    - name: PostgreSQL with an existing private key and certificate
      ansible.builtin.include_role:
        name: redhat.rhel_system_roles.postgresql
      vars:
        postgresql_version: "16"
        postgresql_password: "{{ pwd }}"
        postgresql_ssl_enable: true
        postgresql_cert_name: "/etc/postgresql/server"
        postgresql_server_conf:
          listen_addresses: "'*'"
          password_encryption: scram-sha-256
        postgresql_pg_hba_conf:
          - type: local
            database: all
            user: all
            auth_method: scram-sha-256
          - type: hostssl
            database: all
            user: all
```

```
      address: '127.0.0.1/32'
      auth_method: scram-sha-256
    - type: hostssl
      database: all
      user: all
      address: '::1/128'
      auth_method: scram-sha-256
    - type: hostssl
      database: all
      user: all
      address: '192.0.2.0/24'
      auth_method: scram-sha-256


- name: Open the PostgresQL port in firewalld
  ansible.builtin.include_role:
    name: redhat.rhel_system_roles.firewall
  vars:
    firewall:
      - service: postgresql
        state: enabled
```

The settings specified in the example playbook include the following:

postgresql_version: *<version>*

Sets the version of PostgreSQL to install. The version you can set depends on the PostgreSQL versions that are available in Red Hat Enterprise Linux running on the managed node.

You cannot upgrade or downgrade PostgreSQL by changing the **postgresql_version** variable and running the playbook again.

postgresql_password: *<password>*

Sets the password of the **postgres** database superuser.

You cannot change the password by changing the **postgresql_password** variable and running the playbook again.

postgresql_cert_name: *<private_key_and_certificate_file>*

Defines the path and base name of both the certificate and private key on the managed node without **.crt** and **key** suffixes. During the PostgreSQL configuration, the role creates symbolic links in the **/var/lib/pgsql/data/** directory that refer to these files.

The certificate and private key must exist locally on the managed node. You can use tasks with the **ansible.builtin.copy** module to transfer the files from the control node to the managed node, as shown in the playbook.

postgresql_server_conf: *<list_of_settings>*

Defines **postgresql.conf** settings the role should set. The role adds these settings to the **/etc/postgresql/system-roles.conf** file and includes this file at the end of **/var/lib/pgsql/data/postgresql.conf**. Consequently, settings from the **postgresql_server_conf** variable override settings in **/var/lib/pgsql/data/postgresql.conf**. Re-running the playbook with different settings in **postgresql_server_conf** overwrites the **/etc/postgresql/system-roles.conf** file with the new settings.

postgresql_pg_hba_conf: *<list_of_authentication_entries>*

Configures client authentication entries in the **/var/lib/pgsql/data/pg_hba.conf** file. For details, see see the PostgreSQL documentation.
The example allows the following connections to PostgreSQL:

- Unencrypted connections by using local UNIX domain sockets.
- TLS-encrypted connections to the IPv4 and IPv6 localhost addresses.
- TLS-encrypted connections from the 192.0.2.0/24 subnet. Note that access from remote addresses is only possible if you also configure the **listen_addresses** setting in the **postgresql_server_conf** variable appropriately.

Re-running the playbook with different settings in **postgresql_pg_hba_conf** overwrites the **/var/lib/pgsql/data/pg_hba.conf** file with the new settings.

For details about all variables used in the playbook, see the **/usr/share/ansible/roles/rhel-system-roles.postgresql/README.md** file on the control node.

3. Validate the playbook syntax:

```
$ ansible-playbook --ask-vault-pass --syntax-check ~/playbook.yml
```

Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

4. Run the playbook:

```
$ ansible-playbook --ask-vault-pass ~/playbook.yml
```

Verification

- Use the **postgres** super user to connect to a PostgreSQL server and execute the **\conninfo** meta command:

```
# psql "postgresql://postgres@managed-node-01.example.com:5432" -c '\conninfo'
Password for user postgres:
You are connected to database "postgres" as user "postgres" on host "192.0.2.1" at port "5432".
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
```

If the output displays a TLS protocol version and cipher details, the connection works and TLS encryption is enabled.

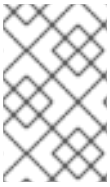
Additional resources

- **/usr/share/ansible/roles/rhel-system-roles.postgresql/README.md** file
- **/usr/share/doc/rhel-system-roles/postgresql/** directory
- Ansible vault

4.8.2. Configuring PostgreSQL with a TLS certificate issued from IdM by using the postgresql RHEL system role

If your application requires a PostgreSQL database server, you can configure the PostgreSQL service with TLS encryption to enable secure communication between the application and the database. If the PostgreSQL host is a member of a Red Hat Enterprise Linux Identity Management (IdM) domain, the **certmonger** service can manage the certificate request and future renewals.

By using the **postgresql** RHEL system role, you can automate this process. You can remotely install and configure PostgreSQL with TLS encryption, and the **postgresql** role uses the **certificate** RHEL system role to configure **certmonger** and request a certificate from IdM.



NOTE

The **postgresql** role cannot open ports in the **firewalld** service. To allow remote access to the PostgreSQL server, add a task to your playbook that uses the **firewall** RHEL system role.

Prerequisites

- You have prepared the control node and the managed nodes
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions on them.
- You enrolled the managed node in an IdM domain.

Procedure

1. Store your sensitive variables in an encrypted file:
   a. Create the vault:

