MySQL Shell 8.0

Abstract

MySQL Shell is an advanced client and code editor for MySQL. This document describes the core features of MySQL Shell. In addition to the provided SQL functionality, similar to `mysql`, MySQL Shell provides scripting capabilities for JavaScript and Python and includes APIs for working with MySQL. X DevAPI enables you to work with both relational and document data, see Using MySQL as a Document Store. AdminAPI enables you to work with InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet.

MySQL Shell 8.0 is highly recommended for use with MySQL Server 8.0 and 5.7. Please upgrade to MySQL Shell 8.0. If you have not yet installed MySQL Shell, download it from the download site.

For notes detailing the changes in each release, see the MySQL Shell Release Notes.

For help with using MySQL, please visit the MySQL Forums, where you can discuss your issues with other MySQL users.

Licensing information. This product may include third-party software, used under license. If you are using a *Commercial* release of MySQL Shell, see MySQL Shell Commercial License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a *Community* release of MySQL Shell, see MySQL Shell Community License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Document generated on: 2026-07-02 (revision: 84706)

Table of Contents

1 MySQL Shell Features 1
2 Installing MySQL Shell 5
2.1 Installing MySQL Shell on Microsoft Windows 5
2.2 Installing MySQL Shell on Linux 5
2.3 Installing MySQL Shell on macOS 7
3 Using MySQL Shell Commands 9
3.1 MySQL Shell Commands 9
4 Getting Started with MySQL Shell 15
4.1 Starting MySQL Shell 15
4.2 MySQL Shell Sessions 15
4.2.1 Creating the `Session` Global Object While Starting MySQL Shell 17
4.2.2 Creating the `Session` Global Object After Starting MySQL Shell 17
4.2.3 Scripting Sessions in JavaScript and Python Mode 18
4.3 MySQL Shell Connections 20
4.3.1 Connecting using Individual Parameters 21
4.3.2 Connecting using login-path and Options Files 23
4.3.3 Connecting using Unix Sockets and Windows Named Pipes 24
4.3.4 Using Encrypted Connections 25
4.3.5 Using LDAP and Kerberos Authentication 26
4.3.6 Using an SSH Tunnel 29
4.3.7 Using Compressed Connections 32
4.4 Pluggable Password Store 35
4.4.1 Pluggable Password Configuration Options 36
4.4.2 Working with Credentials 37
4.5 MySQL Shell Global Objects 37
4.6 Using a Pager 38
4.7 Cloud Service Configuration 39
4.7.1 Oracle Cloud Infrastructure Object Storage 39
4.7.2 S3-compatible Storage 39
4.7.3 Azure Blob Storage 42
5 MySQL Shell Code Execution 45
5.1 Active Language 45
5.2 Interactive Code Execution 46
5.3 Code Autocompletion 48
5.4 Editing Code 52
5.5 Code History 52
5.6 Batch Code Execution 53
5.7 Output Formats 55
5.7.1 Table Format 56
5.7.2 Tab Separated Format 56
5.7.3 Vertical Format 56
5.7.4 JSON Format Output 57
5.7.5 JSON Wrapping 59
5.7.6 Result Metadata 60
5.8 API Command Line Integration 60
5.8.1 Command Line Integration Overview 60
5.8.2 Command Line Integration Details 63
5.9 JSON Integration 72
5.10 Limitations 72
6 MySQL AdminAPI 75
6.1 Using MySQL AdminAPI 75
6.2 Installing AdminAPI Software Components 76
6.2.1 Using Instances Running MySQL 5.7 77
6.2.2 Configuring the Host Name 78
6.2.3 Connecting to Server Instances 78

6.2.4 Persisting Settings 79
6.3 Retrieving a Handler Object 80
6.4 Creating User Accounts for AdminAPI 82
6.5 Verbose Logging 84
6.6 Finding the Primary 84
6.7 Scripting AdminAPI 85
6.8 AdminAPI MySQL Sandboxes 86
6.8.1 Deploying Sandbox Instances 87
6.8.2 Managing Sandbox Instances 88
6.8.3 Setting up InnoDB Cluster and MySQL Router 88
6.9 Tagging Metadata 95
6.10 Using MySQL Router with AdminAPI, InnoDB Cluster, and InnoDB ReplicaSet 100
6.10.1 Bootstrapping MySQL Router 100
6.10.2 Configuring the MySQL Router User 100
6.10.3 Deploying MySQL Router 102
6.10.4 Using ReplicaSets with MySQL Router 104
6.10.5 Testing InnoDB Cluster High Availability 104
6.10.6 Working with a Cluster's Routers 105
6.11 Upgrade Metadata Schema 107
6.12 Locking Mechanism for AdminAPI Operations 108
7 MySQL InnoDB Cluster 113
7.1 InnoDB Cluster Requirements 114
7.2 InnoDB Cluster Limitations 116
7.3 User Accounts for InnoDB Cluster 116
7.4 Deploying a Production InnoDB Cluster 119
7.4.1 Pre-Checking Instance Configuration for InnoDB Cluster Usage 120
7.4.2 Configuring Production Instances for InnoDB Cluster Usage 121
7.4.3 Creating an InnoDB Cluster 125
7.4.4 Adding Instances to an InnoDB Cluster 127
7.4.5 Configuring InnoDB Cluster Ports 130
7.4.6 Using MySQL Clone with InnoDB Cluster 131
7.4.7 Adopting a Group Replication Deployment 135
7.5 Configuring InnoDB Cluster 135
7.5.1 Setting Options for InnoDB Cluster 135
7.5.2 Customizing InnoDB Cluster Member Servers 136
7.5.3 Configuring the Election Process 137
7.5.4 Configuring Failover Consistency 137
7.5.5 Configuring Automatic Rejoin of Instances 138
7.5.6 Configuring the Parallel Replication Applier 139
7.5.7 InnoDB Cluster and Auto-increment 140
7.5.8 InnoDB Cluster and Binary Log Purging 140
7.5.9 Configuring the Group Replication Communication Stack 140
7.6 Securing InnoDB Cluster 142
7.7 Monitoring InnoDB Cluster 145
7.8 Restoring and Rebooting an InnoDB Cluster 156
7.8.1 Rejoining an Instance to a Cluster 156
7.8.2 Restoring a Cluster from Quorum Loss 157
7.8.3 Rebooting a Cluster from a Major Outage 158
7.8.4 Rescanning a Cluster 161
7.8.5 Fencing a Cluster 163
7.9 Modifying or Dissolving an InnoDB Cluster 163
7.10 Upgrade InnoDB Cluster 167
7.10.1 InnoDB Cluster Upgrade 167
7.10.2 Troubleshooting InnoDB Cluster Upgrades 172
8 MySQL InnoDB ClusterSet 175
8.1 InnoDB ClusterSet Requirements 177
8.2 InnoDB ClusterSet Limitations 180
8.3 User Accounts for InnoDB ClusterSet 181

8.4 Deploying InnoDB ClusterSet 183
8.5 Integrating MySQL Router With InnoDB ClusterSet 195
8.6 InnoDB ClusterSet Status and Topology 200
8.7 InnoDB ClusterSet Controlled Switchover 208
8.8 InnoDB ClusterSet Emergency Failover 213
8.9 InnoDB ClusterSet Repair and Rejoin 218
8.9.1 Fencing Clusters in an InnoDB ClusterSet 220
8.9.2 Inconsistent Transaction Sets (GTID Sets) in InnoDB ClusterSet Clusters 223
8.9.3 Repairing Member Servers and Clusters in an InnoDB ClusterSet 225
8.9.4 Removing a Cluster from an InnoDB ClusterSet 227
8.9.5 Rejoining a Cluster to an InnoDB ClusterSet 229
8.10 Upgrade InnoDB ClusterSet 231
9 MySQL InnoDB ReplicaSet 239
9.1 Deploying InnoDB ReplicaSet 240
9.2 Configuring InnoDB ReplicaSet Instances 241
9.3 Creating an InnoDB ReplicaSet 241
9.4 Adding Instances to a ReplicaSet 244
9.4.1 Provisioning Instances for InnoDB ReplicaSet 244
9.4.2 Example of Adding Instances to a ReplicaSet 245
9.5 Adopting an Existing Replication Setup 246
9.6 Changing the Primary Instance 247
9.7 Forcing a New Primary Instance 248
9.8 Tagging ReplicaSets 249
9.9 Checking the Status of InnoDB ReplicaSet 249
9.10 Upgrade InnoDB ReplicaSet 250
10 Extending MySQL Shell 257
10.1 Reporting with MySQL Shell 257
10.1.1 Creating MySQL Shell Reports 258
10.1.2 Registering MySQL Shell Reports 258
10.1.3 Persisting MySQL Shell Reports 260
10.1.4 Example MySQL Shell Report 260
10.1.5 Running MySQL Shell Reports 261
10.1.6 Built-in MySQL Shell Reports 262
10.2 Adding Extension Objects to MySQL Shell 265
10.2.1 Creating User-Defined MySQL Shell Global Objects 265
10.2.2 Creating Extension Objects 266
10.2.3 Persisting Extension Objects 268
10.2.4 Example MySQL Shell Extension Objects 268
10.3 MySQL Shell Plugins 270
10.3.1 Creating MySQL Shell Plugins 270
10.3.2 Creating Plugin Groups 271
10.3.3 Example MySQL Shell Plugins 271
11 MySQL Shell Utilities 275
11.1 Upgrade Checker Utility 276
11.2 JSON Import Utility 281
11.2.1 Running the Utility 281
11.2.2 Importing JSON Documents With the Mysqlsh Command Interface 283
11.2.3 Importing JSON Documents With the `--import` Command 284
11.2.4 Conversions for Representations of BSON Data Types 285
11.3 Table Export Utility 286
11.4 Parallel Table Import Utility 293
11.5 Instance Dump Utility, Schema Dump Utility, and Table Dump Utility 302
11.6 Dump Loading Utility 324
11.7 Diagnostics Utilities 343
11.7.1 collectDiagnostics Utility 343
11.7.2 collectHighLoadDiagnostics Utility 346
11.7.3 collectSlowQueryDiagnostics Utility 347
12 MySQL Shell Logging and Debug 351

12.1 Application Log 352
12.2 Verbose Output 353
12.3 System Logging for User SQL Statements 354
12.4 MySQL Shell SQL Logging 355
12.5 Logging AdminAPI Operations 357
13 Customizing MySQL Shell 359
13.1 Working With Startup Scripts 359
13.2 Adding Module Search Paths 360
13.2.1 Module Search Path Environment Variables 361
13.2.2 Module Search Path Variable in Startup Scripts 361
13.3 Customizing the Prompt 362
13.4 Configuring MySQL Shell Options 363
A MySQL Shell Command Reference 371
A.1 mysqlsh — The MySQL Shell 371

Chapter 1 MySQL Shell Features

Important

MySQL Shell is updated frequently with fixes and new features. It is strongly recommended that you always use the most recent version available. The latest version of MySQL Shell can be used with any version of MySQL 5.7 or 8.0.

The following features are available in MySQL Shell:

- Supported Languages
- Interactive Code Execution
- Batch Code Execution
- Supported APIs
- X Protocol Support
- Extensions
- Utilities
- API Command Line Integration
- Output Formats
- Logging and Debug
- Global Session

Supported Languages

MySQL Shell processes code written in JavaScript, Python and SQL. Any executed code is processed as one of these languages, based on the language that is currently active. There are also specific MySQL Shell commands, prefixed with \, which enable you to configure MySQL Shell regardless of the currently selected language. For more information see Section 3.1, “MySQL Shell Commands”.

From version 8.0.18, MySQL Shell uses Python 3, rather than Python 2.7. For platforms that include a system supported installation of Python 3, MySQL Shell uses the most recent version available, with a minimum supported version of Python 3.6. For platforms where Python 3 is not included or is not at the minimum supported version, MySQL Shell bundles Python 3.7.7 up to MySQL Shell 8.0.25, and Python 3.9.5 from MySQL Shell 8.0.26. MySQL Shell maintains code compatibility with Python 2.6 and Python 2.7, so if you require one of these older versions, you can build MySQL Shell from source using the appropriate Python version.

As of 8.0.32, MySQL Shell bundles Python 3.10.8 for platforms where Python 3 is not included or is not at the minimum supported version.

Note

This is true for all builds except Oracle Linux 7, which bundles Python 3.9.15

Interactive Code Execution

MySQL Shell provides an interactive code execution mode, where you type code at the MySQL Shell prompt and each entered statement is processed, with the result of the processing printed onscreen. Unicode text input is supported if the terminal in use supports it. Color terminals are supported.

Multiple-line code can be written using a command, enabling MySQL Shell to cache multiple lines and then execute them as a single statement. For more information see Multiple-line Support.

Batch Code Execution

In addition to the interactive execution of code, MySQL Shell can also take code from different sources and process it. This method of processing code in a noninteractive way is called *Batch Execution*.

As batch execution mode is intended for script processing of a single language, it is limited to having minimal non-formatted output and disabling the execution of commands. To avoid these limitations, use the `--interactive` command-line option, which tells MySQL Shell to execute the input as if it were an interactive session. In this mode the input is processed *line by line* just as if each line were typed in an interactive session. For more information see Section 5.6, “Batch Code Execution”.

Supported APIs

MySQL Shell includes the following APIs implemented in JavaScript and Python which you can use to develop code that interacts with MySQL.

1. AdminAPI enables you to administer MySQL instances, using them to create InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet deployments, and integrating MySQL Router.

   - InnoDB Cluster provides an integrated solution for high availability and scalability using InnoDB based MySQL databases. InnoDB Cluster is an alternative solution for using Group Replication, without requiring advanced MySQL expertise. See Chapter 7, *MySQL InnoDB Cluster*.

   - InnoDB ClusterSet provides disaster tolerance for Chapter 7, *MySQL InnoDB Cluster* deployments by linking a primary InnoDB Cluster with one or more replicas of itself in alternate locations. See Chapter 8, *MySQL InnoDB ClusterSet*.

   - InnoDB ReplicaSet enables you to administer a set of MySQL instances running asynchronous GTID-based replication. See Chapter 9, *MySQL InnoDB ReplicaSet*.

   AdminAPI also provides operations to configure users for MySQL Router, to make integration with InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet as simple as possible. For more information on AdminAPI, see Chapter 6, *MySQL AdminAPI*.

2. X DevAPI enables developers to work with both relational and document data when MySQL Shell is connected to a MySQL server using the X Protocol. For more information, see Using MySQL as a Document Store. For documentation on the concepts and usage of X DevAPI, see X DevAPI User Guide.

X Protocol Support

MySQL Shell is designed to provide an integrated command-line client for all MySQL products which support X Protocol. The development features of MySQL Shell are designed for sessions using the X Protocol. MySQL Shell can also connect to MySQL Servers that do not support the X Protocol using the classic MySQL protocol. A minimal set of features from the X DevAPI are available for sessions created using the classic MySQL protocol.

Extensions

You can define extensions to the base functionality of MySQL Shell in the form of reports and extension objects. Reports and extension objects can be created using JavaScript or Python, and can be used regardless of the active MySQL Shell language. You can persist reports and extension objects in plugins that are loaded automatically when MySQL Shell starts. MySQL Shell has several built-in reports ready to use. See Chapter 10, *Extending MySQL Shell* for more information.

Utilities

MySQL Shell includes the following utilities for working with MySQL:

- An upgrade checker utility to verify whether MySQL server instances are ready for upgrade. Use `util.checkForServerUpgrade()` to access the upgrade checker.
- A JSON import utility to import JSON documents to a MySQL Server collection or table. Use `util.importJSON()` to access the import utility.
- A parallel table import utility that splits up a single data file and uses multiple threads to load the chunks into a MySQL table.

See Chapter 11, *MySQL Shell Utilities* for more information.

API Command Line Integration

MySQL Shell exposes much of its functionality using an API command syntax that enables you to easily integrate `mysqlsh` with other tools. For example you can create `bash` scripts which administer an InnoDB Cluster with this functionality. Use the `mysqlsh [options] -- shell_object object_method [method_arguments]` syntax to pass operations directly to MySQL Shell global objects, bypassing the REPL interface. See Section 5.8, “API Command Line Integration”.

Output Formats

MySQL Shell can return results in table, tabbed, or vertical format, or as JSON output. To help integrate MySQL Shell with external tools, you can activate JSON wrapping for all output when you start MySQL Shell from the command line. For more information see Section 5.7, “Output Formats”.

Logging and Debug

MySQL Shell can log information about the execution process at your chosen level of detail. Logging information can be sent to any combination of an application log file, an additional viewable destination, and the console. For more information see Chapter 12, *MySQL Shell Logging and Debug*.

Global Session

In MySQL Shell, connections to MySQL Server instances are handled by a session object. When you make the first connection to a MySQL Server instance, which can be done either while starting MySQL Shell or afterwards, a MySQL Shell global object named `session` is created to represent this connection. This session is known as the global session because it can be used in all of the MySQL Shell execution modes. In SQL mode the global session is used for executing statements, and in JavaScript mode and Python mode it is available through an object named `session`. You can create further session objects using functions available in the `mysqlx` and `mysql` JavaScript and Python modules, and you can set one of these session objects as the `session` global object so you can use it in any mode. For more information, see Section 4.2, “MySQL Shell Sessions”.

Chapter 2 Installing MySQL Shell

Table of Contents

2.1 Installing MySQL Shell on Microsoft Windows 5
2.2 Installing MySQL Shell on Linux 5
2.3 Installing MySQL Shell on macOS 7

This section describes how to download, install, and start MySQL Shell, which is an interactive JavaScript, Python, or SQL interface supporting development and administration for MySQL Server. MySQL Shell is a component that you can install separately.

MySQL Shell supports X Protocol and enables you to use X DevAPI in JavaScript or Python to develop applications that communicate with a MySQL Server functioning as a document store. For information about using MySQL as a document store, see Using MySQL as a Document Store.

Important

For the Community and Commercial versions of MySQL Shell: Before installing MySQL Shell, make sure you have the Visual C++ Redistributable for Visual Studio 2017 (available at the Microsoft Visual C++ Redistributable latest supported downloads) installed on your Windows system.

Requirements

MySQL Shell is available on Microsoft Windows, Linux, and macOS for 64-bit platforms.

Important

It is recommended that you always use the most recent version available. The latest version of MySQL Shell can be used with any version of MySQL 5.7 or 8.0.

2.1 Installing MySQL Shell on Microsoft Windows

To install MySQL Shell on Microsoft Windows using the MSI Installer, do the following:

1. Download the **Windows (x86, 64-bit), MSI Installer** package from http://dev.mysql.com/downloads/shell/.
2. When prompted, click **Run**.
3. Follow the steps in the Setup Wizard.

2.2 Installing MySQL Shell on Linux

Note

Installation packages for MySQL Shell are available only for a limited number of Linux distributions, and only for 64-bit systems.

For supported Linux distributions, the easiest way to install MySQL Shell on Linux is to use the MySQL APT repository or MySQL Yum repository. For systems not using the MySQL repositories, MySQL Shell can also be downloaded and installed directly.

Installing MySQL Shell with the MySQL APT Repository

For Linux distributions supported by the MySQL APT repository, follow one of the paths below:

- If you do not yet have the MySQL APT repository as a software repository on your system, do the following:
  - Follow the steps given in Adding the MySQL APT Repository, paying special attention to the following:
    - During the installation of the configuration package, when asked in the dialogue box to configure the repository, make sure you choose MySQL 8.0 as the release series you want.
    - Make sure you do not skip the step for updating package information for the MySQL APT repository:

      ```
      sudo apt-get update
      ```
  - Install MySQL Shell with this command:

    ```
    sudo apt-get install mysql-shell
    ```
- If you already have the MySQL APT repository as a software repository on your system, do the following:
  - Update package information for the MySQL APT repository:

    ```
    sudo apt-get update
    ```
  - Update the MySQL APT repository configuration package with the following command:

    ```
    sudo apt-get install mysql-apt-config
    ```

    When asked in the dialogue box to configure the repository, make sure you choose MySQL 8.0 as the release series you want.
  - Install MySQL Shell with this command:

    ```
    sudo apt-get install mysql-shell
    ```

Installing MySQL Shell with the MySQL Yum Repository

For Linux distributions supported by the MySQL Yum repository, follow these steps to install MySQL Shell:

- Do one of the following:
  - If you already have the MySQL Yum repository as a software repository on your system and the repository was configured with the new release package `mysql80-community-release`.
  - If you already have the MySQL Yum repository as a software repository on your system but have configured the repository with the old release package `mysql-community-release`, it is easiest to install MySQL Shell by first reconfiguring the MySQL Yum repository with the new `mysql80-community-release` package. To do so, you need to remove your old release package first, with the following command :

    ```
    sudo yum remove mysql-community-release
    ```

    For dnf-enabled systems, do this instead:

    ```
    sudo dnf erase mysql-community-release
    ```

    Then, follow the steps given in Adding the MySQL Yum Repository to install the new release package, `mysql80-community-release`.
  - If you do not yet have the MySQL Yum repository as a software repository on your system, follow the steps given in Adding the MySQL Yum Repository.

- Install MySQL Shell with this command:

  ```
  sudo yum install mysql-shell
  ```

  For dnf-enabled systems, do this instead:

  ```
  sudo dnf install mysql-shell
  ```

Installing MySQL Shell from Direct Downloads from the MySQL Developer Zone

RPM, Debian, and source packages for installing MySQL Shell are also available for download at Download MySQL Shell.

2.3 Installing MySQL Shell on macOS

To install MySQL Shell on macOS, do the following:

1. Download the package from http://dev.mysql.com/downloads/shell/.
2. Double-click the downloaded DMG to mount it. Finder opens.
3. Double-click the `.pkg` file shown in the Finder window.
4. Follow the steps in the installation wizard.
5. When the installer finishes, eject the DMG. (It can be deleted.)

Chapter 3 Using MySQL Shell Commands

Table of Contents

3.1 MySQL Shell Commands 9

This section describes the commands which configure MySQL Shell from the interactive code editor. The commands enable you to control the MySQL Shell regardless of the current language being used. For example you can get online help, connect to servers, change the current language being used, run reports, use utilities, and so on. These commands are sometimes similar to the MySQL Shell settings which can be configured using the `mysqlsh` command options, see Appendix A, *MySQL Shell Command Reference*.

3.1 MySQL Shell Commands

MySQL Shell provides commands which enable you to modify the execution environment of the code editor, for example to configure the active programming language or a MySQL Server connection. The following table lists the commands that are available regardless of the currently selected language. As commands need to be available independent of the *execution mode*, they start with an escape sequence, the `\` character.

Command	Alias/Shortcut	Description
`\help`	`\h` or `\?`	Print help about MySQL Shell, or search the online help.
`\quit`	`\q` or `\exit`	Exit MySQL Shell.
`\`		In SQL mode, begin multiple-line mode. Code is cached and executed when an empty line is entered.
`\status`	`\s`	Show the current MySQL Shell status.
`\js`		Switch execution mode to JavaScript.
`\py`		Switch execution mode to Python.
`\sql`		Switch execution mode to SQL.
`\connect`	`\c`	Connect to a MySQL instance.
`\reconnect`		Reconnect to the same MySQL instance.
`\disconnect`		Disconnect the global session.
`\use`	`\u`	Specify the schema to use.
`\source`	`\.` or `source` (no backslash)	Execute a script file using the active language.
`\warnings`	`\W`	Show any warnings generated by a statement.
`\nowarnings`	`\w`	Do not show any warnings generated by a statement.
`\history`		View and edit command line history.

Command	Alias/Shortcut	Description
`\rehash`		Manually update the autocomplete name cache.
`\option`		Query and change MySQL Shell configuration options.
`\show`		Run the specified report using the provided options and arguments.
`\watch`		Run the specified report using the provided options and arguments, and refresh the results at regular intervals.
`\edit`	`\e`	Open a command in the default system editor then present it in MySQL Shell.
`\pager`	`\P`	Configure the pager which MySQL Shell uses to display text.
`\nopager`		Disable any pager which MySQL Shell was configured to use.
`\system`	`\!`	Run the specified operating system command and display the results in MySQL Shell.

Help Command

The `\help` command can be used with or without a parameter. When used without a parameter a general help message is printed including information about the available MySQL Shell commands, global objects and main help categories.

When used with a parameter, the parameter is used to search the available help based on the mode which the MySQL Shell is currently running in. The parameter can be a word, a command, an API function, or part of an SQL statement. The following categories exist:

- `AdminAPI` - details the `dba` global object and the AdminAPI, which enables you to work with InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet.
- `X DevAPI` - details the `mysqlx` module as well as the capabilities of the X DevAPI, which enable you to work with MySQL as a Document Store
- `Shell Commands` - provides details about the available built-in MySQL Shell commands.
- `ShellAPI` - contains information about the `shell` and `util` global objects, as well as the `mysql` module that enables executing SQL on MySQL Servers.
- `SQL Syntax` - entry point to retrieve syntax help on SQL statements.

To search for help on a topic, for example an API function, use the function name as a *`pattern`*. You can use the wildcard characters `?` to match any single character and `*` to match multiple characters in a search. The wildcard characters can be used one or more times in the pattern. The following namespaces can also be used when searching for help:

- `dba` for AdminAPI
- `mysqlx` for X DevAPI
- `mysql` for ShellAPI for classic MySQL protocol

- `shell` for other ShellAPI classes: `Shell`, `Sys`, `Options`
- `commands` for MySQL Shell commands
- `cmdline` for the `mysqlsh` command interface

For example to search for help on a topic, issue `\help pattern` and:

- use `x devapi` to search for help on the X DevAPI
- use `\c` to search for help on the MySQL Shell `\connect` command
- use `getCluster` or `dba.getCluster` to search for help on the AdminAPI `dba.getCluster()` operation
- use `Table` or `mysqlx.Table` to search for help on the X DevAPI `Table` class
- when MySQL Shell is running in JavaScript mode, use `isView`, `Table.isView` or `mysqlx.Table.isView` to search for help on the `isView` function of the `Table` object
- when MySQL Shell is running in Python mode, use `is_view`, `Table.is_view` or `mysqlx.Table.is_view` to search for help on the `isView` function of the `Table` object
- when MySQL Shell is running in SQL mode, if a global session to a MySQL server exists SQL help is displayed. For an overview use `sql syntax` as the search pattern.

Depending on the search pattern provided, one or more results could be found. If only one help topic contains the search pattern in its title, that help topic is displayed. If multiple topic titles match the pattern but one is an exact match, that help topic is displayed, followed by a list of the other topics with pattern matches in their titles. If no exact match is identified, a list of topics with pattern matches in their titles is displayed. If a list of topics is returned, you can select a topic to view from the list by entering the command again with an extended search pattern that matches the title of the relevant topic.

Connect, Reconnect, and Disconnect Commands

The `\connect` command is used to connect to a MySQL Server. See Section 4.3, “MySQL Shell Connections”.

For example:

```
\connect root@localhost:3306
```

If a password is required you are prompted for it.

Use the `--mysqlx` (`--mx`) option to create a session using the X Protocol to connect to MySQL server instance. For example:

```
\connect --mysqlx root@localhost:33060
```

Use the `--mysql` (`--mc`) option to create a ClassicSession, enabling you to use classic MySQL protocol to issue SQL directly on a server. For example:

```
\connect --mysql root@localhost:3306
```

The use of a single dash with the short form options (that is, `-mx` and `-mc`) is deprecated from version 8.0.13 of MySQL Shell.

Use the `--ssh` option (available from MySQL Shell 8.0.28) to create or reuse an SSH tunnel that provides an encrypted connection to the MySQL server instance. The use of AdminAPI commands is not supported over connections made from MySQL Shell using SSH tunneling. Supply the URI for connection to the SSH server in the format `[user@]hostname[:port]`, followed by the MySQL instance URI, for example:

```
\connect --ssh root@198.51.100.4:2222 root@localhost:3306
```

When you use the `--ssh` option, the port for connection to the MySQL server instance must be specified in the MySQL instance URI.

An SSH tunnel set up using the `\connect` command must use the default SSH configuration file and identity file. For instructions to set these and further information on SSH tunnel connections from MySQL Shell, see Section 4.3.6, “Using an SSH Tunnel”. You can set up an SSH tunnel using the `shell.connect()` method or on the command line to get additional setup options. Once established, an SSH tunnel can be shared between connections to the same host from the same user connecting from the same instance, whatever setup method was originally used.

The `\reconnect` command is specified without any parameters or options. If the connection to the server is lost, you can use the `\reconnect` command, which makes MySQL Shell try several reconnection attempts for the session using the existing connection parameters. If those attempts are unsuccessful, you can make a fresh connection using the `\connect` command and specifying the connection parameters.

The `\disconnect` command, available from MySQL Shell 8.0.22, is also specified without any parameters or options. The command disconnects MySQL Shell's global session (the session represented by the `session` global object) from the currently connected MySQL server instance, so that you can close the connection but still continue to use MySQL Shell.

If the connection to the server is lost, you can use the `\reconnect` command, which makes MySQL Shell try several reconnection attempts for the session using the existing connection parameters. If those attempts are unsuccessful, you can make a fresh connection using the `\connect` command and specifying the connection parameters.

Status Command

The `\status` command displays information about the current global connection. This includes information about the server connected to, the character set in use, uptime, and so on.

Source Command

The `\source` command or its alias `\.` can be used in MySQL Shell's interactive mode to execute code from a script file at a given path. For example:

```
\source /tmp/mydata.sql
```

You can execute either SQL, JavaScript or Python code. The code in the file is executed using the active language, so to process SQL code the MySQL Shell must be in SQL mode.

Warning

As the code is executed using the active language, executing a script in a different language than the currently selected execution mode language could lead to unexpected results.

From MySQL Shell 8.0.19, for compatibility with the `mysql` client, in SQL mode only, you can execute code from a script file using the `source` command with no backslash and an optional SQL delimiter. `source` or the alias `\.` (which does not use an SQL delimiter) can be used both in MySQL Shell's interactive mode for SQL, to execute a script directly, and in a file of SQL code processed in batch mode, to execute a further script from within the file. So with MySQL Shell in SQL mode, you could now execute the script in the `/tmp/mydata.sql` file from either interactive mode or batch mode using any of these three commands:

```
source /tmp/mydata.sql;
source /tmp/mydata.sql
\. /tmp/mydata.sql
```

The command `\source /tmp/mydata.sql` is also valid, but in interactive mode only.

In interactive mode, the `\source`, `\.` or `source` command itself is added to the MySQL Shell history, but the contents of the executed script file are not added to the history.

Use Command

The `\use` command enables you to choose which schema is active, for example:

```
\use schema_name
```

The `\use` command requires a global development session to be active. The \use command sets the current schema to the specified *`schema_name`* and updates the `db` variable to the object that represents the selected schema.

History Command

The `\history` command lists the commands you have issued previously in MySQL Shell. Issuing `\history` shows history entries in the order that they were issued with their history entry number, which can be used with the `\history delete` *`entry_number`* command.

The `\history` command provides the following options:

- Use `\history save` to save the history manually.
- Use `\history delete entrynumber` to delete the individual history entry with the given number.
- Use `\history delete` *`firstnumber-lastnumber`* to delete history entries within the range of the given entry numbers. If *`lastnumber`* goes past the last found history entry number, history entries are deleted up to and including the last entry.
- Use `\history delete` *`number-`* to delete the history entries from *`number`* up to and including the last entry.
- Use `\history delete -`*`number`* to delete the specified number of history entries starting with the last entry and working back. For example, `\history delete -10` deletes the last 10 history entries.
- Use `\history clear` to delete the entire history.

The history is not saved between sessions by default, so when you exit MySQL Shell the history of what you issued during the current session is lost. If you want to keep the history across sessions, enable the MySQL Shell `history.autoSave` option. For more information, see Section 5.5, “Code History”.

Rehash Command

When you have disabled the autocomplete name cache feature, use the `\rehash` command to manually update the cache. For example, after you load a new schema by issuing the `\use` *`schema`* command, issue `\rehash` to update the autocomplete name cache. After this autocomplete is aware of the names used in the database, and you can autocomplete text such as table names and so on. See Section 5.3, “Code Autocompletion”.

Option Command

The `\option` command enables you to query and change MySQL Shell configuration options in all modes. You can use the `\option` command to list the configuration options that have been set and show how their value was last changed. You can also use it to set and unset options, either for the session, or persistently in the MySQL Shell configuration file. For instructions and a list of the configuration options, see Section 13.4, “Configuring MySQL Shell Options”.

Pager Commands

You can configure MySQL Shell to use an external pager to read long onscreen output, such as the online help or the results of SQL queries. See Section 4.6, “Using a Pager”.

Show and Watch Commands

The `\show` command runs the named report, which can be either a built-in MySQL Shell report or a user-defined report that has been registered with MySQL Shell. You can specify the standard options for the command, and any options or additional arguments that the report supports. The `\watch` command runs a report in the same way as the `\show` command, but then refreshes the results at regular intervals until you cancel the command using **Ctrl + C**. For instructions, see Section 10.1.5, “Running MySQL Shell Reports”.

Edit Command

The `\edit` (`\e`) command opens a command in the default system editor for editing, then presents the edited command in MySQL Shell for execution. The command can also be invoked using the key combination **Ctrl-X Ctrl-E**. For details, see Section 5.4, “Editing Code”.

System Command

The `\system` (`\!`) command runs the operating system command that you specify as an argument to the command, then displays the output from the command in MySQL Shell. MySQL Shell returns an error if it was unable to execute the command. The output from the command is returned as given by the operating system, and is not processed by MySQL Shell's JSON wrapping function or by any external pager tool that you have specified to display output.

Chapter 4 Getting Started with MySQL Shell

Table of Contents

4.1 Starting MySQL Shell 15
4.2 MySQL Shell Sessions 15
4.2.1 Creating the `Session` Global Object While Starting MySQL Shell 17
4.2.2 Creating the `Session` Global Object After Starting MySQL Shell 17
4.2.3 Scripting Sessions in JavaScript and Python Mode 18
4.3 MySQL Shell Connections 20
4.3.1 Connecting using Individual Parameters 21
4.3.2 Connecting using login-path and Options Files 23
4.3.3 Connecting using Unix Sockets and Windows Named Pipes 24
4.3.4 Using Encrypted Connections 25
4.3.5 Using LDAP and Kerberos Authentication 26
4.3.6 Using an SSH Tunnel 29
4.3.7 Using Compressed Connections 32
4.4 Pluggable Password Store 35
4.4.1 Pluggable Password Configuration Options 36
4.4.2 Working with Credentials 37
4.5 MySQL Shell Global Objects 37
4.6 Using a Pager 38
4.7 Cloud Service Configuration 39
4.7.1 Oracle Cloud Infrastructure Object Storage 39
4.7.2 S3-compatible Storage 39
4.7.3 Azure Blob Storage 42

This section describes how to get started with MySQL Shell, explaining how to connect to a MySQL server instance, and how to choose a session type.

Important

It is recommended that you always use the most recent version available. The latest version of MySQL Shell can be used with any version of MySQL 5.7 or 8.0.

4.1 Starting MySQL Shell

When MySQL Shell is installed you have the `mysqlsh` command available. Open a terminal window (command prompt on Windows) and start MySQL Shell by issuing:

```
> mysqlsh
```

This opens MySQL Shell without connecting to a server, by default in JavaScript mode. You change mode using the `\sql`, `\py`, and `\js` commands.

4.2 MySQL Shell Sessions

In MySQL Shell, connections to MySQL Server instances are handled by a session object. The following types of session object are available:

- `Session`: Use this session object type for new application development to communicate with MySQL Server instances where X Protocol is available. X Protocol offers the best integration with MySQL Server. For X Protocol to be available, X Plugin must be installed and enabled on the MySQL Server instance, which it is by default from MySQL 8.0. In MySQL 5.7, X Plugin must be installed manually. See X Plugin for details. X Plugin listens to the port specified by `mysqlx_port`, which defaults to 33060, so specify this port with connections using a `Session`.

- `ClassicSession`: Use this session object type to interact with MySQL Server instances that do not have X Protocol available. This object is intended for running SQL against servers using classic MySQL protocol. The development API available for this kind of session is very limited. For example, there are none of the X DevAPI CRUD operations, no collection handling, and binding is not supported. For development, prefer `Session` objects whenever possible.

Important

`ClassicSession` is specific to MySQL Shell and cannot be used with other implementations of X DevAPI, such as MySQL Connectors.

When you make the first connection to a MySQL Server instance, which can be done either while starting MySQL Shell or afterwards, a MySQL Shell global object named `session` is created to represent this connection. This particular session object is global because once created, it can be used in all of the MySQL Shell execution modes: SQL mode, JavaScript mode, and Python mode. The connection it represents is therefore referred to as the global session. The variable `session` holds a reference to this session object, and can be used in MySQL Shell in JavaScript mode and Python mode to work with the connection.

The `session` global object can be either the `Session` type of session object or the `ClassicSession` type of session object, according to the protocol you select when making the connection to a MySQL Server instance. You can choose the protocol, and therefore the session object type, using a command option, or specify it as part of the connection data that you provide. To see information about the current global session, issue:

```
mysql-js []> session

<ClassicSession:user@example.com:3330>
```

When the global session is connected, this shows the session object type and the address of the MySQL Server instance to which the global session is connected.

If you choose a protocol explicitly or indicate it implicitly when making a connection, MySQL Shell tries to create the connection using that protocol, and returns an error if this fails. If your connection parameters do not indicate the protocol, MySQL Shell first tries to make the connection using X Protocol (returning the `Session` type of session object), and if this fails, tries to make the connection using classic MySQL protocol (returning the `ClassicSession` type of session object).

To verify the results of your connection attempt, use MySQL Shell's `\status` command or the `shell.status()` method. These display the connection protocol and other information about the connection represented by the `session` global object, or return “Not Connected” if the `session` global object is not connected to a MySQL server. For example:

```
mysql-js []> shell.status()
MySQL Shell version 8.0.18

Session type:                X Protocol
Connection Id:               198
Current schema:
Current user:                user@example.com
SSL:                         Cipher in use: TLS_AES_256_GCM_SHA384 TLSv1.3
Using delimiter:             ;
Server version:              8.0.18 MySQL Community Server - GPL
Protocol version:            X Protocol
Client library:              8.0.18
Connection:                  TCP/IP
TCP port:                    33060
Server characterset:         utf8mb4
Schema characterset:         utf8mb4
Client characterset:         utf8mb4
Conn. characterset:          utf8mb4
Compression:                 Enabled (zstd)
Uptime:                      31 min 42.0000 sec

Threads: 8  Questions: 2622  Slow queries: 0  Opens: 298  Flush tables: 3  Open tables: 217  Queries per se
```

This section focuses on explaining the session objects that represent connections to MySQL Server instances, and the `session` global object. For full instructions and examples for each of the ways mentioned in this section to connect to MySQL Server instances, and the other options that are available for the connections, see Section 4.3, “MySQL Shell Connections”.

4.2.1 Creating the `Session` Global Object While Starting MySQL Shell

When you start MySQL Shell from the command line, you can specify connection parameters using separate command options for each value, such as the user name, host, and port. For instructions and examples to start MySQL Shell and connect to a MySQL Server instance in this way, see Section 4.3.1, “Connecting using Individual Parameters”. When you use this connection method, you can add one of these options to choose the type of session object to create at startup to be the `session` global object:

- `--mysqlx` (`--mx`) creates a `Session` object, which connects to the MySQL Server instance using X Protocol.
- `--mysql` (`--mc`) creates a `ClassicSession` object, which connects to the MySQL Server instance using classic MySQL protocol.

For example, this command starts MySQL Shell and establishes an X Protocol connection to a local MySQL Server instance listening at port 33060:

```
$> mysqlsh --mysqlx -u user -h localhost -P 33060
```

If you are starting MySQL Shell in SQL mode, the `--sqlx` and `--sqlc` options include a choice of session object type, so you can specify one of these instead to make MySQL Shell use X Protocol or classic MySQL protocol for the connection. For a reference for all the `mysqlsh` command line options, see Section A.1, “mysqlsh — The MySQL Shell”.

As an alternative to specifying the connection parameters using individual options, you can specify them using a URI-like connection string. You can pass in this string when you start MySQL Shell from the command line, with or without using the optional `--uri` command option. When you use this connection method, you can include the `scheme` element at the start of the URI-like connection string to select the type of session object to create. `mysqlx` creates a `Session` object using X Protocol, or `mysql` creates a `ClassicSession` object using classic MySQL protocol. For example, either of these commands uses a URI-like connection string to start MySQL Shell and create a classic MySQL protocol connection to a local MySQL Server instance listening at port 3306:

```
$> mysqlsh --uri mysql://user@localhost:3306
$> mysqlsh mysql://user@localhost:3306
```

You can also specify the connection protocol as an option rather than as part of the URI-like connection string, for example:

```
$> mysqlsh --mysql --uri user@localhost:3306
```

For instructions and examples to connect to a MySQL Server instance in this way, see Connecting to the Server Using URI-Like Strings or Key-Value Pairs.

You may omit the connection protocol and let MySQL Shell automatically detect it based on your other connection parameters. For example, if you specify port 33060 and there is no option stating the connection protocol, MySQL Shell attempts to make the connection using X Protocol. If your connection parameters do not indicate the protocol, MySQL Shell first tries to make the connection using X Protocol, and if this fails, tries to make the connection using classic MySQL protocol.

4.2.2 Creating the `Session` Global Object After Starting MySQL Shell

If you started MySQL Shell without connecting to a MySQL Server instance, you can use MySQL Shell's `\connect` command or the `shell.connect()` method to initiate a connection and create the

`session` global object. Alternatively, the `shell.getSession()` method returns the `session` global object.

MySQL Shell's `\connect` command is used with a URI-like connection string, as described above and in Connecting to the Server Using URI-Like Strings or Key-Value Pairs. You can include the `scheme` element at the start of the URI-like connection string to select the type of session object to create, for example:

```
mysql-js> \connect mysqlx://user@localhost:33060
```

Alternatively, you can omit the `scheme` element and use the command's `--mysqlx` (`--mx`) option to create a `Session` object using X Protocol, or `--mysql` (`--mc`) to create a `ClassicSession` object using classic MySQL protocol. For example:

```
mysql-js> \connect --mysqlx user@localhost:33060
```

The `shell.connect()` method can be used in MySQL Shell as an alternative to the `\connect` command to create the `session` global object. This connection method can use a URI-like connection string, with the selected protocol specified as the `scheme` element. For example:

```
mysql-js> shell.connect('mysqlx://user@localhost:33060')
```

With the `shell.connect()` method, you can also specify the connection parameters using key-value pairs, supplied as a JSON object in JavaScript or as a dictionary in Python. The selected protocol (`mysqlx` or `mysql`) is specified as the value for the `scheme` key. For example:

```
mysql-js> shell.connect( {scheme:'mysqlx', user:'user', host:'localhost', port:33060} )
```

For instructions and examples to connect to a MySQL Server instance in these ways, see Connecting to the Server Using URI-Like Strings or Key-Value Pairs.

You may omit the connection protocol and let MySQL Shell automatically detect it based on your other connection parameters, such as specifying the default port for the protocol. To verify the protocol that was used for a connection, use MySQL Shell's `\status` command or the `shell.status()` method.

If you use the `\connect` command or the `shell.connect()` method to create a new connection when the `session` global object already exists (either created during startup or afterwards), MySQL Shell closes the existing connection represented by the `session` global object. This is the case even if you assign the new session object created by the `shell.connect()` method to a different variable. The value of the `session` global object (referenced by the `session` variable) is still updated with the new connection details. If you want to have multiple concurrent connections available, create these using the alternative functions described in Section 4.2.3, “Scripting Sessions in JavaScript and Python Mode”.

4.2.3 Scripting Sessions in JavaScript and Python Mode

You can use functions available in JavaScript and Python mode to create multiple session objects of your chosen types and assign them to variables. These session objects let you establish and manage concurrent connections to work with multiple MySQL Server instances, or with the same instance in multiple ways, from a single MySQL Shell instance.

Functions to create session objects are available in the `mysqlx` and `mysql` JavaScript and Python modules. These modules must be imported before use, which is done automatically when MySQL Shell is used in interactive mode. The function `mysqlx.getSession()` opens an X Protocol connection to a MySQL Server instance using the specified connection data, and returns a `Session` object to represent the connection. The functions `mysql.getClassicSession()` and `mysql.getSession()` open a classic MySQL protocol connection to a MySQL Server instance using the specified connection data, and return a `ClassicSession` object to represent the connection. With these functions, the connection protocol that MySQL Shell uses is built into the function rather

than being selected using a separate option, so you must choose the appropriate function to match the correct protocol for the port.

From MySQL Shell 8.0.20, MySQL Shell provides its own `openSession()` method in the `shell` global object, which can be used in either JavaScript or Python mode. `shell.openSession()` works with both X Protocol and classic MySQL protocol. You specify the connection protocol as part of the connection data, or let MySQL Shell automatically detect it based on your other connection parameters (such as the default port number for the protocol).

The connection data for all these functions can be specified as a URI-like connection string, or as a dictionary of key-value pairs. You can access the returned session object using the variable to which you assign it. This example shows how to open a classic MySQL protocol connection using the `mysql.getClassicSession()` function, which returns a `ClassicSession` object to represent the connection:

```
mysql-js> var s1 = mysql.getClassicSession('user@localhost:3306', 'password');
mysql-js> s1
<ClassicSession:user@localhost:3306>
```

This example shows how to use `shell.openSession()` in Python mode to open an X Protocol connection with compression required for the connection. A `Session` object is returned:

```
mysql-py> s2 = shell.open_session('mysqlx://user@localhost:33060?compression=required', 'password')
mysql-py> s2
<Session:user@localhost:33060>
```

Session objects that you create in JavaScript mode using these functions can only be used in JavaScript mode, and the same happens if the session object is created in Python mode. You cannot create multiple session objects in SQL mode. Although you can only reference session objects using their assigned variables in the mode where you created them, you can use the `shell.setSession()` method in any mode to set as the `session` global object a session object that you have created and assigned to a variable. For example:

```
mysql-js> var s3 = mysqlx.getSession('user@localhost:33060', 'password');
mysql-js> s3
<Session:user@localhost:33060>
mysql-js> shell.setSession(s3);
<Session:user@localhost:33060>
mysql-js> session
<Session:user@localhost:33060>
mysql-js> shell.status();
MySQL Shell version 8.0.18

Session type:                X Protocol
Connection Id:               5
Current schema:
Current user:                user@localhost
...
TCP port:                    33060
...
```

The session object `s3` is now available using the `session` global object, so the X Protocol connection it represents can be accessed from any of MySQL Shell's modes: SQL mode, JavaScript mode, and Python mode. Details of this connection can also now be displayed using the `shell.status()` method, which only displays the details for the connection represented by the `session` global object. If the MySQL Shell instance has one or more open connections but none of them are set as the `session` global object, the `shell.status()` method returns “Not Connected”.

A session object that you set using `shell.setSession()` replaces any existing session object that was set as the `session` global object. If the replaced session object was originally created and assigned to a variable using one of the `mysqlx` or `mysql` functions or `shell.openSession()`, it still exists and its connection remains open. You can continue to use this connection in the MySQL Shell mode where it was originally created, and you can make it into the `session` global object again at any time using `shell.setSession()`. If the replaced session object was created with the

`shell.connect()` method and assigned to a variable, the same is true. If the replaced session object was created while starting MySQL Shell, or using the `\connect` command, or using the `shell.connect()` method but without assigning it to a variable, its connection is closed, and you must recreate the session object if you want to use it again.

4.3 MySQL Shell Connections

MySQL Shell can connect to MySQL Server using both X Protocol and classic MySQL protocol. You can specify the MySQL server instance to which MySQL Shell connects globally in the following ways:

- When you start MySQL Shell, using the command parameters. See Section 4.3.1, “Connecting using Individual Parameters”.
- When MySQL Shell is running, using the `\connect` *`instance`* command. See Section 3.1, “MySQL Shell Commands”.
- When running in Python or JavaScript mode, using the `shell.connect()` method.

These methods of connecting to a MySQL server instance create the global session, which is a connection that can be used in all of the MySQL Shell execution modes: SQL mode, JavaScript mode, and Python mode. A MySQL Shell global object named `session` represents this connection, and the variable `session` holds a reference to it. You can also create multiple additional session objects that represent other connections to MySQL server instances, by using the `shell.openSession()`, `mysqlx.getSession()`, `mysql.getSession()`, or `mysql.getClassicSession()` function. These connections can be used in the modes where you created them, and one of them at a time can be assigned as MySQL Shell's global session so it can be used in all modes. For an explanation of session objects, how to operate on the global session, and how to create and manage multiple connections from a MySQL Shell instance, see Section 4.2, “MySQL Shell Sessions”.

All these different ways of connecting to a MySQL server instance support specifying the connection as follows:

- Parameters specified with a URI-like string use a syntax such as `myuser@example.com:3306/main-schema`. For the full syntax, see Connecting Using URI-Like Connection Strings.
- Parameters specified with key-value pairs use a syntax such as `{user:'myuser', host:'example.com', port:3306, schema:'main-schema'}`. These key-value pairs are supplied in language-natural constructs for the implementation. For example, you can supply connection parameters using key-value pairs as a JSON object in JavaScript, or as a dictionary in Python. For the full syntax, see Connecting Using Key-Value Pairs.

See Connecting to the Server Using URI-Like Strings or Key-Value Pairs for more information.

Important

Regardless of how you choose to connect it is important to understand how passwords are handled by MySQL Shell. By default connections are assumed to require a password. The password (which has a maximum length of 128 characters) is requested at the login prompt, and can be stored using Section 4.4, “Pluggable Password Store”. If the user specified has a passwordless account, which is insecure and not recommended, or if socket peer-credential authentication is in use (for example when using Unix socket connections), you must explicitly specify that no password is provided and the password prompt is not required. To do this, use one of the following methods:

- If you are connecting using a URI-like connection string, place a `:` after the `user` in the string but do not specify a password after it.
- If you are connecting using key-value pairs, provide an empty string using `''` after the `password` key.

- If you are connecting using individual parameters, either specify the `--no-password` option, or specify the `--password=` option with an empty value.

If you do not specify parameters for a connection the following defaults are used:

- `user` defaults to the current system user name.
- `host` defaults to `localhost`.
- `port` defaults to the X Plugin port 33060 when using an X Protocol connection, and port 3306 when using a classic MySQL protocol connection.

To configure the connection timeout use the `connect-timeout` connection parameter. The value of `connect-timeout` must be a non-negative integer that defines a time frame in milliseconds. The timeout default value is 10000 milliseconds, or 10 seconds. For example:

```
// Decrease the timeout to 2 seconds.
mysql-js> \connect user@example.com?connect-timeout=2000
// Increase the timeout to 20 seconds
mysql-js> \connect user@example.com?connect-timeout=20000
```

To disable the timeout set the value of `connect-timeout` to 0, meaning that the client waits until the underlying socket times out, which is platform dependent.

Certain operations that open many connections to servers can take a long time to execute when one or more servers are unreachable, for example, the `shell.connect()` command. The connection timeout may not provide enough time for a response.

From MySQL Shell 8.0.28, you can use the MySQL Shell configuration option `connectTimeout` to set the default connection timeout for any session not using AdminAPI.

Instead of a TCP connection, you can connect using a Unix socket file or a Windows named pipe. For instructions, see Section 4.3.3, “Connecting using Unix Sockets and Windows Named Pipes”.

If the MySQL server instance supports encrypted connections, you can enable and configure the connection to use encryption. For instructions, see Section 4.3.4, “Using Encrypted Connections”.

From MySQL Shell 8.0.27, the use of LDAP and Kerberos authentication is supported for classic MySQL protocol connections. For instructions to use these, see Section 4.3.5, “Using LDAP and Kerberos Authentication”.

From MySQL Shell 8.0.28, MySQL Shell supports SSH tunneling to connect to MySQL server instances. For instructions, see Section 4.3.6, “Using an SSH Tunnel”.

You can also request that the connection uses compression for all data sent between the MySQL Shell and the MySQL server instance. For instructions, see Section 4.3.7, “Using Compressed Connections”.

If the connection to the server is lost, you can use the `\reconnect` command, which makes MySQL Shell try several reconnection attempts for the current global session using the existing connection parameters. The `\reconnect` command is specified without any parameters or options. If those attempts are unsuccessful, you can make a fresh connection using the `\connect` command and specifying the connection parameters.

4.3.1 Connecting using Individual Parameters

In addition to specifying connection parameters using a connection string, it is also possible to define the connection data when starting MySQL Shell using separate command parameters for each value. For a full reference of MySQL Shell command options see Section A.1, “mysqlsh — The MySQL Shell”.

Use the following connection related parameters:

- `--user` (`-u`) *`value`*
- `--host` (`-h`) *`value`*
- `--port` (`-P`) *`value`*
- `--schema` or `--database` (`-D`) *`value`*
- `--socket` (`-S`)

The command options behave similarly to the options used with the `mysql` client described at Connecting to the MySQL Server Using Command Options.

Use the following command options to control whether and how a password is provided for the connection:

- `--password=`*`password`* (`-p`*`password`*) with a value supplies a password (up to 128 characters) to be used for the connection. With the long form `--password=`, you must use an equal sign and not a space between the option and its value. With the short form `-p`, there must be no space between the option and its value. If a space is used in either case, the value is not interpreted as a password and might be interpreted as another connection parameter.

  Specifying a password on the command line should be considered insecure. See End-User Guidelines for Password Security. You can use an option file to avoid giving the password on the command line.

- `--password` with no value and no equal sign, or `-p` without a value, requests the password prompt.
- `--no-password`, or `--password=` with an empty value, specifies that the user is connecting without a password. When connecting to the server, if the user has a passwordless account, which is insecure and not recommended, or if socket peer-credential authentication is in use (for Unix socket connections), you must use one of these methods to explicitly specify that no password is provided and the password prompt is not required.
- `--password1`, `--password2`, and `--password3`, available from MySQL Shell 8.0.28, are the passwords for accounts that require multifactor authentication. You can supply up to three passwords. The options work in the same way as the --password option, and --password1 is treated as equivalent to that option. These options are available from MySQL Shell 8.0.28, where they are only supported for classic MySQL protocol connections made using command-line arguments.

When parameters are specified in multiple ways, for example using both the `--uri` option and specifying individual parameters such as `--user`, the following rules apply:

- If an argument is specified more than once the value of the last appearance is used.
- If both individual connection arguments and `--uri` are specified, the value of `--uri` is taken as the base and the values of the individual arguments override the specific component from the base URI-like string.

  For example to override *`user`* from the URI-like string:

```
$> mysqlsh --uri user@localhost:33065 --user otheruser
```

Connections from MySQL Shell to a server can be encrypted, and can be compressed, if you request these features and the server supports them. For instructions to establish an encrypted connection, see Section 4.3.4, “Using Encrypted Connections”. For instructions to establish a compressed connection, see Section 4.3.7, “Using Compressed Connections”.

The following examples show how to use command parameters to specify connections. Attempt to establish an X Protocol connection with a specified user at port 33065:

```
$> mysqlsh --mysqlx -u user -h localhost -P 33065
```

Attempt to establish a classic MySQL protocol connection with a specified user, requesting compression for the connection:

```
$> mysqlsh --mysql -u user -h localhost -C
```

4.3.2 Connecting using login-path and Options Files

As of MySQL Shell 8.0.32, MySQL login paths and option files are supported. The following MySQL command line options are supported at the start of the command line:

- `--print-defaults`
- `--no-defaults`
- `--defaults-file`
- `--defaults-extra-file`
- `--defaults-group-suffix`
- `--login-path`

MySQL Shell reads a section in the MySQL configuration file, `[mysqlsh]`, which contains the MySQL Shell command line options.

MySQL Shell also reads the `[client]` section of the MySQL configuration file.

Note

Some `[client]` options are not supported by MySQL Shell, such as `local-infile`, and some options have the same name in both, but take different values, such as the `[client]` option `--compress` and the `[mysqlsh]` option `compress=value`.

MySQL Shell returns a specific error for such options, specifying the name of the option and the error.

For information on option file locations, order of precedence on Windows and Linux platforms, and option syntax, see Using Option Files. For information on login-path, see `--login-path`.

For example, if you define the following in your options file, `c:\my.ini` for example:

```
[mysqlsh]
sql

[client]
host=localhost
user=user1
port=3306
database=sakila
```

These options set the following:

- `sql`: the default MySQL Shell mode, SQL.
- `host=localhost`: defines the host as localhost.
- `user=user1`: defines the user as user1.
- `port=3306`: defines the connection port as 3306.
- `database=sakila`: defines the default schema as sakila.

Run `mysqlsh` without any connection string:

```
> mysqlsh

MySQL Shell 8.0.32-commercial

Copyright (c) 2016, 2022, Oracle and/or its affiliates.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.

Type '\help' or '\?' for help; '\quit' to exit.
Creating a Classic session to 'root@localhost:3306/sakila'
Fetching global names, object names from `sakila` for auto-completion... Press ^C to stop.
Your MySQL connection id is 93
Server version: 8.0.31-commercial MySQL Enterprise Server - Commercial
Default schema set to `sakila`.
localhost:3306 ssl  sakila  SQL >
```

Note

Options defined on the command line override all other values.

The order of precedence for MySQL Shell:

- Command line arguments
- Login path
- Option files
- Persisted Shell options

4.3.3 Connecting using Unix Sockets and Windows Named Pipes

On Unix, MySQL Shell connections default to using Unix sockets when the following conditions are met:

- A TCP port is not specified.
- A host name is not specified or it is equal to `localhost`.
- The `--socket` or `-S` option is specified, with or without a path to a socket file.

If you specify `--socket` with no value and no equal sign, or `-S` without a value, the default Unix socket file for the protocol is used. If you specify a path to an alternative Unix socket file, that socket file is used.

If a host name is specified but it is not `localhost`, a TCP connection is established instead. In this case, if a TCP port is not specified the default value of 3306 is used.

On Windows, for MySQL Shell connections using classic MySQL protocol, if you specify the host name as a period (.), MySQL Shell connects using a named pipe.

- If you are connecting using a URI-like connection string, specify `user@.`
- If you are connecting using key-value pairs, specify `{"host": "."}`
- If you are connecting using individual parameters, specify `--host=.` or `-h .`

By default, the pipe name `MySQL` is used. You can specify an alternative named pipe using the `--socket` option or as part of the URI-like connection string.

In URI-like strings, the path to a Unix socket file or Windows named pipe must be encoded, using either percent encoding or by surrounding the path with parentheses. Parentheses eliminate the need to percent encode characters such as the `/` directory separator character. If the path to a Unix socket file is included in a URI-like string as part of the query string, the leading slash must be percent encoded,

but if it replaces the host name, the leading slash must not be percent encoded, as shown in the following examples:

```
mysql-js> \connect user@localhost?socket=%2Ftmp%2Fmysql.sock
mysql-js> \connect user@localhost?socket=(/tmp/mysql.sock)
mysql-js> \connect user@/tmp%2Fmysql.sock
mysql-js> \connect user@(/tmp/mysql.sock)
```

On Windows only, the named pipe must be prepended with the characters `\\.\` as well as being either encoded using percent encoding or surrounded with parentheses, as shown in the following examples:

```
(\\.\named:pipe)
\\.\named%3Apipe
```

Important

On Windows, if one or more MySQL Shell sessions are connected to a MySQL Server instance using a named pipe and you need to shut down the server, you must first close the MySQL Shell sessions. Sessions that are still connected in this way can cause the server to hang during the shutdown procedure. If this does happen, exit MySQL Shell and the server will continue with the shutdown procedure.

For more information on connecting with Unix socket files and Windows named pipes, see Connecting to the MySQL Server Using Command Options and Connecting to the Server Using URI-Like Strings or Key-Value Pairs.

4.3.4 Using Encrypted Connections

Using encrypted connections is possible when connecting to a TLS (sometimes referred to as SSL) enabled MySQL server. Much of the configuration of MySQL Shell is based on the options used by MySQL server, see Using Encrypted Connections for more information.

To configure an encrypted connection at startup of MySQL Shell, use the following command options:

- `--ssl` : Deprecated, to be removed in a future version. Use `--ssl-mode`. This option enables or disables encrypted connections.
- `--ssl-mode` : This option specifies the desired security state of the connection to the server.
- `--ssl-ca=`*`file_name`*: The path to a file in PEM format that contains a list of trusted SSL Certificate Authorities.
- `--ssl-capath=`*`dir_name`*: The path to a directory that contains trusted SSL Certificate Authority certificates in PEM format.
- `--ssl-cert=`*`file_name`*: The name of the SSL certificate file in PEM format to use for establishing an encrypted connection.
- `--ssl-cipher=`*`name`*: The name of the SSL cipher to use for establishing an encrypted connection.
- `--ssl-key=`*`file_name`*: The name of the SSL key file in PEM format to use for establishing an encrypted connection.
- `--ssl-crl=`*`name`*: The path to a file containing certificate revocation lists in PEM format.
- `--ssl-crlpath=`*`dir_name`*: The path to a directory that contains files containing certificate revocation lists in PEM format.
- `--tls-ciphersuites=`*`suites`*: The TLS cipher suites permitted for encrypted connections, specified as a colon separated list of TLS cipher suite names. For example `--tls-ciphersuites=`*`TLS_DHE_PSK_WITH_AES_128_GCM_SHA256:TLS_CHACHA20_POLY1305_SHA256`*. Added in version 8.0.18.

- `--tls-version=version`: The TLS protocols permitted for encrypted connections, specified as a comma separated list. For example `--tls-version=TLSv1.2,TLSv1.3`.

  From MySQL 8.0.28, the TLSv1 and TLSv1.1 protocols are not supported by MySQL Server, and MySQL Shell cannot make a TLS/SSL connection with the protocol set to TLSv1 or TLSv1.1. If you attempt to make a connection using TLS/SSL from any version of MySQL Shell to a MySQL Server instance at 8.0.28 or above, and you specify the TLSv1 or TLSv1.1 protocol using the`--tls-version` option, you will see the following results:

  - For TCP connections, the connection fails, and an error is returned to MySQL Shell.
  - For socket connections, if `--ssl-mode` is set to `REQUIRED`, the connection fails. If `--ssl-mode` is not set to `REQUIRED`, the connection is made but with TLS/SSL disabled.

  The TLSv1 and TLSv1.1 protocols were deprecated from MySQL 8.0.26. For background, refer to the IETF memo Deprecating TLSv1.0 and TLSv1.1. Make connections between MySQL Shell and MySQL Server using the more-secure TLSv1.2 and TLSv1.3 protocols. TLSv1.3 requires that both the MySQL server and the client application be compiled with OpenSSL 1.1.1 or higher. For more information on the support for TLS protocol versions in MySQL Server releases, see Removal of Support for the TLSv1 and TLSv1.1 Protocols.

Alternatively, the SSL options can be encoded as part of a URI-like connection string as part of the query element. The available SSL options are the same as those listed above, but written without the preceding hyphens. For example, `ssl-ca` is the equivalent of `--ssl-ca`.

Paths specified in a URI-like string must be percent encoded, for example:

```
ssluser@127.0.0.1?ssl-ca%3D%2Froot%2Fclientcert%2Fca-cert.pem%26ssl-cert%3D%2Fro\
ot%2Fclientcert%2Fclient-cert.pem%26ssl-key%3D%2Froot%2Fclientcert%2Fclient-key
.pem
```

See Connecting to the Server Using URI-Like Strings or Key-Value Pairs for more information.

To establish an encrypted connection for a scripting session in JavaScript or Python mode, set the SSL information in the `connectionData` dictionary. For example:

```
mysql-js> var session=mysqlx.getSession({host: 'localhost',
                                         user: 'root',
                                         password: 'password',
                                         ssl_ca: "path_to_ca_file",
                                         ssl_cert: "path_to_cert_file",
                                         ssl_key: "path_to_key_file"});
```

Sessions created using `mysqlx.getSession()`, `mysql.getSession()`, or `mysql.getClassicSession()` use `ssl-mode=REQUIRED` as the default if no `ssl-mode` is provided, and neither `ssl-ca` nor `ssl-capath` is provided. If no `ssl-mode` is provided and any of `ssl-ca` or `ssl-capath` is provided, created sessions default to `ssl-mode=VERIFY_CA`.

See Connecting Using Key-Value Pairs for more information.

4.3.5 Using LDAP and Kerberos Authentication

MySQL Enterprise Edition supports authentication methods that enable MySQL Server to use LDAP (Lightweight Directory Access Protocol), LDAP with Kerberos, or native Kerberos to authenticate MySQL users. MySQL Shell 8.0.27 supports both LDAP and Kerberos authentication for classic MySQL protocol connections. This functionality is not supported for X Protocol connections.

The sections that follow describe how to enable connections to MySQL server using LDAP and Kerberos authentication. It is assumed that the server is running with the server-side plugin enabled and that the client-side plugin is available on the client host.

- Simple LDAP Authentication

- SASL-Based LDAP Authentication
- GSSAPI/Kerberos Authentication Through LDAP SASL
- Kerberos Authentication

Simple LDAP Authentication

MySQL and LDAP work together to fetch user, credential, and group information. For an overview of the simple LDAP authentication process, see How LDAP Authentication of MySQL Users Works. To use simple LDAP authentication with MySQL Shell, the following conditions must be satisfied:

- A user account must be created on the MySQL server that is set up to communicate with the LDAP server. The MySQL user must be identified with the `authentication_ldap_simple` server-side plugin and optionally the LDAP user distinguished name (DN). For example:

  ```
  CREATE USER 'admin'@'localhost'
     IDENTIFIED WITH authentication_ldap_simple
     BY 'uid=admin,ou=People,dc=my-domain,dc=com';
  ```

  The BY clause in this example indicates which LDAP entry the MySQL account authenticates against. Specific attributes of the DN may vary depending on the LDAP server.

- MySQL Shell uses the client-side `mysql_clear_password` plugin, which sends the password to the server as cleartext. No password hashing or encryption is used, so a secure connection (using SSL or sockets) between the MySQL Shell and server is required. For more information, see Section 4.3.4, “Using Encrypted Connections” or Section 4.3.3, “Connecting using Unix Sockets and Windows Named Pipes”.

- To minimize the security risk, the `mysql_clear_password` plugin must be enabled explicitly by setting the value of the `--auth-method` command-line option to `clear_text_password` on a secure connection. For example, the following command permits you to establish a global session for the user created in the previous example:

  ```
  $> mysqlsh admin@localhost:3308 --auth-method=clear_text_password
  Please provide the password for 'admin@localhost:3308': admin_password (admin LDAP password)
  ```

Note

You can also set the environment variable, `LIBMYSQL_ENABLE_CLEARTEXT_PLUGIN`, and enable the `mysql_clear_password` plugin for all client connections. However, this method is inherently insecure and is not recommended for any scenario other than testing. For more information, see Client-Side Cleartext Pluggable Authentication.

SASL-Based LDAP Authentication

MySQL Server is able to accept connections from users defined outside the MySQL grant tables in LDAP directories. The client-side and server-side SASL LDAP plugins use SASL messages for secure transmission of credentials within the LDAP protocol (see Using LDAP Pluggable Authentication).

For SASL-based authentication, the MySQL user must be identified with the `authentication_ldap_sasl` server-side plugin and optionally an LDAP entry the MySQL account authenticates against. For example:

```
CREATE USER 'sammy'@'localhost'
   IDENTIFIED WITH authentication_ldap_sasl
   BY 'uid=sammy_ldap,ou=People,dc=my-domain,dc=com';
```

The `authentication_ldap_sasl_client` client-side plugin ships with the MySQL Server packages rather than being built into the `libmysqlclient` client library. MySQL Shell provides the

persistent connection option `shell.options.mysqlPluginDir` that enables you to define where the required plugin is located. Alternatively, you can override the persistent setting by specifying a path with the non-persistent command-line option `--mysql-plugin-dir`. For example, the following command permits you to establish a global session on a Linux host for the user created in the previous example:

```
$> mysqlsh sammy@localhost:3308 --mysql-plugin-dir="/usr/local/mysql/lib/plugin"
Please provide the password for 'sammy@localhost:3308': sammy_password (sammy_ldap LDAP password)
```

For additional usage examples, see LDAP Authentication with Proxying and LDAP Authentication Group Preference and Mapping Specification.

GSSAPI/Kerberos Authentication Through LDAP SASL

MySQL Shell also supports Kerberos authentication through LDAP SASL. Using the Generic Security Service Application Program Interface (GSSAPI) security abstraction interface, a connection of this type authenticates to Kerberos to obtain service credentials, then uses those credentials in turn to enable secure access to other services. GSSAPI/Kerberos is supported as an LDAP authentication method for MySQL servers and MySQL Shell on Linux only.

A GSSAPI library and Kerberos services must be available to MySQL Server for the connection to succeed. See The GSSAPI/Kerberos Authentication Method for server-side configuration information.

The following general example creates proxy user named `lucy@MYSQL.LOCAL` that assumes the privileges of the proxied user named `proxied_krb_usr`. It presumes the realm domain `MYSQL.LOCAL` is configured in the `/etc/krb5.conf` Kerberos configuration file.

Note

The user part of the account name includes the principal domain, so 'lucy@MYSQL.LOCAL' is quoted as a single value for LDAP Kerberos authentication.

```
CREATE USER 'lucy@MYSQL.LOCAL'
   IDENTIFIED WITH authentication_ldap_sasl
   BY '#krb_grp=proxied_krb_user';
CREATE USER 'proxied_krb_user';
GRANT ALL PRIVILEGES ON my_db.* TO 'proxied_krb_user';
GRANT PROXY on 'proxied_krb_user' TO 'lucy@MYSQL.LOCAL';
```

The following command permits you to establish a global session on a Linux host for the user created in the previous example. You must specify the location of the server's plugin directory, either as the persistent `shell.options.mysqlPluginDir` connection option or as a non-persistent command option, for example:

```
$> mysqlsh lucy%40MYSQL.LOCAL:password@localhost:3308/my_db
--mysql-plugin-dir="/usr/local/mysql/lib/plugin"
```

In this example, percent encoding (`%40`) replaces the reserved `@` character in the principal name and *`password`* is the value set for the MySQL Server variable `authentication_ldap_sasl_bind_root_pwd`. For the list of server variables related to Kerberos authentication through LDAP SASL, see Configure the Server-Side SASL LDAP Authentication Plugin for GSSAPI/Kerberos.

Prior to invoking MySQL Shell, you can obtain and cache a ticket-granting ticket from the key distribution center independently of MySQL. In this case, invoke MySQL Shell without specifying a user-name or password option:

```
$> mysqlsh localhost:3308/my_db --auth-method=authentication_ldap_sasl_client
--mysql-plugin-dir="/usr/local/mysql/lib/plugin"
```

Specifying the `--auth-method=authentication_ldap_sasl_client` option is mandatory when user credentials are omitted.

Kerberos Authentication

MySQL Shell is capable of establishing connections for accounts that use the `authentication_kerberos` server-side authentication plugin, provided that the correct Kerberos tickets are available or can be obtained from Kerberos. As of MySQL Enterprise Edition 8.0.27, that capability is available on hosts running Linux and Windows (version 8.0.26 supports Linux only). For detailed setup information, see Kerberos Pluggable Authentication.

Kerberos authentication can combine the user name (for example, `lucy`) and the realm domain specified in the user account (for example, `MYSQL.LOCAL`) to construct the user principal name (UPN), such as `lucy@MYSQL.LOCAL`. To create a MySQL account that corresponds to the UPN `lucy@MYSQL.LOCAL`, use this statement:

```
CREATE USER 'lucy'
   IDENTIFIED WITH authentication_kerberos
   BY 'MYSQL.LOCAL';
```

The client-side plugin uses the UPN and password to obtain a ticket-granting ticket (TGT), uses the TGT to obtain a MySQL service ticket (ST), and uses the ST to authenticate to the MySQL server.

The following command permits you to establish a global session on a Linux host for the user created in the previous example. You must specify the location of the server's plugin directory, either as the persistent `shell.options.mysqlPluginDir` connection option or as a non-persistent command option, for example:

```
$> mysqlsh lucy:3308 --mysql-plugin-dir="/usr/local/mysql/lib/plugin"
Please provide the password for 'lucy@localhost:3308': UPN_password
```

Prior to invoking MySQL Shell, you can obtain and cache a TGT from the key distribution center independently of MySQL. In this case, invoke MySQL Shell without specifying a user-name or password option:

```
$> mysqlsh localhost:3308 --auth-method=authentication_kerberos_client
--mysql-plugin-dir="/usr/local/mysql/lib/plugin"
```

Specifying the `--auth-method=authentication_kerberos_client` option is mandatory when user credentials are omitted.

4.3.6 Using an SSH Tunnel

From release 8.0.28, MySQL Shell supports SSH tunneling for connections to MySQL server instances. An SSH tunnel lets unencrypted traffic pass over an encrypted connection, and enables authorized remote access to servers that are protected from outside connections by a firewall.

The use of AdminAPI commands is not supported over connections made from MySQL Shell using SSH tunneling, with the exception of the commands to deploy, start, stop, kill, and delete sandbox instances (`dba.deploySandboxInstance`, `dba.startSandboxInstance`, `dba.stopSandboxInstance`, `dba.killSandboxInstance`, and `dba.deleteSandboxInstance`). The sandbox commands are always executed locally to the MySQL Shell instance.

Once established, an SSH tunnel can be shared between connections to the same host from the same user connecting from the same remote server instance. The MySQL Shell function `shell.listSshConnections()` lists the currently connected and active SSH tunnels from the MySQL Shell session, with the URI of the SSH server and of the connected MySQL server instance. If you specify the same SSH connection details, MySQL Shell automatically reuses the existing tunnel.

You can select the SSH configuration file and identity file (private key) that are used for the connection. When you set up an SSH tunnel, MySQL Shell selects an SSH configuration file in the following order of priority:

1. An SSH configuration file that you specify as a connection option.
2. An SSH configuration file that you set as a default using the MySQL Shell configuration option `ssh.configFile`. For instructions to set this option, see Section 13.4, “Configuring MySQL Shell Options”.
3. The standard SSH configuration file `~/.ssh/config`.

The known hosts file is read from the default location (`~/.ssh/known_hosts`) unless a different configuration is set in the SSH configuration file.

For the identity file (private key), you can specify a custom file with the `ssh-identity-file` option at connection time. There is no option to set a custom default for the identity file. If you do not specify one, the SSH library uses the following sequence of authentication attempts until one succeeds:

1. If an SSH agent is in use, authentication is attempted with the identity files configured there if available.
2. If an identity file is specified for the target host in the SSH configuration file, authentication is attempted using that file.
3. If neither of those options is available or the authentication attempt fails, authentication is attempted using the standard private key file in the SSH configuration folder (`~/.ssh/id_rsa`).

The default buffer size for data transfer through the SSH tunnel is 10240 bytes. You can change this by setting the MySQL Shell configuration option `ssh.bufferSize`. For instructions to set this option, see Section 13.4, “Configuring MySQL Shell Options”.

SSH tunneling is available when you use any of the MySQL Shell connection methods - the `shell.connect()` method, `mysqlsh` command parameters, or the `\connect` MySQL Shell command.

`shell.connect()`

When you use the `shell.connect()` method to connect while MySQL Shell is running, you can specify a URI for connection to the SSH server, or use key-value pairs for the connection data. The following options are available with this method:

- `ssh`: The URI for connection to the SSH server. The URI format is `[user@]host[:port]`.
- `uri`: The URI for the MySQL server instance that is to be accessed through the SSH tunnel. The URI format is `[scheme://] [user@]host:port`. Do not use the base connection parameters (`scheme`, `user`, `host`, `port`) to specify the MySQL server connection for SSH tunneling, just use this option. The port must be specified.
- `ssh-password`: The password for the connection to the SSH server.
- `ssh-config-file`: An SSH configuration file for the connection to the SSH server.
- `ssh-identity-file`: An identity file to use for the connection to the SSH server.
- `ssh-identity-pass`: The passphrase for the identity file specified by the `ssh-identity-file` option.

These options are also available when you use the `shell.openSession()` method, which works in the same way

as `shell.connect()` but creates and returns a `session` object, rather than setting it as the global session for MySQL Shell. For full instructions to use this connection method and the other options that are available, see Connecting to the Server Using URI-Like Strings or Key-Value Pairs.

`mysqlsh` command parameters

When you connect using command-line options while MySQL is starting up, you can specify a URI for connection to the SSH server. The following options are available with this method:

- `--ssh`: The URI for connection to the SSH server. The URI format is `[user@]host[:port]`. When you use this option, the port for connection to the MySQL server instance must be specified in the MySQL instance URI.
- `--ssh-config-file`: An SSH configuration file for the connection to the SSH server. If you specify this option with an empty value, the custom default SSH configuration file specified by `--ssh.configFile` is ignored, and the `~/.ssh/config` file is used instead.
- `--ssh-identity-file`: An identity file to use for the connection to the SSH server.

For full instructions to use this connection method and the other options that are available, see Section 4.3.1, “Connecting using Individual Parameters”.

`\connect` MySQL Shell command

When you connect using the `\connect` command while MySQL Shell is running, you can specify a URI for connection to the SSH server. There are no additional options for the SSH connection, so you must use the default identity file `~/.ssh/id_rsa`, and the default SSH configuration file, which can be either the standard file `~/.ssh/config` or a custom default that you set using the MySQL Shell configuration option `ssh.configFile` (see Section 13.4, “Configuring MySQL Shell Options”).

To get additional setup options you can create an SSH tunnel using the `shell.connect()` method or on the command line while MySQL Shell is starting, and then reuse it with the `\connect` command. When you are in a MySQL Shell session, you can view the currently connected SSH tunnels using the `shell.listSshConnections()` command.

MySQL Shell’s Secret Store can store passwords and passphrases for connection to the SSH server and for the identity file, to be automatically retrieved for future connections. If you provide a password or passphrase in the connection options it is used instead of any password that is stored in the Secret Store. Note that although there are options to allow it, specifying an explicit password in the connection data is insecure and not recommended. MySQL Shell prompts for a password interactively when one is required, either for the connection to the SSH server or for the identity file. For example:

```
mysql-js> shell.connect({uri:"mysql://root:sandbox@192.0.2.3:3306",
        > ssh:"root@198.51.100.4:2222", "ssh-identity-file":"/home/hanna/.ssh/config_pw"})
Creating a Classic session to 'root@192.0.2.3:3306'
Opening SSH tunnel to 198.51.100.4:2222...
Please provide key passphrase for /home/hanna/.ssh/config_pw: ********
Save password for 'file:/home/hanna/.ssh/config_pw'?
[Y]es/[N]o/Ne[v]er (default No): y
Fetching schema names for autocompletion... Press ^C to stop.
Your MySQL connection id is 7869
Server version: 8.0.28 MySQL Community Server - GPL
```

```
No default schema selected; type \use <schema> to set one.
<ClassicSession:root@192.0.2.3:3306>
```

4.3.7 Using Compressed Connections

From MySQL Shell 8.0.14, you can request compression for MySQL Shell connections that use classic MySQL protocol, and, from MySQL Shell 8.0.20, also for MySQL Shell connections that use X Protocol. When compression is requested for a session, if the server supports compression and can agree a compression algorithm with MySQL Shell, all information sent between the client and the server is compressed. Compression is also applied if requested to connections used by a MySQL Shell utility, such as the upgrade checker utility.

For X Protocol connections, the default is that compression is requested, and uncompressed connections are allowed if the negotiations for a compressed connection do not succeed. For classic MySQL protocol connections, the default is that compression is disabled. After the connection has been made, the MySQL Shell `\status` command shows whether or not compression is in use for a session. The command displays a `Compression:` line that says `Disabled` or `Enabled` to indicate whether the connection is compressed. If compression is enabled, the compression algorithm in use is also displayed.

You can set the `defaultCompress` MySQL Shell configuration option to request compression for every global session. Because the default for X Protocol connections is that compression is requested where the MySQL Shell release supports this, this configuration option only has an effect for classic MySQL protocol connections.

For more information on how connection compression operates for X Protocol connections, see Connection Compression with X Plugin. For more information on how connection compression operates for classic MySQL protocol connections, and on the compression settings and capabilities of a MySQL Server instance, see Connection Compression Control.

4.3.7.1 Compression Control For MySQL Shell 8.0.20 And Later

From MySQL Shell 8.0.20, for X Protocol connections and classic MySQL protocol connections, whenever you create a session object to manage a connection to a MySQL Server instance, you can specify whether compression is required, preferred, or disabled for that connection.

- `required` requests a compressed connection from the server, and the connection fails if the server does not support compression or cannot agree with MySQL Shell on a compression protocol.
- `preferred` requests a compressed connection from the server, and falls back to an uncompressed connection if the server does not support compression or cannot agree with MySQL Shell on a compression protocol. This is the default for X Protocol connections.
- `disabled` requests an uncompressed connection, and the connection fails if the server only permits compressed connections. This is the default for classic MySQL protocol connections.

From MySQL Shell 8.0.20, you can also choose which compression algorithms are allowed for the connection. By default, MySQL Shell proposes the zlib, LZ4, and zstd algorithms to the server for X Protocol connections, and the zlib and zstd algorithms for classic MySQL protocol connections (which do not support the LZ4 algorithm). You can specify any combination of these algorithms. The order in which you specify the compression algorithms is the order of preference in which MySQL Shell proposes them, but the server might not be influenced by this preference, depending on the protocol and the server configuration.

Specifying any compression algorithm or combination of them automatically requests compression for the connection, so you can do that instead of using a separate parameter to specify whether compression is required, preferred, or disabled. With this method of connection compression control, you indicate whether compression is required or preferred by adding the option `uncompressed` (which allows uncompressed connections) to the list of compression algorithms. If you do include `uncompressed`, compression is preferred, and if you do not include it, compression is required. You can also pass in `uncompressed` on its own to specify that compression is disabled. If you specify in

a separate parameter that compression is required, preferred, or disabled, this takes precedence over using `uncompressed` in the list of compression algorithms.

You can also specify a numeric compression level for the connection, which applies to any compression algorithm for X Protocol connections, or to the zstd algorithm only on classic MySQL protocol connections. For X Protocol connections, if the specified compression level is not acceptable to the server for the algorithm that is eventually selected, the server chooses an appropriate setting according to the behaviors listed in Connection Compression with X Plugin. For example, if MySQL Shell requests a compression level of 7 for the zlib algorithm, and the server's `mysqlx_deflate_max_client_compression_level` system variable (which limits the maximum compression level for deflate, or zlib, compression) is set to the default of 5, the server uses the highest permitted compression level of 5.

If the MySQL server instance does not support connection compression for the protocol (which is the case before MySQL 8.0.19 for X Protocol connections), or if it supports connection compression but does not support specifying connection algorithms and a compression level, MySQL Shell establishes the connection without specifying the unsupported parameters.

To request compression for a connection from MySQL Shell 8.0.20, use one of the following methods:

- If you are starting MySQL Shell from the command line and specifying connection parameters using separate command options, use the `--compress` (`-C`) option, specifying whether compression is required, preferred, or disabled for the connection. For example:

  ```
  $> mysqlsh --mysqlx -u user -h localhost -C required
  ```

  The `--compress` (`-C`) option is compatible with earlier releases of MySQL Shell (back to MySQL 8.0.14) and still accepts the boolean settings from those releases. From MySQL Shell 8.0.20, if you specify just `--compress` (`-C`) without a parameter, compression is required for the connection.

  The above example for an X Protocol connection proposes the zlib, LZ4, and zstd algorithms to the server in that order of preference. If you prefer an alternative combination of compression algorithms, you can specify this by using the `--compression-algorithms` option to specify a string with a comma-separated list of permitted algorithms. For X Protocol connections, you can use `zlib`, `lz4`, and `zstd` in any combination and order of preference. For classic MySQL protocol connections, you can use `zlib` and `zstd` in any combination and order of preference. The following example for a classic MySQL protocol connection allows only the zstd algorithm:

  ```
  $> mysqlsh --mysql -u user -h localhost -C preferred --compression-algorithms=zstd
  ```

  You can also use just `--compression-algorithms` without the `--compress` (`-C`) option to request compression. In this case, add `uncompressed` to the list of algorithms if you want to allow uncompressed connections, or omit it if you do not want to allow them. This style of connection compression control is compatible with other MySQL clients such as `mysql` and `mysqlbinlog`. The following example for a classic MySQL protocol connection has the same effect as the example above where `preferred` is specified as a separate option, that is, to propose compression with the zstd algorithm but fall back to an uncompressed connection:

  ```
  $> mysqlsh --mysql -u user -h localhost --compression-algorithms=zstd,uncompressed
  ```

  You can configure the compression level using the `--compression-level` or `--zstd-compression-level` options, which are validated for classic MySQL protocol connections, but not for X Protocol connections. `--compression-level` specifies an integer for the compression level for any algorithm for X Protocol connections, or for the zstd algorithm only on classic MySQL protocol connections. `--zstd-compression-level` specifies an integer from 1 to 22 for the compression level for the zstd algorithm, and is compatible with other MySQL clients such as `mysql` and `mysqlbinlog`. For example, these connection parameters for an X Protocol connection specify that compression is required for the global session and must use the LZ4 or zstd algorithm, with a requested compression level of 5:

  ```
  $> mysqlsh --mysqlx -u user -h localhost -C required --compression-algorithms=lz4,zstd --compression-
  ```

- If you are using a URI-like connection string to specify connection parameters, either from the command line, or with MySQL Shell's `\connect` command, or with the `shell.connect()`, `shell.openSession()`, `mysqlx.getSession()`, `mysql.getSession()`, or `mysql.getClassicSession()` function, use the `compression` parameter in the query string to specify whether compression is required, preferred, or disabled. For example:

```
mysql-js> \connect user@example.com?compression=preferred
```

```
$> mysqlsh mysqlx://user@localhost:33060?compression=disabled
```

  Select compression algorithms using the `compression-algorithms` parameter, and a compression level using the `compression-level` parameter, as for the command line options. (There is no zstd-specific compression level parameter for a URI-like connection string.) You can also use the `compression-algorithms` parameter without the `compression` parameter, including or omitting the `uncompressed` option to allow or disallow uncompressed connections. For example, both these sets of connection parameters specify that compression is preferred but uncompressed connections are allowed, the zlib and zstd algorithms are acceptable, and a compression level of 4 should be used:

```
mysql-js> \connect user@example.com:33060?compression=preferred&compression-algorithms=zlib,zstd&compress
```

```
mysql-js> \connect user@example.com:33060?compression-algorithms=zlib,zstd,uncompressed&compression-level
```

- If you are using key-value pairs to specify connection parameters, either with MySQL Shell's `\connect` command or with the `shell.connect()`, `shell.openSession()`, `mysqlx.getSession()`, `mysql.getSession()`, or `mysql.getClassicSession()` function, use the `compression` parameter in the dictionary of options to specify whether compression is required, preferred, or disabled. For example:

```
mysql-js>  var s1=mysqlx.getSession({host: 'localhost',
                                     user: 'root',
                                     password: 'password',
                                     compression: 'required'});
```

  Select compression algorithms using the `compression-algorithms` parameter, and a compression level using the `compression-level` parameter, as for the command line and URI-like connection string methods. (There is no zstd-specific compression level parameter for key-value pairs.) You can also use the `compression-algorithms` parameter without the `compression` parameter, including or omitting the `uncompressed` option to allow or disallow uncompressed connections.

4.3.7.2 Compression Control For MySQL Shell 8.0.14 Through 8.0.19

In releases from MySQL Shell 8.0.14 through 8.0.19, compression can be requested only for connections that use classic MySQL protocol. The default is that compression is not requested. Compression in these releases uses the zlib compression algorithm. You cannot require compression in these releases, so if compression is not supported by the server, the session falls back to an uncompressed connection.

In these MySQL Shell releases, compression control is limited to enabling (by specifying `true`) or disabling (by specifying `false`) compression for a connection. If you use a MySQL Shell release with this compression control to connect to a server instance at MySQL 8.0.18 or later, where client requests for compression algorithms are supported, enabling compression is equivalent to proposing the algorithm set `zlib`,`uncompressed`.

MySQL Shell cannot request compression in releases before 8.0.14.

To request compression for a connection in MySQL Shell 8.0.14 through 8.0.19, use one of the following methods:

- If you are starting MySQL Shell from the command line and specifying connection parameters using separate command options, use the `--compress` (`-C`) option, for example:

```
$> mysqlsh --mysql -u user -h localhost -C
```

- If you are using a URI-like connection string to specify connection parameters, either from the command line, or with MySQL Shell's `\connect` command, or with the `shell.connect()` method, use the `compression=true` parameter in the query string:

```
mysql-js> \connect user@example.com?compression=true
```

```
$> mysqlsh mysql://user@localhost:3306?compression=true
```

- If you are using key-value pairs to specify connection parameters, either with MySQL Shell's `\connect` command or with the `mysql.getClassicSession()` method, use the `compression` parameter in the dictionary of options:

```
mysql-js> var s1=mysql.getClassicSession({host: 'localhost',
                                          user: 'root',
                                          password: 'password',
                                          compression: 'true'});
```

4.4 Pluggable Password Store

To make working with MySQL Shell more fluent and secure you can persist the password for a server connection using a secret store, such as a keychain. You enter the password for a connection interactively and it is stored with the server URL as credentials for the connection. For example:

```
mysql-js> \connect user@localhost:3310
Creating a session to 'user@localhost:3310'
Please provide the password for 'user@localhost:3310': ********
Save password for 'user@localhost:3310'? [Y]es/[N]o/Ne[v]er (default No): y
```

Once the password for a server URL is stored, whenever MySQL Shell opens a session it retrieves the password from the configured Secret Store Helper to log in to the server without having to enter the password interactively. The same holds for a script executed by MySQL Shell. If no Secret Store Helper is configured the password is requested interactively.

Important

MySQL Shell only persists the server URL and password through the means of a Secret Store and does not persist the password on its own.

Passwords are only persisted when they are entered manually. If a password is provided using either a server URI-like connection string or at the command line when running `mysqlsh` it is not persisted.

The maximum password length that is accepted for connecting to MySQL Shell is 128 characters.

MySQL Shell provides built-in support for the following Secret Stores:

- MySQL login-path, which is available on all platforms supported by the MySQL server. It is provided by the MySQL configuration utility `mysql_config_editor` which offers persistent storage. See mysql_config_editor — MySQL Configuration Utility. In releases before MySQL Shell 8.0.30, `mysql_config_editor` is available as long as the MySQL client package is installed on the system. From MySQL Shell 8.0.30, Linux builds of MySQL Shell bundle `mysql_config_editor` so that the functionality can be used if the MySQL client package is not installed on the system.
- macOS keychain, see here.
- Windows API, see here.

When MySQL Shell is running in interactive mode, password retrieval is performed whenever a new session is initiated and the user is going to be prompted for a password. Before prompting, the Secret

Store Helper is queried for a password using the session's URL. If a match is found this password is used to open the session. If the retrieved password is invalid, a message is added to the log, the password is erased from the Secret Store and MySQL Shell prompts you for a password.

If MySQL Shell is running in noninteractive mode (for example `--no-wizard` was used), password retrieval is performed the same way as in interactive mode. But in this case, if a valid password is not found by the Secret Store Helper, MySQL Shell tries to open a session without a password.

The password for a server URL can be stored whenever a successful connection to a MySQL server is made and the password was not retrieved by the Secret Store Helper. The decision to store the password is made based on the `credentialStore.savePasswords` and `credentialStore.excludeFilters` described here.

Automatic password storage and retrieval is performed when:

- `mysqlsh` is invoked with any connection options, when establishing the first session
- you use the built-in `\connect` command
- you use the `shell.connect()` method
- you use any AdminAPI methods that require a connection

4.4.1 Pluggable Password Configuration Options

To configure the pluggable password store, use the `shell.options` interface, see Section 13.4, “Configuring MySQL Shell Options” . The following options configure the pluggable password store.

shell.options.credentialStore.helper = "*`login-path`*"

A string which specifies the Secret Store Helper used to store and retrieve the passwords. By default, this option is set to a special value `default` which identifies the default helper on the current platform. Can be set to any of the values returned by `shell.listCredentialHelpers()` method. If this value is set to invalid value or an unknown Helper, an exception is raised. If an invalid value is detected during the startup of `mysqlsh`, an error is displayed and storage and retrieval of passwords is disabled. To disable automatic storage and retrieval of passwords, set this option to the special value `<disabled>`, for example by issuing:

```
shell.options.set("credentialStore.helper", "<disabled>")
```

When this option is disabled, usage of all of the credential store MySQL Shell methods discussed here results in an exception.

shell.options.credentialStore.savePasswords = "*`value`*"

A string which controls automatic storage of passwords. Valid values are:

- `always` - passwords are always stored, unless they are already available in the Secret Store or server URL matches `credentialStore.excludeFilters` value.
- `never` - passwords are not stored.
- `prompt` - in interactive mode, if the server URL does not match the value of `shell.credentialStore.excludeFilters`, you are prompted if the password should be stored. The possible answers are `yes` to save this password, `no` to not save this password, `never` to not save this password and to add the URL to `credentialStore.excludeFilters`. The modified value of `credentialStore.excludeFilters` is not persisted, meaning it is in effect only until MySQL Shell is restarted. If MySQL Shell is running in noninteractive mode (for example the `--no-wizard` option was used), the `credentialStore.savePasswords` option is always `never`.

The default value for this option is `prompt`.

shell.options.credentialStore.excludeFilters = ["*@myserver.com:*"];

A list of strings specifying which server URLs should be excluded from automatic storage of passwords. Each string can be either an explicit URL or a glob pattern. If a server URL which is about to be stored matches any of the strings in this options, it is not stored. The valid wildcard characters are: `*` which matches any number of any characters, and `?` which matches a single character.

The default value for this option is an empty list.

4.4.2 Working with Credentials

The following functions enable you to work with the Pluggable Password store. You can list the available Secret Store Helpers, as well as list, store, and retrieve credentials.

var list = shell.listCredentialHelpers();

Returns a list of strings, where each string is a name of a Secret Store Helper available on the current platform. The special values `default` and `<disabled>` are not in the list, but are valid values for the `credentialStore.helper` option.

shell.storeCredential(*url*[, *password*]);

Stores given credentials using the current Secret Store Helper (`credentialStore.helper`). Throws an error if the store operation fails, for example if the current helper is invalid. If the URL is already in the Secret Store, it is overwritten. This method ignores the current value of the `credentialStore.savePasswords` and `credentialStore.excludeFilters` options. If a password is not provided, MySQL Shell prompts for one.

shell.deleteCredential(url);

Deletes the credentials for the given URL using the current Secret Store Helper (`credentialStore.helper`). Throws an error if the delete operation fails, for example the current helper is invalid or there is no credential for the given URL.

shell.deleteAllCredentials();

Deletes all credentials managed by the current Secret Store Helper (`credentialStore.helper`). Throws an error if the delete operation fails, for example the current Helper is invalid.

var list = shell.listCredentials();

Returns a list of all URLs of credentials stored by the current Secret Store Helper (`credentialStore.helper`).

4.5 MySQL Shell Global Objects

MySQL Shell includes a number of built-in global objects that exist in both JavaScript and Python modes. The built-in MySQL Shell global objects are as follows:

- `session` is available when a global session is established, and represents the global session.
- `dba` provides access to InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet administration functions using the AdminAPI. See Chapter 6, *MySQL AdminAPI*.
- `cluster` represents an InnoDB Cluster. Only populated if the `--cluster` option was provided when MySQL Shell was started.
- `rs` represents an InnoDB ReplicaSet (added in version 8.0.20). Only populated if the `--replicaset` option was provided when MySQL Shell was started.
- `db` is available when the global session was established using an X Protocol connection with a default database specified, and represents that schema.