```
$ ansible-vault create ~/vault.yml
New Vault password: <vault_password>
Confirm New Vault password: <vault_password>
```

   b. After the **ansible-vault create** command opens an editor, enter the sensitive data in the ***<key>: <value>*** format:

```
pwd: <password>
```

   c. Save the changes, and close the editor. Ansible encrypts the data in the vault.
2. Create a playbook file, for example ~/**playbook.yml**, with the following content:

```
---
- name: Installing and configuring PostgreSQL
  hosts: managed-node-01.example.com
  vars_files:
    - ~/vault.yml
  tasks:
    - name: PostgreSQL with certificates issued by IdM
      ansible.builtin.include_role:
        name: redhat.rhel_system_roles.postgresql
      vars:
```

```
    postgresql_version: "16"
    postgresql_password: "{{ pwd }}"
    postgresql_ssl_enable: true
    postgresql_certificates:
      - name: postgresql_cert
        dns: "{{ inventory_hostname }}"
        ca: ipa
        principal: "postgresql/{{ inventory_hostname }}@EXAMPLE.COM"
    postgresql_server_conf:
      listen_addresses: "'*'"
      password_encryption: scram-sha-256
    postgresql_pg_hba_conf:
      - type: local
        database: all
        user: all
        auth_method: scram-sha-256
      - type: hostssl
        database: all
        user: all
        address: '127.0.0.1/32'
        auth_method: scram-sha-256
      - type: hostssl
        database: all
        user: all
        address: '::1/128'
        auth_method: scram-sha-256
      - type: hostssl
        database: all
        user: all
        address: '192.0.2.0/24'
        auth_method: scram-sha-256


  - name: Open the PostgresQL port in firewalld
    ansible.builtin.include_role:
      name: redhat.rhel_system_roles.firewall
    vars:
      firewall:
        - service: postgresql
          state: enabled
```

The settings specified in the example playbook include the following:

postgresql_version: *<version>*

Sets the version of PostgreSQL to install. The version you can set depends on the PostgreSQL versions that are available in Red Hat Enterprise Linux running on the managed node.

You cannot upgrade or downgrade PostgreSQL by changing the **postgresql_version** variable and running the playbook again.

postgresql_password: *<password>*

Sets the password of the **postgres** database superuser.

You cannot change the password by changing the **postgresql_password** variable and running the playbook again.

postgresql_certificates: *<certificate_role_settings>*

A list of YAML dictionaries with settings for the **certificate** role.

postgresql_server_conf: *<list_of_settings>*

Defines **postgresql.conf** settings you want the role to set. The role adds these settings to the **/etc/postgresql/system-roles.conf** file and includes this file at the end of **/var/lib/pgsql/data/postgresql.conf**. Consequently, settings from the **postgresql_server_conf** variable override settings in **/var/lib/pgsql/data/postgresql.conf**. Re-running the playbook with different settings in **postgresql_server_conf** overwrites the **/etc/postgresql/system-roles.conf** file with the new settings.

postgresql_pg_hba_conf: *<list_of_authentication_entries>*

Configures client authentication entries in the **/var/lib/pgsql/data/pg_hba.conf** file. For details, see see the PostgreSQL documentation.
The example allows the following connections to PostgreSQL:

- Unencrypted connections by using local UNIX domain sockets.
- TLS-encrypted connections to the IPv4 and IPv6 localhost addresses.
- TLS-encrypted connections from the 192.0.2.0/24 subnet. Note that access from remote addresses is only possible if you also configure the **listen_addresses** setting in the **postgresql_server_conf** variable appropriately.

Re-running the playbook with different settings in **postgresql_pg_hba_conf** overwrites the **/var/lib/pgsql/data/pg_hba.conf** file with the new settings.

For details about all variables used in the playbook, see the **/usr/share/ansible/roles/rhel-system-roles.postgresql/README.md** file on the control node.

3. Validate the playbook syntax:

```
$ ansible-playbook --ask-vault-pass --syntax-check ~/playbook.yml
```

Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

4. Run the playbook:

```
$ ansible-playbook --ask-vault-pass ~/playbook.yml
```

Verification

- Use the **postgres** super user to connect to a PostgreSQL server and execute the **\conninfo** meta command:

```
# psql "postgresql://postgres@managed-node-01.example.com:5432" -c '\conninfo'
Password for user postgres:
You are connected to database "postgres" as user "postgres" on host "192.0.2.1" at port "5432".
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
```

If the output displays a TLS protocol version and cipher details, the connection works and TLS encryption is enabled.

Additional resources

- **/usr/share/ansible/roles/rhel-system-roles.postgresql/README.md** file
- **/usr/share/doc/rhel-system-roles/postgresql/** directory
- Ansible vault