- `shell` provides access to various MySQL Shell functions, for example:
  - `shell.options` provides functions to set and unset MySQL Shell preferences. See Section 13.4, “Configuring MySQL Shell Options”.
  - `shell.reports` provides built-in or user-defined MySQL Shell reports as functions, with the name of the report as the function. See Section 10.1, “Reporting with MySQL Shell”.
- `util` provides various MySQL Shell tools, including the upgrade checker utility, the JSON import utility, and the parallel table import utility. See Chapter 11, *MySQL Shell Utilities*.

Important

The names of the MySQL Shell global objects are reserved as global variables and must not be used, for example, as names of variables. If you assign one of the global variables you override the above functionality, and to restore it you must restart MySQL Shell.

You can also create your own extension objects and register them as additional MySQL Shell global objects to make them available in a global context. For instructions to do this, see Section 10.2, “Adding Extension Objects to MySQL Shell”.

4.6 Using a Pager

You can configure MySQL Shell to use an external pager tool such as `less` or `more`. Once a pager is configured, it is used by MySQL Shell to display the text from the online help or the results of SQL operations. Use the following configuration possibilities:

- Configure the `shell.options[pager] = ""` MySQL Shell option, a string which specifies the external command that displays the paged output. This string can optionally contain command line arguments which are passed to the external pager command. Correctness of the new value is not checked. An empty string disables the pager, as does the MySQL Shell command `\nopager`.

  Default value: empty string.

- Configure the PAGER environment variable, which overrides the default value of `shell.options["pager"]` option. If `shell.options["pager"]` was persisted, it takes precedence over the `PAGER` environment variable.

  The `PAGER` environment variable is commonly used on Unix systems in the same context as expected by MySQL Shell, conflicts are not possible.

- Configure the `--pager` MySQL Shell option, which overrides the initial value of `shell.options["pager"]` option even if it was persisted and `PAGER` environment variable is configured.

- Use the `\pager | \P` *`command`* MySQL Shell command to set the value of `shell`.options["pager"] option. If called with no arguments, restores the initial value of `shell.options["pager"]` option (the one MySQL Shell had at startup. Strings can be marked with `"` characters or not. For example, to configure the pager:
  - pass in no *`command`* or an empty string to restore the initial pager
  - pass in *`more`* to configure MySQL Shell to use the `more` command as the pager
  - pass in *`more -10`* to configure MySQL Shell to use the `more` command as the pager with the option `-10`

The MySQL Shell output that is passed to the external pager tool is forwarded with no filtering. If MySQL Shell is using a prompt with color (see Section 13.3, “Customizing the Prompt”), the output contains ANSI escape sequences. Some pagers might not interpret these escape sequences by

default, such as `less`, for which interpretation can be enabled using the `-R` option. `more` does interpret ANSI escape sequences by default.

4.7 Cloud Service Configuration

MySQL Shell supports exporting of MySQL data to cloud service storage and import of that data from cloud storage to a MySQL instance. The following cloud services are supported:

- Section 4.7.1, “Oracle Cloud Infrastructure Object Storage”
- Section 4.7.2, “S3-compatible Storage”
- Section 4.7.3, “Azure Blob Storage”

For information on exporting MySQL data to cloud storage, see Section 11.5, “Instance Dump Utility, Schema Dump Utility, and Table Dump Utility”. For information on importing MySQL data from cloud storage, see Section 11.6, “Dump Loading Utility”.

For information on using MySQL Shell with MySQL HeatWave Service, see MySQL HeatWave Service Documentation.

4.7.1 Oracle Cloud Infrastructure Object Storage

MySQL Shell uses the parameters defined in the OCI CLI configuration file, `config`, to connect to the Object Storage service. For more information on this file, see SDK and CLI Configuration

MySQL Shell requires the following parameters in the configuration file:

- `user`: OCID of the user.
- `fingerprint`: generated fingerprint of the user's public key.
- `tenancy`: OCID of the user's tenancy.
- `region`: An Oracle Cloud Infrastructure region.
- `key_file`: Full path and filename of the user's public key.

The following is an example of a configuration file:

```
[default]
user=ocid1.user.oc1..alphanumericstring
fingerprint=08:23:60:....:ff:22:dd:55:20
tenancy=ocid1.tenancy.oc1..alphanumericstring
region=us-ashburn-1
key_file=/home/Username/.oci/oci_api_key.pem
```

If you have installed and configured the OCI CLI, MySQL Shell reads the connection parameters from the default location, `.oci/`, automatically. To use an alternate configuration, do so from the command line, using the relevant override options.

4.7.2 S3-compatible Storage

MySQL Shell S3 support has been tested against the following S3-compatible storage services:

- Amazon Web Services S3

  See Amazon Simple Storage Service Documentation for more information.

- Oracle Cloud Infrastructure Object Storage

See Amazon S3 Compatibility API for more information.

MySQL Shell supports configuring AWS credentials in environment variables and in configuration files.

- Configuration Parameter Precedence
- Environment Variables
- Configuration Files

Configuration Parameter Precedence

Configuration parameters are used in order of precedence:

1. Option. For example, `s3Profile` takes precedence over the environment variable `AWS_PROFILE`, and the default profile in the configuration file.
2. Environment variable. For example, the environment variable `AWS_SHARED_CREDENTIALS_FILE` takes precedence over the default location of the `credentials` file, `~/.aws/credentials`.
3. DEFAULT environment variable, if it exists. For example, the environment variable `AWS_REGION` and the environment variable `AWS_DEFAULT_REGION`.
4. Parameter defined in a configuration file.
5. AWS default values.

For example, for the AWS region, in order of precedence:

1. `s3Region` option.
2. `AWS_REGION` environment variable.
3. `AWS_DEFAULT_REGION` environment variable.
4. Region defined in the configuration file.
5. The default value of `us-east-1`.

AWS credentials are read in the following order of precedence:

1. Environment variables, if the `s3Profile` option is not defined.
2. Credentials file, for the specified profile.
3. `credential_process`, as defined in the config file. This configuration type contains an external command which retrieves or generates AWS authentication credentials and writes them to `stdout`.
4. Config file, for the specified profile.

Environment Variables

For information on AWS environment variables, see Environment variables to configure the AWS CLI

You can define the following AWS S3-specific environment variables:

- `AWS_PROFILE`: Specifies the name of the profile to use. This can be the name of a profile in a credentials or config file, or the value `default` to use the default profile. This environment variable overrides the `[default]` profile named in the configuration file. You can override this environment variable with the `--s3Profile` option.
- `AWS_SHARED_CREDENTIALS_FILE`: The location of the file used to store access keys. Such as `~/.aws/credentials`.

- `AWS_CONFIG_FILE`: The location of the file used to store configuration profiles. Such as `~/.aws/config`.
- `AWS_REGION`: Specifies the AWS Region to send the request to. This value overrides the `AWS_DEFAULT_REGION` environment variable and the `profile` defined in the configuration file.
- `AWS_DEFAULT_REGION`: Specifies the AWS Region to send the request to. This value is overridden by the `--s3Region` option and the `AWS_REGION` environment variable, if specified.
- `AWS_ACCESS_KEY_ID`: Specifies an AWS access key associated with an IAM user or role.
- `AWS_SECRET_ACCESS_KEY`: Specifies the secret key associated with the access key. This variable overrides the `aws_secret_access_key` defined in the profile.
- `AWS_SESSION_TOKEN`: Specifies the session token value required if you are using temporary security credentials. This variable overrides the `aws_session_token` defined in the profile.

Configuration Files

MySQL Shell requires the following parameters in one or more configuration files:

- `aws_access_key_id`: specifies the access key associated with the user. If not present, an exception is thrown.
- `aws_secret_access_key`: specifies the secret key associated with the access key. If not present, an exception is thrown.
- `aws_session_token`: specifies the session token required if you are using temporary security credentials. If not present, it is not used to authenticate the user.
- `region`: specifies the cloud service region. If not present, the default value of `us-east-1` is used.
- `credential_process`: (as of 8.0.33) This configuration type contains an external command which retrieves or generates AWS authentication credentials and writes them to `stdout`.

Note

`credential_process` can only be defined in the `config` file.

For more information, see the following:

- AWS Credentials Process
- AWS Configuration and Credential Options
- AWS Sourcing credentials with an external process

If you have installed and configured the AWS CLI, MySQL Shell reads the connection parameters from the default location, `.aws/`, automatically. To use an alternate configuration, do so from the command line, using the relevant override options.

If you are using Oracle Cloud Infrastructure's S3 compatibility API, see Creating a Customer Secret Key for information on creating the credentials required.

Note

By default, the AWS CLI creates two configuration files, `config`, which stores parameters such as region and output format, and `credentials`, which stores access keys and session tokens. It is also possible to place all configuration parameters in a single file, `config`. For more information, see AWS Configuration and authentication settings . However, if you have

defined access key and secret access keys in both files, those defined in the `credentials` file take precedence.

The following example shows a default pair of AWS CLI configuration files:

```
/home/.aws/credentials

[default]
aws_access_key_id = AKIA6AV......
aws_secret_access_key = XHRY579I.....

/home/.aws/config

[default]
region = us-west-1
output = json
```

4.7.3 Azure Blob Storage

The Azure configuration values are evaluated in the following order of precedence:

- Option. See the Azure-specific sections of Chapter 11, *MySQL Shell Utilities* for the applicable options.
- Environment variable. See Environment Variables.
- Configuration file. See Configuration Files.

For more detailed information on Microsoft Azure CLI configuration, see the Azure CLI documentation.

Note

If you intend to use SAS Tokens, they must provide the following:

- Access to the target container.
- Read, Write, and List permissions for dump and export operations.
- Read and List permissions for load and import operations.

Environment Variables

Azure connection settings can be read from the following environment variables:

- `AZURE_STORAGE_ACCOUNT`: The default storage account name.
- `AZURE_STORAGE_KEY`: The default storage key.
- `AZURE_STORAGE_CONNECTION_STRING`: The default connection string. If this is defined, AZURE_STORAGE_ACCOUNT and AZURE_STORAGE_KEY are ignored.
- `AZURE_STORAGE_SAS_TOKEN`: The default SAS token.

If an SAS token is defined, it is used for the authentication, any defined account key is ignored.

Configuration Files

If you use the Azure config file, you must add one or more of the following parameters to the `[storage]` section of the file:

- `connection_string`: The default connection string. If this is defined, `account` and `key` are ignored.
- `account`: The default storage account name.
- `key`: The default access key.
- `sas_token`: The default SAS token.

If an SAS token is defined, it is used for the authentication, any defined account key is ignored.

For more information, see Microsoft Azure CLI configuration values and environment variables.

Chapter 5 MySQL Shell Code Execution

Table of Contents

5.1 Active Language 45
5.2 Interactive Code Execution 46
5.3 Code Autocompletion 48
5.4 Editing Code 52
5.5 Code History 52
5.6 Batch Code Execution 53
5.7 Output Formats 55
    5.7.1 Table Format 56
    5.7.2 Tab Separated Format 56
    5.7.3 Vertical Format 56
    5.7.4 JSON Format Output 57
    5.7.5 JSON Wrapping 59
    5.7.6 Result Metadata 60
5.8 API Command Line Integration 60
    5.8.1 Command Line Integration Overview 60
    5.8.2 Command Line Integration Details 63
5.9 JSON Integration 72
5.10 Limitations 72

This section explains how code execution works in MySQL Shell.

5.1 Active Language

MySQL Shell can execute SQL, JavaScript or Python code, but only one language can be active at a time. The active mode determines how the executed statements are processed:

- If using SQL mode, statements are processed as SQL which means they are sent to the MySQL server for execution.
- If using JavaScript mode, statements are processed as JavaScript code.
- If using Python mode, statements are processed as Python code.

Note

From version 8.0.18, MySQL Shell uses Python 3. For platforms that include a system supported installation of Python 3, MySQL Shell uses the most recent version available, with a minimum supported version of Python 3.4.3. For platforms where Python 3 is not included, MySQL Shell bundles Python 3.7.4. MySQL Shell maintains code compatibility with Python 2.6 and Python 2.7, so if you require one of these older versions, you can build MySQL Shell from source using the appropriate Python version.

When running MySQL Shell in interactive mode, activate a specific language by entering the commands: `\sql`, `\js`, `\py`.

When running MySQL Shell in batch mode, activate a specific language by passing any of these command-line options: `--js`, `--py` or `--sql`. The default mode if none is specified is JavaScript.

Use MySQL Shell to execute the content of the file `code.sql` as SQL.

```
$> mysqlsh --sql < code.sql
```

Use MySQL Shell to execute the content of the file `code.js` as JavaScript code.

```
$> mysqlsh < code.js
```

Use MySQL Shell to execute the content of the file `code.py` as Python code.

```
$> mysqlsh --py < code.py
```

From MySQL Shell 8.0.16, you can execute single SQL statements while another language is active, by entering the `\sql` command immediately followed by the SQL statement. For example:

```
mysql-py> \sql select * from sakila.actor limit 3;
```

The SQL statement does not need any additional quoting, and the statement delimiter is optional. The command only accepts a single SQL query on a single line. With this format, MySQL Shell does not switch mode as it would if you entered the `\sql` command. After the SQL statement has been executed, MySQL Shell remains in JavaScript or Python mode.

From MySQL Shell 8.0.18, you can execute operating system commands while any language is active, by entering the `\system` or `\!` command immediately followed by the command to execute. For example:

```
mysql-py> \system echo Hello from MySQL Shell!
```

MySQL Shell displays the output from the operating system command, or returns an error if it was unable to execute the command.

5.2 Interactive Code Execution

The default mode of MySQL Shell provides interactive execution of database operations that you type at the command prompt. These operations can be written in JavaScript, Python or SQL depending on the current Section 5.1, “Active Language”. When executed, the results of the operation are displayed on-screen.

As with any other language interpreter, MySQL Shell is very strict regarding syntax. For example, the following JavaScript snippet opens a session to a MySQL server, then reads and prints the documents in a collection:

```
var mySession = mysqlx.getSession('user:pwd@localhost');
var result = mySession.getSchema('world_x').getCollection('countryinfo').find().execute();
var record = result.fetchOne();
while(record){
  print(record);
  record = result.fetchOne();
}
```

As seen above, the call to `find()` is followed by the `execute()` function. CRUD database commands are only actually executed on the MySQL Server when `execute()` is called. However, when working with MySQL Shell interactively, `execute()` is implicitly called whenever you press `Return` on a statement. Then the results of the operation are fetched and displayed on-screen. The rules for when you need to call `execute()` or not are as follows:

- When using MySQL Shell in this way, calling `execute()` becomes optional on:
  - `Collection.add()`
  - `Collection.find()`
  - `Collection.remove()`
  - `Collection.modify()`
  - `Table.insert()`

    - `Table.select()`
    - `Table.delete()`
    - `Table.update()`
- Automatic execution is disabled if the object is assigned to a variable. In such a case calling `execute()` is mandatory to perform the operation.
- When a line is processed and the function returns any of the available `Result` objects, the information contained in the Result object is automatically displayed on screen. The functions that return a Result object include:
    - The SQL execution and CRUD operations (listed above)
    - Transaction handling and drop functions of the session objects in both `mysql` and `mysqlx` modules: -
        - `startTransaction()`
        - `commit()`
        - `rollback()`
        - `dropSchema()`
        - `dropCollection()`
        - `ClassicSession.runSql()`

Based on the above rules, the statements needed in the MySQL Shell in interactive mode to establish a session, query, and print the documents in a collection are as follows:

```
mysql-js> var mySession = mysqlx.getSession('user:pwd@localhost');
mysql-js> mySession.getSchema('world_x').getCollection('countryinfo').find();
```

No call to `execute()` is needed and the Result object is automatically printed.

Multiple-line Support

It is possible to specify statements over multiple lines. When in Python or JavaScript mode, multiple-line mode is automatically enabled when a block of statements starts like in function definitions, if/then statements, for loops, and so on. In SQL mode multiple line mode starts when the command `\` is issued.

Once multiple-line mode is started, the subsequently entered statements are cached.

For example:

```
mysql-sql> \
... create procedure get_actors()
... begin
...   select first_name from sakila.actor;
... end
...
```

Note

You cannot use multiple-line mode when you use the `\sql` command with a query to execute single SQL statements while another language is active. The command only accepts a single SQL query on a single line.

5.3 Code Autocompletion

MySQL Shell supports autocompletion of text preceding the cursor by pressing the **Tab** key. The Section 3.1, “MySQL Shell Commands” can be autocompleted in any of the language modes. For example typing `\con` and pressing the `Tab` key autocompletes to `\connect`. Autocompletion is available for SQL, JavaScript, and Python language keywords depending on the current Section 5.1, “Active Language”.

Autocompletion supports the following text objects:

- In SQL mode, autocompletion is aware of schema names, table names, column names of the current active schema.
- In JavaScript and Python modes autocompletion is aware of object members, for example:
  - global object names such as `session`, `db`, `dba`, `shell`, `mysql`, `mysqlx`, and so on.
  - members of global objects such as `session.connect()`, `dba.configureLocalInstance()`, and so on.
  - global user defined variables
  - chained object property references such as `shell.options.verbose`.
  - chained X DevAPI method calls such as `col.find().where().execute().fetchOne()`.

By default autocompletion is enabled, to change this behavior see Configuring Autocompletion.

Once you activate autocompletion, if the text preceding the cursor has exactly one possible match, the text is automatically completed. If autocompletion finds multiple possible matches, it beeps or flashes the terminal. If the `Tab` key is pressed again, a list of the possible completions is displayed. If no match is found then no autocompletion happens.

- Autocompleting SQL
- Autocompleting JavaScript and Python
- Configuring Autocompletion

Autocompleting SQL

In SQL mode, context-aware autocompletion completes any word with relevant completions. As of MySQL Shell 8.0.31, the following can be autocompleted:

- Schemas
- Tables
- Views
- Columns
- Stored procedures
- Functions
- Triggers
- Events
- Engines
- User-defined functions

- Runtime functions
- Log file groups
- User variables
- System variables
- Tablespaces
- Users
- Character sets
- Collations
- Plugins

If you connect to a MySQL instance but do not select a schema, autocompletion is available for global objects, charsets, engines, schemas and so on. For example on a default MySQL installation, `USE` suggests the names of all schemas detected unless one or more relevant characters from the schema name are provided:

```
SQL > use
information_schema  mysql  performance_schema  sys
```

If a schema is selected, additional schema information is loaded and available for autocompletion (tables, events, and so on). If you switch from one schema to another, the objects loaded from the previous schema are still available for autocompletion. However, any new object added during the session will not be available for autocompletion until the `\rehash` command is run.

To fetch a list of suggestions or complete a partial word from the selected schema, enter the initial fragment and press the Tab button twice. For example:

1. At the SQL prompt, enter the following fragment: `SE`.
2. Press the Tab key twice.

   The following suggestions are displayed below your input:

   ```
   SET SELECT
   ```

3. At the SQL prompt, enter the following fragment: `SEL`.
4. Press the Tab key twice.

   The fragment autocompletes to `SELECT`.

If there are many possible results, you are prompted to display the results or not. For example:

```
Display all 118 possibilities? (y or n)
```

SQL Autocompletion API

The autocompletion API is exposed to developers through the following functions:

- JavaScript: `shell.autoCompleteSql(statement, options)`
- Python: `shell.auto_complete_sql(statement, options)`

statement: "*`string`*"	A partial SQL statement for autocompletion.

These return feasible candidates for the autocompletion.

Options:

serverVersion: "*`string`*"	Required. Server grammar version. This takes the format major.minor.patch. `serverVersion:"8.0.31"`, for example.
sqlMode: "*`string`*"	Required. The SQL Mode to use. A comma-separated string, `sqlMode:"STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"`, for example. For more information, see Server SQL Modes.
statementOffset: *`number`*	Optional. The zero-based offset position of the caret in the statement. Default value is the length of the statement.
uppercaseKeywords: [true|false]	Default `true`. Whether the returned keywords are in upper case.
filtered: [true|false]	Default `true`. Whether explicit candidate names returned in the result should be filtered using the prefix which is being auto-completed.

This function returns a dictionary describing candidates for statement autocompletion using the following syntax:

```
{
 "context": {
   "prefix": string,
   "qualifier": list of strings,
   "references": list of dictionaries,
   "labels": list of strings,
 },
 "keywords": list of strings,
 "functions": list of strings,
 "candidates": list of strings,
}
```

- `context`: the context of the autocomplete operation.
- `prefix`: the fragment being autocompleted.
- `qualifier`: present if a qualified name is available.

  For example:

  - `SELECT s`: the prefix is `'s'`, no qualifier is present.
  - `SELECT schema1.t`: the prefix is `'t'`, the qualifier is `['schema1']`.
  - `SELECT schema1.table1.c`: the prefix is `'c'`, the qualifier is `['schema1','table1']`.
  - `SELECT schema1.table1.column1 FR`: the prefix is `'FR'`, no qualifier is present.
- `references`: references detected in the statement.
  - `schema`: name of the schema.
  - `table`: name of the table referenced in the statement.
  - `alias`: alias of the table.
- `labels`: labels in labelled blocks.

- `keywords`: candidate keyword suggestions.
- `functions`: candidate MySQL library (runtime) functions whose names are also keywords.
- `candidates`: lists one or more of the supported candidates. Schemas, tables, views, and so on.

For example:

```
JS > shell.autoCompleteSql("select * from  ",{serverVersion: "8.0.30", sqlMode: "STRICT_TRANS
  {
      "candidates": [
          "schemas",
          "tables",
          "views"
      ],
      "context": {
          "prefix": ""
      },
      "functions": [
          "JSON_TABLE()"
      ],
      "keywords": [
          "DUAL",
          "LATERAL"
      ]
  }
```

Autocompleting JavaScript and Python

In both JavaScript and Python modes, the string to be completed is determined from right to left, beginning at the current cursor position when **Tab** is pressed. Contents inside method calls are ignored, but must be syntactically correct. This means that strings, comments and nested method calls must all be properly closed and balanced. This allows chained methods to be handled properly. For example, when you are issuing:

```
print(db.user.select().where("user in ('foo', 'bar')").e
```

Pressing the **Tab** key would cause autocompletion to try to complete the text **`db.user.select().where().e`** but this invalid code yields undefined behavior. Any whitespace, including newlines, between tokens separated by a `.` is ignored.

Configuring Autocompletion

By default the autocompletion engine is enabled. This section explains how to disable autocompletion and how to use the `\rehash` MySQL Shell command. Autocompletion uses a cache of database name objects that MySQL Shell is aware of. When autocompletion is enabled, this name cache is automatically updated. For example whenever you load a schema, the autocompletion engine updates the name cache based on the text objects found in the schema, so that you can autocomplete table names and so on.

To disable this behavior you can:

- Start MySQL Shell with the `--no-name-cache` command option.
- Modify the `autocomplete.nameCache` and `devapi.dbObjectHandles` keys of the `shell.options` to disable the autocompletion while MySQL Shell is running.

When the autocompletion name cache is disabled, you can manually update the text objects autocompletion is aware of by issuing `\rehash`. This forces a reload of the name cache based on the current active schema.

To disable autocompletion while MySQL Shell is running use the following `shell.options` keys:

- `autocomplete.nameCache: boolean` toggles autocompletion name caching for use by SQL.
- `devapi.dbObjectHandles: boolean` toggles autocompletion name caching for use by the X DevAPI `db` object, for example `db.mytable`, `db.mycollection`.

Both keys are set to `true` by default, and set to `false` if the `--no-name-cache` command option is used. To change the autocompletion name caching for SQL while MySQL Shell is running, issue:

```
shell.options['autocomplete.nameCache']=true
```

Use the `\rehash` command to update the name cache manually.

To change the autocompletion name caching for JavaScript and Python while MySQL Shell is running, issue:

```
shell.options['devapi.dbObjectHandles']=true
```

Again you can use the `\rehash` command to update the name cache manually.

5.4 Editing Code

MySQL Shell's `\edit` command (available from MySQL Shell 8.0.18) opens a command in the default system editor for editing, then presents the edited command in MySQL Shell for execution. The command can also be invoked using the short form `\e` or key combination **Ctrl-X Ctrl-E**. If you specify an argument to the command, this text is placed in the editor. If you do not specify an argument, the last command in the MySQL Shell history is placed in the editor.

The `EDITOR` and `VISUAL` environment variables are used to identify the default system editor. If the default system editor cannot be identified from these environment variables, MySQL Shell uses `notepad.exe` on Windows and `vi` on any other platform. Command editing takes place in a temporary file, which MySQL Shell deletes afterwards.

When you have finished editing, you must save the file and close the editor, MySQL Shell then presents your edited text ready for you to execute by pressing **Enter**, or if you do not want to proceed, to cancel by pressing **Ctrl-C**.

For example, here the user runs the MySQL Shell built-in report `threads` with a custom set of columns, then opens the command in the system editor to add display names for some of the columns:

```
\show threads --foreground -o tid,cid,user,host,command,state,lastwait,lastwaitl
\e
\show threads --foreground -o tid=thread_id,cid=conn_id,user,host,command,state,lastwait=last_wait_event,la
```

5.5 Code History

Code which you issue in MySQL Shell is stored in the history, which can then be accessed using the up and down arrow keys. You can also search the history using the incremental history search feature. To search the history, use **Ctrl+R** to search backwards, or **Ctrl+S** to search forwards through the history. Once the search is active, typing characters searches for any strings that match them in the history and displays the first match. Use **Ctrl+S** or **Ctrl+R** to search for further matches to the current search term. Typing more characters further refines the search. During a search, you can press the arrow keys to continue stepping through the history from the current search result. Press Enter to accept the displayed match. Use **Ctrl+C** to cancel the search.

The `history.maxSize` MySQL Shell configuration option sets the maximum number of entries to store in the history. The default is 1000. If the number of history entries exceeds the configured maximum, the oldest entries are removed and discarded. If the maximum is set to 0, no history entries are stored.

By default the history is not saved between sessions, so when you exit MySQL Shell the history of what you issued during the current session is lost. You can save your history between sessions by enabling the MySQL Shell `history.autoSave` option. For example, to make this change permanent issue:

```
mysqlsh-js> \option --persist history.autoSave=1
```

When the `history.autoSave` option is enabled the history is stored in the MySQL Shell configuration path, which is the `~/.mysqlsh` directory on Linux and macOS, or the `%AppData%\MySQL\mysqlsh` folder on Windows. This path can be overridden on all platforms by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`. The saved history is created automatically by MySQL Shell and is readable only by the owner user. If the history file cannot be read or written to, MySQL Shell logs an error message and skips the read or write operation. Prior to version 8.0.16, history entries were saved to a single `history` file, which contained the code issued in all of the MySQL Shell languages. In MySQL Shell version 8.0.16 and later, the history is split per active language and the files are named `history.sql`, `history.js` and `history.py`.

Issuing the MySQL Shell `\history` command shows history entries in the order that they were issued, together with their history entry number, which can be used with the `\history delete` *`entry_number`* command. You can manually delete individual history entries, a specified numeric range of history entries, or the tail of the history. You can also use `\history clear` to delete the entire history manually. When you exit MySQL Shell, if the `history.autoSave` configuration option has been set to `true`, the history entries that remain in the history file are saved, and their numbering is reset to start at 1. If the `shell.options["history.autoSave"]` configuration option is set to `false`, which is the default, the history file is cleared.

Only code which you type interactively at the MySQL Shell prompt is added to the history. Code that is executed indirectly or internally, for example when the `\source` command is executed, is not added to the history. When you issue multi-line code, the new line characters are stripped in the history entry. If the same code is issued multiple times it is only stored in the history once, reducing duplication.

You can customize the entries that are added to the history using the `--histignore` command option. Additionally, when using MySQL Shell in SQL mode, you can configure strings which should not be added to the history. This history ignore list is also applied when you use the `\sql` command with a query to execute single SQL statements while another language is active.

By default strings that match the glob patterns `IDENTIFIED` or `PASSWORD` are not added to the history. To configure further strings to match use either the `--histignore` command option, or `shell.options["history.sql.ignorePattern"]`. Multiple strings can be specified, separated by a colon (:). The history matching uses case-insensitive glob pattern like matching. Supported wildcards are * (match any 0 or more characters) and ? (match exactly 1 character). The default strings are specified as `"*IDENTIFIED*:*PASSWORD*"`.

The most recent executed statement is always available by pressing the Up arrow, even if the history ignore list applies to it. This is so that you can make corrections without retyping all the input. If filtering applies to the last executed statement, it is removed from the history as soon as another statement is entered, or if you exit MySQL Shell immediately after executing the statement.

5.6 Batch Code Execution

As well as interactive code execution, MySQL Shell provides batch code execution from:

- A file loaded for processing.
- A file containing code that is redirected to the standard input for execution.
- Code from a different source that is redirected to the standard input for execution.

Tip

As an alternative to batch execution of a file, you can also control MySQL Shell from a terminal, see Section 5.8, “API Command Line Integration”.

In batch mode, all the command logic described at Section 5.2, “Interactive Code Execution” is not available, only valid code for the active language can be executed. When processing SQL code,

it is executed statement by statement using the following logic: read/process/print result. When processing non-SQL code, it is loaded entirely from the input source and executed as a unit. Use the `--interactive` (or `-i`) command-line option to configure MySQL Shell to process the input source as if it were being issued in interactive mode; this enables all the features provided by the Interactive mode to be used in batch processing.

Note

In this case, whatever the source is, it is read line by line and processed using the interactive pipeline.

The input is processed based on the current programming language selected in MySQL Shell, which defaults to JavaScript. You can change the default programming language using the `defaultMode` MySQL Shell configuration option. Files with the extensions `.js`, `.py`, and `.sql` are always processed in the appropriate language mode, regardless of the default programming language.

This example shows how to load JavaScript code from a file for batch processing:

```
$> mysqlsh --file code.js
```

Here, a JavaScript file is redirected to standard input for execution:

```
$> mysqlsh < code.js
```

The following example shows how to redirect SQL code to standard input for execution on Linux platforms:

```
$> echo "show databases;" | mysqlsh --sql --uri user@192.0.2.20:33060
```

Note

To run this command on Windows platforms, you must remove the quotation marks surrounding the string in the `echo` command.

From MySQL Shell 8.0.22, the `--pym` command line option is available to execute the specified Python module as a script in Python mode. The option works in the same way as Python's `-m` command line option.

Executable Scripts

On Linux you can create executable scripts that run with MySQL Shell by including a `#!` line as the first line of the script. This line should provide the full path to MySQL Shell and include the `--file` option. For example:

```
#!/usr/local/mysql-shell/bin/mysqlsh --file
print("Hello World\n");
```

The script file must be marked as executable in the filesystem. Running the script invokes MySQL Shell and it executes the contents of the script.

SQL Execution in Scripts

SQL query execution for X Protocol sessions normally uses the `sql()` function, which takes an SQL statement as a string, and returns a SqlExecute object that you use to bind and execute the query and return the results. This method is described at Using SQL with Session. However, SQL query execution for classic MySQL protocol sessions uses the `runSql()` function, which takes an SQL statement and its parameters, binds the specified parameters into the specified query and executes the query in a single step, returning the results.

If you need to create a MySQL Shell script that is independent of the protocol used for connecting to the MySQL server, MySQL Shell provides a `session.runSql()` function for X Protocol, which works in the same way as the `runSql()` function in classic MySQL protocol sessions. You can use

this function in MySQL Shell only in place of `sql()`, so that your script works with either an X Protocol session or a classic MySQL protocol session. `Session.runSql()` returns a SqlResult object, which matches the specification of the ClassicResult object returned by the classic MySQL protocol function, so the results can be handled in the same way.

Note

`Session.runSql()` is exclusive to the MySQL Shell X DevAPI implementation in JavaScript and Python, and is not part of the standard X DevAPI.

To browse the query results, you can use the `fetchOneObject()` function, which works for both the classic MySQL protocol and X Protocol. This function returns the next result as a scripting object. Column names are used as keys in the dictionary (and as object attributes if they are valid identifiers), and row values are used as attribute values in the dictionary. Updates made to the object are not persisted on the database.

For example, this code in a MySQL Shell script works with either an X Protocol session or a classic MySQL protocol session to retrieve and output the name of a city from the given country:

```
var resultSet = mySession.runSql("SELECT * FROM city WHERE countrycode = ' AUT'");
var row = resultSet.fetchOneObject();
print(row['Name']);
```

5.7 Output Formats

MySQL Shell can print results in table, tabbed, or vertical format, or as pretty or raw JSON output. From MySQL Shell 8.0.14, the MySQL Shell configuration option `resultFormat` can be used to specify any of these output formats as a persistent default for all sessions, or just for the current session. Changing this option takes effect immediately. For instructions to set MySQL Shell configuration options, see Section 13.4, “Configuring MySQL Shell Options”. Alternatively, the command line option `--result-format` or its aliases (`--table`, `--tabbed`, `--vertical`) can be used at startup to specify the output format for a session. For a list of the command line options, see Section A.1, “mysqlsh — The MySQL Shell”.

If the `resultFormat` configuration option has not been specified, when MySQL Shell is in interactive mode, the default format for printing a result set is a formatted table, and when MySQL Shell is in batch mode, the default format for printing a result set is tab separated output. When you set a default using the `resultFormat` configuration option, this default applies in both interactive mode and batch mode.

The MySQL Shell function `shell.dumpRows()` can format a result set returned by a query in any of the output formats supported by MySQL Shell, and dump it to the console. (Note that the result set is consumed by the function.)

To help integrate MySQL Shell with external tools, you can use the `--json` option to control JSON wrapping for all MySQL Shell output when you start MySQL Shell from the command line. When JSON wrapping is turned on, MySQL Shell generates either pretty-printed JSON (the default) or raw JSON, and the value of the `resultFormat` MySQL Shell configuration option is ignored. When JSON wrapping is turned off, or was not requested for the session, result sets are output as normal in the format specified by the `resultFormat` configuration option.

The `outputFormat` configuration option is now deprecated. This option combined the JSON wrapping and result printing functions. If this option is still specified in your MySQL Shell configuration file or scripts, the behavior is as follows:

- With the `json` or `json/raw` value, `outputFormat` activates JSON wrapping with pretty or raw JSON respectively.
- With the `table`, `tabbed`, or `vertical` value, `outputFormat` turns off JSON wrapping and sets the `resultFormat` configuration option for the session to the appropriate value.

5.7.1 Table Format

The table format is used by default for printing result sets when MySQL Shell is in interactive mode. The results of the query are presented as a formatted table for a better view and to aid analysis.

To get this output format when running in batch mode, start MySQL Shell with the `--result-format=table` command line option (or its alias `--table`), or set the MySQL Shell configuration option `resultFormat` to `table`.

Example 5.1 Output in Table Format

```
MySQL  localhost:33060+ ssl  world_x  JS > shell.options.set('resultFormat','table')
MySQL  localhost:33060+ ssl  world_x  JS > session.runSql("select * from city where countrycode='AUT'")
+------+------------+-------------+---------------+------------------------+
| ID   | Name       | CountryCode | District      | Info                   |
+------+------------+-------------+---------------+------------------------+
| 1523 | Wien       | AUT         | Wien          | {"Population": 1608144} |
| 1524 | Graz       | AUT         | Steiermark    | {"Population": 240967}  |
| 1525 | Linz       | AUT         | North Austria | {"Population": 188022}  |
| 1526 | Salzburg   | AUT         | Salzburg      | {"Population": 144247}  |
| 1527 | Innsbruck  | AUT         | Tiroli        | {"Population": 111752}  |
| 1528 | Klagenfurt | AUT         | Kärnten       | {"Population": 91141}   |
+------+------------+-------------+---------------+------------------------+
6 rows in set (0.0030 sec)
```

5.7.2 Tab Separated Format

The tab separated format is used by default for printing result sets when running MySQL Shell in batch mode, to have better output for automated analysis.

To get this output format when running in interactive mode, start MySQL Shell with the `--result-format=tabbed` command line option (or its alias `--tabbed`), or set the MySQL Shell configuration option `resultFormat` to `tabbed`.

Example 5.2 Output in Tab Separated Format

```
MySQL  localhost:33060+ ssl  world_x  JS > shell.options.set('resultFormat','tabbed')
MySQL  localhost:33060+ ssl  world_x  JS > session.runSql("select * from city where countrycode='AUT'")
ID      Name    CountryCode     District        Info
1523    Wien    AUT     Wien    {"Population": 1608144}
1524    Graz    AUT     Steiermark      {"Population": 240967}
1525    Linz    AUT     North Austria   {"Population": 188022}
1526    Salzburg        AUT     Salzburg        {"Population": 144247}
1527    Innsbruck       AUT     Tiroli  {"Population": 111752}
1528    Klagenfurt      AUT     Kärnten {"Population": 91141}
6 rows in set (0.0041 sec)
```

5.7.3 Vertical Format

The vertical format option prints result sets vertically instead of in a horizontal table, in the same way as when the `\G` query terminator is used for an SQL query. Vertical format is more readable where longer text lines are part of the output.

To get this output format, start MySQL Shell with the `--result-format=vertical` command line option (or its alias `--vertical`), or set the MySQL Shell configuration option `resultFormat` to `vertical`.

Example 5.3 Output in Vertical Format

```
MySQL  localhost:33060+ ssl  world_x  JS > shell.options.set('resultFormat','vertical')
MySQL  localhost:33060+ ssl  world_x  JS > session.runSql("select * from city where countrycode='AUT'")
*************************** 1. row ***************************
         ID: 1523
       Name: Wien
```

```
CountryCode: AUT
   District: Wien
       Info: {"Population": 1608144}
*************************** 2. row ***************************
         ID: 1524
       Name: Graz
CountryCode: AUT
   District: Steiermark
       Info: {"Population": 240967}
*************************** 3. row ***************************
         ID: 1525
       Name: Linz
CountryCode: AUT
   District: North Austria
       Info: {"Population": 188022}
*************************** 4. row ***************************
         ID: 1526
       Name: Salzburg
CountryCode: AUT
   District: Salzburg
       Info: {"Population": 144247}
*************************** 5. row ***************************
         ID: 1527
       Name: Innsbruck
CountryCode: AUT
   District: Tiroli
       Info: {"Population": 111752}
*************************** 6. row ***************************
         ID: 1528
       Name: Klagenfurt
CountryCode: AUT
   District: Kärnten
       Info: {"Population": 91141}
6 rows in set (0.0027 sec)
```

5.7.4 JSON Format Output

MySQL Shell provides a number of JSON format options to print result sets:

`json` or `json/pretty`	These options both produce pretty-printed JSON.
`ndjson` or `json/raw`	These options both produce raw JSON delimited by newlines.
`json/array`	This option produces raw JSON wrapped in a JSON array.

You can select these output formats by starting MySQL Shell with the `--result-format=`*`value`* command line option, or setting the MySQL Shell configuration option `resultFormat`.

In batch mode, to help integrate MySQL Shell with external tools, you can use the `--json` option to control JSON wrapping for all output when you start MySQL Shell from the command line. When JSON wrapping is turned on, MySQL Shell generates either pretty-printed JSON (the default) or raw JSON, and the value of the `resultFormat` MySQL Shell configuration option is ignored. For instructions, see Section 5.7.5, “JSON Wrapping”.

Example 5.4 Output in Pretty-Printed JSON Format (`json` or `json/pretty`)

```
MySQL  localhost:33060+ ssl  world_x  JS > shell.options.set('resultFormat','json')
MySQL  localhost:33060+ ssl  world_x  JS > session.runSql("select * from city where countrycode='AUT'")
{
    "ID": 1523,
    "Name": "Wien",
    "CountryCode": "AUT",
    "District": "Wien",
    "Info": {
        "Population": 1608144
    }
}
{
```

```
    "ID": 1524,
    "Name": "Graz",
    "CountryCode": "AUT",
    "District": "Steiermark",
    "Info": {
        "Population": 240967
    }
}
{
    "ID": 1525,
    "Name": "Linz",
    "CountryCode": "AUT",
    "District": "North Austria",
    "Info": {
        "Population": 188022
    }
}
{
    "ID": 1526,
    "Name": "Salzburg",
    "CountryCode": "AUT",
    "District": "Salzburg",
    "Info": {
        "Population": 144247
    }
}
{
    "ID": 1527,
    "Name": "Innsbruck",
    "CountryCode": "AUT",
    "District": "Tiroli",
    "Info": {
        "Population": 111752
    }
}
{
    "ID": 1528,
    "Name": "Klagenfurt",
    "CountryCode": "AUT",
    "District": "Kärnten",
    "Info": {
        "Population": 91141
    }
}
6 rows in set (0.0031 sec)
```

Example 5.5 Output in Raw JSON Format with Newline Delimiters (`ndjson` or `json/raw`)

```
MySQL  localhost:33060+ ssl  world_x  JS > shell.options.set('resultFormat','ndjson')
MySQL  localhost:33060+ ssl  world_x  JS > session.runSql("select * from city where countrycode='AUT'")
{"ID":1523,"Name":"Wien","CountryCode":"AUT","District":"Wien","Info":{"Population":1608144}}
{"ID":1524,"Name":"Graz","CountryCode":"AUT","District":"Steiermark","Info":{"Population":240967}}
{"ID":1525,"Name":"Linz","CountryCode":"AUT","District":"North Austria","Info":{"Population":188022}}
{"ID":1526,"Name":"Salzburg","CountryCode":"AUT","District":"Salzburg","Info":{"Population":144247}}
{"ID":1527,"Name":"Innsbruck","CountryCode":"AUT","District":"Tiroli","Info":{"Population":111752}}
{"ID":1528,"Name":"Klagenfurt","CountryCode":"AUT","District":"Kärnten","Info":{"Population":91141}}
6 rows in set (0.0032 sec)
```

Example 5.6 Output in Raw JSON Format Wrapped in a JSON Array (`json/array`)

```
MySQL  localhost:33060+ ssl  world_x  JS > shell.options.set('resultFormat','json/array')
MySQL  localhost:33060+ ssl  world_x  JS > session.runSql("select * from city where countrycode='AUT'")
[
{"ID":1523,"Name":"Wien","CountryCode":"AUT","District":"Wien","Info":{"Population":1608144}},
{"ID":1524,"Name":"Graz","CountryCode":"AUT","District":"Steiermark","Info":{"Population":240967}},
{"ID":1525,"Name":"Linz","CountryCode":"AUT","District":"North Austria","Info":{"Population":188022}},
{"ID":1526,"Name":"Salzburg","CountryCode":"AUT","District":"Salzburg","Info":{"Population":144247}},
{"ID":1527,"Name":"Innsbruck","CountryCode":"AUT","District":"Tiroli","Info":{"Population":111752}},
{"ID":1528,"Name":"Klagenfurt","CountryCode":"AUT","District":"Kärnten","Info":{"Population":91141}}
]
6 rows in set (0.0032 sec)
```

5.7.5 JSON Wrapping

To help integrate MySQL Shell with external tools, you can use the `--json` option to control JSON wrapping for all MySQL Shell output when you start MySQL Shell from the command line. The `--json` option only takes effect for the MySQL Shell session for which it is specified.

Specifying `--json`, `--json=pretty`, or `--json=raw` turns on JSON wrapping for the session. With `--json=pretty` or with no value specified, pretty-printed JSON is generated. With `--json=raw`, raw JSON is generated.

When JSON wrapping is turned on, any value that was specified for the `resultFormat` MySQL Shell configuration option in the configuration file or on the command line (with the `--result-format` option or one of its aliases) is ignored.

Specifying `--json=off` turns off JSON wrapping for the session. When JSON wrapping is turned off, or was not requested for the session, result sets are output as normal in the format specified by the `resultFormat` MySQL Shell configuration option.

Example 5.7 MySQL Shell Output with Pretty-Printed JSON Wrapping (`--json` or `--json=pretty`)

```
$> echo "select * from world_x.city where countrycode='AUT'" | mysqlsh --json --sql --uri user@localhos
or
$> echo "select * from world_x.city where countrycode='AUT'" | mysqlsh --json=pretty --sql --uri user@l
{
    "hasData": true,
    "rows": [
        {
            "ID": 1523,
            "Name": "Wien",
            "CountryCode": "AUT",
            "District": "Wien",
            "Info": {
                "Population": 1608144
            }
        },
        {
            "ID": 1524,
            "Name": "Graz",
            "CountryCode": "AUT",
            "District": "Steiermark",
            "Info": {
                "Population": 240967
            }
        },
        {
            "ID": 1525,
            "Name": "Linz",
            "CountryCode": "AUT",
            "District": "North Austria",
            "Info": {
                "Population": 188022
            }
        },
        {
            "ID": 1526,
            "Name": "Salzburg",
            "CountryCode": "AUT",
            "District": "Salzburg",
            "Info": {
                "Population": 144247
            }
        },
        {
            "ID": 1527,
            "Name": "Innsbruck",
            "CountryCode": "AUT",
            "District": "Tiroli",
            "Info": {
```

```
                "Population": 111752
            }
        },
        {
            "ID": 1528,
            "Name": "Klagenfurt",
            "CountryCode": "AUT",
            "District": "Kärnten",
            "Info": {
                "Population": 91141
            }
        }
    ],
    "executionTime": "0.0067 sec",
    "affectedRowCount": 0,
    "affectedItemsCount": 0,
    "warningCount": 0,
    "warningsCount": 0,
    "warnings": [],
    "info": "",
    "autoIncrementValue": 0
}
```


Example 5.8 MySQL Shell Output with Raw JSON Wrapping (`--json=raw`)

```
$> echo "select * from world_x.city where countrycode='AUT'" | mysqlsh --json=raw --sql --uri user@localhos
{"hasData":true,"rows":[{"ID":1523,"Name":"Wien","CountryCode":"AUT","District":"Wien","Info":{"Population"
```

5.7.6 Result Metadata

When an operation is executed, in addition to any results returned, some additional information is returned. This includes information such as the number of affected rows, warnings, duration, and so on, when any of these conditions is true:

- JSON format is being used for the output
- MySQL Shell is running in interactive mode.

When JSON format is used for the output, the metadata is returned as part of the JSON object. In interactive mode, the metadata is printed after the results.

5.8 API Command Line Integration

MySQL Shell exposes much of its functionality through an API command-line integration using a syntax that provides access to objects and their functions without opening the interactive interface. This enables you easily integrate `mysqlsh` with other tools. For example if you want to automate how you create an InnoDB Cluster using a `bash` script, you could use the command-line integration to call AdminAPI operations. This functionality is similar to using the `--execute` option, but the command-line integration uses a simplified argument syntax which reduces the quoting and escaping that can be required by terminals. Unlike batch mode, the command-line integration is stateless. This means that operations which return an object to be used by further operations are not possible. The command-line integration calls operations, or global object's functions, and returns.

5.8.1 Command Line Integration Overview

This section provides an overview of the command-line integration and some basic usage examples. For more detailed information, see Section 5.8.2, “Command Line Integration Details”.

- MySQL Shell Command Line Integration Syntax
- The Objects Available in the Command Line Integration
- MySQL Shell Command Line Integration Argument Syntax
- MySQL Shell Command Line Integration Examples

The following built-in MySQL Shell global objects are available:

- `session`: represents the current global session.
- `db`: represents the default database for the global session, if that session was established using an X Protocol connection with a default database specified. See Using MySQL as a Document Store.
- `dba`: provides access to AdminAPI, used to manage InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet deployments. See Chapter 6, *MySQL AdminAPI*.
- `cluster`: represents an InnoDB Cluster.
- `clusterset`: represents an InnoDB ClusterSet.
- `rs`: represents an InnoDB ReplicaSet.
- `shell`: provides access to MySQL Shell functions, such as `shell.options` for configuring MySQL Shell options (see Section 13.4, "Configuring MySQL Shell Options").
- `util`: provides access to MySQL Shell utilities. See Chapter 11, *MySQL Shell Utilities*.

For more information, see Section 4.5, "MySQL Shell Global Objects".

MySQL Shell Command Line Integration Syntax

Important

As of MySQL Shell 8.0.32, MySQL Shell reads MySQL Server option files and login paths by default. As a result, if you connect to a MySQL Server which uses an option file, it will be used, by default, and attempt to create a global session using that configuration. If you do not want to use the options file, you must add `--no-defaults` to your command line.

You access the command-line integration by starting the `mysqlsh` application and passing in the special `--` option. When you start MySQL Shell in this way, the `--` indicates the end of the list of options (such as the server to connect to, which language to use, and so on) and everything after it is passed to the command-line integration. The command-line integration supports a specific syntax, which is based on the objects and methods used in the MySQL Shell interactive interface. To execute an operation using command-line integration syntax, in your terminal issue:

```
mysqlsh [options] -- [shell_object]+ object_method [arguments]
```

The syntax elements are:

- *`shell_object`* is a string which maps to a MySQL Shell global object. The command-line integration supports nested objects. To call a function in a nested object, provide the list of objects in the hierarchy separated by spaces, to reach the desired object.
- *`object_method`* is the name of the method provided by the last *`shell_object`*. The method names can be provided following either the JavaScript, or Python naming convention, or an alternative command-line integration friendly format, where all known functions use all lower case letters, and words are separated by hyphens. The name of a *`object_method`* is automatically converted from the standard JavaScript style camelCase name, where all case changes are replaced with a `-` and turned into lowercase. For example, `createCluster` becomes `create-cluster`.
- *`arguments`* are the arguments passed to the *`object_method`* when it is called.

`shell_object` must match one of the exposed global objects, and any nested objects must be a child object of the previous object provided in the list. The *`object_method`* must match one of the last object in the list's methods, and must be defined in one of the valid formats (JavaScript, Python or command line friendly). If they do not correspond to a valid object and its methods, MySQL Shell exits with status 10.

See the examples at MySQL Shell Command Line Integration Examples.

The Objects Available in the Command Line Integration

To find out which objects and methods are available in the command-line integration it is best to query the MySQL Shell you are working with. This is because in addition to the standard objects bundled with MySQL Shell, additional objects from plugins might also be exposed.

To get the list of objects supported by the command-line integration:

```
$ mysqlsh -- --help
```

This displays a list of objects and a brief description of what the object provides.

To get a list of the functions available in the command-line integration for an *`object`*:

```
$ mysqlsh -- object --help
```

For more information, see Section 5.8.2.4, “Command Line Help”.

MySQL Shell Command Line Integration Argument Syntax

The *`arguments`* list is optional and all arguments must follow a syntax suitable for command-line use as described in this section. Special characters (such as spaces or \) and quoting are processed by your system's shell (`bash`, `cmd`, and so on) before they are passed to MySQL Shell. If you are unfamiliar with how your system shell deals with those character sequences as it parses a command, you should try to avoid them. For example, to pass a parameter with quotes as part of the parameter such as “list, of, names”, using just that syntax on the command line is not enough. You need to use your system's shell syntax for escaping those quotes. If you do not, then MySQL Shell might not receive the actual quotation marks. See Section 5.8.2.2, “Defining Arguments”.

There are two types of arguments that can be used in the list of arguments: anonymous arguments and named arguments. Anonymous arguments are used to define simple type parameters such as strings, numbers, boolean, null. Named arguments are used to define the values for list parameters and the options in a dictionary parameter, they are key-value pairs, where the values are simple types. Their usage must adhere to the following pattern:

```
[positional_argument | named_argument]*
```

All parts of the syntax are optional and can be given in any order. These arguments are then converted into the arguments passed to the method call in the following order:

- Named arguments that come from lists cause the values to be appended to the list parameter that originated the named argument
- Named arguments that come from dictionaries cause the values to be added to the dictionary parameter that originated the named argument
- If a dictionary parameter exists with no explicit options defined, this causes it to accept *any* named argument that does not belong to another List or Dictionary parameter
- Any remaining arguments provided to the function call are processed in the order they are provided

MySQL Shell Command Line Integration Examples

Using the command-line integration, calling MySQL Shell API functions is easier and less cumbersome than with the `--execute` option. The following examples show how to use this functionality:

- To check a server instance is suitable for upgrade and return the results as JSON for further processing:

```
$ mysqlsh -- util check-for-server-upgrade --user=root --host=localhost --port=3301 --password='passwor
```

The equivalent command in MySQL Shell interactive mode:

```
mysql-js> util.checkForServerUpgrade({user:'root', host:'localhost', port:3301}, {password:'passwor
```

- To deploy an InnoDB Cluster sandbox instance, listening on port 1234 and specifying the password used to connect:

```
$ mysqlsh -- dba deploy-sandbox-instance 1234 --password=password
```

The equivalent command in MySQL Shell interactive mode:

```
mysql-js> dba.deploySandboxInstance(1234, {password: password})
```

- To create an InnoDB Cluster using the sandbox instance listening on port 1234 and specifying the name `mycluster`:

```
$ mysqlsh root@localhost:1234 -- dba create-cluster mycluster
```

The equivalent command in MySQL Shell interactive mode:

```
mysql-js> dba.createCluster('mycluster')
```

- To check the status of an InnoDB Cluster using the sandbox instance listening on port 1234:

```
$ mysqlsh root@localhost:1234 -- cluster status
```

The equivalent command in MySQL Shell interactive mode:

```
mysql-js> cluster.status()
```

- To configure MySQL Shell to turn the command history on:

```
$ mysqlsh -- shell options set_persist history.autoSave true
```

The equivalent command in MySQL Shell interactive mode:

```
mysql-js> shell.options.set_persist('history.autoSave', true);
```

5.8.2 Command Line Integration Details

This section provides detailed information about the MySQL Shell command-line integration.

5.8.2.1 Command Line Integration for MySQL Shell API Functions

The MySQL Shell provides global objects that expose different functionality, such as `dba` for InnoDB Cluster and InnoDB ReplicaSet management operations, `util` for the utility functions, and so on. Global objects provide functions which are called from the scripting modes in the MySQL Shell. In addition to the interactive MySQL Shell integration, you can use the command-line integration to call object functions directly from the terminal, enabling you to easily integrate with other tools.

When you use the APIs included with MySQL Shell in the interactive mode, the typical function syntax is as follows:

```
object.functionName(parameter1, parameter2, ..., parameterN)
```

The parameters define the order in which the data should be provided to the API function. In most cases, API functions expect the parameters in a specific data type, however there are a few exceptions where a specific parameter can handle multiple data types. The data types used for parameters in API functions can be one of the following:

- Scalars: string, numbers, booleans, null
- Lists

- Dictionaries: key-value pairs where the key is a string
- Objects

List parameters are typically restricted to contain elements of a pre-defined data type, for example a list of strings, however, there could be list parameters that support items of different data types.

Dictionary parameters accept *`key`*-*`value`* pairs, where *`keys`* are strings. The *`value`* associated to a key is usually expected to be of a pre-defined data type. However, there might be cases where different data types are supported for values by the same key. Dictionary parameters can therefore be one of the following types:

- A pre-defined set of keys-value pairs is allowed, in which case specifying keys not in the pre-defined set results in an error.
- No pre-defined set of key-value pairs exists, the dictionary accepts any key

In other words, some dictionary parameters specify which keys are valid. For those parameters, attempting to use a key outside of that set results in an error. When no pre-defined set of values exists, any value of any data type can be used. Dictionary parameters that do not have a pre-defined list of keys, accept any key-value pair as long as the key is not in the pre-defined set of a different dictionary parameter.

To use the command-line integration to call API functions exposed by global objects without having to start an interactive session in the MySQL Shell you must provide the required data in the correct way. This includes defining the way an API function is called, as well as the way its parameters are mapped from command-line arguments to API arguments.

Important

Not all of the MySQL Shell functions are exposed by the command-line integration. For example a function such as `dba.getCluster()` relies on returning an object which is then used in further operations. Such operations are not exposed by the command-line integration.

Similarly, the MySQL Shell command-line integration does not support Objects as parameters. Any API function with a parameter of type object cannot be used with the command-line integration. The lifetime of the object is limited to the lifetime of the MySQL Shell invocation that created it. Since `mysqlsh` exits immediately after executing an object method through this API syntax, any objects received from or passed into an API call would immediately be out of scope. This should be considered while developing MySQL Shell Plugins that you want to expose with the command-line integration.

The general format to call a MySQL Shell API function from the command-line is:

```
$ mysqlsh [shell options] -- [shell_object]+ object_function [anonymous_arguments|named arguments]*
```

Where:

- *`shell_object`*: specifies a global object with functions exposed for command-line usage. Supports nested objects in a list separated by spaces.
- *`object_function`*: specifies the API function of the last *`shell_object`* which should be executed.
- *`[anonymous_arguments|named arguments]*`*: specifies the arguments passed to the *`object_function`* call

For most of the available APIs a single object is required, for example:

```
$ mysqlsh -- shell status
```

But for nested objects, the list of objects must be indicated. For example, to call a function exposed by `shell.options`, such as `setPersist(optionName, value)`, use the syntax:

```
$ mysqlsh -- shell options set-persist defaultMode py
```

A similar situation might happen with nested objects defined in MySQL Shell Plugins.

The arguments you pass to functions can be divided into the following types:

- Anonymous Arguments: which are raw values provided to the command. For example, in the following call `1`, `one` and `true` are anonymous arguments:

  ```
  $ mysqlsh -- object command 1 one true
  ```

- Named Arguments: which are key-value pairs provided in the form of `--key=value`. For example in the following call, `--sample` and `--path` are named arguments:

  ```
  $ mysqlsh -- object command 1 one true --sample=3 --path=some/path
  ```

Given this division of arguments, the general format to call an API function from the command-line integration is:

```
$ mysqlsh [shell options] -- object command [anonymous arguments][named arguments]
```

The order of any *`anonymous arguments`* is important as they are processed in a positional way. On the other hand, *`named arguments`* can appear anywhere as they are processed first and are associated to the corresponding parameter. Once named arguments are processed, the anonymous arguments are processed in a positional way.

5.8.2.2 Defining Arguments

As mentioned in Section 5.8.2.1, “Command Line Integration for MySQL Shell API Functions”, most of the APIs available in MySQL Shell expect a specific data type for the arguments being provided. Values in command-line arguments can be provided using the JSON specification with the following considerations. Some terminals do their own pre-processing of the data which can impact the way the data is provided to MySQL Shell, and this varies depending on the terminal being used. For example:

- Some terminals split arguments if whitespace is found.
- Consecutive whitespace could be ignored by the splitting logic.
- Quotes could be removed.

MySQL Shell interprets the values as provided by the terminal it is running in, therefore you must provide the data to the terminal in a way that is correctly formatted. For example:

-

  Important

  Some terminals require quotes to be escaped

- String arguments should be quoted in the following cases:
  - They contain whitespace
  - The argument is for a list parameter and contains commas
  - They contain escaped characters
- The API parameter can accept different data types and the value (based on the JSON specification) could be the wrong data type.
- When defining parameters using JSON, quote string values and string keys. Avoid using whitespace outside of quoted items.

The following examples illustrate some of the handling of parameters.

- To pass in multiple parameters, each a single string, no quoting is required:

```
$ mysqlsh -- object function simple string
```

  In this case, MySQL Shell gets two arguments - argument 1 is `simple`, and argument 2 is `string`.

- If you want these two strings to be treated as a single parameter, they must be surrounded by quote marks, as follows

```
$ mysqlsh -- object function "simple string"
```

  In this case, MySQL Shell gets one argument - argument 1 is `simple string`.

- To use an argument which contains characters such as a backslash, the string must be quoted. Otherwise the character is ignored. For example:

```
$ mysqlsh -- object function simple\tstring
```

  In this case, MySQL Shell gets one argument - `simpletstring`, the backslash character (`\`) has been ignored.

  To ensure the backslash character is passed to MySQL Shell, surround the string with quotes:

```
$ mysqlsh -- object function "simple\tstring"
```

  In this case, MySQL Shell gets one argument - `simple\tstring`.

When using the command-line integration, defining a JSON array has its own caveats. For example, in the MySQL Shell interactive mode you define a JSON array as:

```
["simple",123]
```

To use the same array in the command-line integration requires specific quoting. The following example illustrates how to correctly quote the JSON array:

- Attempting to pass the JSON array in the same way as the interactive mode does not work:

```
$ mysqlsh -- object function ["simple", 123]
```

  In this case, MySQL Shell gets two arguments - argument 1 is `[simple,` and argument 2 is `123]`.

- Not using spaces in the array helps, but it is still an invalid JSON array:

```
$ mysqlsh -- object function ["simple",123]
```

  In this case, MySQL Shell gets one argument - `[simple,123]`.

- To make a valid JSON array, add escaped quotes within the already quoted string element, for example:

```
$ mysqlsh -- object function ["\"simple\"",123]
```

  In this case, MySQL Shell gets one argument - `["simple",123]`.

To use a JSON array which contains JSON objects requires quoting in a similar way. For example, in the MySQL Shell interactive mode you define a JSON array which contains JSON objects as:

```
{"firstName":"John","lastName":"Smith"}
```

The following example illustrates how to correctly quote the same array in the command-line integration:

- Attempting to pass the JSON array in the same way as the interactive mode does not work:

```
$ mysqlsh -- object function {"firstName":"John","lastName":"Smith"}
```

In this case, MySQL Shell gets two arguments - argument 1 is *`firstName:John`* and argument 2 is *`lastName:Smith`*.

- Using escaped quotes for string data leads to:

```
$ mysqlsh -- object function {\"firstName\":\"John\",\"lastName\":\"Smith\"}
```

In this case, MySQL Shell gets two arguments - argument 1 is *`"firstName":"John"`* and argument 2 is *`"lastName":"Smith"`*.

- To fix this, you need to additionally quote the whole JSON object, to get:

```
$ mysqlsh -- object function "{\"firstName\":\"John\",\"lastName\":\"Smith\"}"
```

In this case, MySQL Shell gets one argument - *`{"firstName":"John","lastName":"Smith"}`*.

Due to the difficulties shown and the fact that the way the terminals in different platforms behave might be different, the following formats are supported.

String Arguments

Strings require quoting *only* in the following cases:

- The value contains spaces
- The value itself contains commas and is for a list parameter (to avoid splitting)
- The value contains escaped characters
- The value is a number, `null`, `true`, `false` but it is meant to be a string. In these cases the value should be quoted with inner escaped quotes. In other words, if a string value is "true", it should be defined in a CLI call as ""true"".

List Arguments

In addition to a JSON array, an argument for a list parameter can be provided as:

- a comma separated list of values
- separate anonymous arguments

When a list parameter is being processed (in positional order), all of the remaining anonymous arguments are part of the list. The following MySQL Shell CLI calls are equivalent:

- Using a comma separated list of values:

```
$ mysqlsh root@localhost -- util dump-schemas sakila,employees
```

- Using consecutive anonymous arguments:

```
$ mysqlsh root@localhost -- util dump-schemas sakila employees
```

- Using a JSON array:

```
$ mysqlsh root@localhost -- util dump-schemas ["\"sakila\"","\"employees\""]
```

Dictionary Arguments

Dictionaries are created using key-value pairs, the value for a key in a dictionary argument can also be specified using named arguments:

```
--key=value
```

The following MySQL Shell CLI call illustrates how the `threads` and `osBucketName` keys are defined for the options parameter in the `util.dumpInstance()` function:

```
$ mysqlsh -- util dump-instance my-dump --threads=8 --osBucketName=my-bucket
```

List Keys

You can define the values of a list key in a dictionary in the following ways:

- Defining the value as a JSON array.
- Defining the value as a comma separated list of values.
- Defining values for the key repeatedly.

For example, in the following calls, the definition of the `excludeSchemas` key passed to the `util.dumpInstance()` operation is equivalent:

- Using a comma separated list of values:

  ```
  $ mysqlsh root@localhost -- util dump-instance --outputUrl="my-dump" --excludeSchemas=sakila,employees
  ```

- Using a JSON array:

  ```
  $ mysqlsh root@localhost -- util dump-instance --outputUrl="my-dump" --excludeSchemas=["\"sakila\"","\"em
  ```

- Defining several values for the `--excludeSchemas` key:

  ```
  $ mysqlsh root@localhost -- util dump-instance --outputUrl="my-dump" --excludeSchemas=sakila --excludeSch
  ```

Dictionary Keys

Nested dictionaries are supported with the following restrictions:

- Only one level of nesting is supported.
- Validation for inner pre-defined keys is not supported.
- Validation for inner expected data types is not supported.

The syntax to define a value for a key in a nested dictionary is as follows:

```
--key=innerKey=value
```

For example, to define the `decodeColumns` key and pass it to the `util.importTable()` operation:

```
$ mysqlsh -- util import-table --decodeColumns=myColumn=1
```

Additional Named Arguments

As shown in the previous section, dictionary parameters are supported through named arguments using the `--key=value` syntax. There is another case when arguments must be specified as named arguments: parameters which are defined after a list parameter. The most convenient way to provide arguments that belong to a list parameter is by using anonymous arguments, for example as shown in the example at List Arguments:

```
$ mysqlsh root@localhost -- util dump-schemas sakila employees
```

However, this example is missing the argument for the `outputUrl` parameter, which is mandatory for the `util.dumpSchemas()` operation. Because all of the remaining anonymous arguments are included as items in the schemas list, there is no way to specify the `outputUrl` as an anonymous argument. For example the following would *not* work:

```
$ mysqlsh root@localhost -- util dump-schemas sakila employees path/to/dump
```

In this call, the path `path/to/dump` would be interpreted as another item in the schemas list. For this reason, *any* parameter defined after a list parameter must be specified as a named argument when calling the function from the command-line. For example:

```
$ mysqlsh root@localhost -- util dump-schemas sakila employees --outputUrl=path/to/dump
```

5.8.2.3 Data Type Handling

In general, the data type of an argument is resolved using the following criteria, in order of priority:

- The expected data type for the target parameter.
- The data type of the value based on the JSON specification.
- User specified data type.

The last case is a complicated (and rare) case applicable for named arguments only. For example, suppose you have a MySQL Shell Plugin function such as:

```
def set_object_attributes(variables)
```

Where `variables` is a dictionary with *no* pre-defined set of values, thus it accepts any key, and therefore accepts any data type for the value. To set a string attribute named `streetNumber` with a *string* value of 123, issue:

```
$ mysqlsh -- plugin set-object-attributes --streetNumber=123
```

Because there is no expected data type, the value `123` is interpreted as a numeric value according to the JSON specification, but we wanted to store that as a string, not as a number.

Note

Currently there is no case of an API function like this unless user creates a plugin as explained above.

User Data Types

To avoid issues with MySQL Shell trying to guess the type of input data, the command-line integration supports forcing a specific data type, by specifying a named argument using the following syntax:

```
--key:type=value
```

Where *`type`* is one of:

- str
- int
- uint
- float
- bool
- list
- dict
- json

To store the value as a string, issue:

```
$ mysqlsh -- plugin set-object-attributes --streetNumber:str=1234
```

Important

This format is allowed in any named argument, but it is only required when there is no expected data type for the argument. If there is an expected data type for the parameter and you specify a different data type, an error is raised.

Data Type Resolution

When you do not specify the data type, MySQL Shell attempts to resolve the data type using the following logic. This data interpretation logic is based on the JSON specification but has some MySQL Shell specific additions and limitations:

- Strings:
  - Support both double quoted and single quoted strings.
  - Support for hexadecimals such as `\x`*`NN`* where *`NN`* is a hexadecimal digit. This is used to represent ASCII characters in hexadecimal format.
  - Support for vertical tabs escaped character
- The following literals can also be defined:
- undefined: define a value as undefined (not really needed in CLI so usage is discouraged).
- true/false: creates a boolean value.
- null: define a null value.

Any value not covered by the JSON specification and the rules above is interpreted as a plain string.

5.8.2.4 Command Line Help

You can access the MySQL Shell online help when calling commands from the command-line integration using the `--help` (-h) CLI argument. Help is supported at the global, object and command level.

Note

The built-in help CLI argument does not map to any API argument and is supported in all the objects and commands available in CLI.

The descriptions of the commands and parameters is taken from the existing documentation for the target API function.

Global CLI Help

To retrieve the list of global objects available for CLI calls, use the following syntax:

```
$ mysqlsh -- --help
```

In this example, `--` initiates the command-line integration section of the command. Using the `--help` or `-h` option alone after that lists the global objects available within this interface.

Object Help

To access the object help from the command-line integration, use the following syntax:

```
$ mysqlsh -- object --help
```

where *`object`* is what you want help on, such as the `dba` global object. This call displays:

- A brief description of the object.

- A list of the available commands and a short description of them.

To retrieve the help for nested objects, provide the entire list of objects before the `--help` argument. For example, to get help on the `shell.options` functions, issue:

```
$ mysqlsh -- shell options --help
```

Command Help

To display help on commands from the command-line integration, use the following syntax:

```
$ mysqlsh -- object command --help
```

This call displays details about the *`command`*, including:

- A brief description of what the command does.
- The signature for calling the command.
- The list of anonymous arguments and a brief description of each.
- The list of named arguments, their expected data types, and a brief description explaining the purpose of each argument.

For the case of commands in nested objects, the entire list of objects should be provided before the command, for example:

```
$ mysqlsh shell options set-persist --help
```

For parameters that expect a specific data type, the argument is listed as:

```
--name=type
Brief description of the parameter.
```

The type information represents the expected data type for the argument, for example: `str`, `int`, `uint`, `bool`, `list`, `float`, or `dict`.

For example, the `consistent` key of the `dump-schemas` parameter:

```
$ mysqlsh -- util dump-schemas --help
...
--consistent=<bool>
Enable or disable consistent data dumps. Default: true.
...
```

For parameters that support different data types, the argument is listed as:

```
--name[:type]=value
Brief description of the parameter.
```

For example, the `columns` key of the `util.importTable()` operation.

```
$ mysqlsh -- util import-table --help
...
--columns[:<type>]=<value>
Array of strings and/or integers (default: empty array) - This...
...
```

5.8.2.5 Support for MySQL Shell Plugins

To use Section 10.3, “MySQL Shell Plugins” with the command-line integration, the functions must be explicitly defined for CLI support. When an object defined in a MySQL Shell Plugin is enabled for command-line integration, *only* the specific functions that were enabled are available for CLI calls. From MySQL Shell version 8.0.24, when you add function members to an object, they support the `cli` boolean option. When `cli` is set to `true`, the function is available from the command-line

integration. The `cli` option defaults to false, therefore functions are not available from the command-line integration unless specifically enabled. Any object with a function that has the `cli` option enabled causes its parent objects to be available in the command-line integration as well.

To make a function available through the command-line integration, set the `cli` option to `true` when you add the extension object member. For example:

```
shell.addExtensionObjectMember(object, "exampleFunction", exampleFunction,
                       {
                         brief:"Retrieves brief information",
                         details: ["Retrieves detailed information"],
                         cli: true,
                         parameters:
                         [
                           {
                             name: "param_a",
                             type: "string",
                             brief: "param_a brief"
                           }
                         ]
                       });
```

You could then use the `exampleFunction()` function from the command-line integration as follows:

```
mysqlsh -- customObj exampleFunction 1
```

If you have added an extension object member using a MySQL Shell version earlier than 8.0.24, and you want to use it with the command-line integration, you must enable the `cli` option. Use the `addExtensionObjectMember` method as illustrated here to add the object member again, this time enabling the `cli` option.

5.9 JSON Integration

From MySQL Shell 8.0.27, you can activate a JSON shell mode to help with integration of MySQL Shell with other applications that could use its functionality. In this mode, MySQL Shell accepts commands formatted as JSON documents.

To activate the JSON shell mode, define the `MYSQLSH_JSON_SHELL` environment variable. The following commands can then be used:

`{"execute":`*`json-string`*`}`	Executes the given code in the active MySQL Shell mode (JavaScript, Python or SQL). The code is executed as a complete unit, and an error is returned if it is incomplete.
`{"command":`*`json-string`*`}`	Executes the given MySQL Shell command (see Section 3.1, “MySQL Shell Commands”).
`{"complete": {"data":`*`json-string`*`[, "offset": `*`uint`*`}}}`	Determines the options for auto-completion based on the given data and the current MySQL Shell context.

5.10 Limitations

This section describes the limitations of the various supported MySQL Shell modes.

SQL Limitations for X Protocol Sessions

The following statements are not possible with an X Protocol session:

- `ALTER INSTANCE`
- `INSTALL COMPONENT`

- `INSTALL PLUGIN`
- `LOAD DATA LOCAL INFILE`
- `LOCK INSTANCE FOR BACKUP`
- `UNINSTALL COMPONENT`
- `UNINSTALL PLUGIN`
- `UNLOCK INSTANCE`
- `SHOW BINARY LOGS`
- `SHUTDOWN`

Chapter 6 MySQL AdminAPI

Table of Contents

6.1 Using MySQL AdminAPI 75
6.2 Installing AdminAPI Software Components 76
6.2.1 Using Instances Running MySQL 5.7 77
6.2.2 Configuring the Host Name 78
6.2.3 Connecting to Server Instances 78
6.2.4 Persisting Settings 79
6.3 Retrieving a Handler Object 80
6.4 Creating User Accounts for AdminAPI 82
6.5 Verbose Logging 84
6.6 Finding the Primary 84
6.7 Scripting AdminAPI 85
6.8 AdminAPI MySQL Sandboxes 86
6.8.1 Deploying Sandbox Instances 87
6.8.2 Managing Sandbox Instances 88
6.8.3 Setting up InnoDB Cluster and MySQL Router 88
6.9 Tagging Metadata 95
6.10 Using MySQL Router with AdminAPI, InnoDB Cluster, and InnoDB ReplicaSet 100
6.10.1 Bootstrapping MySQL Router 100
6.10.2 Configuring the MySQL Router User 100
6.10.3 Deploying MySQL Router 102
6.10.4 Using ReplicaSets with MySQL Router 104
6.10.5 Testing InnoDB Cluster High Availability 104
6.10.6 Working with a Cluster's Routers 105
6.11 Upgrade Metadata Schema 107
6.12 Locking Mechanism for AdminAPI Operations 108

This chapter covers MySQL AdminAPI, provided with MySQL Shell, which enables you to administer MySQL instances, using them to create InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet deployments, and integrating MySQL Router.

6.1 Using MySQL AdminAPI

AdminAPI is provided by MySQL Shell. AdminAPI is accessed through the `dba` global variable and its associated methods. The `dba` variable's methods provide the operations which enable you to deploy, configure, and administer InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet. For example, use the `dba.createCluster()` method to create an InnoDB Cluster. In addition, AdminAPI supports administration of some MySQL Router related tasks, such as creating or upgrading a user account that works with InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet.

AdminAPI supports the following deployment scenarios:

- *Production deployment:* If you want to use a full production environment, you need to configure the required number of machines and then deploy your server instances to the machines.
- *Sandbox deployment:* If you would like to test a deployment before committing to a full production deployment, the provided sandbox feature enables you to set up a test environment on your local machine. Sandbox server instances are created for you with the required configuration. You can experiment to become familiar with the technologies employed.

Important

An AdminAPI sandbox deployment is not suitable for use in a full production environment.

MySQL Shell provides two language modes, JavaScript and Python, in addition to a native SQL mode. Throughout this guide MySQL Shell is used primarily in JavaScript mode. When MySQL Shell starts it is in JavaScript mode by default. Switch modes by issuing `\js` for JavaScript mode, and `\py` for Python mode. Ensure you are in JavaScript mode by issuing the `\js`.

Important

MySQL Shell enables you to connect to servers over a socket connection, but AdminAPI requires TCP connections to a server instance. Socket based connections are not supported in AdminAPI.

This section assumes familiarity with MySQL Shell; see MySQL Shell 8.0 for further information. MySQL Shell also provides online help for the AdminAPI. To list all available `dba` commands, use the `dba.help()` method. For online help on a specific method, use the general format `object.help('methodname')`. For example, using JavaScript:

```
mysql-js> dba.help('getCluster')

Retrieves a cluster from the Metadata Store.

SYNTAX

  dba.getCluster([name][, options])

WHERE

  name: Parameter to specify the name of the cluster to be returned.
  options: Dictionary with additional options.
  ...
```

Or using Python:

```
mysql-py>dba.help('get_cluster')
NAME
      get_cluster - Retrieves a cluster from the Metadata Store.

SYNTAX
      dba.get_cluster([name][, options])

WHERE
      name: Parameter to specify the name of the cluster to be returned.
      options: Dictionary with additional options.
...
```

In addition to this documentation, there is developer documentation for all AdminAPI methods in the MySQL Shell JavaScript API Reference or MySQL Shell Python API Reference, available from Connectors and APIs.

6.2 Installing AdminAPI Software Components

How you install the software components required by AdminAPI depends on the type of deployment you intend to use:

- For a production deployment, install the components to each machine. A production deployment uses multiple remote host machines running MySQL server instances, so you need to connect to each machine using a tool such as SSH or Windows remote desktop to carry out tasks such as installing components.
- For a sandbox deployment, install the components to a single machine. A sandbox deployment is local to a single machine, therefore the installation needs to only be done once on the local machine.

Important

Always use the most recent versions of MySQL Shell and MySQL Router that are available to you, and ensure that their version is the same as or higher than

> the MySQL Server release. MySQL Shell and MySQL Router can manage older MySQL Server releases, but older versions of the products cannot manage features in newer MySQL Server releases.

Download and install the software components using the following documentation:

- MySQL Server - see Installing MySQL.
- MySQL Shell - see Chapter 2, *Installing MySQL Shell*.
- MySQL Router - see Installing MySQL Router.

Once you have installed the required software, this section has further information on using AdminAPI. Follow the procedures to set up Chapter 7, *MySQL InnoDB Cluster*, Chapter 8, *MySQL InnoDB ClusterSet*, or Chapter 9, *MySQL InnoDB ReplicaSet*.

6.2.1 Using Instances Running MySQL 5.7

This documentation assumes you are using MySQL instances running the latest version of MySQL 8, and MySQL Shell 8. AdminAPI also supports using instances running MySQL 5.7, but many of the features described require instances running MySQL 8. The following differences exist for instances running MySQL 5.7:

- Instances running MySQL 5.7 do not support `SET PERSIST`, so they cannot be configured remotely, or automatically, unlike instances running MySQL 8. Instead, when configuring MySQL 5.7 instances, each time you must connect to the instance and use the `dba.configureLocalInstance()` operation. This operation persists the settings to the instances option file when it is available locally. See Section 6.2.4, “Persisting Settings”.
- Instances running MySQL 5.7 do not support automatic node provisioning, so before joining them to the cluster, you must manually synchronize them with the other cluster instances. This means either relying on Group Replication's distributed recovery, which requires binary logs with GTIDs enabled and potentially a long wait when there is a large number of transactions to recover, or using a tool such as MySQL Enterprise Backup to manually copy the data.

  With the addition of the MySQL Clone plugin in version 8.0, instances can be provisioned by AdminAPI automatically. When you add a version 8.0 instance supporting MySQL Clone, AdminAPI automatically chooses the best way to bring the joining instance into synchrony with the existing instances. For example, if the cluster contains a large number of transactions, MySQL Clone is used to recover the data directly, and any transactions processed by the cluster during the clone operation are then synchronized using distributed recovery. You can monitor the progress of the operation directly from MySQL Shell, no other tools are required. This makes tasks such as adding instances to expand the InnoDB Cluster and improve the chances of high availability effortless. For more information, see Section 7.4.6, “Using MySQL Clone with InnoDB Cluster”.
- Instances running MySQL 5.7 are not compatible with InnoDB ReplicaSet.
- Instances running MySQL 5.7 are not compatible with InnoDB ClusterSet.
- The InnoDB Cluster topology (whether it is running in single-primary or multi-primary mode) cannot be dynamically changed when using MySQL 5.7 Servers. For more information, see Changing a Cluster's Topology.
- Instances running MySQL 5.7 are not compatible with the parallel replication applier. For more information see Section 7.5.6, “Configuring the Parallel Replication Applier”.
- Instances running MySQL 5.7 do not support the `autoRejoinTries` and `exitStateAction` options, which configure how many times instances try to rejoin a cluster and what happens when an instance leaves. For more information, see Section 7.5.5, “Configuring Automatic Rejoin of Instances”.

- Instances running MySQL 5.7 do not support the `consistency` option. For more information, see Section 7.5.4, “Configuring Failover Consistency”.
- Instances running MySQL 5.7 do not support the `expelTimeout` option, which configures how long the cluster waits before expelling an instance which has lost contact with the other instances.

To use these features, please upgrade your instances to MySQL 8.

With instances running MySQL 5.7, ensure that you use `dba.configureInstance()` before adding the instances to a cluster to persist the configuration changes. For non-sandbox server instances on MySQL 5.7 (instances which you have configured manually rather than using `dba.deploySandboxInstance()`), if you do not use the `dba.configureInstance()` operation, MySQL Shell can not persist anyInnoDB ClusterSet configuration changes in the instance's configuration file. This leads to one or both of the following scenarios:

1. The Group Replication configuration is not persistent in the instance's configuration file, and upon restart, the instance does not rejoin the cluster.
2. The instance is not valid for cluster usage. Although the instance can be verified with `dba.checkInstanceConfiguration()`, and MySQL Shell makes the required configuration changes in order to make the instance ready for cluster usage, those changes are not persisted in the configuration file and so are lost once a restart happens.

If both situations happen, you cannot use the `dba.rebootClusterFromCompleteOutage()` operation to get the cluster back online. This is because without the `dba.configureInstance()` operation, the instance loses any configuration changes made by MySQL Shell, and because they were not persisted, the instance reverts to the previous state before being configured for the cluster. This causes Group Replication to stop responding, and eventually the command times out.

6.2.2 Configuring the Host Name

In a production deployment, the instances in which you use run-on separate machines, therefore each machine must have a unique hostname and be able to resolve the hostnames of the other machines, which run server instances. If this is not the case, you can:

1. Configure each machine to map the IP of each other machine to a hostname. See your operating system documentation for details. This configuration is the recommended solution.
2. Set up a Domain Name System (DNS) service.
3. Configure the `report_host` variable in the MySQL configuration of each instance to a suitable externally reachable address.

AdminAPI supports using IP addresses instead of host names. From MySQL Shell 8.0.18, AdminAPI supports IPv6 addresses if the target MySQL Server version is higher than 8.0.13.

When using MySQL Shell 8.0.18 or higher, if all cluster instances are running 8.0.14 or higher, then you can use an IPv6 address or a hostname that resolves to an IPv6 address in connection strings and with options such as `localAddress` and `ipAllowlist`. For more information on using IPv6, see Support For IPv6 And For Mixed IPv6 And IPv4 Groups.

Previous versions support IPv4 addresses only.

To verify whether the hostname of a MySQL server you have correctly configured, process the following query. This query shows how the instance reports its address to other servers and try to connect to that MySQL server from other hosts using the returned address:

```
SELECT coalesce(@@report_host, @@hostname);
```

6.2.3 Connecting to Server Instances

MySQL Shell enables you to work with various APIs, and supports specifying connections as explained in Connecting to the Server Using URI-Like Strings or Key-Value Pairs. You can specify connections using either URI-like strings, or key-value pairs. The Additional Connection parameters are not supported in AdminAPI. This documentation demonstrates AdminAPI using URI-like connection strings.

For AdminAPI operations, you can only connect to server instances in an InnoDB Cluster using TCP/IP connections and classic MySQL protocol. The use of Unix sockets and named pipes is not supported for AdminAPI operations, and the use of X Protocol is not supported for AdminAPI operations. The same limitations apply to connections between the server instances themselves.

Note

Client applications can use X Protocol and Unix sockets and named pipes to connect to instances in an InnoDB Cluster. The limitations only apply to administration operations using AdminAPI commands, and to connections between the instances.

For example, to connect as the user *`myuser`* to the MySQL server instance at *`www.example.com`*, on port *`3306`* use the connection string:

```
myuser@www.example.com:3306
```

To use this connection string with an AdminAPI operation such as `dba.configureInstance()`, you need to ensure the connection string is interpreted as a string. For example, by surrounding the connection string with either single (') or double (") quote marks.

If you are using the JavaScript implementation of AdminAPI issue:

```
mysql-js> > dba.configureInstance('myuser@www.example.com:3306')
```

If you are using the Python implementation of AdminAPI issue:

```
mysql-py> dba.configure_instance('myuser@www.example.com:3306')
```

You are prompted for your password if you are running MySQL Shell in the default interactive mode. AdminAPI supports MySQL Shell's Section 4.4, “Pluggable Password Store”, and once you store the password you used to connect to the instance, you will no longer be prompted for it.

MySQL Shell defaults to trying X Protocol for connection to a server instance. If you do not specify the connection type when you make a connection for an AdminAPI operation, MySQL Shell's automatic protocol detection briefly creates a session for X Protocol, before it creates a classic MySQL protocol session.

The behavior has no effect unless you are connecting to an InnoDB Cluster with only two secondary (read-only) instances using a port that a MySQL Router is managing. In this case, load balancing is not managed correctly between the two instances, and the same instance is always used. To avoid this side-effect, you can specify a classic MySQL protocol session explicitly by adding the `--mc` or `--mysql` option.

Certain operations that open many connections to servers can take a long time to execute when one or more servers are indeed unreachable, for example, the `cluster.status()` command. The connection timeout may not provide enough time for a response.

From MySQL Shell 8.0.28, you can use the MySQL Shell configuration option `dba.connectTimeout` to set the default connection timeout in seconds for any session using AdminAPI.

6.2.4 Persisting Settings

The AdminAPI commands you use to work with an InnoDB Cluster, InnoDB ClusterSet, InnoDB ReplicaSet, and the individual member server instances in these deployments modify the configuration of MySQL Server on the instance. Depending on the way MySQL Shell is connected to an instance and

the version of MySQL Server installed on the instance, these configuration changes can be persisted to the instance automatically.

By making settings to the instance persistent, you ensure that after the instance restarts, configuration changes are retained. For background information see `SET PERSIST`. This persistence is essential for reliable usage. For example, if settings are not persistent, an instance added to a cluster does not rejoin the cluster after a restart because configuration changes are lost.

Instances which meet the following requirements support persisting configuration changes automatically:

- The instance is running MySQL version 8.0.11 or later.
- `persisted_globals_load` is set to `ON`.
- The instance has not been started with the `--no-defaults` option.

Instances which do not meet these requirements do not support persisting configuration changes automatically, and when AdminAPI operations result in changes to the instance's settings to be persisted you receive warnings such as:

```
WARNING: On instance 'localhost:3320' membership change cannot be persisted since MySQL version 5.7.21
does not support the SET PERSIST command (MySQL version >= 8.0.5 required). Please use the
<Dba>.configureLocalInstance command locally to persist the changes.
```

When AdminAPI commands are issued against the MySQL instance which MySQL Shell is currently running on, in other words, the local instance, MySQL Shell persists configuration changes directly to the instance. On local instances which support persisting configuration changes automatically, configuration changes are persisted to the instance's `mysqld-auto.cnf` file, and the configuration change does not require any further steps.

You must make the changes locally on local instances that do not support persisting configuration changes automatically. For more information, see Configuring Instances with `dba.configureLocalInstance()`.

When run against a remote instance, in other words, an instance other than the one which MySQL Shell is currently running on, if the instance supports persisting configuration changes automatically, the AdminAPI commands persist configuration changes to the instance's `mysql-auto.conf` option file.

If a remote instance does not support persisting configuration changes automatically, the AdminAPI commands can not automatically configure the instance's option file. So, the AdminAPI commands can read information from the instance, for example, to display the current configuration. But changes to the configuration cannot be persisted to the instance's option file. In this case, you need to persist the changes locally. For more information, see Configuring Instances with `dba.configureLocalInstance()`.

6.3 Retrieving a Handler Object

When you are working with AdminAPI, you use a handler object which represents the InnoDB Cluster, InnoDB ClusterSet, or InnoDB ReplicaSet. You assign this object to a variable, and then use the operations available to monitor and administer the InnoDB Cluster, InnoDB ClusterSet, or InnoDB ReplicaSet.

To retrieve the handler object, you establish a connection to one of the active instances which belong to the InnoDB Cluster, InnoDB ClusterSet, or InnoDB ReplicaSet. For example, when you create a cluster using `dba.createCluster()`, the operation returns a `Cluster` object which can be assigned to a variable. You use this handler object to work with the cluster. For example, to add instances or check the cluster's status. If you want to retrieve a `Cluster` object again at a later date, for example after restarting MySQL Shell, use the `dba.getCluster([name],[options])` function. For example, using JavaScript:

```
mysql-js> var cluster1 = dba.getCluster()
```

Or using Python:

```
mysql-py> cluster1 = dba.get_cluster()
```

To retrieve the `ClusterSet` object representing an InnoDB ClusterSet deployment, use the `dba.getClusterSet()` or *`cluster`*`.getClusterSet()` function. For example, using JavaScript:

```
mysql-js> myclusterset = dba.getClusterSet()
```

Or using Python:

```
mysql-py> myclusterset = dba.get_cluster_set()
```

Note

When you use a `ClusterSet` object, the server instance from which you got it must still be online in the InnoDB ClusterSet. If that server instance goes offline, the object no longer works, and you need to get it again from a server that is still online in the InnoDB ClusterSet.

Use the `dba.getReplicaSet()` operation to retrieve a `ReplicaSet` object. For example, using JavaScript:

```
mysql-js> var replicaset1 = dba.getReplicaSet()
```

Or using Python:

```
mysql-py> replicaset1 = dba.get_replica_set()
```

If you do not specify a *`name`* then the default object is returned. The returned object uses a new session, independent from MySQL Shell's global session. This ensures that if you change the MySQL Shell global session, the `Cluster`, `ClusterSet`, or `ReplicaSet` object maintains its session to the server instance.

By default MySQL Shell attempts to connect to the primary instance when you retrieve a handler. Set the `connectToPrimary` option to configure this behavior.

- If `connectToPrimary` is `true` and the active global MySQL Shell session is not to a primary instance, MySQL Shell queries for the primary instance. If there is no quorum in a cluster, the operation fails.
- If `connectToPrimary` is `false`, the retrieved object uses the server instance specified for the active session, in other words the same instance as MySQL Shell's current global session.
- If `connectToPrimary` is not specified, MySQL Shell treats `connectToPrimary` as `true`, and falls back to `connectToPrimary` being `false`.

To force connecting to a secondary, establish a connection to the secondary instance and use the `connectToPrimary` option by issuing the following in JavaScript:

```
mysql-js> shell.connect(secondary_member)
mysql-js> var cluster1 = dba.getCluster(testCluster, {connectToPrimary:false})

WARNING: You are connected to an instance in state 'Read Only'
Write operations on the InnoDB cluster will not be allowed.

<Cluster:testCluster>
```

Or, by issuing the following in Python:

```
mysql-py> shell.connect(secondary_member)
mysql-py> cluster1 = dba.get_cluster(testCluster, connectToPrimary='false')

WARNING: You are connected to an instance in state 'Read Only'
```

```
Write operations on the InnoDB cluster will not be allowed.

<Cluster:testCluster>
```

Note

Secondary instances have `super_read_only=ON`, so you cannot write changes to them.

6.4 Creating User Accounts for AdminAPI

The user accounts used to configure and administer a member server instance in an InnoDB Cluster, InnoDB ClusterSet, or InnoDB ReplicaSet deployment must have full read and write privileges on the metadata tables, in addition to full MySQL administrator privileges (`SUPER`, `GRANT OPTION`, `CREATE`, `DROP` and so on). For more information, see Privileges Provided by MySQL.

You can use the `root` account on the servers for this purpose, but if you do this, the `root` account on every member server in the deployment must have the same password. Using the `root` account is not recommended for security reasons.

Instead, the recommended method is to set up user accounts using AdminAPI's JavaScript `dba.configureInstance()` and *`cluster.setupAdminAccount()`* operations. The format of the user names accepted by these operations follows the standard MySQL account name format, see Specifying Account Names.

If you prefer to set up the user accounts, the required permissions are listed in Configuring InnoDB Cluster Administrator Accounts Manually. If only read operations are needed, for example, for monitoring purposes, you can use an account with more restricted privileges, as detailed in this topic.

Important

Each account used to configure or administer an InnoDB Cluster, InnoDB ClusterSet, or InnoDB ReplicaSet deployment must exist on all the member server instances in the deployment, with the same user name, and the same password.

Server Configuration Account

A server configuration account is required on each server instance that is to join an InnoDB Cluster, InnoDB ClusterSet, or InnoDB ReplicaSet deployment. You set this account up using a `dba.configureInstance()` JavaScript command or `dba.configure_instance()` Python command, with the `clusterAdmin` option.

For better security, specify the password at the interactive prompt, otherwise specify it using the `clusterAdminPassword` option. Create the same account, with the same user name and password, in the same way on every server instance that will be part of the deployment, both the instance you connect to create the deployment and the instances that will join after that.

You can define a password expiration using the `clusterAdminPasswordExpiration` option. This option can be set to a number of days, `NEVER` to never expire, or `DEFAULT`, to use the system default.

If you are using SSL certificates for authentication, you can add the certificate issuer and subject using the `clusterAdminCertIssuer` and `clusterAdminCertSubject` options, respectively.

The server configuration account that you create using the `dba.configureInstance()` operation is *not* replicated to other servers in the InnoDB Cluster, InnoDB ClusterSet, or InnoDB ReplicaSet deployment. MySQL Shell disables binary logging for the `dba.configureInstance()` operation. For this reason, you must create the account on every server instance individually.

The `clusterAdmin` option must be used with a MySQL Shell connection based on a user which has the privileges to create users with suitable privileges. In this JavaScript example the root user is used:

```
mysql-js> dba.configureInstance('root@ic-1:3306', {clusterAdmin: "'icadmin'@'ic-1%'"});
```

Again, in this Python example the root user is used:

```
mysql-py> dba.configure_instance('root@ic-1:3306', clusterAdmin="'icadmin'@'ic-1%'");
```

Administrator Accounts

Administrator accounts can be used to administer a deployment after you have completed the configuration process. You can set up more than one of them. To create an administrator account, you issue a *`cluster`*`.setupAdminAccount()` JavaScript command after you have added all the instances to the InnoDB Cluster or InnoDB ReplicaSet. Or issue the Python command: `<Cluster>setup_admin_account()`.

The command creates an account with the user name and password that you specify, with all the required permissions. A transaction to create an account with *`cluster`*`.setupAdminAccount()` is written to the binary log and sent to all the other server instances in the cluster to create the account on them.

To use the `setupAdminAccount()` operation, you must be connected as a MySQL user with privileges to create users, for example as root. The `setupAdminAccount(`*`user`*`)` operation also enables you to upgrade an existing MySQL account with the necessary privileges before a `dba.upgradeMetadata()` JavaScript operation, or the `dba.upgrade_metadata()` Python operation.

The mandatory *`user`* argument is the name of the MySQL account you want to create to be used to administer the deployment. The format of the user names accepted by the `setupAdminAccount()` operation follows the standard MySQL account name format. For more information, see Specifying Account Names. The user argument format is *`username`*`[@`*`host`*`]` where *`host`* is optional and if it is not provided it defaults to the `%` wildcard character.

For example, to create a user named *`icadmin`* to administer an InnoDB Cluster assigned to the variable *`myCluster`* using JavaScript, issue:

```
mysql-js> myCluster.setupAdminAccount('icadmin')

Missing the password for new account icadmin@%. Please provide one.
Password for new account: ********
Confirm password: ********

Creating user icadmin@%.
Setting user password.
Account icadmin@% was successfully created.
```

Or using Python:

```
mysql-py> myCluster.setup_admin_account('icadmin')

Missing the password for new account icadmin@%. Please provide one.
Password for new account: ********
Confirm password: ********

Creating user icadmin@%.
Setting user password.
Account icadmin@% was successfully created.
```

As of MySQL Shell 8.0.33, the following options were added to `setupAdminAccount()`:

- `requireCertIssuer`: Optional SSL certificate issuer for the account.
- `requireCertSubject`: Optional SSL certificate subject for the account.
- `passwordExpiration: numberOfDays | Never | Default`: Password expiration setting for the account.

Note

If either `requireCertIssuer` or `requireCertSubject` are set, or both, the existing password becomes optional.

Updating Old Accounts

If you have a server configuration account or administrator account created with a version prior to MySQL Shell 8.0.20, use the `update` option with the `setupAdminAccount()` operation to upgrade the privileges of the existing user. This is relevant during an upgrade, to ensure that the user accounts are compatible. For example, to upgrade the user named `icadmin`, using JavaScript, issue:

```
mysql-js> myCluster.setupAdminAccount('icadmin', {'update':1})
Updating user icadmin@%.
Account icadmin@% was successfully updated.
```

Or using Python:

```
mysql-py> myCluster.setup_admin_account('icadmin',update=1})
Updating user icadmin@%.
Account icadmin@% was successfully updated.
```

This is a special use of the *`cluster`*`.setupAdminAccount()` command that is not written to the binary log.

6.5 Verbose Logging

When working with a production deployment it can be useful to configure verbose logging for MySQL Shell. For example, the information in the log can help you to find and resolve any issues that might occur when you are preparing server instances to work as part of InnoDB Cluster. To start MySQL Shell with a verbose logging level, use the `--log-level` option:

```
$> mysqlsh --log-level=DEBUG3
```

The `DEBUG3` level is recommended. For more information, see `--log-level`. When `DEBUG3` is set the MySQL Shell log file contains lines such as `Debug: execute_sql( ... )` which contain the SQL queries that are executed as part of each AdminAPI call. The log file generated by MySQL Shell is located in `~/.mysqlsh/mysqlsh.log` for Unix-based systems; on Microsoft Windows systems it is located in `%APPDATA%\MySQL\mysqlsh\mysqlsh.log`. For more information, see Chapter 12, *MySQL Shell Logging and Debug*.

In addition to enabling the MySQL Shell log level, you can configure the amount of output AdminAPI provides in MySQL Shell after issuing each command. To enable the amount of AdminAPI output, in MySQL Shell issue:

```
mysql-js> dba.verbose=2
```

This enables the maximum output from AdminAPI calls. The available levels of output are:

- 0 or OFF is the default. This provides minimal output and is the recommended level when not troubleshooting.
- 1 or ON adds verbose output from each call to the AdminAPI.
- 2 adds debug output to the verbose output providing full information about what each call to AdminAPI executes.

MySQL Shell can optionally log the SQL statements used by AdminAPI operations (with the exception of sandbox operations), and can also display them in the terminal as they are executed. For more information, see Section 12.5, “Logging AdminAPI Operations”.

6.6 Finding the Primary

When you are working with a single-primary InnoDB Cluster or an InnoDB ReplicaSet, you need to connect to the primary instance for administration tasks so that configuration changes are written to the metadata. To find the current primary you can:

- Use the `--redirect-primary` option at MySQL Shell start up to ensure that the target server is part of an InnoDB Cluster or InnoDB ReplicaSet. If the target instance is not the primary, MySQL Shell finds the primary and connects to it.
- Use the `shell.connectToPrimary([instance, password])` operation (added in version 8.0.20), which checks whether the target instance belongs to a cluster or ReplicaSet. If so, MySQL Shell opens a new session to the primary, sets the active global MySQL Shell session to the established session and returns it.

  If an *`instance`* is not provided, the operation attempts to use the active global MySQL Shell session. If an *`instance`* is not provided and there is no active global MySQL Shell session, an exception is thrown. If the target instance does not belong to a cluster or ReplicaSet the operation fails with an error.
- Use the status operation, find the primary in the result, and manually connect to that instance.

6.7 Scripting AdminAPI

In addition to the interactive mode illustrated in this section, MySQL Shell supports running scripts in batch mode. This enables you to automate processes using AdminAPI with scripts written in JavaScript or Python, which can be run using MySQL Shell's `--file` option. For example:

```
$> mysqlsh --file setup-innodb-cluster.js
```

Note

Any command line options specified after the script file name are passed to the script and *not* to MySQL Shell. You can access those options using the `os.argv` array in JavaScript, or the `sys.argv` array in Python. In both cases, the first option picked up in the array is the script name.

The contents of an example script files are shown here, using JavaScript:

```
print('InnoDB Cluster sandbox set up\n');
print('=================================\n');
print('Setting up a MySQL InnoDB Cluster with 3 MySQL Server sandbox instances,\n');
print('installed in ~/mysql-sandboxes, running on ports 3310, 3320 and 3330.\n\n');

var dbPass = shell.prompt('Please enter a password for the MySQL root account: ', {type:"password"});

try {
   print('\nDeploying the sandbox instances.');
   dba.deploySandboxInstance(3310, {password: dbPass});
   print('.');
   dba.deploySandboxInstance(3320, {password: dbPass});
   print('.');
   dba.deploySandboxInstance(3330, {password: dbPass});
   print('.\nSandbox instances deployed successfully.\n\n');

   print('Setting up InnoDB Cluster...\n');
   shell.connect('root@localhost:3310', dbPass);

   var cluster = dba.createCluster("prodCluster");

   print('Adding instances to the Cluster.');
   cluster.addInstance({user: "root", host: "localhost", port: 3320, password: dbPass});
   print('.');
   cluster.addInstance({user: "root", host: "localhost", port: 3330, password: dbPass});
   print('.\nInstances successfully added to the Cluster.');
```

```
   print('\nInnoDB Cluster deployed successfully.\n');
} catch(e) {
   print('\nThe InnoDB Cluster could not be created.\n\nError: ' +
   + e.message + '\n');
}
```

Or using Python:

```
print('InnoDB Cluster sandbox set up\n');
print('================================\n');
print('Setting up a MySQL InnoDB Cluster with 3 MySQL Server sandbox instances,\n');
print('installed in ~/mysql-sandboxes, running on ports 3310, 3320 and 3330.\n\n');

dbPass = shell.prompt('Please enter a password for the MySQL root account: ', type ="password");

try:
       print('\nDeploying the sandbox instances.');
       dba.deploy_sandbox_instance(3310, password = dbPass);
       print('.');
       dba.deploy_sandbox_instance(3320, password = dbPass);
       print('.');
       dba.deploy_sandbox_instance(3330, password = dbPass);
       print('.\nSandbox instances deployed successfully.\n\n');

       print('Setting up InnoDB Cluster...\n');
       shell.connect('root@localhost:3310', dbPass);

       cluster = dba.create_cluster("prodCluster");

       print('Adding instances to the Cluster.');
       cluster.add_instance('root@localhost:3320', password = dbPass);
       print('.');
       cluster.add_instance('root@localhost:3330', password = dbPass);
       print('.\nInstances successfully added to the Cluster.');

       print('\nInnoDB Cluster deployed successfully.\n');

except ValueError:
       print('\nThe InnoDB Cluster could not be created.\n\nError.\n');
```

AdminAPI MySQL Shell Command Line Integration

AdminAPI is also supported by MySQL Shell's Section 5.8, “API Command Line Integration”. This command line integration enables you to easily integrate AdminAPI into your environment. For example, to check the status of an InnoDB Cluster using the sandbox instance listening on port 1234:

```
$ mysqlsh root@localhost:1234 -- cluster status
```

This maps to the equivalent command in MySQL Shell:

```
mysql-js> cluster.status()
```

6.8 AdminAPI MySQL Sandboxes

This section explains how to set up a sandbox deployment with AdminAPI. Deploying and using local sandbox instances of MySQL is a good way to start your exploration of AdminAPI. You can test the functionality locally before deploying on your production servers. AdminAPI has built-in functionality for creating sandbox instances that are correctly configured to work with InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet in a locally deployed scenario.

Important

Sandbox instances are only suitable for deploying and running on your local machine for testing purposes. In a production environment, the MySQL Server instances are deployed to various host machines on the network. For more information, see Section 7.4, “Deploying a Production InnoDB Cluster”.

Unlike a production deployment, where you work with instances and specify them by a connection string, sandbox instances run locally on the same machine as that you are running MySQL Shell. To select a sandbox instance, you supply the port number on which the MySQL sandbox instance is listening.

6.8.1 Deploying Sandbox Instances

Rather than using a production setup, where each instance runs on a separate host, AdminAPI provides the `dba.deploySandboxInstance(port_number)` operation. The *`port_number`* argument is the TCP port number where the MySQL Server instance listens for connections. To deploy a new sandbox instance which is bound to port 3310, issue:

```
mysql-js> dba.deploySandboxInstance(3310)
```

By default the sandbox is created in a directory named `$HOME/mysql-sandboxes/port` on Unix systems. For Microsoft Windows systems the directory is `%userprofile%\MySQL\mysql-sandboxes\port`. Each sandbox instance is stored in a directory named after the *`port_number`*.

You are prompted for the root user's password.

Important

Each sandbox instance uses the root user and password, and it must be the same on all sandbox instances which should work together. This is not recommended in production.

To deploy another sandbox server instance, repeat the steps followed for the sandbox instance at port 3310, choosing different port numbers for each instance.

To change the directory which sandboxes are stored in, for example to run multiple sandboxes on one host for testing purposes, use the MySQL Shell `sandboxDir` option. For example, to use a sandbox in the `/home/user/sandbox1` directory, issue:

```
mysql-js> shell.options.sandboxDir='/home/user/sandbox1'
```

All subsequent sandbox related operations are then executed against the instances found at `/home/user/sandbox1`.

When you deploy sandboxes, MySQL Shell searches for the `mysqld` binary, which it then uses to create the sandbox instance. You can configure where MySQL Shell searches for the `mysqld` binary by configuring the `PATH` environment variable. This can be useful to test a new version of MySQL locally before deploying it to production. For example, to use a `mysqld` binary at the path `/home/user/mysql-latest/bin/` issue:

```
PATH=/home/user/mysql-latest/bin/:$PATH
```

Then run MySQL Shell from the terminal where the `PATH` environment variable is set. Any sandboxes you deploy, then use the `mysqld` binary found at the configured path.

The following options are supported by the `dba.deploySandboxInstance()` operation:

- `allowRootFrom` configures which host the root user can connect from. Defaults to `root@%`.
- `ignoreSslError` configures secure connections on the sandbox instance. When `ignoreSslError` is true, which is the default, no error is issued during the operation if SSL support cannot be provided and the server instance is deployed without SSL support. When `ignoreSslError` is set to `false`, the sandbox instance is deployed with SSL support, issuing an error if SSL support cannot be configured.
- `mysqldOptions` configures additional options on the sandbox instance. Defaults to an empty string, and accepts a list of strings that specify options and values. For example `mysqldOptions: ["lower_case_table_names=1", "report_host="10.1.2.3"]}`. The specified options are written to the sandbox instance's option file.

- `portX` configures the port used for X Protocol connections. The default is calculated by multiplying the `port` value by 10. The value is an integer between 1024 and 65535.

6.8.2 Managing Sandbox Instances

Once a sandbox instance is running, it is possible to change its status at any time using the following commands. Specify the port number for the instance to identify it:

- To stop a sandbox instance using JavaScript, issue `dba.stopSandboxInstance(instance)`. This stops the instance gracefully, unlike `dba.killSandboxInstance(instance)`.
- To stop a sandbox instance using Python, issue: `dba.stop_sandbox_instance(instance)`. This stops the instance gracefully, unlike `dba.kill_sandbox_instance(instance)`.
- To start a sandbox instance using JavaScript, issue: `dba.startSandboxInstance(instance)`.
- To start a sandbox instance using Python, issue: `dba.start_sandbox_instance(instance)`.
- To kill a sandbox instance using JavaScript, issue: `dba.killSandboxInstance(instance)`. This stops the instance without gracefully stopping it and is useful in simulating unexpected halts.
- To kill a sandbox instance using Python, issue: `dba.kill_sandbox_instance(instance)`. This stops the instance without gracefully stopping it and is useful in simulating unexpected halts.
- To delete a sandbox instance using JavaScript, issue: `dba.deleteSandboxInstance(instance)`. This completely removes the sandbox instance from your file system.
- To delete a sandbox instance using Python, issue: `dba.delete_SandboxInstance(instance)`. This completely removes the sandbox instance from your file system.

Sandbox instances are considered to be transient and are not designed for production use. They are therefore not supported for version upgrades. In a sandbox deployment, each sandbox instance uses a copy of the `mysqld` binary found in the `$PATH` in the local `mysql-sandboxes` directory. If the version of `mysqld` changes, for example after an upgrade, sandboxes based on the previous version fail to start. This is because the sandbox binary is outdated compared to the dependencies found under the `basedir`.

If you do want to retain a sandbox instance after an upgrade, a workaround is to manually copy the upgraded `mysqld` binary into the `bin` directory of each sandbox. Then start the sandbox by issuing `dba.startSandboxInstance()`. The operation fails with a timeout, and the error log contains:

```
2020-03-26T11:43:12.969131Z 5 [System] [MY-013381] [Server] Server upgrade
from '80019' to '80020' started.
2020-03-26T11:44:03.543082Z 5 [System] [MY-013381] [Server] Server upgrade
from '80019' to '80020' completed.
```

Although the operation seems to fail with a timeout, the sandbox has started successfully.

6.8.3 Setting up InnoDB Cluster and MySQL Router

In the following example, we complete the following tasks using a sandbox deployment with AdminAPI to deploy an InnoDB Cluster with MySQL Router.

Deploying and using local sandbox instances of MySQL allows you to test out the functionality locally, before deployment on your production servers. AdminAPI has built-in functionality for creating sandbox instances that are pre-configured to work with InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet in a locally deployed scenario.

This example contains the following sections:

- Installation
- Creating InnoDB Cluster

- Bootstrapping MySQL Router
- Test MySQL Router Configuration

Warning

Sandbox instances are only suitable for deploying and running on your local machine for testing purposes.

Installation

Install the following components:

- MySQL Server: For more information, see Installing MySQL.
- MySQL Shell: For more information, see Installing MySQL Shell.
- MySQL Router: For more information, see Installing MySQL Router.

Creating an InnoDB Cluster Sandbox Configuration

To provide tolerance for one failure, create an InnoDB Cluster with three instances. In this example, we will be using three sandbox instances running on the same machine. In a real-world setup, those three instances would be running on different hosts on the network.

1. To start MySQL Shell, issue:

```
> mysqlsh
```

2. To create and start the MySQL sandbox instances, use the `dba.deploySandboxInstance()` function that is part of the X AdminAPI. Issue the following three statements in the MySQL Shell and enter a root password for each instance:

```
mysql-js> dba.deploySandboxInstance(3310)
mysql-js> dba.deploySandboxInstance(3320)
mysql-js> dba.deploySandboxInstance(3330)
```

Note

Use the same root password for all instances.

Creating InnoDB Cluster

To create an InnoDB Cluster, complete the following steps:

1. Connect to the MySQL instance you want to be the primary instance in the InnoDB Cluster by issuing:

```
mysql-js> shell.connect('root@localhost:3310')
```

2. Issue the `dba.createCluster()` command to create the Cluster, and use the assigned variable `cluster` to hold the outputted value:

```
mysql-js> cluster = dba.createCluster('devCluster')
```

This command outputs:

```
A new InnoDB cluster will be created on instance 'localhost:3310'.

Validating instance configuration at localhost:3310...
NOTE: Instance detected as a sandbox.
Please note that sandbox instances are only suitable for deploying test clusters for use within the

This instance reports its own address as 127.0.0.1:3310

Instance configuration is suitable.
```

```
NOTE: Group Replication will communicate with other members using '127.0.0.1:33101'.
      Use the localAddress option to override.

Creating InnoDB cluster 'devCluster' on '127.0.0.1:3310'...

Adding Seed Instance...
Cluster successfully created. Use Cluster.addInstance() to add MySQL instances.
At least 3 instances are needed for the cluster to be able to withstand up to
one server failure.

<Cluster:devCluster>
```

3. Verify that the creation was successful by using the `cluster.status()` function with the assigned variable `cluster`:

```
mysql-js> cluster.status()
```

   The following status is output:

```
 {
“clusterName”: “devCluster”,
“defaultReplicaSet”: {
        "name": "default",
"primary": "127.0.0.1:3310",
"ssl": "REQUIRED",
"status": "OK_NO_TOLERANCE",
"statusText": "Cluster is NOT tolerant to any failures.",
"topology": {
   "127.0.0.1:3310": {
       "address": "127.0.0.1:3310",
       "memberRole": "PRIMARY",
       "mode": "R/W",
       "readReplicas": {},
       "replicationLag": null,
       "role": "HA",
       "status": "ONLINE",
       "version": "8.0.28"
   }
},
"topologyMode": "Single-Primary"
}, “groupInformationSourceMember”:
“127.0.0.1:3310” }
```

4. The Cluster is up and running but not yet tolerant to a failure. Add another MySQL Server instances to the Cluster using the `<Cluster>.addInstance()` function:

```
{
mysql-js> cluster.addInstance('root@localhost:3320')
NOTE: The target instance '127.0.0.1:3320' has not been pre-provisioned (GTID set is empty).
      The Shell is unable to decide whether incremental state recovery can correctly provision it.
The safest and most convenient way to provision a new instance is through automatic clone provisioning,
which will completely overwrite the state of '127.0.0.1:3320' with a physical snapshot from an existing
cluster member. To use this method by default, set the 'recoveryMethod' option to 'clone'.

The incremental state recovery may be safely used if you are sure all updates ever executed in the
cluster were done with GTIDs enabled, there are no purged transactions and the new instance contains
the same GTID set as the cluster or a subset of it. To use this method by default, set the
'recoveryMethod' option to 'incremental'.

Please select a recovery method [C]lone/[I]ncremental recovery/[A]bort (default Clone):
mysql-js> cluster.addInstance('root@localhost:3330')
}
```

5. Select a recovery method from the prompt. The options are:

   - **Clone**: Clones the instance that you are adding to the primary Cluster, deleting any transactions the instance contains. The MySQL Clone plugin is automatically installed. Assuming you are adding either an empty instance (has not processed any transactions) or an instance that contains transactions you prefer not to retain, select the Clone option.

- **Incremental recovery**: Recovers all transactions processed by the Cluster to the joining instance using asynchronous replication. Incremental recovery is appropriate if you are sure all updates ever processed by the Cluster were completed with global transaction IDs (`GTID`) enabled. There are no purged transactions, and the new instance contains the same `GTID` set as the Cluster or a subset of it.

In this example, select **C** for *Clone*:

```
     Please select a recovery method [C]lone/[I]ncremental recovery/[A]bort (default Clone): C
Validating instance configuration at localhost:3320...
NOTE: Instance detected as a sandbox.
Please note that sandbox instances are only suitable for deploying test clusters for
use within the same host.

This instance reports its own address as 127.0.0.1:3320

Instance configuration is suitable.
NOTE: Group Replication will communicate with other members using '127.0.0.1:33201'.
      Use the localAddress option to override.

A new instance will be added to the InnoDB cluster. Depending on the amount of
data on the cluster this might take from a few seconds to several hours.

Adding instance to the cluster...

Monitoring recovery process of the new cluster member. Press ^C to stop monitoring
and let it continue in background.
Clone based state recovery is now in progress.

NOTE: A server restart is expected to happen as part of the clone process. If the
server does not support the RESTART command or does not come back after a
while, you may need to manually start it back.

* Waiting for clone to finish...
NOTE: 127.0.0.1:3320 is being cloned from 127.0.0.1:3310
** Stage DROP DATA: Completed
** Clone Transfer
FILE COPY  ############################################################  100%  Completed
PAGE COPY  ############################################################  100%  Completed
REDO COPY  ############################################################  100%  Completed

NOTE: 127.0.0.1:3320 is shutting down...

* Waiting for server restart... ready
* 127.0.0.1:3320 has restarted, waiting for clone to finish...
** Stage RESTART: Completed
* Clone process has finished: 72.61 MB transferred in about 1 second (~72.61 MB/s)

State recovery already finished for '127.0.0.1:3320'

The instance '127.0.0.1:3320' was successfully added to the cluster.
```

6. Add the third instance created and again select **C** for the Clone recovery method:

```
mysql-js> cluster.addInstance('root@localhost:3330')
```

7. Check the status of the Cluster, by issuing:

```
mysql-js> cluster.status()
```

This outputs the following:

```
{
"clusterName": "devCluster",
"defaultReplicaSet": {
   "name": "default",
   "primary": "127.0.0.1:3310",
   "ssl": "REQUIRED",
   "status": "OK",
```

```
   "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
       "topology": {
           "127.0.0.1:3310": {
               "address": "127.0.0.1:3310",
               "memberRole": "PRIMARY",
               "mode": "R/W",
               "readReplicas": {},
               "replicationLag": null,
               "role": "HA",
               "status": "ONLINE",
               "version": "8.0.28"
           },
           "127.0.0.1:3320": {
               "address": "127.0.0.1:3320",
               "memberRole": "SECONDARY",
               "mode": "R/O",
               "readReplicas": {},
               "replicationLag": null,
               "role": "HA",
               "status": "ONLINE",
               "version": "8.0.28"
           },
           "127.0.0.1:3330": {
               "address": "127.0.0.1:3330",
               "memberRole": "SECONDARY",
               "mode": "R/O",
               "readReplicas": {},
               "replicationLag": null,
               "role": "HA",
               "status": "ONLINE",
               "version": "8.0.28"
           }
       },
       "topologyMode": "Single-Primary"
   },
   "groupInformationSourceMember": "127.0.0.1:3310"
}
The setup of the InnoDB Cluster was successful!
```

8. The Cluster can now tolerate up to one failure. Quit MySQL Shell by issuing:`\q`

Bootstrapping MySQL Router

After MySQL InnoDB Cluster is set up, test the high availability of the Cluster. For this purpose, use MySQL Router. If one instance fails, the MySQL Router updates its routing configuration automatically and ensures that new connections are routed to the remaining instances.

Before MySQL Router can perform the routing operations, make it aware of the new InnoDB Cluster. To do this, use the `–bootstrap` option and point MySQL Router to the current `R/W` MySQL Server instance (primary instance) of the Cluster. Store the Router's configuration in a folder called `mysql-router` using the `-d` option.

1. Open a terminal in your home directory:

   - On a Linux system, issue:

     ```
     [demo-user@losthost]$> mysqlrouter --bootstrap root@localhost:3310 -d mysqlrouter
     ```

   - On a Windows system, issue:

     ```
     C:\Users\demo-user> mysqlrouter --bootstrap root@localhost:3310 -d mysql-router
     ```

     MySQL Router then prints the TCP/IP ports that it will use for routing connections. For more information, see Deploying MySQL Router.

2. When MySQL Router has been successfully configured, start it up in a background thread:

- On a Windows system use the `start /B` command and point the Router to the configuration file that was generated by using the `–bootstrap` option:

```
C:\> start /B mysqlrouter -c %HOMEPATH%\mysql-router\mysqlrouter.conf
```

- Or call the `Windows PowerShell` script in the `mysqlrouter` folder, created previously:

```
\mysqlrouter\start.ps1
```

- On a Linux system using *systemd*, issue:

```
sudo systemctl start mysqlrouter.service
```

- Or on a Linux system, call the `Shell` script in the `mysqlrouter` folder, created previously:

```
/mysqlrouter/start.sh
```

Test MySQL Router Configuration

Now that an InnoDB Cluster and MySQL Router are running, test the Cluster setup.

Instead of connecting to one of the MySQL Server instances directly, connect through the MySQL Router.

1. Issue the following connection command:

```
> mysqlsh root@localhost:6446
```

2. Provide the root password to connect to the InnoDB Cluster.

3. Check the status of the InnoDB Cluster by creating a variable `cluster` and assigning it with the value of the `dba.getCluster()` operation:

```
mysql-js> cluster = dba.getCluster()
```

```
mysql-js> cluster.status()
```

4. Switch to SQL mode:

```
mysql-js> \sql
```

5. Query the port the instance is running on, by issuing:

```
mysql-sql> SELECT @@port;

        +--------+
        | @@port |
        +--------+
        |   3310 |
        +--------+
1 row in set (0.0007 sec)
```

6. Switch back to the JavaScript mode:

```
mysql-js> \js
```

7. Use the `dba.killSandboxInstance()` function to *stop* the MySQL Server instance:

```
            dba.killSandboxInstance(3310)

Killing MySQL instance...

Instance localhost:3310 successfully killed.
```

8. Check if MySQL Router is correctly routing traffic by running `SELECT @@port` command against the instance that was just killed and check the result:

- Switch to SQL mode:

```
mysql-js> \sql
```

- Check the port of MySQL:

```
mysql-sql> SELECT @@port;
```

9. An error is returned; `ERROR: 2013 (HY000): Lost connection to MySQL server during query`. This error means that the instance running on port 3310 is no longer running.

10. Check the port again:

```
mysql-sql> SELECT @@port;
+--------+
| @@port |
+--------+
|   3320 |
+--------+
```

11. This output shows that the instance running on port 3320 was promoted to be the new `Read/Write` primary instance.

12. Return to the JavaScript mode, and check the status of the Cluster:

```
mysql-js> cluster.status()
{
   "clusterName": "devCluster",
   "defaultReplicaSet": {
       "name": "default",
       "primary": "127.0.0.1:3320",
       "ssl": "REQUIRED",
       "status": "OK_NO_TOLERANCE",
       "statusText": "Cluster is NOT tolerant to any failures. 1 member is not active.",
       "topology": {
           "127.0.0.1:3310": {
               "address": "127.0.0.1:3310",
               "memberRole": "SECONDARY",
               "mode": "n/a",
               "readReplicas": {},
               "role": "HA",
               "shellConnectError": "MySQL Error 2003: Could not open connection to '127.0.0.1:3310':
                                     Can't connect to MySQL server on '127.0.0.1:3310' (10061)",
               "status": "(MISSING)"
           },
           "127.0.0.1:3320": {
               "address": "127.0.0.1:3320",
               "memberRole": "PRIMARY",
               "mode": "R/W",
               "readReplicas": {},
               "replicationLag": null,
               "role": "HA",
               "status": "ONLINE",
               "version": "8.0.28"
           },
           "127.0.0.1:3330": {
               "address": "127.0.0.1:3330",
               "memberRole": "SECONDARY",
               "mode": "R/O",
               "readReplicas": {},
               "replicationLag": null,
               "role": "HA",
               "status": "ONLINE",
               "version": "8.0.28"
           }
       },
       "topologyMode": "Single-Primary"
   },
   "groupInformationSourceMember": "127.0.0.1:3320"
}
```

13. The MySQL Server instance formally running on port 3310 is `MISSING`.

14. Restart this instance, by issuing the `dba.startSandboxInstance()` operation with the port number:

```
mysql-js> dba.startSandboxInstance(3310)
```

15. Checking the status of the Cluster shows that the instance has been restored as active in the Cluster, but as a `SECONDARY` member:

```
mysql-js > cluster.status()
{
  "clusterName": "devCluster",
  "defaultReplicaSet": {
      "name": "default",
      "primary": "127.0.0.1:3320",
      "ssl": "REQUIRED",
      "status": "OK",
      "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
      "topology": {
          "127.0.0.1:3310": {
              "address": "127.0.0.1:3310",
              "memberRole": "SECONDARY",
              "mode": "R/O",
              "readReplicas": {},
              "replicationLag": null,
              "role": "HA",
              "status": "ONLINE",
              "version": "8.0.28"
          },
          "127.0.0.1:3320": {
              "address": "127.0.0.1:3320",
              "memberRole": "PRIMARY",
              "mode": "R/W",
              "readReplicas": {},
              "replicationLag": null,
              "role": "HA",
              "status": "ONLINE",
              "version": "8.0.28"
          },
          "127.0.0.1:3330": {
              "address": "127.0.0.1:3330",
              "memberRole": "SECONDARY",
              "mode": "R/O",
              "readReplicas": {},
              "replicationLag": null,
              "role": "HA",
              "status": "ONLINE",
              "version": "8.0.28"
          }
      },
      "topologyMode": "Single-Primary"
  },
  "groupInformationSourceMember": "127.0.0.1:3320"
}
```

16. All instances are back online, and the Cluster can tolerate one failure again.

6.9 Tagging Metadata

From version 8.0.21, a configurable tag framework is available to allow the metadata of InnoDB Cluster, InnoDB ClusterSet, or InnoDB ReplicaSet to be marked with additional information. Tags make it possible to associate custom key-value pairs to a Cluster, ReplicaSet, or instance. Tags have been reserved for use by MySQL Router that enables a compatible MySQL Router to support hiding instances from applications. The following tags are reserved for this purpose:

- `_hidden` instructs MySQL Router: Excludes the instance from the list of possible destinations for client applications.

- `_disconnect_existing_sessions_when_hidden`: Instructs the router to disconnect existing connections from instances that are marked to be hidden.

For more information, see Removing Instances from Routing.

In addition, the tags framework is user-configurable. Custom tags can consist of any ASCII character and provide a `namespace`, which serves as a dictionary key-value pairs that can be associated with Clusters, ReplicaSets, or their specific instances. Tag values can be any JSON value. This configuration enables you to add your own attributes on top of the metadata.

Showing Tags

The *`Cluster`*`.options()` operation shows information about the tags assigned to individual cluster instances as well as to the cluster itself. For example, the InnoDB Cluster assigned to `myCluster` could show:

```
mysql-js> myCluster.options()
{
    "cluster": {
        "name": "test1",
        "tags": {
            "ic-1:3306": [
                {
                    "option": "_disconnect_existing_sessions_when_hidden",
                    "value": true
                },
                {
                    "option": "_hidden",
                    "value": false
                }
            ],
            "ic-2:3306": [],
            "ic-3:3306": [],
            "global": [
                {
                    "option": "location:",
                    "value": "US East"
                }
            ]
        }
    }
}
```

This cluster has a global tag named `location` which has the value `US East`, and instance `ic-1` has been tagged.

Setting Tags on a Cluster Instance

You can set tags at the instance level, which enables you for example to mark an instance as not available, so that applications and router treat it as offline. Use the *`Cluster`*`.setInstanceOption(`*`instance, option, value`*`)` operation to set the value of a tag for the instance.

The *`instance`* argument is a connection string to the target instance. The *`option`* argument must be a string with the format *`namespace`*`:`*`option`*. The *`value`* parameter is the value that should be assigned to `option` in the specified `namespace`. If the value is `null`, the `option` is removed from the specified `namespace`. For instances which belong to a cluster, the `setInstanceOption()` operation only accepts the `tag` namespace. Any other namespace results in an `ArgumentError`.

For example, to use JavaScript to set the tag *`test`* to `true` on the *`myCluster`* instance `ic-1`, issue:

```
mysql-js> myCluster.setInstanceOption("icadmin@ic-1:3306", "tag:test", true);
```

Or using Python to set the tag *`test`* to `true` on the *`myCluster`* instance `ic-1`, issue:

```
mysql-py> myCluster.set_instance_option("icadmin@ic-1:3306", "tag:test", True);
```

Removing Instances from Routing

When AdminAPI and MySQL Router are working together, they support specific tags that enable you to mark instances as hidden and remove them from routing. MySQL Router then excludes such tagged instances from the routing destination candidates list. This functionality enables you to safely take a server instance offline so that applications and MySQL Router ignore it. For example, while you perform maintenance tasks, such as a server upgrade or configuration changes.

When the `_hidden` tag is set to true, this instructs MySQL Router to exclude the instance from the list of possible destinations for client applications. The instance remains online, but is not routed to for new incoming connections. The `_disconnect_existing_sessions_when_hidden` tag controls how existing connections to the instance are closed. This tag is assumed to be true, and it instructs any MySQL Router instances bootstrapped against the InnoDB Cluster, InnoDB ClusterSet, or InnoDB ReplicaSet to disconnect any existing connections from the instance when the `_hidden` tag is true.

When `_disconnect_existing_sessions_when_hidden` is false, any existing client connections to the instance are not closed if `_hidden` is true. The reserved `_hidden` and `_disconnect_existing_sessions_when_hidden` tags are specific to instances and cannot be used at the cluster level.

Warning

When the `use_gr_notifications` MySQL Router option is enabled, it defaults to 60 seconds. This means that when you set tags, it takes up to 60 seconds for MySQL Router to detect the change. To reduce the waiting time, change `use_gr_notifications` to a lower value.

For example, suppose you want to remove the `ic-1` instance which is part of an InnoDB Cluster assigned to `myCluster` from the routing destinations. Use the `setInstanceOption()` JavaScript operation to enable the `_hidden` and `_disconnect_existing_sessions_when_hidden` tags:

```
mysql-js> myCluster.setInstanceOption("icadmin@ic-1:3306", "tag:_hidden", true);
```

Or use the `set_instance_option()` Python operation to enable the `_hidden` and `_disconnect_existing_sessions_when_hidden` tags:

```
   mysql-js> myCluster.set_instance_option("icadmin@ic-1:3306", "tag:_hidden", true);
```

You can verify the change in the metadata by checking the options. For example the change made to `ic-1` would show in the options as:

```
mysql-js> myCluster.options()
{
    "cluster": {
        "name": "test1",
        "tags": {
            "ic-1:3306": [
                {
                    "option": "_disconnect_existing_sessions_when_hidden",
                    "value": true
                },
                {
                    "option": "_hidden",
                    "value": true
                }
            ],
            "ic-2:3306": [],
            "ic-3:3306": [],
            "global": []
        }
    }
```

```
}
```

You can verify that MySQL Router has detected the change in the metadata by viewing the log file. A MySQL Router that has detected the change made to `ic-1` would show a change such as:

```
2020-07-03 16:32:16 metadata_cache INFO [7fa9d164c700] Potential changes detected in cluster 'testCluster'
2020-07-03 16:32:16 metadata_cache INFO [7fa9d164c700] view_id = 4, (3 members)
2020-07-03 16:32:16 metadata_cache INFO [7fa9d164c700]     ic-1:3306 / 33060 - mode=RW
2020-07-03 16:32:16 metadata_cache INFO [7fa9d164c700]     ic-1:3306 / 33060 - mode=RO
2020-07-03 16:32:16 metadata_cache INFO [7fa9d164c700]     ic-1:3306 / 33060 - mode=RO hidden=yes disconnec
2020-07-03 16:32:16 routing INFO [7fa9d164c700] Routing routing:testCluster_x_ro listening on 64470 got rec
2020-07-03 16:32:16 routing INFO [7fa9d164c700] Routing routing:testCluster_x_rw listening on 64460 got rec
2020-07-03 16:32:16 routing INFO [7fa9d164c700] Routing routing:testCluster_rw listening on 6446 got reques
2020-07-03 16:32:16 routing INFO [7fa9d164c700] Routing routing:testCluster_ro listening on 6447 got reques
```

To bring the instance back online, use the `setInstanceOption()` operation to remove the tags, and MySQL Router automatically adds the instance back to the routing destinations, and it becomes online for applications. For example:

```
mysql-js> myCluster.setInstanceOption(icadmin@ic-1:3306, "tag:_hidden", false);
```

Verify the change in the metadata by checking the options again:

```
mysql-js> myCluster.options()
{
    "cluster": {
        "name": "test1",
        "tags": {
            "ic-1:3306": [
                {
                    "option": "_disconnect_existing_sessions_when_hidden",
                    "value": true
                },
                {
                    "option": "_hidden",
                    "value": false
                }
            ],
            "ic-2:3306": [],
            "ic-3:3306": [],
            "global": []
        }
    }
}
```

As of 8.0.32, you can also view the `_hidden` status on members using the attribute `hiddenFromRouter` in the output of the following commands:

- *`Cluster`*`.status()`
- *`Cluster`*`.describe()`
- *`ReplicaSet`*`.status()`

This attribute is `true` for any member hidden from MySQL Router traffic using the `_hidden` metadata tag.

Setting Tags on a Cluster

The *`Cluster`*`.setOption(`*`option, value`*`)` operation enables you to change the value of a namespace option for the whole cluster. The *`option`* argument must be a string with the format *`namespace`*`:`*`option`*.

The *`value`* parameter is the value to be assigned to `option` in the specified `namespace`. If the value is `null`, the `option` is removed from the specified `namespace`. For Clusters, the `setOption()` operation accepts the `tag` namespace. Any other namespace results in an `ArgumentError`.

Tip

Tags set at the cluster level do not override tags set at the instance level. You cannot use `Cluster.setOption()` to remove all tags set at the instance level.

There is no requirement for all the instances to be online, only that the cluster has quorum. To tag the InnoDB Cluster assigned to `myCluster` with the `location` tag set to US East, issue the following in JavaScript:

```
mysql-js> myCluster.setOption("tag:location", "US East")
mysql-js> myCluster.options()
{
    "cluster": {
        "name": "test1",
        "tags": {
            "ic-1:3306": [],
            "ic-2:3306": [],
            "ic-3:3306": [],
            "global": [
                {
                    "option": "location:",
                    "value": "US East"
                }
            ]
        }
    }
}
```

Or issue the following in Python:

```
mysql-py> myCluster.set_option("tag:location", "US East")
mysql-pys> myCluster.options()
{
    "cluster": {
        "name": "test1",
        "tags": {
            "ic-1:3306": [],
            "ic-2:3306": [],
            "ic-3:3306": [],
            "global": [
                {
                    "option": "location:",
                    "value": "US East"
                }
            ]
        }
    }
}
```

Removing Tags from a Cluster

To remove a tag from a Cluster, use the `Cluster.setOption(option, value)` operation with a `null` value in JavaScript, and a `None` value in Python.

To remove the tag from InnoDB Cluster assigned to `myCluster` with the `location` tag, issue the following in JavaScript:

```
mysql-js> myCluster.setOption("tag:location", null)
```

Or using Python:

```
mysql-js> myCluster.set_option("tag:location", None)
```

User Defined Tagging

AdminAPI supports the `tag` namespace, where you can store information in the key-value pairs associated with a given Cluster, ReplicaSet or instance. The options under the `tag` namespace are not

constrained, meaning you can tag with whatever information you choose, as long as it is a valid MySQL ASCII identifier.

You can use any name and value for a tag, as long as the name follows the following syntax: _ or letters followed by alphanumeric and _ characters.

The `namespace` option is a colon separated string with the format `namespace:option`, where `namespace` is the name of the namespace and `option` is the actual option name. You can set and remove tags at the instance level, or at the Cluster or ReplicaSet level.

Tag names can have any value as long as it starts with a letter or underscore, optionally followed by alphanumeric and _ characters, for example, `^[a-zA-Z_][0-9a-zA-Z_]*`. Only built-in tags are allowed to start with the underscore _ character.

How you use custom tags is up to you. You could set a custom tag on a Cluster to mark the location of the Cluster. For example, set a custom tag named location with a value of `EMEA` on the Cluster.

6.10 Using MySQL Router with AdminAPI, InnoDB Cluster, and InnoDB ReplicaSet

This section describes how to integrate MySQL Router with InnoDB Cluster and InnoDB ReplicaSet. For instructions to integrate MySQL Router with InnoDB ClusterSet, see Section 8.5, “Integrating MySQL Router With InnoDB ClusterSet”.

For background information on MySQL Router, see MySQL Router 8.0.

6.10.1 Bootstrapping MySQL Router

You bootstrap MySQL Router against an InnoDB ReplicaSet or InnoDB Cluster to automatically configure routing. The bootstrap process is a specific way of running MySQL Router, which does not start the usual routing and instead configures the `mysqlrouter.conf` file based on the metadata.

To bootstrap MySQL Router at the command-line, pass in the `--bootstrap` option when you start the `mysqlrouter` command, and it retrieves the topology information from the metadata and configures routing connections to the server instances. Alternatively, on Windows use the MySQL Installer to bootstrap MySQL Router. For more information, see MySQL Router Configuration with MySQL Installer.

Once MySQL Router has been bootstrapped, client applications then connect to the ports it publishes. MySQL Router automatically redirects client connections to the instances based on the incoming port, for example 6646 is used by default for `read-write` connections using classic MySQL protocol.

In the event of a topology change, for example, an unexpected failure of an instance, MySQL Router detects the change and adjusts the routing to the remaining instances automatically. This automatic adjustment removes the need for client applications to handle failover, or to be aware of the underlying topology. For more information, see Routing for MySQL InnoDB Cluster.

Note

Do not attempt to configure MySQL Router manually to redirect to the server instances. Always use the `--bootstrap` option as this ensures that MySQL Router takes its configuration from the metadata. See Cluster Metadata and State.

6.10.2 Configuring the MySQL Router User

When MySQL Router connects to a Cluster, ClusterSet, or ReplicaSet, it requires a user account that has the correct privileges. From MySQL Router version 8.0.19 this internal user can be specified using the `--account` option. In previous versions, MySQL Router created internal accounts at each

bootstrap of the cluster, which could result in many accounts building up over time. From MySQL Shell version 8.0.20, you can use AdminAPI to set up the user account required for MySQL Router.

Use the `setupRouterAccount(user, [options])` operation to create a MySQL user account or upgrade an existing account so that it can be used by MySQL Router to operate on an InnoDB Cluster or InnoDB ReplicaSet. This is the recommended method of configuring MySQL Router with InnoDB Cluster and InnoDB ReplicaSet.

To add a new MySQL Router account named *`myRouter1`* to the InnoDB Cluster referenced by the variable *`testCluster`*, issue:

```
mysqlsh> testCluster.setupRouterAccount('myRouter1')
```

In this case, no domain is specified and so the account is created with the wildcard (`%`) character, which ensures that the created user can connect from any domain. To limit the account to only be able to connect from the *`example.com`* domain in JavaScript, issue:

```
mysql-js> testCluster.setupRouterAccount('myRouter1@example.com')
```

Or using Python:

```
mysql-py> testCluster.setup_router_account('myRouter1@example.com')
```

The operation prompts for a password, and then sets up the MySQL Router user with the correct privileges. If the InnoDB Cluster or InnoDB ReplicaSet has multiple instances, the created MySQL Router user is propagated to all of the instances.

When you already have a MySQL Router user configured, for example if you are using a version prior to 8.0.20, you can use the `setupRouterAccount()` operation to reconfigure the existing user. In this case, pass in the `update` option set to true. For example, to reconfigure the *`myOldRouter`* user, issue the following in JavaScript:

```
mysql-js> testCluster.setupRouterAccount('myOldRouter', {'update':1})
```

Or using Python:

```
mysql-py> testCluster.setup_router_account('myOldRouter', {'update':1})
```

You can also update the MySQL Router user's password with the dictionary options `{password: "newPassword", update: 1}` . The following JavaScript example updates the password of the MySQL Router user, `myRouter1` to `newPassword1#`:

```
mysql-js> testCluster.setupRouterAccount('myRouter1', {password: "newPassword1#",'update':1})
```

Or using Python:

```
mysql-py> testCluster.setup_router_account('myRouter1', {password: "newPassword1#", 'update':1})
```

As of MySQL Shell 8.0.33, SSL certificates are supported. The following options were added to `setupRouterAccount()`:

- `requireCertIssuer`: Optional SSL certificate issuer for the account.
- `requireCertSubject`: Optional SSL certificate subject for the account.
- `passwordExpiration: numberOfDays | Never | Default`: Password expiration setting for the account.
  - `numberOfDays`: The number of days before the password expires.
  - `Never`: The password never expires.
  - `Default`: The system default is used.

6.10.3 Deploying MySQL Router

The recommended deployment of MySQL Router is on the same host as the application. When using a sandbox deployment, everything is running on a single host. Therefore you deploy MySQL Router to the same host. When using a production deployment, we recommend deploying one MySQL Router instance to each machine used to host one of your client applications. It is also possible to deploy MySQL Router to a common machine through which your application instances connect. For more information, see Installing MySQL Router.

To bootstrap MySQL Router based on an InnoDB Cluster or InnoDB ReplicaSet, you need the URI-like connection string to an online instance. Run the `mysqlrouter` command and provide the `--bootstrap=`*`instance`* option, where *`instance`* is the URI-like connection string to an online instance. MySQL Router connects to the instance and uses the included metadata cache plugin to retrieve the metadata, consisting of a list of server instance addresses and their role. For example:

```
$> mysqlrouter --bootstrap icadmin@ic-1:3306 --account=mysqlrouter
```

You are prompted for the instance password and encryption key for MySQL Router to use. This key is used to encrypt the instance password used by MySQL Router to connect to the cluster. The ports you can use for client connections are also displayed. For additional bootstrap related options, see Bootstrapping Options.

Tip

At this point MySQL Router has not been started so that it would route connections. Bootstrapping is a separate process.

The MySQL Router bootstrap process creates a `mysqlrouter.conf` file, with the settings based on the metadata retrieved from the address passed to the `--bootstrap` option, in the above example `icadmin@ic-1:3306`. Based on the metadata retrieved, MySQL Router automatically configures the `mysqlrouter.conf` file, including a `metadata_cache` section.

If you are using MySQL Router 8.0.14 and later, the `--bootstrap` option automatically configures MySQL Router to track and store active MySQL metadata server addresses at the path configured by `dynamic_state`. This ensures that when MySQL Router is restarted it knows which MySQL metadata server addresses are current. For more information, see the `dynamic_state` documentation.

In earlier MySQL Router versions, metadata server information was defined during MySQL Router's initial bootstrap operation and stored statically as `bootstrap_server_addresses` in the configuration file, which contained the addresses for all server instances in the cluster. For example:

```
[metadata_cache:prodCluster]
router_id=1
bootstrap_server_addresses=mysql://icadmin@ic-1:3306,mysql://icadmin@ic-2:3306,mysql://icadmin@ic-3:3306
user=mysql_router1_jy95yozko3k2
metadata_cluster=prodCluster
ttl=300
```

Tip

If using MySQL Router 8.0.13 or earlier, when you change the topology of a cluster by adding another server instance after you have bootstrapped MySQL Router, you need to update `bootstrap_server_addresses` based on the updated metadata. Either restart MySQL Router using the `--bootstrap` option, or manually edit the `bootstrap_server_addresses` section of the `mysqlrouter.conf` file and restart MySQL Router.

The generated MySQL Router configuration creates TCP ports which you use to connect to the cluster. By default, ports for communicating with the cluster using both classic MySQL protocol and X Protocol are created. To use X Protocol the server instances must have X Plugin installed and configured, which is the default for MySQL 8.0 and later. The default available TCP ports are:

- *6446* - for classic MySQL protocol read-write sessions, which MySQL Router redirects incoming connections to primary server instances.
- *6447* - for classic MySQL protocol read-only sessions, which MySQL Router redirects incoming connections to one of the secondary server instances.
- *64460* - for X Protocol read-write sessions, which MySQL Router redirects incoming connections to primary server instances.
- *64470* - for X Protocol read-only sessions, which MySQL Router redirects incoming connections to one of the secondary server instances.

Depending on your MySQL Router configuration the port numbers might be different to the above. For example if you use the `--conf-base-port` option, or the `group_replication_single_primary_mode` variable. The exact ports are listed when you start MySQL Router.

The way incoming connections are redirected depends on the underlying topology used. For example, when using a single-primary cluster, by default MySQL Router publishes a X Protocol and a classic MySQL protocol port, which clients connect to for read-write sessions and which are redirected to the cluster's single primary. With a multi-primary cluster, read-write sessions are redirected to one of the primary instances in a round-robin fashion. For example, the first connection to port 6446 is redirected to the ic-1 instance. The second connection to port 6446 is redirected to the ic-2 instance, and so on.

For incoming read-only connections, MySQL Router redirects connections to one of the secondary instances, also in a round-robin fashion. To modify this behavior see the `routing_strategy` option.

Once bootstrapped and configured, start MySQL Router. If you used a system wide install with the `--bootstrap` option then issue:

```
$> mysqlrouter &
```

If you installed MySQL Router to a directory using the `--directory` option, use the `start.sh` script found in the directory you installed to.

Alternatively set up a service to start MySQL Router automatically when the system boots, see Starting MySQL Router. You can now connect a MySQL client, such as MySQL Shell to one of the incoming MySQL Router ports as described above and see how the client gets transparently connected to one of the server instances.

```
$> mysqlsh --uri root@localhost:6442
```

To verify which instance you are connected to, issue an SQL query against the `port` status variable. For example:

```
mysql-js> \sql
Switching to SQL mode... Commands end with ;
mysql-sql> select @@port;
+--------+
| @@port |
+--------+
|   3310 |
+--------+
```

Or, for example, using:

```
mysql-js> \sql
Switching to SQL mode... Commands end with ;
mysql-sql> SHOW VARIABLES WHERE Variable_name = 'port';
+--------+
| @@port |
+--------+
|   3310 |
+--------+
```

6.10.4 Using ReplicaSets with MySQL Router

You can use MySQL Router 8.0.19 and later to bootstrap against an InnoDB ReplicaSet. For more information, see Section 6.10, “Using MySQL Router with AdminAPI, InnoDB Cluster, and InnoDB ReplicaSet”. The only difference in the generated MySQL Router configuration file is the addition of the `cluster_type` option. When MySQL Router is bootstrapped against a ReplicaSet, the generated configuration file includes:

```
cluster_type=rs
```

When you use MySQL Router with InnoDB ReplicaSet, be aware that:

- The read-write port of MySQL Router directs client connections to the primary instance of the ReplicaSet.
- The read-only port of MySQL Router direct client connections to a secondary instance of the ReplicaSet, although it could also direct them to the primary.
- MySQL Router obtains information about the ReplicaSet's topology from the primary instance.
- MySQL Router automatically recovers when the primary instance becomes unavailable and a different instance is promoted.

You work with the MySQL Router instances which have been bootstrapped against a ReplicaSet in the same way as with InnoDB Cluster. For more information on *`ReplicaSet`*`.listRouters()` and *`ReplicaSet`*`.removeRouterMetadata()`, see Section 6.10.6, “Working with a Cluster's Routers” .

6.10.5 Testing InnoDB Cluster High Availability

To test if InnoDB Cluster high availability works, simulate an unexpected halt by killing an instance. The cluster detects the fact that the instance left the cluster and reconfigures itself. How the cluster reconfigures itself depends on whether you are using a single-primary or multi-primary cluster, and the role the instance serves within the cluster.

In single-primary mode:

- If the current primary leaves the cluster, one of the secondary instances is elected as the new primary, with instances prioritized by the lowest `server_uuid`. MySQL Router redirects read-write connections to the newly elected primary.
- If a current secondary leaves the cluster, MySQL Router stops redirecting read-only connections to the instance.

For more information see Single-Primary Mode.

In multi-primary mode:

- If a current "R/W" instance leaves the cluster, MySQL Router redirects read-write connections to other primaries. If the instance which left was the last primary in the cluster then the cluster is completely gone and you cannot connect to any MySQL Router port.

For more information see Multi-Primary Mode.

There are various ways to simulate an instance leaving a cluster, for example you can forcibly stop the MySQL server on an instance, or use the AdminAPI `dba.killSandboxInstance()` if testing a sandbox deployment. In this example, there is a single-primary sandbox cluster deployment with three server instances and the instance listening at port 3310 is the current primary. The instance leaves the cluster unexpectedly, simulated by `killing` an the instance:

For example, by issuing the JavaScript command:

```
mysql-js> dba.killSandboxInstance(3310)
```

Or, by issuing the Python command:

```
mysql-py> dba.kill_sandbox_instance(3310)
```

The cluster detects the change and elects a new primary automatically.

Assuming your session is connected to port 6446, the default read-write classic MySQL protocol port, MySQL Router should detect the change to the cluster's topology and redirect your session to the newly elected primary. To verify this, switch to SQL mode in MySQL Shell using the `\sql` command and select the instance's `port` variable to check which instance your session has been redirected to.

The first `SELECT` statement fails as the connection to the original primary was lost, this means the current session has been closed. MySQL Shell automatically reconnects for you and when you issue the command again the new port is confirmed.

```
mysql-js> \sql
Switching to SQL mode... Commands end with ;
mysql-sql> SELECT @@port;
ERROR: 2013 (HY000): Lost connection to MySQL server during query
The global session got disconnected.
Attempting to reconnect to 'root@localhost:6446'...
The global session was successfully reconnected.
mysql-sql> SELECT @@port;
+--------+
| @@port |
+--------+
|   3330 |
+--------+
1 row in set (0.00 sec)
```

In this example, the instance at port 3330 has been elected as the new primary. This election shows that the InnoDB Cluster has provided automatic failover, and that MySQL Router has automatically reconnected us to the new primary instance, and that we have high availability.

6.10.6 Working with a Cluster's Routers

You can bootstrap multiple instances of MySQL Router against InnoDB Cluster or InnoDB ReplicaSet. From version 8.0.19, to show a list of all registered MySQL Router instances, issue:

```
cluster.listRouters()
```

The result provides information about each registered MySQL Router instance, such as its name in the metadata, the hostname, ports, and so on. For example, issue the following JavaScript command:

```
mysql-js> Cluster.listRouters()
{
    "clusterName": "example",
    "routers": {
        "ic-1:3306": {
            "hostname": "ic-1:3306",
            "lastCheckIn": "2020-01-16 11:43:45",
            "roPort": 6447,
            "roXPort": 64470,
            "rwPort": 6446,
            "rwXPort": 64460,
            "version": "8.0.19"
        }
    }
}
```

Or issue the following Python command:

```
mysql-py> Cluster.list_routers()
{
    "clusterName": "example",
    "routers": {
        "ic-1:3306": {
            "hostname": "ic-1:3306",
            "lastCheckIn": "2020-01-16 11:43:45",
```

```
            "roPort": 6447,
            "roXPort": 64470,
            "rwPort": 6446,
            "rwXPort": 64460,
            "version": "8.0.19"
        }
    }
}
```

The returned information shows:

- The name of the MySQL Router instance.
- Last check-in timestamp, which is generated by a periodic ping from the MySQL Router stored in the metadata.
- Hostname where the MySQL Router instance is running.
- Read-Only and Read-Write ports, which the MySQL Router publishes for classic MySQL protocol connections.
- Read-Only and Read-Write ports, which the MySQL Router publishes for X Protocol connections.
- Version of this MySQL Router instance. The support for returning `version` was added in 8.0.19. If this operation is run against an earlier version of MySQL Router, the version field is `null`.

Additionally, the *`Cluster`*`.listRouters()` operation can show a list of instances that do not support the metadata version supported by MySQL Shell. Use the `onlyUpgradeRequired` option. For example, by issuing *`Cluster`*`.listRouters({'onlyUpgradeRequired':'true'})`.

The returned list shows only the MySQL Router instances registered with the *`Cluster`*, which require an upgrade of their metadata. For more information, see Section 6.11, “Upgrade Metadata Schema”.

MySQL Router instances are not automatically removed from the metadata, so for example as you bootstrap more instances the InnoDB Cluster metadata contains a growing number of references to instances. To remove a registered MySQL Router instance from a cluster's metadata, use the *`Cluster`*`.removeRouterMetadata(`*`router`*`)` operation, added in version 8.0.19.

Use the *`Cluster`*`.listRouters()` operation to get the name of the MySQL Router instance you want to remove, and pass it in as *`router`*. For example, suppose the MySQL Router instances registered with a cluster were:

```
mysql-js> Cluster.listRouters(){

    "clusterName": "testCluster",
    "routers": {
        "myRouter1": {
            "hostname": "example1.com",
            "lastCheckIn": null,
            "routerId": "1",
            "roPort": "6447",
            "rwPort": "6446"
            "version": null
        },
        "myRouter2": {
            "hostname": "example2.com",
            "lastCheckIn": "2019-11-27 16:25:00",
            "routerId": "3",
            "roPort": "6447",
            "rwPort": "6446"
            "version": "8.0.19"
        }
    }
}
```

Based on the fact that the instance named “myRouter1” has `null` for “lastCheckIn” and “version”. Remove this old instance from the metadata by issuing the following JavaScript command:

```
mysql-js> cluster.removeRouterMetadata('myRouter1')
```

Or, by issuing the following Python command:

```
mysql-py> cluster.remove_router_metadata('myRouter1')
```

The MySQL Router instance specified is unregistered from the cluster by removing it from the InnoDB Cluster metadata.

6.11 Upgrade Metadata Schema

As AdminAPI evolves, some releases might require you to upgrade the metadata of existing ClusterSets, ReplicaSets, and Clusters to ensure they are compatible with newer versions of MySQL Shell. For example, the addition of InnoDB ReplicaSet in version 8.0.19 means that the metadata schema has been upgraded to version 2.0. Regardless of whether you plan to use InnoDB ReplicaSet or not, to use MySQL Shell 8.0.19 or later with a cluster deployed using an earlier version of MySQL Shell, you must upgrade the metadata of your Cluster.

Warning

Without upgrading the metadata you cannot use MySQL Shell to change the configuration of a cluster created with earlier versions. For example, you can only perform read operations against the cluster such as:

- *Cluster*.status()
- *Cluster*.describe()
- *Cluster*.options()

The dba.upgradeMetadata() operation compares the version of the metadata schema found on the ClusterSet, ReplicaSet, or InnoDB Cluster that MySQL Shell is currently connected to, with the version of the metadata schema supported by this MySQL Shell version. If the metadata found version is lower, an upgrade process is started. The dba.upgradeMetadata() function then updates any automatically created MySQL Router users to have the correct privileges. Manually created MySQL Router users with a name not starting with mysql_router_ are not automatically upgraded. This is an important step in upgrading your ClusterSet, ReplicaSet, or InnoDB Cluster, only then can the MySQL Router metadata be upgraded. To view information on which of the MySQL Router instances registered with a ClusterSet, ReplicaSet, or Cluster require the metadata upgrade, use the .listRouters() function. For example, to list the Router instances associated with a Cluster, using the assigned variable cluster issue:

```
cluster.listRouters({'onlyUpgradeRequired':'true'})
{
    "clusterName": "mycluster",
    "routers": {
        "example.com::": {
            "hostname": "example.com",
            "lastCheckIn": "2019-11-26 10:10:37",
            "roPort": 6447,
            "roXPort": 64470,
            "rwPort": 6446,
            "rwXPort": 64460,
            "version": "8.0.18"
        }
    }
}
```

In this example, the onlyUpgradeRequired options is included in the listRouters() function. The onlyUpgradeRequired is a Boolean value that enables filtering , so only router instances that support older versions of the Metadata Schema and require upgrading are included in the returned JSON object.

To upgrade a ClusterSet, ReplicaSet, or Cluster's metadata, connect MySQL Shell's global session to your ClusterSet, ReplicaSet, or Cluster and use the `dba.upgradeMetadata()` operation to upgrade the ClusterSet, ReplicaSet, or Cluster's metadata to the new metadata. For example:

```
mysql-js> shell.connect('user@example.com:3306')

mysql-js> dba.upgradeMetadata()
InnoDB Cluster Metadata Upgrade

The cluster you are connected to is using an outdated metadata schema version
1.0.1 and needs to be upgraded to 2.0.0.

Without doing this upgrade, no AdminAPI calls except read only operations will
be allowed.

The grants for the MySQL Router accounts that were created automatically when
bootstrapping need to be updated to match the new metadata version's
requirements.
Updating router accounts...
NOTE: 2 router accounts have been updated.

Upgrading metadata at 'example.com:3306' from version 1.0.1 to version 2.0.0.
Creating backup of the metadata schema...
Step 1 of 1: upgrading from 1.0.1 to 2.0.0...
Removing metadata backup...
Upgrade process successfully finished, metadata schema is now on version 2.0.0
```

If the installed metadata version is lower, an upgrade process is started.

The `dba.upgradeMetadata()` function accepts the following options:

- `dryRun`: is a Boolean value used to enable a dry run of the upgrade process. If `dryRun` is used, the `dba.upgradeMetadata()` function determines whether a metadata upgrade or restore is required and informs you without actually executing the operation.
- `interactive`: is a Boolean value used to disable or enable the wizards in the command execution, meaning that prompts and confirmations will be provided or not provided according to the value set. The default value is equal to MySQL Shell wizard mode.

If you encounter an error related to the ClusterSet, ReplicaSet, or Cluster administration user missing privileges, use the relevant `.setupAdminAccount()` operation with the update option to grant the user the correct privileges:

- Create or upgrade a MySQL user account with the necessary privileges to administer an InnoDB Cluster: `<Cluster>.setupAdminAccount(user, options)`

  See InnoDB Cluster administrator accounts.

- Create or upgrade a MySQL user account with the necessary privileges to administer an InnoDB ReplicaSet: `<ReplicaSet>.setupAdminAccount(user, options)`

6.12 Locking Mechanism for AdminAPI Operations

As of version 8.0.33, AdminAPI extends the locking mechanism introduced for InnoDB ReplicaSet in 8.0.20 to InnoDB Cluster and InnoDB ClusterSet.

Previously, different instances of MySQL Shell could connect and process AdminAPI operations simultaneously on the same resource. This could lead to inconsistent states and errors, for example, if `Cluster.addInstance()` and `Cluster.setPrimaryInstance()` were processed in parallel.

Locking Types

AdminAPI uses the MySQL Locking Service to provide the following locking types:

- Read or shared lock: allows concurrent execution of some operations while blocking exclusive operations. If an operation attempts to acquire a shared lock, but cannot, due to the presence of an exclusive lock, the operation is aborted without making any changes. If the current operation has a shared lock, and the new operation requires a shared lock, the new operation is allowed access.
- Write or exclusive lock: blocks execution of all other operations until the current operation is complete and the exclusive lock is released. If an operation attempts to acquire an exclusive lock, but cannot, due to the presence of an existing lock, the operation is aborted without making any changes.

See The Locking Service for more information.

The following tables list the locking per AdminAPI operation:

- DBA Locking: lists the locks for `dba.operationName` operations.
- Cluster Locking: lists the locks for `Cluster.operationName` operations.
- ClusterSet Locking: lists the locks for `ClusterSet.operationName` operations.
- InnoDB ReplicaSet Locking: lists the locks for `ReplicaSet.operationName` operations.

Note

Operations which do not require locks are not listed.

In practice, if you try to perform an operation while another operation that cannot be performed concurrently is still running, you get an error indicating that a lock on a needed resource could not be acquired. In this case, you should wait for the running operation which holds the lock to complete, and only then try to process the next operation. For example:

```
mysql-js> rs.addInstance("admin@rs2:3306");

ERROR: The operation cannot be executed because it failed to acquire the lock on
instance 'rs1:3306'. Another operation requiring exclusive access to the
instance is still in progress, please wait for it to finish and try again.

ReplicaSet.addInstance: Failed to acquire lock on instance 'rs1:3306' (MYSQLSH
51400)
```

In this example, `ReplicaSet.addInstance()` failed because a lock on the primary instance (`rs1:3306`) could not be acquired, because a `ReplicaSet.setPrimaryInstance()` operation (or other similar operation) was still running.

Note

If an instance restarts as part of a clone operation or a requested restart, the lock is released. As a result, there is a short period, measured in milliseconds, when another Shell session could gain access to the instance on restart and lock it. However, the original locks on Cluster and/or ClusterSet remain, so a new command which could lock the newly restarted instance cannot request Cluster or ClusterSet locks.

DBA Locking

This section lists the locks for `dba.operationName` operations.

Table 6.1 DBA Operation Locks

Operation	Lock Type
`configureInstance()`	Exclusive on the target instance
`configureLocalInstance()`	Exclusive on the target instance

Operation	Lock Type
`createCluster()`	Exclusive on the target instance
`rebootClusterFromCompleteOutage()`	Exclusive on all contactable cluster members. If the cluster is a replica cluster, and part of a ClusterSet, it is also rejoined to the ClusterSet as part of the operation. In this scenario, the operation also acquires the same locks as `clusterset.rejoinCluster()`.
`upgradeMetadata()`	• Cluster: Exclusive lock on the Cluster and on the target instance. • ClusterSet: Exclusive lock on the ClusterSet, primary Cluster, and the target instance.
`createReplicaSet()`	Exclusive lock on the target instance.
`configureReplicaSetInstance()`	Exclusive lock on the target instance.
`upgradeMetadata()`	Exclusive lock on the ClusterSet and primary cluster, if the topology is a ClusterSet, or on the Cluster, if the topology is a standalone cluster, and on the target instance.

Cluster Locking

This section lists the locks for *`cluster.operationName`* operations.

Table 6.2 Cluster Operation Locks

Operation	Cluster Lock Type	Target Instance Lock Type
`addInstance()`	Exclusive	Exclusive
`createClusterSet()`	Exclusive	
`dissolve()`	Exclusive	
`fenceAllTraffic()`	Exclusive	
`fenceWrites()`	Exclusive	
`forceQuorum()`	Exclusive	
`rejoinInstance()`	Shared	Exclusive
`removeInstance()`	Exclusive	Exclusive
`rescan()`	Exclusive	
`resetRecoveryAccountsPassword()`	Exclusive	
`setInstanceOption()`		Exclusive Except for the options `tag` and `clusterName`.

Operation	Cluster Lock Type	Target Instance Lock Type
`setOption()`	Exclusive Except for the options `tag` and `clusterName`.	
`setPrimaryInstance()`	Exclusive	
`setupAdminAccount()`	Shared	
`setupRouterAccount()`	Shared	
`switchToMultiPrimaryMode()`	Exclusive	
`switchToSinglePrimaryMode()`	Exclusive	
`unfenceWrites()`	Exclusive	

ClusterSet Locking

This section lists the locks for *`clusterSet.operationName`* operations.

Table 6.3 ClusterSet Operation Locks

Operation	ClusterSet Lock Type	Primary Cluster Lock Type	Target Cluster Lock Type
`createReplicaCluster()`	Shared		Exclusive on the instance used to create the new replica cluster.
`forcePrimaryCluster()`			Exclusive on all replica clusters.
`rejoinCluster()`	Shared	Shared	Exclusive
`removeCluster()`	Exclusive.	Exclusive	Exclusive
`setOption()`	Exclusive	Exclusive (only if `replicationAllowedHost` is set)	
`setPrimaryCluster()`	Exclusive	Exclusive	

InnoDB ReplicaSet Locking

This section lists the locks for *`replicaSet.operationName`* operations.

Table 6.4 ReplicaSet Operation Locks

Operation	Primary Instance Lock Type	Target Instance Lock Type
`forcePrimaryInstance()`		Exclusive on target instance and on all contactable members of the ReplicaSet at the time of the operation.
`setPrimaryInstance()`	Exclusive	Exclusive on target instance and on all contactable members of the ReplicaSet at the time of the operation.
`addInstance()`	Shared	Exclusive
`rejoinInstance()`	Shared	Exclusive
`removeInstance()`	Shared	Exclusive

Operation	Primary Instance Lock Type	Target Instance Lock Type
`removeRouterMetadata()`	Shared	

Chapter 7 MySQL InnoDB Cluster

Table of Contents

7.1 InnoDB Cluster Requirements 114
7.2 InnoDB Cluster Limitations 116
7.3 User Accounts for InnoDB Cluster 116
7.4 Deploying a Production InnoDB Cluster 119
7.4.1 Pre-Checking Instance Configuration for InnoDB Cluster Usage 120
7.4.2 Configuring Production Instances for InnoDB Cluster Usage 121
7.4.3 Creating an InnoDB Cluster 125
7.4.4 Adding Instances to an InnoDB Cluster 127
7.4.5 Configuring InnoDB Cluster Ports 130
7.4.6 Using MySQL Clone with InnoDB Cluster 131
7.4.7 Adopting a Group Replication Deployment 135
7.5 Configuring InnoDB Cluster 135
7.5.1 Setting Options for InnoDB Cluster 135
7.5.2 Customizing InnoDB Cluster Member Servers 136
7.5.3 Configuring the Election Process 137
7.5.4 Configuring Failover Consistency 137
7.5.5 Configuring Automatic Rejoin of Instances 138
7.5.6 Configuring the Parallel Replication Applier 139
7.5.7 InnoDB Cluster and Auto-increment 140
7.5.8 InnoDB Cluster and Binary Log Purging 140
7.5.9 Configuring the Group Replication Communication Stack 140
7.6 Securing InnoDB Cluster 142
7.7 Monitoring InnoDB Cluster 145
7.8 Restoring and Rebooting an InnoDB Cluster 156
7.8.1 Rejoining an Instance to a Cluster 156
7.8.2 Restoring a Cluster from Quorum Loss 157
7.8.3 Rebooting a Cluster from a Major Outage 158
7.8.4 Rescanning a Cluster 161
7.8.5 Fencing a Cluster 163
7.9 Modifying or Dissolving an InnoDB Cluster 163
7.10 Upgrade InnoDB Cluster 167
7.10.1 InnoDB Cluster Upgrade 167
7.10.2 Troubleshooting InnoDB Cluster Upgrades 172

MySQL InnoDB Cluster provides a complete high availability solution for MySQL. By using AdminAPI, which is included with MySQL Shell, you can easily configure and administer a group of at least three MySQL server instances to function as an InnoDB Cluster.

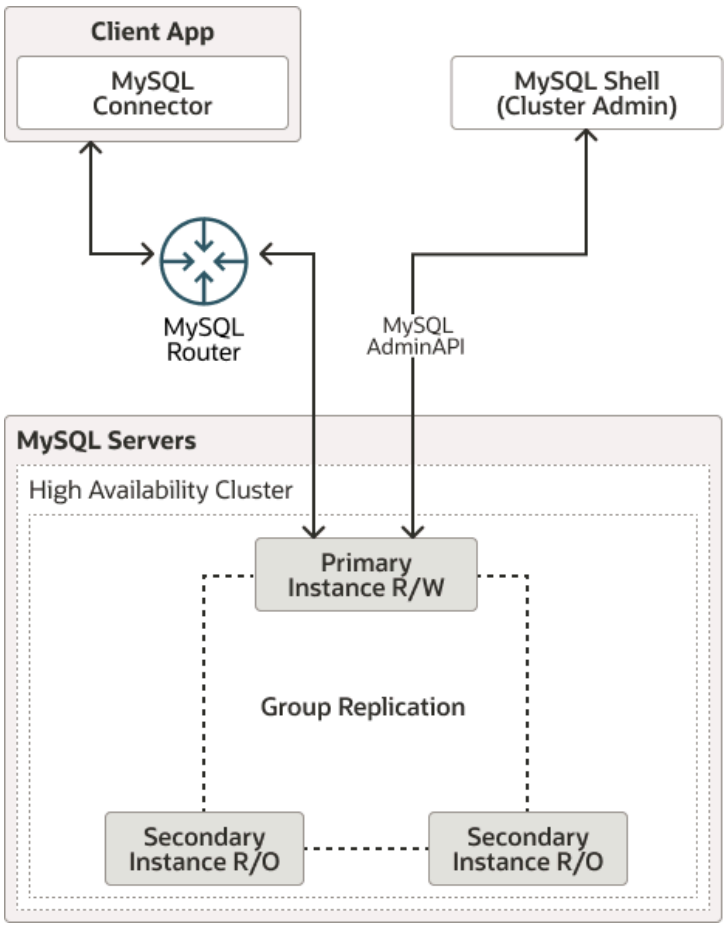
Each MySQL server instance in an InnoDB Cluster runs MySQL Group Replication, which provides the mechanism to replicate data within an InnoDB Cluster, with built-in failover. AdminAPI removes the need to work directly with Group Replication in an InnoDB Cluster, but for more information see Group Replication which explains the details. From MySQL 8.0.27, you can also set up InnoDB ClusterSet (see Chapter 8, *MySQL InnoDB ClusterSet*) to provide disaster tolerance for InnoDB Cluster deployments by linking a primary InnoDB Cluster with one or more replicas of itself in alternate locations, such as different datacenters.

MySQL Router can automatically configure itself based on the cluster you deploy, connecting client applications transparently to the server instances. In the event of an unexpected failure of a server instance the cluster reconfigures automatically. In the default single-primary mode, an InnoDB Cluster has a single read-write server instance - the primary. Multiple secondary server instances are replicas

of the primary. If the primary fails, a secondary is automatically promoted to the role of primary. MySQL Router detects this and forwards client applications to the new primary. Advanced users can also configure a cluster to have multiple primaries.

The following diagram shows an overview of how the technologies work together:

Figure 7.1 InnoDB Cluster overview



Important

InnoDB Cluster does not provide support for MySQL NDB Cluster. NDB Cluster depends on the `NDB` storage engine as well as a number of programs specific to NDB Cluster which are not furnished with MySQL Server 8.0; `NDB` is available only as part of the MySQL NDB Cluster distribution. In addition, the MySQL server binary (`mysqld`) that is supplied with MySQL Server 8.0 cannot be used with NDB Cluster. For more information about MySQL NDB Cluster, see MySQL NDB Cluster 8.0. MySQL Server Using InnoDB Compared with NDB Cluster, provides information about the differences between the `InnoDB` and `NDB` storage engines.

7.1 InnoDB Cluster Requirements

Before installing a production deployment of InnoDB Cluster, ensure that the server instances you intend to use meet the following requirements.

- InnoDB Cluster uses Group Replication and therefore your server instances must meet the same requirements. See Group Replication Requirements. AdminAPI provides the `dba.checkInstanceConfiguration()` method to verify that an instance meets the Group

Replication requirements, and the `dba.configureInstance()` method to configure an instance to meet the requirements.

Note

When using a sandbox deployment the instances are configured to meet these requirements automatically.

- Data for use with Group Replication, and therefore also data for use with InnoDB Cluster, must be stored in the `InnoDB` transactional storage engine. The use of other storage engines, including the temporary `MEMORY` storage engine, might cause errors in Group Replication. Convert any tables in other storage engines to use `InnoDB` before using the instance with Group Replication and InnoDB Cluster. You can prevent the use of other storage engines by setting the `disabled_storage_engines` system variable on server instances, for example:

```
disabled_storage_engines="MyISAM,BLACKHOLE,FEDERATED,ARCHIVE,MEMORY"
```

- There must be no inbound replication channels on any server instance when you set up the cluster. The channels created automatically by Group Replication (`group_replication_applier` and `group_replication_recovery`) are allowed on a replication group that is being adopted. InnoDB Cluster does not support manually configured asynchronous replication channels outside of those that are managed using AdminAPI. If you are migrating an existing replication topology to an InnoDB Cluster deployment and need to skip this validation temporarily during the setup process, you can use the `force` option when you create the cluster to bypass it.
- `group_replication_tls_source` must not be set to `mysql_admin`.
- The Performance Schema must be enabled on any instance which you want to use with InnoDB Cluster.
- The provisioning scripts that MySQL Shell uses to configure servers for use in InnoDB Cluster require access to Python. On Windows MySQL Shell includes Python and no user configuration is required. On Unix Python must be found as part of the shell environment. To check that your system has Python configured correctly issue:

```
$ /usr/bin/env python
```

  If a Python interpreter starts, no further action is required. If the previous command fails, create a soft link between `/usr/bin/python` and your chosen Python binary. For more information, see Supported Languages.
- From version 8.0.17, instances must use a unique `server_id` within an InnoDB Cluster. When you use the *`Cluster`*`.addInstance(`*`instance`*`)` operation, if the `server_id` of *`instance`* is already used by an instance in the cluster then the operation fails with an error.
- From version 8.0.23, instances should be configured to use the parallel replication applier. See Section 7.5.6, “Configuring the Parallel Replication Applier”.
- During the process of configuring an instance for InnoDB Cluster, the majority of the system variables required for using an instance are configured. But AdminAPI does not configure the `transaction_isolation` system variable, which means that it defaults to `REPEATABLE READ`. This does not impact a single-primary cluster, but if you are using a multi-primary cluster then unless you rely on `REPEATABLE READ` semantics in your applications, we recommend using the `READ COMMITTED` isolation level. See Group Replication Limitations.
- The relevant configuration options for the instance, particularly the Group Replication configuration options, must be in a single option file. InnoDB Cluster only supports a single option file for server instances and does not support the use of the `--defaults-extra-file` option to specify an additional option file. For any AdminAPI operation working with the instance's option file, the main file must be specified. If you want to use multiple option files for configuration options that do not relate to InnoDB Cluster, you must configure the files manually, make sure they are updated correctly considering the precedence rules for the use of multiple option files, and ensure that the settings

relating to InnoDB Cluster are not incorrectly overwritten by options in an extra unrecognized option file.

7.2 InnoDB Cluster Limitations

This section describes the known limitations of InnoDB Cluster. As InnoDB Cluster uses Group Replication, you should also be aware of its limitations, see Group Replication Limitations.

-

  Important

  Due to an error in a metadata query, MySQL Shell 8.0.27 cannot be used to manage an InnoDB Cluster running MySQL Server 8.0.25. To work around this issue, upgrade MySQL Server to the 8.0.26 or 8.0.27 release on the InnoDB Cluster member instances before using MySQL Shell 8.0.27 with the cluster. The issue will be fixed in MySQL Shell 8.0.28.

- InnoDB Cluster does not manage manually configured asynchronous replication channels. Group Replication and AdminAPI do not ensure that the asynchronous replication is active on the primary only, and state is not replicated across instances. This can lead to various scenarios where replication no longer works, as well as potentially causing a split brain. Replication between one InnoDB Cluster and another is supported only by InnoDB ClusterSet, which is available from MySQL 8.0.27 and manages replication from an active primary read-write InnoDB Cluster to multiple read-only replica clusters. For information on that solution, see Chapter 8, *MySQL InnoDB ClusterSet*.

- InnoDB Cluster is intended to be deployed in a local area network. Deploying a single InnoDB Cluster over a wide area network has a noticeable impact on write performance. A stable and low latency network is important for InnoDB Cluster member servers to communicate with each other using the underlying Group Replication technology for consensus on transactions. InnoDB ClusterSet, however, is designed to be deployed across multiple datacenters, with each InnoDB Cluster in a single datacenter and asynchronous replication channels linking them. For information on that solution, see Chapter 8, *MySQL InnoDB ClusterSet*.

- For AdminAPI operations, you can only connect to server instances in an InnoDB Cluster using TCP/IP connections and classic MySQL protocol. The use of Unix sockets and named pipes is not supported for AdminAPI operations, and the use of X Protocol is not supported for AdminAPI operations. The same limitations apply to connections between the server instances themselves.

  Note

  Client applications can use X Protocol and Unix sockets and named pipes to connect to instances in an InnoDB Cluster. The limitations only apply to administration operations using AdminAPI commands, and to connections between the instances.

- AdminAPI and InnoDB Cluster support the use of instances running MySQL Server 5.7. However, there are additional limitations for these instances, and some of the features described do not apply when you use them. Section 6.2.1, “Using Instances Running MySQL 5.7” lists the additional limitations.

- Concurrent data definition statements and data manipulation statements issued against the same object but on different servers is not supported when using multi-primary mode. During the issue of Data Definition Language (DDL) statements on an object, issuing concurrent Data Manipulation Language (DML) on the same object but from a different server instance has the risk of conflicting DDL executing on different instances not being detected. For more information, see Group Replication Limitations.

7.3 User Accounts for InnoDB Cluster

- Configuring InnoDB Cluster Administrator Accounts Manually
- Internal User Accounts Created by InnoDB Cluster

- Resetting Recovery Account Passwords

The member servers in an InnoDB Cluster make use of three types of user accounts. One InnoDB Cluster server configuration account is used to configure the server instances for the cluster. One or more InnoDB Cluster administrator accounts can be created for administrators to manage the server instances after the cluster has been set up. One or more MySQL Router accounts can be created for MySQL Router instances to connect to the cluster. Each of the user accounts must exist on all of the member servers in the InnoDB Cluster, with the same user name and the same password.

InnoDB Cluster server configuration account

This account is used to create and configure the member servers of an InnoDB Cluster. Each member server has only one server configuration account. The same user account name and password must be used on every member server in the cluster. You can use the `root` account on the servers for this purpose, but if you do this, the `root` account on every member server in the cluster must have the same password. This is not recommended for security reasons.

The preferred approach is to create the InnoDB Cluster server configuration account using a `dba.configureInstance()` command with the `clusterAdmin` option. For better security, specify the password at the interactive prompt, otherwise specify it using the `clusterAdminPassword` option. Create the same account, with the same user name and password, in the same way on every server instance that will be part of the InnoDB Cluster - both the instance to which you connect to create the cluster, and the instances that will join the cluster after that.

The `dba.configureInstance()` command grants the account the required permissions automatically. You may set up the account manually if you prefer, granting it the permissions listed in Configuring InnoDB Cluster Administrator Accounts Manually. The account needs full read and write privileges on the InnoDB Cluster metadata tables, in addition to full MySQL administrator privileges.

The InnoDB Cluster server configuration account that you create using the `dba.configureInstance()` operation is *not* replicated to other servers in the InnoDB Cluster. MySQL Shell disables binary logging for the `dba.configureInstance()` operation. This means that you must create the account on every server instance individually.

InnoDB Cluster administrator accounts

These accounts can be used to administer an InnoDB Cluster after you have completed the configuration process. You can set up more than one of them. Each account must exist on every member server in an InnoDB Cluster with the same user name and password.

To create an InnoDB Cluster administrator account for an InnoDB ClusterSet deployment, you issue a `*cluster*.setupAdminAccount()` command after you have added all the instances to that cluster. The command creates an account with the user name and password that you specify, with all the required permissions. A transaction to create an account with `*cluster*.setupAdminAccount()` is written to the binary log and sent to all the other server instances in the cluster to create the account on them.

Note

If the primary InnoDB Cluster was set up in a version before MySQL Shell 8.0.20,

the *cluster*.setupAdminAccount() command might have been used with the update option to update the privileges of the InnoDB Cluster server configuration account. This is a special use of the command that is not written to the binary log.

MySQL Router accounts

These accounts are used by MySQL Router to connect to server instances in an InnoDB Cluster. You can set up more than one of them. Each account must exist on every member server in an InnoDB Cluster with the same user name and password. The process to create a MySQL Router account is the same as for an InnoDB Cluster administrator account, but using a *cluster*.setupRouterAccount() command. For instructions to create or upgrade a MySQL Router account, see Section 6.10.2, "Configuring the MySQL Router User".

Configuring InnoDB Cluster Administrator Accounts Manually

If you want to manually configure an InnoDB Cluster administration user, that user requires the privileges listed here, all with GRANT OPTION.

Note

This list of privileges is based on the current version of MySQL Shell. The privileges are subject to change between releases. Therefore the recommended way to set up administration accounts is using the dba.configureInstance() or *cluster*.setupAdminAccount() operation.

Important

Each account used to administer an InnoDB Cluster, InnoDB ClusterSet, or InnoDB ReplicaSet deployment must exist on all the member server instances in the deployment, with the same user name, and the same password.

- Global privileges on *.* for RELOAD, SHUTDOWN, PROCESS, FILE, SELECT, SUPER, REPLICATION SLAVE, REPLICATION CLIENT, REPLICATION_APPLIER, CREATE USER, SYSTEM_VARIABLES_ADMIN, PERSIST_RO_VARIABLES_ADMIN, BACKUP_ADMIN, CLONE_ADMIN, and EXECUTE.

  Note

  SUPER includes the following required privileges: SYSTEM_VARIABLES_ADMIN, SESSION_VARIABLES_ADMIN, REPLICATION_SLAVE_ADMIN, GROUP_REPLICATION_ADMIN, REPLICATION_SLAVE_ADMIN, ROLE_ADMIN.

- Schema specific privileges for mysql_innodb_cluster_metadata.*, mysql_innodb_cluster_metadata_bkp.*, and mysql_innodb_cluster_metadata_previous.* are ALTER, ALTER ROUTINE, CREATE, CREATE ROUTINE, CREATE TEMPORARY TABLES, CREATE VIEW, DELETE, DROP, EVENT, EXECUTE, INDEX, INSERT, LOCK TABLES, REFERENCES, SHOW VIEW, TRIGGER, UPDATE; and for mysql.* are INSERT, UPDATE, DELETE.

If only read operations are needed, for example to create a user for monitoring purposes, an account with more restricted privileges can be used. To retrieve the privileges required by the user *your_user* to monitor InnoDB Cluster run the following command:

```
dba.configureInstance('root:password@localhost:3306',{'clusterAdmin':'your_user'})
```

where root is either the root user or a user with the required privileges to create users with the required privileges, and `your_user` is the InnoDB Cluster administrator user you want to create. The command returns the full list of privileges required by the administrator user.

For more information, see Account Management Statements.

Internal User Accounts Created by InnoDB Cluster

As part of using Group Replication, InnoDB Cluster creates internal recovery users which enable connections between the servers in the cluster. These users are internal to the cluster, and the user name of the generated users follows a naming scheme of `mysql_innodb_cluster_`*`server_id`*`@%`, where `server_id` is unique to the instance. In versions earlier than 8.0.17 the user name of the generated users followed a naming scheme of `mysql_innodb_cluster_r[`*`10_numbers`*`]`.

The hostname used for these internal users is set to '%'. Before v8.0.17, the `ipAllowlist` affected hostname behavior by creating one account per host in the `ipAllowlist`. For more information, see Creating an Allowlist of Servers.

Each internal user has a randomly generated password. From version 8.0.18, AdminAPI enables you to change the generated password for internal users. See Resetting Recovery Account Passwords. The randomly generated users are given the following grants:

```
GRANT REPLICATION SLAVE ON *.* to internal_user;
```

The internal user accounts are created on the seed instance and then replicated to the other instances in the cluster. The internal users are:

- Generated when creating a new cluster by issuing `dba.createCluster()`
- Generated when adding a new instance to the cluster by issuing `Cluster.addInstance()`
- Generated using the authentication plugin in use by the primary member

Before v8.0.17, `ipAllowlist` caused `Cluster.rejoinInstance()` to remove old internal users and generate new ones instead of reusing them.

For more information on the internal users required by Group Replication, see User Credentials For Distributed Recovery.

Resetting Recovery Account Passwords

From version 8.0.18, you can use the `Cluster.resetRecoveryAccountsPassword()` operation to reset the passwords for the internal recovery accounts created by InnoDB Cluster, for example to follow a custom password lifetime policy. Use the `Cluster.resetRecoveryAccountsPassword()` operation to reset the passwords for all internal recovery accounts used by the cluster. The operation sets a new random password for the internal recovery account on each instance which is online. If an instance cannot be reached, the operation fails. You can use the `force` option to ignore such instances, but this is not recommended, and it is safer to bring the instance back online before using this operation. This operation only applies to the passwords created by InnoDB Cluster and cannot be used to update manually created passwords.

Note

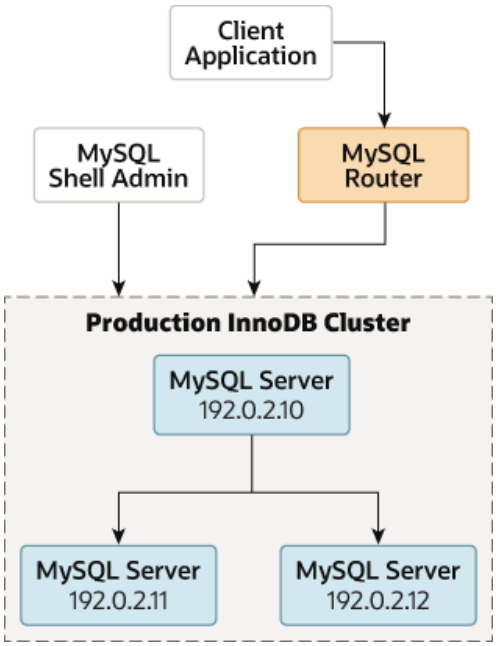
The user which executes this operation must have all the required administer privileges, in particular `CREATE USER`, in order to ensure that the password of recovery accounts can be changed regardless of the password verification-required policy. In other words, independent of whether the `password_require_current` system variable is enabled or not.

7.4 Deploying a Production InnoDB Cluster

When working in a production environment, the MySQL server instances which make up an InnoDB Cluster run on multiple host machines as part of a network rather than on single machine as described in Section 6.8, “AdminAPI MySQL Sandboxes”. Before proceeding with these instructions you must install the required software to each machine that you intend to add as a server instance to your cluster, see Section 6.2, “Installing AdminAPI Software Components”.

The following diagram illustrates the scenario you work with in this section:

Figure 7.2 Production Deployment



Important

Unlike a sandbox deployment, where all instances are deployed locally to one machine which AdminAPI has local file access to and can persist configuration changes, for a production deployment you must persist any configuration changes on the instance. How you do this depends on the version of MySQL running on the instance, see Section 6.2.4, “Persisting Settings”.

To pass a server's connection information to AdminAPI, use URI-like connection strings or a data dictionary; see Connecting to the Server Using URI-Like Strings or Key-Value Pairs. In this documentation, URI-like strings are shown.

This section assumes that you have:

- installed the MySQL components to your instances
- installed MySQL Shell and can connect by specifying instances
- created a suitable administration user

7.4.1 Pre-Checking Instance Configuration for InnoDB Cluster Usage

Before creating a production deployment from server instances you need to check that MySQL on each instance is correctly configured. The `dba.configureInstance()` function does this as part of configuring the instance, but you can optionally use the `dba.checkInstanceConfiguration(`*`instance`*`)` function . This checks whether the instance satisfies the requirements listed in Section 7.1, “InnoDB Cluster Requirements” without changing any

configuration on the instance. This does not check any data that is on the instance, see Checking Instance State for more information.

The user which you use to connect to the instance must have suitable privileges, for example as configured at Configuring InnoDB Cluster Administrator Accounts Manually. The following demonstrates issuing this in a running MySQL Shell:

```
mysql-js> dba.checkInstanceConfiguration('icadmin@ic-1:3306')
Please provide the password for 'icadmin@ic-1:3306': ***
Validating MySQL instance at ic-1:3306 for use in an InnoDB cluster...

This instance reports its own address as ic-1
Clients and other cluster members will communicate with it through this address by default.
If this is not correct, the report_host MySQL system variable should be changed.

Checking whether existing tables comply with Group Replication requirements...
No incompatible tables detected

Checking instance configuration...

Some configuration options need to be fixed:
+--------------------------+---------------+----------------+----------------------------------------
| Variable                 | Current Value | Required Value | Note
+--------------------------+---------------+----------------+----------------------------------------
| enforce_gtid_consistency | OFF           | ON             | Update read-only variable and restart the
| gtid_mode                | OFF           | ON             | Update read-only variable and restart the
| server_id                | 1             |                | Update read-only variable and restart the
+--------------------------+---------------+----------------+----------------------------------------

Please use the dba.configureInstance() command to repair these issues.

{
    "config_errors": [
        {
            "action": "restart",
            "current": "OFF",
            "option": "enforce_gtid_consistency",
            "required": "ON"
        },
        {
            "action": "restart",
            "current": "OFF",
            "option": "gtid_mode",
            "required": "ON"
        },
        {
            "action": "restart",
            "current": "1",
            "option": "server_id",
            "required": ""
        }
    ],
    "status": "error"
}
```

Repeat this process for each server instance that you plan to use as part of your cluster. The report generated after running `dba.checkInstanceConfiguration()` provides information about any configuration changes that will be required to use the instance in an InnoDB Cluster deployment. The `action` field in the `config_error` section of the report tells you whether MySQL on the instance requires a restart to detect any change made to the configuration file.

7.4.2 Configuring Production Instances for InnoDB Cluster Usage

AdminAPI provides the `dba.configureInstance()` function that checks if an instance is suitably configured for InnoDB Cluster usage, and configures the instance if it finds any settings which are not compatible with InnoDB Cluster. You run the `dba.configureInstance()` command against an instance and it checks all of the settings required to enable the instance to be used for InnoDB Cluster usage. If the instance does not require configuration changes, there is no need to modify the

configuration of the instance, and the `dba.configureInstance()` command output confirms that the instance is ready for InnoDB Cluster usage.

If any changes are required to make the instance compatible with InnoDB Cluster, a report of the incompatible settings is displayed, and you can choose to let the command make the changes to the instance's option file. Depending on the way MySQL Shell is connected to the instance, and the version of MySQL running on the instance, you can make these changes permanent by persisting them to a remote instance's option file, see Section 6.2.4, “Persisting Settings”.

Instances which do not support persisting configuration changes automatically require that you configure the instance locally, see Configuring Instances with `dba.configureLocalInstance()`. Alternatively you can make the changes to the instance's option file manually, see Using Option Files for more information. Regardless of the way you make the configuration changes, you might have to restart MySQL to ensure the configuration changes are detected.

The syntax of the `dba.configureInstance()` command is:

```
dba.configureInstance([instance][, options])
```

where *`instance`* is an instance definition, and *`options`* is a data dictionary with additional options to configure the operation. The operation returns a descriptive text message about the result.

The *`instance`* definition is the connection data for the instance. For example:

```
dba.configureInstance('user@example:3306')
```

For more information, see Connecting to the Server Using URI-Like Strings or Key-Value Pairs. If the target instance already belongs to an InnoDB Cluster an error is generated and the process fails.

The options dictionary can contain the following:

- `mycnfPath` - the path to the MySQL option file of the instance. Note that InnoDB Cluster only supports a single option file for server instances, and does not support the use of the `--defaults-extra-file` option to specify an additional option file. For any AdminAPI operation working with the instance's option file the main file must be specified.
- `outputMycnfPath` - alternative output path to write the MySQL option file of the instance.
- `password` - the password to be used by the connection.
- `clusterAdmin` - the name of an InnoDB Cluster administrator user to be created. The supported format is the standard MySQL account name format. Supports identifiers or strings for the user name and host name. By default if unquoted it assumes input is a string. See Section 6.4, “Creating User Accounts for AdminAPI”.
- `clusterAdminPassword` - the password for the InnoDB Cluster administrator account being created using `clusterAdmin`. Although you can specify using this option, this is a potential security risk. If you do not specify this option, but do specify the `clusterAdmin` option, you are prompted for the password at the interactive prompt.
- `clearReadOnly` - a Boolean value used to confirm that `super_read_only` should be set to off, see Instance Configuration in Super Read-only Mode. This option is deprecated, and scheduled for removal in a future version.
- `interactive` - a Boolean value used to disable the interactive wizards in the command execution, so that prompts are not provided to the user and confirmation prompts are not shown.
- `restart` - a Boolean value used to indicate that a remote restart of the target instance should be performed to finalize the operation.

Although the connection password can be contained in the instance definition, this is insecure and not recommended. Use the MySQL Shell Section 4.4, “Pluggable Password Store” to store instance passwords securely.

Once `dba.configureInstance()` is issued against an instance, the command checks if the instance's settings are suitable for InnoDB Cluster usage. A report is displayed which shows the settings required by InnoDB Cluster . If the instance does not require any changes to its settings you can use it in an InnoDB Cluster, and can proceed to Section 7.4.3, “Creating an InnoDB Cluster”. If the instance's settings are not valid for InnoDB Cluster usage the `dba.configureInstance()` command displays the settings which require modification. Before configuring the instance you are prompted to confirm the changes shown in a table with the following information:

- `Variable` - the invalid configuration variable.
- `Current Value` - the current value for the invalid configuration variable.
- `Required Value` - the required value for the configuration variable.

How you proceed depends on whether the instance supports persisting settings, see Section 6.2.4, “Persisting Settings”. When `dba.configureInstance()` is issued against the MySQL instance which MySQL Shell is currently running on, in other words the local instance, it attempts to automatically configure the instance. When `dba.configureInstance()` is issued against a remote instance, if the instance supports persisting configuration changes automatically, you can choose to do this. If a remote instance does not support persisting the changes to configure it for InnoDB Cluster usage, you have to configure the instance locally. See Configuring Instances with `dba.configureLocalInstance()`.

In general, a restart of the instance is not required after `dba.configureInstance()` configures the option file, but for some specific settings a restart might be required. This information is shown in the report generated after issuing `dba.configureInstance()`. If the instance supports the `RESTART` statement, MySQL Shell can shutdown and then start the instance. This ensures that the changes made to the instance's option file are detected by mysqld. For more information see `RESTART`.

Note

After executing a `RESTART` statement, the current connection to the instance is lost. If auto-reconnect is enabled, the connection is reestablished after the server restarts. Otherwise, the connection must be reestablished manually.

The `dba.configureInstance()` method verifies that a suitable user is available for cluster usage, which is used for connections between members of the cluster, see Section 6.4, “Creating User Accounts for AdminAPI”.

If you do not specify a user to administer the cluster, in interactive mode a wizard enables you to choose one of the following options:

- enable remote connections for the root user, not recommended in a production environment
- create a new user
- no automatic configuration, in which case you need to manually create the user

Tip

If the instance has `super_read_only=ON` then you might need to confirm that AdminAPI can set `super_read_only=OFF`. See Instance Configuration in Super Read-only Mode for more information.

Configuring Instances with `dba.configureLocalInstance()`

Instances which do not support persisting configuration changes automatically (see Section 6.2.4, “Persisting Settings”) require you to connect to the server, run MySQL Shell, connect to the instance locally and issue `dba.configureLocalInstance()`. This enables MySQL Shell to modify the instance's option file after running the following commands against a remote instance:

- `dba.configureInstance()`
- `dba.createCluster()`
- *`Cluster`*`.addInstance()`
- *`Cluster`*`.removeInstance()`
- *`Cluster`*`.rejoinInstance()`

Important

Failing to persist configuration changes to an instance's option file can result in the instance not rejoining the cluster after the next restart.

The relevant configuration options for the instance, particularly the Group Replication configuration options, must be in a single option file. InnoDB Cluster only supports a single option file for server instances and does not support the use of the `--defaults-extra-file` option to specify an additional option file. For any AdminAPI operation working with the instance's option file the main file must be specified.

The recommended method to persist configuration changes is to log in to the remote machine, for example using SSH, run MySQL Shell as the root user and then connect to the local MySQL server. For example, use the `--uri` option to connect to the local *`instance`*:

```
$> sudo -i mysqlsh --uri=instance
```

Alternatively use the `\connect` command to log in to the local instance. Then issue `dba.configureInstance(`*`instance`*`)`, where *`instance`* is the connection information to the local instance, to persist any changes made to the local instance's option file.

```
mysql-js> dba.configureLocalInstance('icadmin@ic-2:3306')
```

Repeat this process for each instance in the cluster which does not support persisting configuration changes automatically. For example if you add 2 instances to a cluster which do not support persisting configuration changes automatically, you must connect to each server and persist the configuration changes required for InnoDB Cluster before the instance restarts. Similarly if you modify the cluster structure, for example changing the number of instances, you need to repeat this process for each server instance to update the InnoDB Cluster metadata accordingly for each instance in the cluster.

Instance Configuration in Super Read-only Mode

Whenever Group Replication stops, the `super_read_only` variable is set to `ON` to ensure no writes are made to the instance. When you try to use such an instance with the following AdminAPI commands you are given the choice to set `super_read_only=OFF` on the instance:

- `dba.configureInstance()`
- `dba.configureLocalInstance()`
- `dba.dropMetadataSchema()`

When AdminAPI encounters an instance which has `super_read_only=ON`, in interactive mode you are given the choice to set `super_read_only=OFF`. For example:

```
mysql-js> var myCluster = dba.dropMetadataSchema()
Are you sure you want to remove the Metadata? [y/N]: y
The MySQL instance at 'localhost:3310' currently has the super_read_only system
variable set to protect it from inadvertent updates from applications. You must
first unset it to be able to perform any changes to this instance.
For more information see:
https://dev.mysql.com/doc/refman/en/server-system-variables.html#sysvar_super_read_only.
```

```
Do you want to disable super_read_only and continue? [y/N]: y

Metadata Schema successfully removed.
```

The number of current active sessions to the instance is shown. You must ensure that no applications can write to the instance inadvertently. By answering `y` you confirm that AdminAPI can write to the instance. If there is more than one open session to the instance listed, exercise caution before permitting AdminAPI to set `super_read_only=OFF`.

7.4.3 Creating an InnoDB Cluster

Once you have prepared your instances, use the `dba.createCluster()` function to create the cluster, using the instance which MySQL Shell is connected to as the seed instance for the cluster. The seed instance is replicated to the other instances that you add to the cluster, making them replicas of the seed instance. In this procedure the ic-1 instance is used as the seed. When you issue `dba.createCluster(name)` MySQL Shell creates a classic MySQL protocol session to the server instance connected to the MySQL Shell's current global session. For example, to create a cluster called *`testCluster`* and assign the returned cluster to a variable called `cluster`:

```
mysql-js> var cluster = dba.createCluster('testCluster')
Validating instance at icadmin@ic-1:3306...
This instance reports its own address as ic-1
Instance configuration is suitable.
Creating InnoDB cluster 'testCluster' on 'icadmin@ic-1:3306'...
Adding Seed Instance...
Cluster successfully created. Use Cluster.addInstance() to add MySQL instances.
At least 3 instances are needed for the cluster to be able to withstand up to
one server failure.
```

This pattern of assigning the returned cluster to a variable enables you to then execute further operations against the cluster using the Cluster object's methods. The returned Cluster object uses a new session, independent from the MySQL Shell's global session. This ensures that if you change the MySQL Shell global session, the Cluster object maintains its session to the instance.

To be able to administer a cluster, you must ensure that you have a suitable user which has the required privileges. The recommended approach is to create an administration user. If you did not create an administration user when configuring your instances, use the *`Cluster`*`.setupAdminAccount()` operation. For example to create a user named *`icadmin`* that can administer the InnoDB Cluster assigned to the variable *`cluster`*, issue:

```
mysql-js> cluster.setupAdminAccount("icadmin")
```

See Configuring InnoDB Cluster Administrator Accounts Manually for more information on InnoDB Cluster administrator accounts.

The `dba.createCluster()` operation supports MySQL Shell's `interactive` option. When `interactive` is on, prompts appear in the following situations:

- If the instance belongs to a Group Replication group, and `adoptFromGr: true` is not set as an option, you are asked if you want to adopt the replication group.
- If `force: true` is not set as an option, you are asked to confirm the creation of a multi-primary cluster.

When you run `dba.createCluster()`, and when you add a further server instance to the InnoDB Cluster by running *`Cluster`*`.addInstance()`, the following errors are logged to the MySQL server instance's error log. These messages are harmless and relate to the way AdminAPI starts Group Replication:

```
2020-02-10T10:53:43.727246Z 12 [ERROR] [MY-011685] [Repl] Plugin
group_replication reported: 'The group name option is mandatory'
2020-02-10T10:53:43.727292Z 12 [ERROR] [MY-011660] [Repl] Plugin
group_replication reported: 'Unable to start Group Replication on boot'
```

Note

If you encounter an error related to metadata being inaccessible you might have the loopback network interface configured. For correct InnoDB Cluster usage disable the loopback interface.

To check the cluster has been created, use the cluster instance's `status()` function. See Checking a cluster's Status with *`Cluster`*`.status()`.

Tip

Once server instances belong to a cluster it is important to only administer them using MySQL Shell and AdminAPI. Attempting to manually change the configuration of Group Replication on an instance once it has been added to a cluster is not supported. Similarly, modifying server variables critical to InnoDB Cluster, such as `server_uuid`, after an instance is configured using AdminAPI is not supported.

When you create a cluster using MySQL Shell 8.0.14 and later, you can set the amount of time to wait before instances are expelled from the cluster, for example when they become unreachable. Pass the `expelTimeout` option to the `dba.createCluster()` operation, which configures the `group_replication_member_expel_timeout` variable on the seed instance. The `expelTimeout` option can take an integer value in the range of 0 to 3600. All instances running MySQL server 8.0.13 and later which are added to a cluster with `expelTimeout` configured are automatically configured to have the same `expelTimeout` value as configured on the seed instance.

For information on the other options which you can pass to `dba.createCluster()`, see Section 7.9, “Modifying or Dissolving an InnoDB Cluster”.

As of MySQL Shell 8.0.33, it is possible to enable or disable `group_replication_paxos_single_leader` using `dba.createCluster()`.

Note

This can only be set by MySQL Shell on MySQL Server 8.0.31, or higher, because MySQL Shell requires the information provided by `WRITE_CONSENSUS_SINGLE_LEADER_CAPABLE` in the `replication_group_communication_information` table, which was introduced in MySQL 8.0.31.

InnoDB Cluster replicationAllowedHost

When you create a cluster using MySQL Shell 8.0.28 and later, if you have security requirements that all accounts created automatically by AdminAPI have strict authentication requirements, you can set a value for the `replicationAllowedHost` cluster configuration option. The `replicationAllowedHost` option means that all accounts created automatically can only connect from allowed hosts, using strict subnet-based filtering. Previously, the internal user accounts created by InnoDB Cluster, by default, were accessible from anywhere.

The `replicationAllowedHost` option can take a string value. For example, to create a cluster called `testCluster` and set the `replicationAllowedHost` option to `192.0.2.0/24`, issue:

```
mysql-js> dba.createCluster('testCluster', {replicationAllowedHost:'192.0.2.0/24'})
```

Configuring the Communication Stack

As of MySQL Shell 8.0.30, InnoDB Cluster supports the MySQL communication stack introduced for Group Replication in MySQL 8.0.27.

The option, `communicationStack: XCOM|MYSQL` sets the value of the Group Replication system variable `group_replication_communication_stack`.

For example:

```
mysql-js> dba.createCluster("testCluster", {communicationStack: "xcom"})
```

The `MYSQL` communication stack is the default for all new clusters created for MySQL 8.0.27, or higher.

For more information, see Section 7.5.9, “Configuring the Group Replication Communication Stack”.

7.4.4 Adding Instances to an InnoDB Cluster

You need a minimum of three instances in an InnoDB Cluster to make it tolerant to the failure of one instance. Adding further instances increases the tolerance to failure of an InnoDB Cluster.

From version 8.0.17, Group Replication implements compatibility policies which consider the patch version of the instances, and the *`Cluster`*`.addInstance()` operation detects this and in the event of an incompatibility the operation terminates with an error. See Checking the MySQL Version on Instances and Combining Different Member Versions in a Group.

If the instance already contains data, use the `cluster.checkInstanceState()` function first to verify the existing data does not prevent the instance from joining a cluster. See Checking Instance State.

Use the *`Cluster`*`.addInstance(`*`instance`*`)` function to add an instance to the cluster, where *`instance`* is connection information to a configured instance, see Section 7.4.2, “Configuring Production Instances for InnoDB Cluster Usage”. For example:

```
mysql-js> cluster.addInstance('icadmin@ic-2:3306')
A new instance will be added to the InnoDB cluster. Depending on the amount of
data on the cluster this might take from a few seconds to several hours.
Please provide the password for 'icadmin@ic-2:3306': ********
Adding instance to the cluster ...
Validating instance at ic-2:3306...
This instance reports its own address as ic-2
Instance configuration is suitable.
The instance 'icadmin@ic-2:3306' was successfully added to the cluster.
```

The options dictionary of the `addInstance(instance[, options])` function provides the following attributes:

- label: an identifier for the instance being added.
- recoveryMethod: Preferred method of state recovery. May be auto, clone, or incremental. Default is auto.
- waitRecovery: Integer value to indicate if the command shall wait for the recovery process to finish and its verbosity level.
- password: the instance connection password.
- memberSslMode: SSL mode used on the instance.
- ipWhitelist: The list of hosts allowed to connect to the instance for group replication. Deprecated.
- ipAllowlist: The list of hosts allowed to connect to the instance for group replication.
- localAddress: string value with the Group Replication local address to be used instead of the automatically generated one.
- groupSeeds: string value with a comma-separated list of the Group Replication peer addresses to be used instead of the automatically generated one. Deprecated and ignored.
- interactive: Boolean value used to disable/enable the wizards in the command execution, meaning that prompts and confirmations will be provided or not according to the value set. The default value is equal to MySQL Shell wizard mode.
- exitStateAction: string value indicating the group replication exit state action.

- memberWeight: integer value with a percentage weight for automatic primary election on failover.
- autoRejoinTries: integer value to define the number of times an instance will attempt to rejoin the cluster after being expelled.

When a new instance is added to the cluster, the local address for this instance is automatically added to the `group_replication_group_seeds` variable on all online cluster instances in order to allow them to use the new instance to rejoin the group, if needed.

Note

The instances listed in `group_replication_group_seeds` are used according to the order in which they appear in the list. This ensures user specified settings are used first and preferred. See Section 7.5.2, “Customizing InnoDB Cluster Member Servers” for more information.

If you are using MySQL 8.0.17 or later you can choose how the instance recovers the transactions it requires to synchronize with the cluster. Only when the joining instance has recovered all of the transactions previously processed by the cluster can it then join as an online instance and begin processing transactions. For more information, see Section 7.4.6, “Using MySQL Clone with InnoDB Cluster”.

Also in 8.0.17 and later, you can configure how *`Cluster`*`.addInstance()` behaves, letting recovery operations proceed in the background or monitoring different levels of progress in MySQL Shell.

Depending on which option you choose to recover the instance from the cluster, you see different output in MySQL Shell. Suppose that you are adding the instance ic-2 to the cluster, and ic-1 is the seed or donor.

- When you use MySQL Clone to recover an instance from the cluster, the output looks like:

```
Validating instance at ic-2:3306...
This instance reports its own address as ic-2:3306
Instance configuration is suitable.
A new instance will be added to the InnoDB cluster. Depending on the amount of
data on the cluster this might take from a few seconds to several hours.
Adding instance to the cluster...
Monitoring recovery process of the new cluster member. Press ^C to stop monitoring
and let it continue in background.
Clone based state recovery is now in progress.
NOTE: A server restart is expected to happen as part of the clone process. If the
server does not support the RESTART command or does not come back after a
while, you may need to manually start it back.
* Waiting for clone to finish...
NOTE: ic-2:3306 is being cloned from ic-1:3306
** Stage DROP DATA: Completed
** Clone Transfer
FILE COPY  ############################################################  100%  Completed
PAGE COPY  ############################################################  100%  Completed
REDO COPY  ############################################################  100%  Completed
NOTE: ic-2:3306 is shutting down...
* Waiting for server restart... ready
* ic-2:3306 has restarted, waiting for clone to finish...
** Stage RESTART: Completed
* Clone process has finished: 2.18 GB transferred in 7 sec (311.26 MB/s)
State recovery already finished for 'ic-2:3306'
The instance 'ic-2:3306' was successfully added to the cluster.
```

  The warnings about server restart should be observed, you might have to manually restart an instance. See RESTART Statement.

- When you use incremental recovery to recover an instance from the cluster, the output looks like:

```
Incremental distributed state recovery is now in progress.
* Waiting for incremental recovery to finish...
NOTE: 'ic-2:3306' is being recovered from 'ic-1:3306'
* Distributed recovery has finished
```

To cancel the monitoring of the recovery phase, issue **CONTROL+C**. This stops the monitoring but the recovery process continues in the background. The `waitRecovery` integer option can be used with the *`Cluster`*`.addInstance()` operation to control the behavior of the command regarding the recovery phase. The following values are accepted:

- 0: do not wait and let the recovery process finish in the background;
- 1: wait for the recovery process to finish;
- 2: wait for the recovery process to finish; and show detailed static progress information;
- 3: wait for the recovery process to finish; and show detailed dynamic progress information (progress bars);

By default, if the standard output which MySQL Shell is running on refers to a terminal, the `waitRecovery` option defaults to 3. Otherwise, it defaults to 2. See Monitoring Recovery Operations.

To verify the instance has been added, use the cluster instance's `status()` function. For example this is the status output of a sandbox cluster after adding a second instance:

```
mysql-js> cluster.status()
{
    "clusterName": "testCluster",
    "defaultReplicaSet": {
        "name": "default",
        "primary": "ic-1:3306",
        "ssl": "REQUIRED",
        "status": "OK_NO_TOLERANCE",
        "statusText": "Cluster is NOT tolerant to any failures.",
        "topology": {
            "ic-1:3306": {
                "address": "ic-1:3306",
                "mode": "R/W",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            },
            "ic-2:3306": {
                "address": "ic-2:3306",
                "mode": "R/O",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            }
        }
    },
    "groupInformationSourceMember": "mysql://icadmin@ic-1:3306"
}
```

How you proceed depends on whether the instance is local or remote to the instance MySQL Shell is running on, and whether the instance supports persisting configuration changes automatically, see Section 6.2.4, “Persisting Settings”. If the instance supports persisting configuration changes automatically, you do not need to persist the settings manually and can either add more instances or continue to the next step. If the instance does not support persisting configuration changes automatically, you have to configure the instance locally. See Configuring Instances with `dba.configureLocalInstance()`. This is essential to ensure that instances rejoin the cluster in the event of leaving the cluster.

Tip

If the instance has `super_read_only=ON` then you might need to confirm that AdminAPI can set `super_read_only=OFF`. See Instance Configuration in Super Read-only Mode for more information.

Once you have your cluster deployed you can configure MySQL Router to provide high availability, see Section 6.10, “Using MySQL Router with AdminAPI, InnoDB Cluster, and InnoDB ReplicaSet”.

Checking Instance State

The `cluster.checkInstanceState()` function can be used to verify the existing data on an instance does not prevent it from joining a cluster. This process works by validating the instance's global transaction identifier (GTID) state compared to the GTIDs already processed by the cluster. For more information on GTIDs see GTID Format and Storage. This check enables you to determine if an instance which has processed transactions can be added to the cluster.

The following demonstrates issuing this in a running MySQL Shell:

```
mysql-js> cluster.checkInstanceState('icadmin@ic-4:3306')
```

The output of this function can be one of the following:

- OK new: the instance has not executed any GTID transactions, therefore it cannot conflict with the GTIDs executed by the cluster
- OK recoverable: the instance has executed GTIDs which do not conflict with the executed GTIDs of the cluster seed instances
- ERROR diverged: the instance has executed GTIDs which diverge with the executed GTIDs of the cluster seed instances
- ERROR lost_transactions: the instance has more executed GTIDs than the executed GTIDs of the cluster seed instances

Instances with an OK status can be added to the cluster because any data on the instance is consistent with the cluster. In other words the instance being checked has not executed any transactions which conflict with the GTIDs executed by the cluster, and can be recovered to the same state as the rest of the cluster instances.

7.4.5 Configuring InnoDB Cluster Ports

Instances that belong to a cluster use different ports for different types of communication. If you are using the `XCOM` communication stack, in addition to the default `port` at 3306, which is used for client connections over classic MySQL protocol, and the `mysqlx_port`, which defaults to 33060 and is used for X Protocol client connections, there is also a port for internal connections between the instances in the cluster which is not used for client connections. This port is configured by the `localAddress` option, which configures the `group_replication_local_address` system variable, and this port must be open so that the instances in the cluster can communicate with each other. For example, if your firewall is blocking this port then the instances cannot communicate with each other, and the cluster cannot function. Similarly, if your instances are using SELinux, you need to ensure that all of the required ports used by InnoDB Cluster are open so that the instances can communicate with each other. See Setting the TCP Port Context for MySQL Features and MySQL Shell Ports.

When you create a cluster or add instances to a cluster, by default the `localAddress` port is calculated by multiplying the target instance's `port` value by 10 and then adding one to the result. For example, when the `port` of the target instance is the default value of 3306, the calculated `localAddress` port is 33061. You should ensure that port numbers used by your cluster instances are compatible with the way `localAddress` is calculated. For example, if the server instance being used to create a cluster has a `port` number higher than 6553, the `dba.createCluster()` operation fails because the calculated `localAddress` port number exceeds the maximum valid port which is 65535. To avoid this situation either use a lower `port` value on the instances you use for InnoDB Cluster, or manually assign the `localAddress` value, for example:

```
mysql-js> dba.createCluster('testCluster', {'localAddress':'icadmin@ic-1:33061'}
```

If you are using the `MYSQL` communication stack, the localAddress value is generated automatically using the same network address as the MySQL server. An extra internal port/address is not required. See Section 7.5.9, “Configuring the Group Replication Communication Stack”.

`localAddress` can be defined manually, but the port used must be one MySQL is listening on, as defined by `bind_address`.

7.4.6 Using MySQL Clone with InnoDB Cluster

In MySQL 8.0.17, InnoDB Cluster integrates the MySQL Clone plugin to provide automatic provisioning of joining instances. The process of retrieving the cluster's data so that the instance can synchronize with the cluster is called distributed recovery. When an instance needs to recover a cluster's transactions we distinguish between the *donor*, which is the cluster instance that provides the data, and the *receiver*, which is the instance that receives the data from the donor. In previous versions, Group Replication provided only asynchronous replication to recover the transactions required for the joining instance to synchronize with the cluster so that it could join the cluster. For a cluster with a large amount of previously processed transactions it could take a long time for the new instance to recover all of the transactions before being able to join the cluster. Or a cluster which had purged GTIDs, for example as part of regular maintenance, could be missing some of the transactions required to recover the new instance. In such cases the only alternative was to manually provision the instance using tools such as MySQL Enterprise Backup, as shown in Using MySQL Enterprise Backup with Group Replication.

MySQL Clone provides an alternative way for an instance to recover the transactions required to synchronize with a cluster. Instead of relying on asynchronous replication to recover the transactions, MySQL Clone takes a snapshot of the data on the donor instance and then transfers the snapshot to the receiver.

Warning

All previous data in the receiver is destroyed during a clone operation. All MySQL settings not stored in tables are however maintained.

Once a clone operation has transferred the snapshot to the receiver, if the cluster has processed transactions while the snapshot was being transferred, asynchronous replication is used to recover any required data for the receiver to be synchronized with the cluster. This can be much more efficient than the instance recovering all of the transactions using asynchronous replication, and avoids any issues caused by purged GTIDs, enabling you to quickly provision new instances for InnoDB Cluster. For more information, see The Clone Plugin and Cloning for Distributed Recovery

In contrast to using MySQL Clone, incremental recovery is the process where an instance joining a cluster uses only asynchronous replication to recover an instance from the cluster. When an InnoDB Cluster is configured to use MySQL Clone, instances which join the cluster use either MySQL Clone or incremental recovery to recover the cluster's transactions. By default, the cluster automatically chooses the most suitable method, but you can optionally configure this behavior, for example to force cloning, which replaces any transactions already processed by the joining instance. When you are using MySQL Shell in interactive mode, the default, if the cluster is not sure it can proceed with recovery it provides an interactive prompt. This section describes the different options you are offered, and the different scenarios which influence which of the options you can choose.

In addition, the output of `Cluster.status()` for members in `RECOVERING` state includes recovery progress information to enable you to easily monitor recovery operations, whether they are using MySQL Clone or incremental recovery. InnoDB Cluster provides additional information about instances using MySQL Clone in the output of `Cluster.status()`.

7.4.6.1 Working with a Cluster that uses MySQL Clone

An InnoDB Cluster that uses MySQL Clone provides the following additional behavior.

`dba.createCluster()` and MySQL Clone

From version 8.0.17, by default when a new cluster is created on an instance where the MySQL Clone plugin is available then it is automatically installed and the cluster is configured to support cloning. The InnoDB Cluster recovery accounts are created with the required `BACKUP_ADMIN` privilege.

Set the `disableClone` Boolean option to `true` to disable MySQL Clone for the cluster. In this case a metadata entry is added for this configuration and the MySQL Clone plugin is uninstalled if it is installed. You can set the `disableClone` option when you issue `dba.createCluster()`, or at any time when the cluster is running using `Cluster.setOption()`.

`Cluster.addInstance(instance)` and MySQL Clone

MySQL Clone can be used for a joining *`instance`* if the new instance is running MySQL 8.0.17 or later, and there is at least one donor in the cluster (included in the `group_replication_group_seeds` list) running MySQL 8.0.17 or later. A cluster using MySQL Clone follows the behavior documented at Section 7.4.4, “Adding Instances to an InnoDB Cluster”, with the addition of a possible choice of how to transfer the data required to recover the instance from the cluster. How `Cluster.addInstance(instance)` behaves depends on the following factors:

- Whether MySQL Clone is supported.
- Whether incremental recovery is possible or not, which depends on the availability of binary logs. For example, if a donor instance has all binary logs required (`GTID_PURGED` is empty) then incremental recovery is possible. If no cluster instance has all binary logs required then incremental recovery is not possible.
- Whether incremental recovery is appropriate or not. Even though incremental recovery might be possible, because it has the potential to clash with data already on the instance, the GTID sets on the donor and receiver are checked to make sure that incremental recovery is appropriate. The following are possible results of the comparison:
  - New: the receiver has an empty `GTID_EXECUTED` GTID set
  - Identical: the receiver has a GTID set identical to the donor’s GTID set
  - Recoverable: the receiver has a GTID set that is missing transactions but these can be recovered from the donor
  - Irrecoverable: the donor has a GTID set that is missing transactions, possibly they have been purged
  - Diverged: the GTID sets of the donor and receiver have diverged

  When the result of the comparison is determined to be Identical or Recoverable, incremental recovery is considered appropriate. When the result of the comparison is determined to be Irrecoverable or Diverged, incremental recovery is not considered appropriate.

  For an instance considered New, incremental recovery cannot be considered appropriate because it is impossible to determine if the binary logs have been purged, or even if the `GTID_PURGED` and `GTID_EXECUTED` variables were reset. Alternatively, it could be that the server had already processed transactions before binary logs and GTIDs were enabled. Therefore in interactive mode, you have to confirm that you want to use incremental recovery.
- The state of the `gtidSetIsComplete` option. If you are sure a cluster has been created with a complete GTID set, and therefore instances with empty GTID sets can be added to it without extra confirmations, set the cluster level `gtidSetIsComplete` Boolean option to `true`.

Warning

Setting the `gtidSetIsComplete` option to `true` means that joining servers are recovered regardless of any data they contain, use with caution. If you try to add an instance which has applied transactions you risk data corruption.

The combination of these factors influence how instances join the cluster when you issue `Cluster.addInstance()`. The `recoveryMethod` option is set to `auto` by default, which means that in MySQL Shell's interactive mode, the cluster selects the best way to recover the instance from

the cluster, and the prompts advise you how to proceed. In other words the cluster recommends using MySQL Clone or incremental recovery based on the best approach and what the server supports. If you are not using interactive mode and are scripting MySQL Shell, you must set `recoveryMethod` to the type of recovery you want to use - either `clone` or `incremental`. This section explains the different possible scenarios.

When you are using MySQL Shell in interactive mode, the main prompt with all of the possible options for adding the instance is:

```
Please select a recovery method [C]lone/[I]ncremental recovery/[A]bort (default Clone):
```

Depending on the factors mentioned, you might not be offered all of these options. The scenarios described later in this section explain which options you are offered. The options offered by this prompt are:

- *Clone*: choose this option to clone the donor to the instance which you are adding to the cluster, deleting any transactions the instance contains. The MySQL Clone plugin is automatically installed. The InnoDB Cluster recovery accounts are created with the required `BACKUP_ADMIN` privilege. Assuming you are adding an instance which is either empty (has not processed any transactions) or which contains transactions you do not want to retain, select the Clone option. The cluster then uses MySQL Clone to completely overwrite the joining instance with a snapshot from an donor cluster member. To use this method by default and disable this prompt, set the cluster's `recoveryMethod` option to `clone`.
- *Incremental recovery* choose this option to use incremental recovery to recover all transactions processed by the cluster to the joining instance using asynchronous replication. Incremental recovery is appropriate if you are sure all updates ever processed by the cluster were done with GTIDs enabled, there are no purged transactions and the new instance contains the same GTID set as the cluster or a subset of it. To use this method by default, set the `recoveryMethod` option to `incremental`.

The combination of factors mentioned influences which of these options is available at the prompt as follows:

Note

If the `group_replication_clone_threshold` system variable has been manually changed outside of AdminAPI, then the cluster might decide to use Clone recovery instead of following these scenarios.

- In a scenario where
  - incremental recovery is possible
  - incremental recovery is not appropriate
  - Clone is supported

  you can choose between any of the options. It is recommended that you use MySQL Clone, the default.
- In a scenario where
  - incremental recovery is possible
  - incremental recovery is appropriate

  you are not provided with the prompt, and incremental recovery is used.
- In a scenario where
  - incremental recovery is possible

  - incremental recovery is not appropriate
  - Clone is not supported or is disabled

  you cannot use MySQL Clone to add the instance to the cluster. You are provided with the prompt, and the recommended option is to proceed with incremental recovery.

- In a scenario where
  - incremental recovery is not possible
  - Clone is not supported or is disabled

  you cannot add the instance to the cluster and an `ERROR: The target instance must be either cloned or fully provisioned before it can be added to the target cluster. Cluster.addInstance: Instance provisioning required (RuntimeError)` is shown. This could be the result of binary logs being purged from all cluster instances. It is recommended to use MySQL Clone, by either upgrading the cluster or setting the `disableClone` option to `false`.

- In a scenario where
  - incremental recovery is not possible
  - Clone is supported

  you can only use MySQL Clone to add the instance to the cluster. This could be the result of the cluster missing binary logs, for example when they have been purged.

Once you select an option from the prompt, by default the progress of the instance recovering the transactions from the cluster is displayed. This monitoring enables you to check the recovery phase is working and also how long it should take for the instance to join the cluster and come online. To cancel the monitoring of the recovery phase, issue **CONTROL+C**.

Cluster.checkInstanceState() and MySQL Clone

When the *Cluster*.checkInstanceState() operation is run to verify an instance against a cluster that is using MySQL Clone, if the instance does not have the binary logs, for example because they were purged but Clone is available and not disabled (`disableClone` is `false`) the operation provides a warning that the Clone can be used. For example:

```
The cluster transactions cannot be recovered on the instance, however,
Clone is available and can be used when adding it to a cluster.

{
"reason": "all_purged",
"state": "warning"
}
```

Similarly, on an instance where Clone is either not available or has been disabled and the binary logs are not available, for example because they were purged, then the output includes:

```
The cluster transactions cannot be recovered on the instance.

{
"reason": "all_purged",
"state": "warning"
}
```

dba.checkInstanceConfiguration() and MySQL Clone

When the `dba.checkInstanceConfiguration()` operation is run against an instance that has MySQL Clone available but it is disabled, a warning is displayed.

7.4.7 Adopting a Group Replication Deployment

If you have an existing deployment of Group Replication and you want to use it to create a cluster, pass the `adoptFromGR` option to the `dba.createCluster()` function. The created InnoDB Cluster matches whether the replication group is running as single-primary or multi-primary.

To adopt an existing Group Replication group, connect to a group member using MySQL Shell. In the following example a single-primary group is adopted. We connect to `gr-member-2`, a secondary instance, while `gr-member-1` is functioning as the group's primary. Create a cluster using `dba.createCluster()`, passing in the `adoptFromGR` option. For example:

```
mysql-js> var cluster = dba.createCluster('prodCluster', {adoptFromGR: true});

A new InnoDB cluster will be created on instance 'root@gr-member-2:3306'.

Creating InnoDB cluster 'prodCluster' on 'root@gr-member-2:3306'...
Adding Seed Instance...

Cluster successfully created. Use cluster.addInstance() to add MySQL instances.
At least 3 instances are needed for the cluster to be able to withstand up to
one server failure.
```

Tip

If the instance has `super_read_only=ON` then you might need to confirm that AdminAPI can set `super_read_only=OFF`. See Instance Configuration in Super Read-only Mode for more information.

If you do not specify `adoptFromGR: true`, and the target server instance belongs to a replication group, MySQL Shell prompts you to confirm whether you want to adopt the replication group. From MySQL Shell 8.0.29, if you specify `adoptFromGR: false`, the operation is stopped with no prompt if the instance is found to belong to a replication group.

The new cluster matches the mode of the group. If the adopted group was running in single-primary mode then a single-primary cluster is created. If the adopted group was running in multi-primary mode then a multi-primary cluster is created.

Note

It is not possible to define the communication stack used by the cluster in the same command as `adoptFromGR`, the cluster must initially use the communication stack used by the adopted group. If necessary, you can change the communication stack after the group is adopted using `rebootClusterFromCompleteOutage`. See Section 7.5.9, “Configuring the Group Replication Communication Stack”.

7.5 Configuring InnoDB Cluster

This section describes how to use AdminAPI for further detailed configuration of an InnoDB Cluster during the cluster creation process and after you have created it. You can use this information to modify the settings that AdminAPI applies by default when you create a cluster.

7.5.1 Setting Options for InnoDB Cluster

You can check and modify the settings in place for an InnoDB Cluster while the instances are online. To check the current settings of a cluster, use the following operation:

- *`Cluster`*`.options()`, which lists the configuration options for the cluster and its instances. A Boolean option `all` can also be specified to include information about all Group Replication system variables in the output.

You can configure the options of an InnoDB Cluster at a cluster level or instance level, while instances remain online. This avoids the need to remove, reconfigure and then again add the instance to change InnoDB Cluster options. Use the following operations:

- `Cluster.setOption(option, value)` to change the settings of all cluster instances globally or cluster global settings such as `clusterName`.
- `Cluster.setInstanceOption(instance, option, value)` to change the settings of individual cluster instances

The way which you use InnoDB Cluster options with the operations listed depends on whether the option can be changed to be the same on all instances or not. These options are changeable at both the cluster (all instances) and per instance level:

- `autoRejoinTries`: integer value to define the number of times an instance attempts to rejoin the cluster after being expelled. See Section 7.5.5, “Configuring Automatic Rejoin of Instances”.
- `exitStateAction`: string value indicating the Group Replication exit state action. See Section 7.5.5, “Configuring Automatic Rejoin of Instances”.
- `memberWeight`: integer value with a percentage weight for automatic primary election on failover. See Section 7.5.3, “Configuring the Election Process”.
- `ipAllowList`: comma-separated list of IP addresses or subnet CIDR notation. For example: `192.168.1.0/24,10.0.0.1`. By default the value is set to `AUTOMATIC`, allowing addresses from the instance private network to be automatically set for the allowlist.

  Note

  This option can only be set if the `communicationStack` is set to `XCOM`.

- `tag:option`: built-in and user-defined tags to be associated to the cluster. See Section 6.9, “Tagging Metadata”.

These options are changeable at the cluster level only:

- `clusterName`: string value to define the cluster name
- `disableClone`: Boolean value used to disable the clone usage on the cluster. See `dba.createCluster()` and MySQL Clone.
- `replicationAllowedHost`: string value to define strict subnet based filtering, so that internally managed replication accounts can only connect from allowed hosts. See InnoDB Cluster replicationAllowedHost.
- `expelTimeout`: integer value to define the time period in seconds that cluster members should wait for a non-responding member before evicting it from the cluster. See Section 7.4.3, “Creating an InnoDB Cluster”.
- `failoverConsistency`: string value indicating the consistency guarantees that the cluster provides. See Section 7.5.5, “Configuring Automatic Rejoin of Instances”.
- `transactionSizeLimit`: positive integer value which sets the Group Replication system variable `group_replication_transaction_size_limit`. This sets the maximum transaction size in bytes which the cluster accepts. Larger transactions are rolled back and not broadcast to the cluster. All members added to the cluster use the same value.

This option is changeable at the per instance level only:

- `label`: a string identifier of the instance

7.5.2 Customizing InnoDB Cluster Member Servers

When you create a cluster and add instances to it, values such as the group name and the local address are configured automatically by AdminAPI. The default values are recommended for most deployments, but advanced users can override the defaults by passing the following options to the `dba.createCluster()` and *`Cluster`*`.addInstance()` commands:

- Pass the `groupName` option to the `dba.createCluster()` command to customize the name of the replication group created by InnoDB Cluster. This sets the `group_replication_group_name` system variable. The name must be a valid UUID.
- Pass the `localAddress` option to the `dba.createCluster()` and `cluster.addInstance()` commands to customize the address which an instance provides for connections from other instances. Specify the address in the format *`host:port`*. This sets the `group_replication_local_address` system variable on the instance. The address must be accessible to all instances in the cluster, and must be reserved for internal cluster communication only. In other words, do not use this address for communication with the instance.

For more information see the documentation of the system variables configured by these AdminAPI options.

7.5.3 Configuring the Election Process

You can optionally configure how a single-primary cluster elects a new primary, for example to prefer one instance as the new primary to fail over to. Use the `memberWeight` option and pass it to the `dba.createCluster()` and `Cluster.addInstance()` methods when creating your cluster. The `memberWeight` option accepts an integer value between 0 and 100, which is a percentage weight for automatic primary election on failover. When an instance has a higher percentage number set by `memberWeight`, it is more likely to be elected as primary in a single-primary cluster. When a primary election takes place, if multiple instances have the same `memberWeight` value, the instances are then prioritized based on their server UUID in lexicographical order (the lowest) and by picking the first one.

Setting the value of `memberWeight` configures the `group_replication_member_weight` system variable on the instance. Group Replication limits the value range from 0 to 100, automatically adjusting it if a higher or lower value is provided. Group Replication uses a default value of 50 if no value is provided. See Single-Primary Mode for more information.

For example to configure a cluster where `ic-3` is the preferred instance to fail over to in the event that `ic-1`, the current primary, leaves the cluster unexpectedly use `memberWeight` as follows:

```
dba.createCluster('cluster1', {memberWeight:35})
var mycluster = dba.getCluster()
mycluster.addInstance('icadmin@ic2', {memberWeight:25})
mycluster.addInstance('icadmin@ic3', {memberWeight:50})
```

7.5.4 Configuring Failover Consistency

Group Replication provides the ability to specify the failover guarantees (eventual or “read your writes”) if a primary failover happens in single-primary mode (see Configuring Transaction Consistency Guarantees). You can configure the failover guarantees of an InnoDB Cluster at creation by passing the `consistency` option (prior to version 8.0.16 this option was the `failoverConsistency` option, which is now deprecated) to the `dba.createCluster()` operation, which configures the `group_replication_consistency` system variable on the seed instance. This option defines the behavior of a new fencing mechanism used when a new primary is elected in a single-primary group. The fencing restricts connections from writing and reading from the new primary until it has applied any pending backlog of changes that came from the old primary (sometimes referred to as “read your writes”). While the fencing mechanism is in place, applications effectively do not see time going backward for a short period of time while any backlog is applied. This ensures that applications do not read stale information from the newly elected primary.

The `consistency` option is only supported if the target MySQL server version is 8.0.14 or later, and instances added to a cluster which has been configured with the `consistency` option are

automatically configured to have `group_replication_consistency` the same on all cluster members that have support for the option. The variable default value is controlled by Group Replication and is `EVENTUAL`, change the `consistency` option to `BEFORE_ON_PRIMARY_FAILOVER` to enable the fencing mechanism. Alternatively use `consistency=0` for `EVENTUAL` and `consistency=1` for `BEFORE_ON_PRIMARY_FAILOVER`.

Note

Using the `consistency` option on a multi-primary InnoDB Cluster has no effect but is allowed because the cluster can later be changed into single-primary mode with the `Cluster.switchToSinglePrimaryMode()` operation.

7.5.5 Configuring Automatic Rejoin of Instances

Instances running MySQL 8.0.16 and later support the Group Replication automatic rejoin functionality, which enables you to configure instances to automatically rejoin the cluster after being expelled. See Responses to Failure Detection and Network Partitioning for background information. AdminAPI provides the `autoRejoinTries` option to configure the number of tries instances make to rejoin the cluster after being expelled. By default instances do not automatically rejoin the cluster. You can configure the `autoRejoinTries` option at either the cluster level or for an individual instance using the following commands:

- `dba.createCluster()`
- `Cluster.addInstance()`
- `Cluster.setOption()`
- `Cluster.setInstanceOption()`

The `autoRejoinTries` option accepts positive integer values between 0 and 2016 and the default value is 3. When using the automatic rejoin functionality, your cluster is more tolerant to faults, especially temporary ones such as unreliable networks. But if quorum has been lost, you should not expect members to automatically rejoin the cluster, because majority is required to rejoin instances.

Instances running MySQL version 8.0.12 and later have the `group_replication_exit_state_action` variable, which you can configure using the AdminAPI `exitStateAction` option. This controls what instances do in the event of leaving the cluster unexpectedly. By default the `exitStateAction` option is `READ_ONLY`, which means that instances which leave the cluster unexpectedly become read-only. If `exitStateAction` is set to `OFFLINE_MODE` (available from MySQL 8.0.18), instances which leave the cluster unexpectedly become read-only and also enter offline mode, where they disconnect existing clients and do not accept new connections (except from clients with administrator privileges). If `exitStateAction` is set to `ABORT_SERVER` then in the event of leaving the cluster unexpectedly, the instance shuts down MySQL, and it has to be started again before it can rejoin the cluster. Note that when you are using the automatic rejoin functionality, the action configured by the `exitStateAction` option only happens in the event that all attempts to rejoin the cluster fail.

There is a chance you might connect to an instance and try to configure it using the AdminAPI, but at that moment the instance could be rejoining the cluster. This could happen whenever you use any of these operations:

- `Cluster.status()`
- `dba.getCluster()`
- `Cluster.rejoinInstance()`
- `Cluster.addInstance()`
- `Cluster.removeInstance()`

- `Cluster.rescan()`
- `Cluster.checkInstanceState()`

These operations might provide extra information while the instance is automatically rejoining the cluster. In addition, when you are using `Cluster.removeInstance()`, if the target instance is automatically rejoining the cluster the operation aborts unless you pass in `force:true`.

7.5.6 Configuring the Parallel Replication Applier

From version 8.0.23 instances support and enable parallel replication applier threads, sometimes referred to as a multi-threaded replica. Using multiple replica applier threads in parallel improves the throughput of both the replication applier and incremental recovery.

This means that on instances running 8.0.23 and later, the following system variables must be configured:

- `binlog_transaction_dependency_tracking=WRITESET`
- `slave_preserve_commit_order=ON`
- `slave_parallel_type=LOGICAL_CLOCK`
- `transaction_write_set_extraction=XXHASH64`

By default, the number of applier threads (configured by the `slave_parallel_workers` system variable) is set to 4.

When you upgrade a cluster that has been running a version of MySQL server and MySQL Shell earlier than 8.0.23, the instances are not configured to use the parallel replication applier. If the parallel applier is not enabled, the output of the `Cluster.status()` operation shows a message in the `instanceErrors` field, for example:

```
...
"instanceErrors": [
 "NOTE: The required parallel-appliers settings are not enabled on
  the instance. Use dba.configureInstance() to fix it."
...
```

In this situation you should reconfigure your instances, so that they use the parallel replication applier. For each instance that belongs to the InnoDB Cluster, update the configuration by issuing `dba.configureInstance(instance)`. Note that usually `dba.configureInstance()` is used before adding the instance to a cluster, but in this special case there is no need to remove the instance and the configuration change is made while it is online.

Information about the parallel replication applier is displayed in the output of the `Cluster.status(extended=1)` operation. For example, if the parallel replication applier is enabled, then the `topology` section output for the instance shows the number of threads under `applierWorkerThreads`. The system variables configured for the parallel replication applier are shown in the output of the `Cluster.options()` operation.

You can configure the number of threads which an instance uses for the parallel replication applier with the `applierWorkerThreads` option, which defaults to 4 threads. The option accepts integers in the range of 0 to 1024 and can only be used with the `dba.configureInstance()` and `dba.configureReplicaSetInstance()` operations. For example, to use 8 threads, issue:

```
mysql-js> dba.configureInstance(instance, {applierWorkerThreads: 8, restart: true})
```

Note

The change to the number of threads used by the parallel replication applier only occurs after the instance is restarted and has rejoined the cluster.

To disable the parallel replication applier, set the `applierWorkerThreads` option to 0.

7.5.7 InnoDB Cluster and Auto-increment

When you are using an instance as part of an InnoDB Cluster, the `auto_increment_increment` and `auto_increment_offset` variables are configured to avoid the possibility of auto increment collisions for multi-primary clusters up to a size of 9 (the maximum supported size of a Group Replication group). The logic used to configure these variables can be summarized as:

- If the group is running in single-primary mode, then set `auto_increment_increment` to 1 and `auto_increment_offset` to 2.
- If the group is running in multi-primary mode, then when the cluster has 7 instances or less set `auto_increment_increment` to 7 and `auto_increment_offset` to 1 + `server_id` % 7. If a multi-primary cluster has 8 or more instances set `auto_increment_increment` to the number of instances and `auto_increment_offset` to 1 + `server_id` % the number of instances.

7.5.8 InnoDB Cluster and Binary Log Purging

In MySQL 8, the binary log is automatically purged (as defined by `binlog_expire_logs_seconds`). This means that a cluster which has been running for a longer time than `binlog_expire_logs_seconds` could eventually not contain an instance with a complete binary log that contains all of the transactions applied by the instances. This could result in instances needing to be provisioned automatically, for example using MySQL Enterprise Backup, before they could join the cluster. Instances running 8.0.17 and later support the MySQL Clone plugin, which resolves this issue by providing an automatic provisioning solution which does not rely on incremental recovery, see Section 7.4.6, “Using MySQL Clone with InnoDB Cluster”. Instances running a version earlier than 8.0.17 only support incremental recovery, and the result is that, depending on which version of MySQL the instance is running, instances might have to be provisioned automatically. Otherwise operations which rely on distributed recovery, such as `Cluster.addInstance()` and so on might fail.

On instances running earlier versions of MySQL the following rules are used for binary log purging:

- Instances running a version earlier than 8.0.1 have no automatic binary log purging because the default value of `expire_logs_days` is 0.
- Instances running a version later than 8.0.1 but earlier than 8.0.4 purge the binary log after 30 days because the default value of `expire_logs_days` is 30.
- Instances running a version later than 8.0.10 purge the binary log after 30 days because the default value of `binlog_expire_logs_seconds` is 2592000 and the default value of `expire_logs_days` is 0.

Thus, depending on how long the cluster has been running binary logs could have been purged and you might have to provision instances manually. Similarly, if you manually purged binary logs you could encounter the same situation. Therefore you are strongly advised to upgrade to a version of MySQL later than 8.0.17 to take full advantage of the automatic provisioning provided by MySQL Clone for distributed recovery, and to minimize downtime while provisioning instances for your InnoDB Cluster.

7.5.9 Configuring the Group Replication Communication Stack

As of MySQL Shell 8.0.30, InnoDB Cluster and ClusterSet support the MySQL communication stack introduced for Group Replication in MySQL 8.0.27.

The new option, `communicationStack: XCOM|MYSQL` sets the value of the Group Replication system variable `group_replication_communication_stack`.

Note

It is not possible to use the `communicationStack` option with `adoptfromGR`.

Communication Stack Types

The following communication stacks are supported:

- MYSQL: (default for MySQL Server 8.0.27 or newer)
  - Simplifies the creation of InnoDB Clusters by using MySQL Server's connection security in place of the Group Replication implementation.
  - Removes the need for an extra network address, or port, for internal Group Replication communications.
  - Using the MYSQL protocol means that standard methods of user authentication can be used for granting, or revoking, access to the group in place of the allow list.
  - Supports network namespaces for Group Replication.

  Oracle recommends using the `MYSQL` communication stack instead of `XCOM`.
- XCOM: (default for MySQL Server 8.0.26 or older). You can use the XCOM communication stack with MySQL 8.0.27, or higher, but it must be explicitly defined in the creation or reboot commands.

  XCOM secures group communication connections and distributed recovery connections between members using the Group Replication implementation of the security protocols, including TLS/SSL and the use of an allowlist for incoming Group Communication System (GCS) connections.

Selecting Communication Stack

The communication stack selection is set by the `communicationStack` option in the `dba.createCluster()` and `<clusterSet>.createReplicaCluster()` commands.

For example:

```
js> dba.createCluster("testCluster", {communicationStack: "mysql"})

js> clusterset.createReplicaCluster("hostname:3306", "replica", {communicationStack: "mysql"})
```

Each command checks the MySQL server to ensure it can use the `MYSQL` protocol. If it does not support `MYSQL`, an error is displayed and the command fails.

The `addInstance`, `rejoinInstance`, and `rescan` commands also check the target instance for communication stack support and set the required configuration options accordingly.

- `ipAllowList`.
  - `XCOM`: Set automatically, by default.
  - `MYSQL`: unset. `ipAllowList` is not permitted with the `MYSQL` communication stack.
- `localAddress`
  - `XCOM`: (advanced option, not recommended) Automatically generated. Requires additional network address, or port.
  - `MYSQL`: Automatically updated to use the value reported by the MySQL server.

    `localAddress` can be defined manually, but the port must be one MySQL is listening on, as defined by `bind_address`.
- `groupSeeds`

  - `XCOM`: automatically generated. Requires additional network address, or port.
  - `MYSQL`: is automatically updated to use the `bind_address` value used by each MySQL instance.
- Updates the SSL settings. The same SSL settings are used by both communication protocols. The settings are copied from the MySQL server if `memberSslMode` is set to VERIFY_CA or VERIFY_IDENTITY. No changes are made if `memberSslMode` is set to REQUIRED.

Note

`group_replication_recovery_use_ssl` is always enabled if `memberSslMode` is set to any value other than DISABLED.

Switching Communication Stack

It is possible to switch communication stack during a reboot from complete outage operation.

For example:

```
js> dba.rebootClusterFromCompleteOutage("testcluster", {switchCommunicationStack: "mysql"})
```

Switching from the `MYSQL` protocol to `XCOM` requires an additional network address for the `localAddress` and may also require `ipAllowList` values.

If switching from the XCOM to the MYSQL stack, the following changes are made:

- `ipAllowList` is unset.
- `localAddress` is updated to use the value reported by the MySQL server.
- `groupSeeds` is updated to use the `bind_address` value used by each MySQL instance.
- Updates the SSL settings. The settings are copied from the MySQL server if `memberSslMode` is set to VERIFY_CA or VERIFY_IDENTITY. No changes are made if `memberSslMode` is set to REQUIRED.

Note

`group_replication_recovery_use_ssl` is always enabled if `memberSslMode` is set to any value other than DISABLED.

7.6 Securing InnoDB Cluster

Server instances can be configured to use secure connections. For general information on using secure connections with MySQL see Using Encrypted Connections. This section explains how to configure a cluster to use encrypted connections. An additional security possibility is to configure which servers can access the cluster, see Creating an Allowlist of Servers.

Important

If you are using the `XCOM` communication stack, once you have configured a cluster to use encrypted connections you must add the servers to the `ipAllowlist`. For example, when using the commercial version of MySQL, SSL is enabled by default and you need to configure the `ipAllowlist` option for all instances. See Creating an Allowlist of Servers.

When using `dba.createCluster()` to set up a cluster, if the server instance provides encryption then it is automatically enabled on the seed instance. Pass the `memberSslMode` option to the `dba.createCluster()` method to specify a different SSL mode. The SSL mode of a cluster can only be set at the time of creation. The `memberSslMode` option is a string that configures the SSL mode to be used, it defaults to `AUTO`. The following modes are supported:

- `DISABLED`: Ensure SSL encryption is disabled for the seed instance in the cluster.
- `AUTO`: Automatically enable SSL encryption if the server instance supports it, or disable encryption if the server does not support it.
- `REQUIRED`: Enable SSL encryption for the seed instance in the cluster. If it cannot be enabled, an error is raised.
- `VERIFY_CA`: Like `REQUIRED`, but additionally verify the server Certificate Authority (CA) certificate against the configured CA certificates. The connection attempt fails if no valid matching CA certificates are found.
- `VERIFY_IDENTITY`: Like `VERIFY_CA`, but additionally perform host name identity verification by checking the host name the client uses for connecting to the server against the identity in the certificate that the server sends to the client.

For example, to set the cluster to use `REQUIRED`, issue:

```
mysql-js> var myCluster = dba.createCluster({memberSslMode: 'REQUIRED'})
```

If you choose to use the `VERIFY_CA` or `VERIFY_IDENTITY` mode, on each cluster instance you must manually supply the CA certificates using the `ssl_ca` and/or `ssl_capath` option. For more information on these modes, see `--ssl-mode=`*`mode`*.

When you use the *`Cluster`*`.addInstance()` and *`Cluster`*`.rejoinInstance()` operations, SSL encryption on the instance is enabled or disabled based on the setting used for the cluster. Use the `memberSslMode` option with either of these operations to set the instance to use a different mode of encryption.

When using `dba.createCluster()` with the `adoptFromGR` option to adopt an existing Group Replication group, no SSL settings are changed on the adopted cluster:

- `memberSslMode` cannot be used with `adoptFromGR`.
- If the SSL settings of the adopted cluster are different from the ones supported by the MySQL Shell, in other words SSL for Group Replication recovery and Group Communication, both settings are not modified. This means you are not be able to add new instances to the cluster, unless you change the settings manually for the adopted cluster.

MySQL Shell always enables or disables SSL for the cluster for both Group Replication recovery and Group Communication, see Securing Group Communication Connections with Secure Socket Layer (SSL). A verification is performed and an error issued in case those settings are different for the seed instance (for example as the result of a `dba.createCluster()` using `adoptFromGR`) when adding a new instance to the cluster. SSL encryption must be enabled or disabled for all instances in the cluster. Verifications are performed to ensure that this invariant holds when adding a new instance to the cluster.

The `dba.deploySandboxInstance()` command attempts to deploy sandbox instances with SSL encryption support by default. If it is not possible, the server instance is deployed without SSL support. See Section 6.8.1, “Deploying Sandbox Instances”.

Securing Communications Between Cluster Members

As of MySQL Shell 8.0.33, it is possible to configure clusters and replica clusters to use SSL to encrypt replication channels, and enable replicas to verify host identity and use SSL certificates for authentication.

When creating a cluster with `dba.createCluster()` you can define the authentication type used for the internal replication accounts with the `memberAuthType` option. This option takes one of the following values:

- `PASSWORD`: Account authenticates with password only.

- `CERT_ISSUER`: Account authenticates with a client certificate, which must match the expected issuer. This value is equivalent to `VERIFY_CA`.
- `CERT_SUBJECT`: Account authenticates with a client certificate, which must match the expected issuer and subject. This value is equivalent to `VERIFY_IDENTITY`.
- `CERT_ISSUER_PASSWORD`: Account authenticates with a combination of `PASSWORD` and `CERT_ISSUER` values.
- `CERT_SUBJECT_PASSWORD`: Account authenticates with a combination of `PASSWORD` and `CERT_SUBJECT` values.

Important

ClusterSets inherit the `memberAuthType` defined on the primary cluster. All replica clusters in a ClusterSet will also use the `memberAuthType` defined on the primary.

SSL certificates are defined using the following options:

- `CERT_ISSUER`: Defines the certificate issuer required by all replication accounts in the topology if `memberAuthType` contains `CERT_ISSUER` or `CERT_SUBJECT`.
- `CERT_SUBJECT`: Defines the certificate subject of the instance. Required if `memberAuthType` contains `CERT_SUBJECT`.

Note

It is not possible to use `adoptFromGR=true` with any option except `memberAuthType=password`.

The following example creates a cluster, `cluster1` which sets client SSL connections and connections opened by Group Replication from one server to another to VERIFY_IDENTITY, and sets the authentication of the internal replication accounts to require a client certificate:

```
        cluster = dba.createCluster("cluster1", { "memberSslMode": "VERIFY_IDENTITY", "memberAuthType":"CER
        "certIssuer":"/CN=MyCertAuthority", "certSubject": "/CN=mysql-1.local"});
```

The following example shows how to add an instance to a cluster using `"memberAuthType":"CERT_SUBJECT"`:

```
        cluster.addInstance("mysql-2.local", {"certSubject": "/CN=mysql-2.local"});
```

For more information on replication and encrypted connections, see Setting Up Replication to Use Encrypted Connections.

Creating an Allowlist of Servers

Note

This applies only to the `XCOM` communication stack.

`createCluster()`, `addInstance()`, and `rejoinInstance()` methods enable you to optionally specify a list of approved servers, referred to as an allowlist. By specifying the allowlist explicitly in this way you can increase the security of your cluster because only servers in the allowlist can connect to the cluster.

You can also define an allowList on a running cluster, using *`Cluster`*`.setOption()` to specify the allowList for all members of the cluster, and *`Cluster`*`.setInstanceOption()` to specify the allowList for an individual member. See Section 7.5.1, “Setting Options for InnoDB Cluster”.

Using the `ipAllowlist` option (previously `ipWhitelist`, now deprecated) configures the `group_replication_ip_allowlist` system variable on the instance. By default, if not specified explicitly, the allowlist is automatically set to the private network addresses that the server has network interfaces on. To configure the allowlist, specify the servers to add with the `ipAllowlist` option when using the method. IP addresses must be specified in IPv4 format. Pass the servers as a comma separated list, surrounded by quotes. For example:

```
mysql-js> cluster.addInstance("icadmin@ic-3:3306", {ipAllowlist: "203.0.113.0/24, 198.51.100.110"})
```

This configures the instance to only accept connections from servers at addresses `203.0.113.0/24` and `198.51.100.110`. The allowlist can also include host names, which are resolved only when a connection request is made by another server.

Warning

Host names are inherently less secure than IP addresses in an allowlist. MySQL carries out FCrDNS verification, which provides a good level of protection, but can be compromised by certain types of attack. Specify host names in your allowlist only when strictly necessary, and ensure that all components used for name resolution, such as DNS servers, are maintained under your control. You can also implement name resolution locally using the hosts file, to avoid the use of external components.

7.7 Monitoring InnoDB Cluster

This section describes how to use AdminAPI to monitor an InnoDB Cluster.

- Using *`Cluster`*`.describe()`
- Checking a cluster's Status with *`Cluster`*`.status()`
- Monitoring Recovery Operations
- InnoDB Cluster and Group Replication Protocol
- Checking the MySQL Version on Instances

Using *`Cluster`*`.describe()`

To get information about the structure of the InnoDB Cluster itself, use the *`Cluster`*`.describe()` function:

```
mysql-js> cluster.describe();
{
    "clusterName": "testCluster",
    "defaultReplicaSet": {
        "name": "default",
        "topology": [
            {
                "address": "ic-1:3306",
                "label": "ic-1:3306",
                "role": "HA"
            },
            {
                "address": "ic-2:3306",
                "label": "ic-2:3306",
                "role": "HA"
            },
            {
                "address": "ic-3:3306",
                "label": "ic-3:3306",
                "role": "HA"
            }
        ]
    }
```

```
}
```

The output from this function shows the structure of the InnoDB Cluster including all of its configuration information, and so on. The address, label and role values match those described at Checking a cluster's Status with *Cluster*.status() .

Checking a cluster's Status with *Cluster*.status()

Cluster objects provide the status() method that enables you to check how a cluster is running. Before you can check the status of the InnoDB Cluster, you need to get a reference to the InnoDB Cluster object by connecting to any of its instances. However, if you want to make changes to the configuration of the cluster, you must connect to a "R/W" instance. Issuing status() retrieves the status of the cluster based on the view of the cluster which the server instance you are connected to is aware of and outputs a status report.

Important

The instance's state in the cluster directly influences the information provided in the status report. Therefore ensure the instance you are connected to has a status of ONLINE.

For information about how the InnoDB Cluster is running, use the cluster's status() method:

```
mysql-js> var cluster = dba.getCluster()
mysql-js> cluster.status()
{
    "clusterName": "testcluster",
    "defaultReplicaSet": {
        "name": "default",
        "primary": "ic-1:3306",
        "ssl": "REQUIRED",
        "status": "OK",
        "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
        "topology": {
            "ic-1:3306": {
                "address": "ic-1:3306",
                "memberRole": "PRIMARY",
                "mode": "R/W",
                "readReplicas": {},
                "replicationLag": "applier_queue_applied",
                "role": "HA",
                "status": "ONLINE"
                "version": "8.0.30"
            },
            "ic-2:3306": {
                "address": "ic-2:3306",
                "memberRole": "SECONDARY",
                "mode": "R/O",
                "readReplicas": {},
                "replicationLag": "applier_queue_applied",
                "role": "HA",
                "status": "ONLINE"
                "version": "8.0.30"
            },
            "ic-3:3306": {
                "address": "ic-3:3306",
                "memberRole": "SECONDARY",
                "mode": "R/O",
                "readReplicas": {},
                "replicationLag": "applier_queue_applied",
                "role": "HA",
                "status": "ONLINE"
                "version": "8.0.30"
            }
        }
        "topologyMode": "Single-Primary"
    },
    "groupInformationSourceMember": "mysql://icadmin@ic-1:3306"
```

```
}
```

The output of *Cluster*.status() provides the following information:

- clusterName: name assigned to this cluster during dba.createCluster().
- defaultReplicaSet: the server instances which belong to an InnoDB Cluster and contain the data set.
- primary: displayed when the cluster is operating in single-primary mode only. Shows the address of the current primary instance. If this field is not displayed, the cluster is operating in multi-primary mode.
- ssl: whether secure connections are used by the cluster or not. Shows values of REQUIRED or DISABLED, depending on how the memberSslMode option was configured during either createCluster() or addInstance(). The value returned by this parameter corresponds to the value of the group_replication_ssl_mode server variable on the instance. See Section 7.6, "Securing InnoDB Cluster".
- status: The status of the InnoDB Cluster. The status describes the high availability provided by this cluster. The status is one of the following:
  - OK: The cluster is online and can tolerate up to n failures. There are three or more members in the cluster, and they are functioning.
  - OK_PARTIAL: The cluster is online and can tolerate up to n failures. At least three of the member servers in the cluster are in Group Replication's online state. However, one or more member servers are not currently participating as active members of the cluster.
  - OK_NO_TOLERANCE: The cluster is not tolerant to any failures.
  - OK_NO_TOLERANCE_PARTIAL: The cluster is not tolerant to any failures. One or two member servers in the cluster are online, but one or more servers are in an offline, recovering, error, or unreachable state. The cluster does not have sufficient tolerance for failures because of the unavailability of some members.
  - NO_QUORUM: The cluster does not have quorum, meaning that a majority of the replication group's member servers are unavailable for agreeing on a decision, and cannot process write transactions.
  - OFFLINE: All members of the group are offline.
  - ERROR: There are no online members in the cluster.
  - UNREACHABLE: There is no connectivity to any online members.
  - UNKNOWN: There is no connectivity to any online members.
  - FENCED_WRITES: The cluster is fenced from write traffic.
- topology: The status of the MySQL Server instance. The status is one of the following:
  - Host name of instance: The host name of an instance, for example "localhost:3310".
  - memberRole the Member Role as reported by the Group Replication plugin, see the MEMBER_ROLE column of the replication_group_members table.
  - mode: whether the server is read-write ("R/W") or read-only ("R/O"). From version 8.0.17, this is derived from the current state of the super_read_only variable on the instance, and whether the cluster has quorum. In previous versions the value of mode was derived from whether the instance was serving as a primary or secondary instance. Usually if the instance is a primary, then the mode is "R/W", and if the instance is a secondary the mode is "R/O". Any instances

in a cluster that have no visible quorum are marked as "R/O", regardless of the state of the super_read_only variable.

Note

If the member status is anything other than ONLINE, mode is reported as n/a.

- replicationLag: returns one of the following values:
  - The time difference between the last transaction commit timestamp and the last transaction applied timestamp, in HH:MM:SS format.

    If multiple workers are used, the value is retrieved from the worker executing the oldest transaction.
  - null: The replication connection or SQL thread is not running.
  - applier_queue_applied: The applier queue has applied everything. That is, if the last queued transaction and the last applied transaction are the same, or the applying transaction is 0.
- role: what function this instance provides in the cluster. Currently only HA, for high availability.
- status: The status of this element of the cluster. The status is one of the following:
  - ONLINE: The instance is online and participating in the cluster.
  - OFFLINE: The instance has lost connection to the other instances.
  - RECOVERING: The instance is attempting to synchronize with the cluster by retrieving transactions it needs before it can become an online member.
  - UNREACHABLE: The instance has lost communication with the cluster.
  - ERROR: The instance has encountered an error during the recovery phase or while applying a transaction.

    Important

    Once an instance enters ERROR state, the super_read_only option is set to ON. To leave the ERROR state you must manually configure the instance with super_read_only=OFF.
  - (MISSING): The state of an instance which is part of the configured cluster, but is currently unavailable.

    Note

    The MISSING state is specific to InnoDB Cluster, it is not a state generated by Group Replication. MySQL Shell uses this state to indicate instances that are registered in the metadata, but cannot be found in the live cluster view.
- groupInformationSourceMember: the internal connection used to get information about the cluster, shown as a URI-like connection string. Usually the connection initially used to create the cluster.

- version: the MySQL Server version running on the instance. See Checking the MySQL Version on Instances for more information.

To display more information about the cluster use the extended option. From version 8.0.17, the extended option supports integer or Boolean values. To configure the additional information that *Cluster*.status({'extended':*value*}) provides, use the following values:

- 0: disables the additional information, the default
- 1: includes information about the Group Replication Protocol Version, Group name, communication stack, cluster member UUIDs, cluster member roles and states as reported by Group Replication, and the list of fenced system variables
- 2: includes information about transactions processed by connection and applier
- 3: includes more detailed statistics about the replication performed by each cluster member.

Setting extended using Boolean values is the equivalent of setting the integer values 0 and 1. In versions prior to 8.0.17, the extended option was only Boolean. Similarly prior versions used the queryMembers Boolean option to provide more information about the instances in the cluster, which is the equivalent of setting extended to 3. The queryMembers option is deprecated and scheduled to be removed in a future release.

When you issue *Cluster*.status({'extended':1}), or the extended option is set to true, the output includes:

- the following additional attributes for the defaultReplicaSet object:
  - GRProtocolVersion: the Group Replication Protocol Version being used in the cluster.

Tip

InnoDB Cluster manages the Group Replication Protocol version being used automatically, see InnoDB Cluster and Group Replication Protocol for more information.

  - communicationStack: the communication stack in use by the cluster. Possible values are XCOM or MYSQL. See Section 7.5.9, “Configuring the Group Replication Communication Stack” for more information.
  - groupName: the group's name, a UUID.
  - groupViewChangeUuid: the value of group_replication_view_change_uuid.
  - groupViewId: the current view identifier for this group. This value is taken from the VIEW_ID column of the replication_group_member_stats table.
  - paxosSingleLeader: displays the value of group_replication_paxos_single_leader.

Note

This is only available on MySQL Server 8.0.31, or higher, because MySQL Shell requires the information provided by WRITE_CONSENSUS_SINGLE_LEADER_CAPABLE in the replication_group_communication_information table, which was introduced in MySQL 8.0.31.

- the following additional attributes for each object of the topology object:
  - fenceSysVars a list containing the name of the fenced system variables which are configured by AdminAPI. Currently the fenced system variables considered are read_only, super_read_only and offline_mode. The system variables are listed regardless of their value.

- instanceErrors for each instance, displaying any diagnostic information that can be detected for the instance. For example, if the instance is a secondary and the super_read_only variable is not set to ON, then a warning is shown. This information can be used to troubleshoot errors.
- memberId Each cluster member UUID.
- memberState the Member State as reported by the Group Replication plugin, see the MEMBER_STATE column of the replication_group_members table.

To see information about recovery and regular transaction I/O, applier worker thread statistics and any lags; applier coordinator statistics, if the parallel replication applier is enabled; error, and other information from the receiver and applier threads, use a value of 2 or 3 for extended. When you use these values, a connection to each instance in the cluster is opened so that additional instance specific statistics can be queried. The exact statistics that are included in the output depend on the state and configuration of the instance and the server version. This information matches that shown in the replication_group_member_stats table, see the descriptions of the matching columns for more information. Instances which are ONLINE have a transactions section included in the output. Instances which are RECOVERING have a recovery section included in the output. When you set extended to 2, in either case, these sections can contain the following:

- appliedCount: see COUNT_TRANSACTIONS_REMOTE_APPLIED
- checkedCount: see COUNT_TRANSACTIONS_CHECKED
- committedAllMembers: see TRANSACTIONS_COMMITTED_ALL_MEMBERS
- conflictsDetectedCount: see COUNT_CONFLICTS_DETECTED
- inApplierQueueCount: see COUNT_TRANSACTIONS_REMOTE_IN_APPLIER_QUEUE
- inQueueCount: see COUNT_TRANSACTIONS_IN_QUEUE
- lastConflictFree: see LAST_CONFLICT_FREE_TRANSACTION
- proposedCount: see COUNT_TRANSACTIONS_LOCAL_PROPOSED
- rollbackCount: see COUNT_TRANSACTIONS_LOCAL_ROLLBACK

When you set extended to 3, the connection section shows information from the replication_connection_status table. A value of 3 is the equivalent of setting the deprecated queryMembers option to true. The connection section can contain the following:

The currentlyQueueing section has information about the transactions currently queued:

- immediateCommitTimestamp: see QUEUEING_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP
- immediateCommitToNowTime: see QUEUEING_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP minus NOW()
- originalCommitTimestamp: see QUEUEING_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP
- originalCommitToNowTime: see QUEUEING_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP minus NOW()
- startTimestamp: see QUEUEING_TRANSACTION_START_QUEUE_TIMESTAMP
- transaction: see QUEUEING_TRANSACTION
- lastHeartbeatTimestamp: see LAST_HEARTBEAT_TIMESTAMP

The lastQueued section has information about the most recently queued transaction:

- `endTimestamp`: see `LAST_QUEUED_TRANSACTION_END_QUEUE_TIMESTAMP`
- `immediateCommitTimestamp`: see `LAST_QUEUED_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP`
- `immediateCommitToEndTime`: `LAST_QUEUED_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP` minus `NOW()`
- `originalCommitTimestamp`: see `LAST_QUEUED_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP`
- `originalCommitToEndTime`: `LAST_QUEUED_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP` minus `NOW()`
- `queueTime`: `LAST_QUEUED_TRANSACTION_END_QUEUE_TIMESTAMP` minus `LAST_QUEUED_TRANSACTION_START_QUEUE_TIMESTAMP`
- `startTimestamp`: see `LAST_QUEUED_TRANSACTION_START_QUEUE_TIMESTAMP`
- `transaction`: see `LAST_QUEUED_TRANSACTION`
- `receivedHeartbeats`: see `COUNT_RECEIVED_HEARTBEATS`
- `receivedTransactionSet`: see `RECEIVED_TRANSACTION_SET`
- `threadId`: see `THREAD_ID`

Instances which are using a multithreaded replica have a `workers` section which contains information about the worker threads, and matches the information shown by the `replication_applier_status_by_worker` table.

The `lastApplied` section shows the following information about the last transaction applied by the worker:

- `applyTime`: see `LAST_APPLIED_TRANSACTION_END_APPLY_TIMESTAMP` minus `LAST_APPLIED_TRANSACTION_START_APPLY_TIMESTAMP`
- `endTimestamp`: see `LAST_APPLIED_TRANSACTION_END_APPLY_TIMESTAMP`
- `immediateCommitTimestamp`: see `LAST_APPLIED_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP`
- `immediateCommitToEndTime`: see `LAST_APPLIED_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP` minus `NOW()`
- `originalCommitTimestamp`: see `LAST_APPLIED_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP`
- `originalCommitToEndTime`: see `LAST_APPLIED_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP` minus `NOW()`
- `startTimestamp`: see `LAST_APPLIED_TRANSACTION_START_APPLY_TIMESTAMP`
- `transaction`: see `LAST_APPLIED_TRANSACTION`

The `currentlyApplying` section shows the following information about the transaction currently being applied by the worker:

- `immediateCommitTimestamp`: see `APPLYING_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP`
- `immediateCommitToNowTime`: see `APPLYING_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP` minus `NOW()`

- originalCommitTimestamp: see APPLYING_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP
- originalCommitToNowTime: see APPLYING_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP minus NOW()
- startTimestamp: see APPLYING_TRANSACTION_START_APPLY_TIMESTAMP
- transaction: see APPLYING_TRANSACTION

The lastProcessed section has the following information about the last transaction processed by the worker:

- bufferTime: LAST_PROCESSED_TRANSACTION_END_BUFFER_TIMESTAMP minus LAST_PROCESSED_TRANSACTION_START_BUFFER_TIMESTAMP
- endTimestamp: see LAST_PROCESSED_TRANSACTION_END_BUFFER_TIMESTAMP
- immediateCommitTimestamp: see LAST_PROCESSED_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP
- immediateCommitToEndTime: LAST_PROCESSED_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP minus LAST_PROCESSED_TRANSACTION_END_BUFFER_TIMESTAMP
- originalCommitTimestamp: see LAST_PROCESSED_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP
- originalCommitToEndTime: LAST_PROCESSED_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP minus LAST_PROCESSED_TRANSACTION_END_BUFFER_TIMESTAMP
- startTimestamp: see LAST_PROCESSED_TRANSACTION_START_BUFFER_TIMESTAMP
- transaction: see LAST_PROCESSED_TRANSACTION

If the parallel replication applier is enabled, then the number of objects in the workers array in transactions or recovery matches the number of configured workers and an additional coordinator object is included. The information shown matches the information in the replication_applier_status_by_coordinator table. The object can contain:

The currentlyProcessing section has the following information about the transaction being processed by the worker:

- immediateCommitTimestamp: see PROCESSING_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP
- immediateCommitToNowTime: PROCESSING_TRANSACTION_IMMEDIATE_COMMIT_TIMESTAMP minus NOW()
- originalCommitTimestamp: see PROCESSING_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP
- originalCommitToNowTime: PROCESSING_TRANSACTION_ORIGINAL_COMMIT_TIMESTAMP minus NOW()
- startTimestamp: see PROCESSING_TRANSACTION_START_BUFFER_TIMESTAMP
- transaction: see PROCESSING_TRANSACTION

worker objects have the following information if an error was detected in the replication_applier_status_by_worker table:

- lastErrno: see LAST_ERROR_NUMBER

- `lastError`: see `LAST_ERROR_MESSAGE`
- `lastErrorTimestamp`: see `LAST_ERROR_TIMESTAMP`

`connection` objects have the following information if an error was detected in the `replication_connection_status` table:

- `lastErrno`: see `LAST_ERROR_NUMBER`
- `lastError`: see `LAST_ERROR_MESSAGE`
- `lastErrorTimestamp`: see `LAST_ERROR_TIMESTAMP`

`coordinator` objects have the following information if an error was detected in the `replication_applier_status_by_coordinator` table:

- `lastErrno`: see `LAST_ERROR_NUMBER`
- `lastError`: see `LAST_ERROR_MESSAGE`
- `lastErrorTimestamp`: see `LAST_ERROR_TIMESTAMP`

Monitoring Recovery Operations

The output of *`Cluster`*`.status()` shows information about the progress of recovery operations for instances in `RECOVERING` state. Information is shown for instances recovering using either MySQL Clone, or incremental recovery. Monitor these fields:

- The `recoveryStatusText` field includes information about the type of recovery being used. When MySQL Clone is working the field shows “Cloning in progress”. When incremental recovery is working the field shows “Distributed recovery in progress”.
- When MySQL Clone is being used, the `recovery` field includes a dictionary with the following fields:
  - `cloneStartTime`: The timestamp of the start of the clone process
  - `cloneState`: The state of the clone progress
  - `currentStage`: The current stage which the clone process has reached
  - `currentStageProgress`: The current stage progress as a percentage of completion
  - `currentStageState`: The current stage state

  Example *`Cluster`*`.status()` output, trimmed for brevity:

```
...
"recovery": {
"cloneStartTime": "2019-07-15 12:50:22.730",
"cloneState": "In Progress",
"currentStage": "FILE COPY",
"currentStageProgress": 61.726837675213865,
"currentStageState": "In Progress"
},
"recoveryStatusText": "Cloning in progress",
...
```

- When incremental recovery is being used and the `extended` option is set to 1 or greater, the `recovery` field includes a dictionary with the following fields:
  - `state`: The state of the `group_replication_recovery` channel
  - `recoveryChannel` : Displayed for instances performing incremental recovery or in which the recovery channel status is not off. Incremental recovery utilises the receiver thread to receive

transactions from the source, and the applier thread applies the received transactions on the instance. Provides the following information:

- `applierQueuedTransactionSetSize`: The number of transactions currently queued, which are waiting to be applied.
- `applierState`: The current state of the replication applier, either `ON` or `OFF`.
- `applierStatus`: The current status of the applier threads. An aggregation of the states shown in the `applierThreadState` field. Can be one of:
  - `APPLIED_ALL`: there are no queued transactions waiting to be applied
  - `APPLYING`: there are transactions being applied
  - `ON`: thread is connected and there are no queued transactions
  - `ERROR`: there was an error while applying transactions
  - `OFF`: the applier thread is disabled
- `applierThreadState`: The current state of any applier threads. Provides detailed information about exactly what the applier thread is doing. For more information, see Replication SQL Thread States.
- `receiverStatus`: The current status of the receiver thread. An aggregation of the states shown in the `receiverThreadState` field. Can be one of:
  - `ON`: the receiver thread has successfully connected and is ready to receive
  - `CONNECTING`: the receiver thread is connecting to the source
  - `ERROR`: there was an error while receiving transactions
  - `OFF`: the receiver thread has gracefully disconnected
- `receiverThreadState`: The current state of the receiver thread. Provides detailed information about exactly what the receiver thread is doing. For more information, see Replication I/O (Receiver) Thread States.
- `source`: The source of the transactions which are being applied.

Example *`Cluster`*`.status()` output, trimmed for brevity:

```
...
"recovery": {
                    "recoveryChannel": {
                        "applierQueuedTransactionSetSize": 2284,
                        "applierStatus": "APPLYING",
                        "applierThreadState": "Opening tables",
                        "receiverStatus": "ON",
                        "receiverThreadState": "Queueing master event to the relay log",
                        "source": "ic-2:3306"
                    },
                    "state": "ON"
                },
...
```

InnoDB Cluster and Group Replication Protocol

From MySQL 8.0.16, Group Replication has the concept of a communication protocol for the group, see Setting a Group's Communication Protocol Version for background information. The Group Replication communication protocol version usually has to be managed explicitly, and set to accommodate the oldest MySQL Server version that you want the group to support. However, InnoDB

Cluster automatically and transparently manages the communication protocol versions of its members, whenever the cluster topology is changed using AdminAPI operations. A cluster always uses the most recent communication protocol version that is supported by all the instances that are currently part of the cluster or joining it.

- When an instance is added to, removed from, or rejoins the cluster, or a rescan or reboot operation is carried out on the cluster, the communication protocol version is automatically set to a version supported by the instance that is now at the earliest MySQL Server version.
- When you carry out a rolling upgrade by removing instances from the cluster, upgrading them, and adding them back into the cluster, the communication protocol version is automatically upgraded when the last remaining instance at the old MySQL Server version is removed from the cluster prior to its upgrade.

To see the communication protocol version being used in a cluster, use the *Cluster*.status() function with the extended option enabled. The communication protocol version is returned in the GRProtocolVersion field, provided that the cluster has quorum and no cluster members are unreachable.

Checking the MySQL Version on Instances

The following operations can report information about the MySQL Server version running on the instance:

- *Cluster*.status()
- *Cluster*.describe()
- *Cluster*.rescan()

The behavior varies depending on the MySQL Server version of the *Cluster* object session.

- *Cluster*.status()

  If either of the following requirements are met, a version string attribute is returned for each instance JSON object of the topology object:

  - The *Cluster* object's current session is version 8.0.11 or later.
  - The *Cluster* object's current session is running a version earlier than version 8.0.11 but the extended option is set to 3 (or the deprecated queryMembers is true).

  For example on an instance running version 8.0.16:

```
"topology": {
    "ic-1:3306": {
        "address": "ic-1:3306",
        "mode": "R/W",
        "readReplicas": {},
        "role": "HA",
        "status": "ONLINE",
        "version": "8.0.16"
}
```

  For example on an instance running version 5.7.24:

```
"topology": {
    "ic-1:3306": {
        "address": "ic-1:3306",
        "mode": "R/W",
        "readReplicas": {},
        "role": "HA",
        "status": "ONLINE",
        "version": "5.7.24"
```

```
}
```

- *Cluster*.describe()

  If the *Cluster* object's current session is version 8.0.11 or later, a version string attribute is returned for each instance JSON object of the topology object

  For example on an instance running version 8.0.16:

```
"topology": [
    {
        "address": "ic-1:3306",
        "label": "ic-1:3306",
        "role": "HA",
        "version": "8.0.16"
    }
]
```

- *Cluster*.rescan()

  If the *Cluster* object's current session is version 8.0.11 or later, and the *Cluster*.rescan() operation detects instances which do not belong to the cluster, a version string attribute is returned for each instance JSON object of the newlyDiscoveredInstance object.

  For example on an instance running version 8.0.16:

```
"newlyDiscoveredInstances": [
    {
        "host": "ic-4:3306",
        "member_id": "82a67a06-2ba3-11e9-8cfc-3c6aa7197deb",
        "name": null,
        "version": "8.0.16"
    }
]
```

7.8 Restoring and Rebooting an InnoDB Cluster

This section describes how to rejoin a server instance to an InnoDB Cluster, restore an InnoDB Cluster from quorum loss or reboot it after an outage, and rescan an InnoDB Cluster after changes.

7.8.1 Rejoining an Instance to a Cluster

If an instance leaves the cluster, for example because it lost connection, and for some reason it could not automatically rejoin the cluster, it might be necessary to rejoin it to the cluster at a later stage. To rejoin an instance to a cluster issue *Cluster*.rejoinInstance(*instance*).

Tip

If the instance has super_read_only=ON then you might need to confirm that AdminAPI can set super_read_only=OFF. See Instance Configuration in Super Read-only Mode for more information.

In the case where an instance has not had its configuration persisted (see Section 6.2.4, “Persisting Settings”), upon restart the instance does not rejoin the cluster automatically. The solution is to issue cluster.rejoinInstance() so that the instance is added to the cluster again and ensure the changes are persisted. Once the InnoDB Cluster configuration is persisted to the instance's option file it rejoins the cluster automatically.

If you are rejoining an instance which has changed in some way then you might have to modify the instance to make the rejoin process work correctly. For example, when you restore a MySQL Enterprise Backup backup, the server_uuid changes. Attempting to rejoin such an instance fails because InnoDB Cluster instances are identified by the server_uuid variable. In such a situation,

information about the instance's old `server_uuid` must be removed from the InnoDB Cluster metadata and then a *`Cluster`*`.rescan()` must be executed to add the instance to the metadata using the new `server_uuid`. For example:

```
cluster.removeInstance("root@instanceWithOldUUID:3306", {force: true})

cluster.rescan()
```

In this case you must pass the `force` option to the *`Cluster`*`.removeInstance()` method because the instance is unreachable from the cluster's perspective and we want to remove it from the InnoDB Cluster metadata anyway.

`rejoinInstance()` also checks the communication stack used by the instance and ensures it is supported by the cluster. If the cluster supports the communication stack, `rejoinInstance()` adds the instance to the cluster.

7.8.2 Restoring a Cluster from Quorum Loss

If an instance (or instances) fail, then a cluster can lose its quorum, which is the ability to vote in a new primary. This can happen when there is a failure of enough instances that there is no longer a majority of the instances which make up the cluster to vote on Group Replication operations. See Fault-tolerance. When a cluster loses quorum you can no longer process write transactions with the cluster, or change the cluster's topology, for example by adding, rejoining, or removing instances. However if you have an instance online which contains the InnoDB Cluster metadata, it is possible to restore a cluster with quorum. This assumes you can connect to an instance that contains the InnoDB Cluster metadata, and that instance can contact the other instances you want to use to restore the cluster.

Important

This operation is potentially dangerous because it can create a split-brain scenario if incorrectly used and should be considered a last resort. Make absolutely sure that there are no partitions of this group that are still operating somewhere in the network, but not accessible from your location.

Connect to an instance which contains the cluster's metadata, then use the *`Cluster`*`.forceQuorumUsingPartitionOf(`*`instance`*`)` operation, which restores the cluster based on the metadata on *`instance`*, and then all the instances that are `ONLINE` from the point of view of the given instance definition are added to the restored cluster.

```
mysql-js> cluster.forceQuorumUsingPartitionOf("icadmin@ic-1:3306")

  Restoring replicaset 'default' from loss of quorum, by using the partition composed of [icadmin@ic-1:

  Please provide the password for 'icadmin@ic-1:3306': ******
  Restoring the InnoDB cluster ...

  The InnoDB cluster was successfully restored using the partition from the instance 'icadmin@ic-1:3306

  WARNING: To avoid a split-brain scenario, ensure that all other members of the replicaset
  are removed or joined back to the group that was restored.
```

In the event that an instance is not automatically added to the cluster, for example if its settings were not persisted, use *`Cluster`*`.rejoinInstance()` to manually add the instance back to the cluster.

The restored cluster might not, and does not have to, consist of all of the original instances which made up the cluster. For example, if the original cluster consisted of the following five instances:

- `ic-1`
- `ic-2`
- `ic-3`

- `ic-4`
- `ic-5`

and the cluster experiences a split-brain scenario, with `ic-1`, `ic-2`, and `ic-3` forming one partition while `ic-4` and `ic-5` form another partition. If you connect to `ic-1` and issue `Cluster.forceQuorumUsingPartitionOf('icadmin@ic-1:3306')` to restore the cluster the resulting cluster would consist of these three instances:

- `ic-1`
- `ic-2`
- `ic-3`

because `ic-1` sees `ic-2` and `ic-3` as `ONLINE` and does not see `ic-4` and `ic-5`.

7.8.3 Rebooting a Cluster from a Major Outage

If your cluster experiences a complete outage you can reconfigure it using `dba.rebootClusterFromCompleteOutage()`. This operation enables you to connect to one of the cluster's MySQL instances and use its metadata to recover the cluster.

A complete outage means that group replication has stopped on all member instances.

Note

Ensure all cluster members are started before running `dba.rebootClusterFromCompleteOutage()`. The command will fail if any of the cluster members are unreachable.

This check is ignored if the cluster is INVALIDATED and is a member of a ClusterSet.

Connect to the most up-to-date instance and run the following command:

```
JS> var cluster = dba.rebootClusterFromCompleteOutage()
```

If all members have the same GTID set, the member to which you are currently connected becomes the primary. See Selecting a Primary with rebootClusterFromCompleteOutage.

The `dba.rebootClusterFromCompleteOutage()` operation follows these steps to ensure the cluster is correctly reconfigured:

- Cluster metadata and the cluster topology is retrieved from the current instance.
- If a cluster member is in RECOVERING or ERROR, and all other members are OFFLINE or ERROR, `dba.rebootClusterFromCompleteOutage()` attempts to stop Group Replication on that member. If Group Replication fails to stop, the command stops and displays an error.
- The InnoDB Cluster metadata found on the instance which MySQL Shell is currently connected to is checked to see if it contains the GTID superset. If the currently connected instance does not contain the GTID superset, the operation aborts with that information.

  See GTID Superset.
- If the instance contains the GTID superset, the cluster is recovered based on the metadata stored in that instance.
- MySQL Shell checks which instances of the cluster are currently reachable and fails if any member is currently unreachable.

Note

It is possible to bypass this check with the `force` option. This reboots the cluster using the remaining contactable members.

See Force Option.

- Similarly, MySQL Shell detects instances which are currently not reachable. It is not possible to add or remove former members to the cluster as part of the `dba.rebootClusterFromCompleteOutage()` command, if they are currently unreachable.
- If enabled on the primary instance of the cluster, while in single-primary mode, `super_read_only` is disabled.

GTID Superset

To reboot the cluster, you must connect to the member with the GTID superset, which means the instance which had applied the most transactions before the outage.

To determine which member has the GTID superset, do one of the following:

- Connect to an instance and run `dba.rebootClusterFromCompleteOutage()` with `dryRun: true`. The generated report returns information similar to the following:.

  ```
  Switching over to instance '127.0.0.1:4001' to be used as seed.
  ```

  This indicates the member with the GTID superset.

  Running `dba.rebootClusterFromCompleteOutage()` against a member with a lower GTID set results in an error.

- Connect to each instance in turn and run the following in `SQL` mode:

  ```
  SHOW VARIABLES LIKE 'gtid_executed';
  ```

  The instance which has applied the largest GTID Sets of transactions contains the GTID superset.

Note

It is possible to override this behavior, and use an instance with a lower GTID set, by running `dba.rebootClusterFromCompleteOutage()` with the `force` option.

This makes the selected member the primary and discards any transactions not included in the selected member's GTID set.

If this process fails, and the cluster metadata has become badly corrupted, you might need to drop the metadata and create the cluster again from scratch. You can drop the cluster metadata using `dba.dropMetadataSchema()`.

Warning

The `dba.dropMetadataSchema()` method should only be used as a last resort, when it is not possible to restore the cluster. It cannot be undone.

If you are using MySQL Router with the cluster, when you drop the metadata, all current connections are dropped and new connections are forbidden. This causes a full outage.

Options

`dba.rebootClusterFromCompleteOutage()` has the following options:

- `force: true | false (default)`: If true, the operation must be executed even if some members of the Cluster cannot be reached, or the primary instance selected has a diverging or lower GTID_SET. See Force Option
- `dryRun: true | false (default)`: performs all validations and steps of the command, but no changes are made. A report is displayed when finished. See Testing rebootClusterFromCompleteOutage.
- `primary`: Instance definition representing the instance that must be selected as the primary. See Selecting a Primary with rebootClusterFromCompleteOutage.
- `switchCommunicationStack: mysql | xcom`: The Group Replication protocol stack to be used by the Cluster after the reboot. See Section 7.5.9, “Configuring the Group Replication Communication Stack”.
- `ipAllowList`: The list of hosts allowed to connect to the instance for Group Replication traffic when using the `XCOM` protocol stack.
- `localAddress`: string value with the Group Replication local address to use instead of the automatically generated one when using the `XCOM` protocol stack.

Force Option

The `force` option enables you to ignore the availability of Cluster members or GTID-set divergence in the selected member and reboot the Cluster.

For example, rebooting the Cluster `myCluster`:

```
JS> var cluster = dba.rebootClusterFromCompleteOutage("myCluster",{force: true})
```

The `force` option is not permitted in the following situations:

- If the Cluster belongs to a ClusterSet and is INVALIDATED or the primary Cluster is not in global status OK,
- The Cluster belongs to a ClusterSet, is the primary Cluster, and is INVALIDATED.

It is not possible to add or rejoin instances with `rebootClusterFromCompleteOutage`.
If you used `force` to ignore unreachable members and reboot your Cluster, you must use `cluster.rejoinInstance()` to add the unreachable members to the Cluster.

Selecting a Primary with rebootClusterFromCompleteOutage

You can define the Cluster primary in one of the following ways:

- Define the `primary` option in the `dba.rebootClusterFromCompleteOutage()` command.

  For example, rebooting the Cluster `myCluster` and setting the member running on the local machine, on port 4001, as the primary:

  ```
  var cluster = dba.rebootClusterFromCompleteOutage("myCluster",{primary: "127.0.0.1:4001"})
  ```

- By using the `primary` option with the `force` option on a Cluster member with a lower GTID set than another member.

Testing rebootClusterFromCompleteOutage

You can test the changes by using the `dryRun` option. This option validates the command and its options and generates a log of results. An exception is thrown if there is a problem with the proposed changes.

The following example shows a dry run of rebooting the Cluster, `myCluster`, setting the primary to the local member running on port 4001, and the log message it returns:

```
JS > var cluster = dba.rebootClusterFromCompleteOutage("myCluster",{primary: "127.0.0.1:4001", dryRun:

NOTE: dryRun option was specified. Validations will be executed, but no changes will be applied.
Cluster instances: '127.0.0.1:4000' (OFFLINE), '127.0.0.1:4001' (OFFLINE), '127.0.0.1:4002' (OFFLINE)
Switching over to instance '127.0.0.1:4001' to be used as seed.
dryRun finished.
```

Considerations for ClusterSet and ReplicaSet

`rebootClusterFromCompleteOutage` performs the following checks and generates a warning if the Cluster does not meet the requirements:

- Confirms the Replica Cluster was not forcibly removed from the ClusterSet.
- Confirms the ClusterSet's primary Cluster is reachable.
- Checks the Cluster for errant transactions which are not View Change Log Events (VCLE). See How Distributed Recovery Works.
- Confirms the Cluster's executed transaction set (`GTID_EXECUTED`) is not empty.

The command automatically rejoins a Replica Cluster to the ClusterSet, ensuring the ClusterSet replication channel is configured for all Cluster members.

Switching Communication Stack

You can switch communication stack during a `dba.rebootClusterFromCompleteOutage()` operation.

For example:

```
js> dba.rebootClusterFromCompleteOutage("testcluster", {switchCommunicationStack: "mysql"})
```

Switching from the `MYSQL` protocol to `XCOM` requires an additional network address for the `localAddress` and may also require you to define `ipAllowList` values.

7.8.4 Rescanning a Cluster

If you make configuration changes to a cluster outside of the AdminAPI commands, for example by changing an instance's configuration manually to resolve configuration issues or after the loss of an instance, you need to update the InnoDB Cluster metadata so that it matches the current configuration of instances. In these cases, use the *`Cluster`*`.rescan()` operation, which enables you to update the InnoDB Cluster metadata either manually or using an interactive wizard. The *`Cluster`*`.rescan()` operation can detect new active instances that are not registered in the metadata and add them, or obsolete instances (no longer active) still registered in the metadata, and remove them. You can automatically update the metadata depending on the instances found by the command, or you can specify a list of instance addresses to either add to the metadata or remove from the metadata. You can also update the topology mode stored in the metadata, for example after changing from single-primary mode to multi-primary mode outside of AdminAPI.

Note

`group_replication_transaction_size_limit` is set to the maximum value in Replica Clusters. The original value is stored in the metadata schema and is restored by *`Cluster`*`.rescan()` in the event of a switchover or failover.

The syntax of the command is `Cluster.rescan([options])`. The `options` dictionary supports the following:

- `interactive`: Boolean value used to disable or enable the wizards in the command execution. Controls whether prompts and confirmations are provided. The default value is equal to MySQL Shell wizard mode, specified by `shell.options.useWizards`.
- `addInstances`: list with the connection data of the new active instances to add to the metadata, or "auto" to automatically add missing instances to the metadata. The value "auto" is case-insensitive.
  - Instances specified in the list are added to the metadata, without prompting for confirmation
  - In interactive mode, you are prompted to confirm the addition of newly discovered instances that are not included in the `addInstances` option
  - In non-interactive mode, newly discovered instances that are not included in the `addInstances` option are reported in the output, but you are not prompted to add them
- `removeInstances`: list with the connection data of the obsolete instances to remove from the metadata, or "auto" to automatically remove obsolete instances from the metadata.
  - Instances specified in the list are removed from the metadata, without prompting for confirmation
  - In interactive mode, you are prompted to confirm the removal of obsolete instances that are not included in the `removeInstances` option
  - In non-interactive mode, obsolete instances that are not included in the `removeInstances` option are reported in the output but you are not prompted to remove them
- `updateTopologyMode`: Boolean value used to indicate if the topology mode (single-primary or multi-primary) in the metadata should be updated (true) or not (false) to match the one being used by the cluster. By default, the metadata is not updated (false).
  - If the value is `true` then the InnoDB Cluster metadata is compared to the current mode being used by Group Replication, and the metadata is updated if necessary. Use this option to update the metadata after making changes to the topology mode of your cluster outside of AdminAPI.
  - If the value is `false` then InnoDB Cluster metadata about the cluster's topology mode is not updated even if it is different from the topology used by the cluster's Group Replication group
  - If the option is not specified and the topology mode in the metadata is different from the topology used by the cluster's Group Replication group, then:
    - In interactive mode, you are prompted to confirm the update of the topology mode in the metadata
    - In non-interactive mode, if there is a difference between the topology used by the cluster's Group Replication group and the InnoDB Cluster metadata, it is reported and no changes are made to the metadata
  - When the metadata topology mode is updated to match the Group Replication mode, the auto-increment settings on all instances are updated as described at Section 7.5.7, "InnoDB Cluster and Auto-increment".
- `updateViewChangeUuid`: Boolean value used to indicate if a value should be generated and set for the `group_replication_view_change_uuid` system variable on the cluster instances. This system variable supplies an alternative UUID for view change events generated by the group. For MySQL Server instances at release 8.0.27 and above, for an InnoDB Cluster that is part of an InnoDB ClusterSet, the `group_replication_view_change_uuid` system variable is required and must be set to the same value on all member servers in the cluster. From MySQL Shell 8.0.27, an InnoDB Cluster that is created using the `dba.createCluster()` command gets a value

generated and set for the system variable on all the member servers. An InnoDB Cluster created before MySQL Shell 8.0.27 might not have the system variable set, but the InnoDB ClusterSet creation process checks for this and fails with a warning if it is absent.

By default, `updateViewChangeUuid` is set to `false`, and if the system variable is not found or does not match on any of the instances, a warning message is returned to let you know you must set a value for the system variable and reboot the InnoDB Cluster. If you set `updateViewChangeUuid` to `true`, the rescan operation generates and sets a value for `group_replication_view_change_uuid` on all the member servers, following which you must reboot the cluster to implement the changes. Before MySQL Shell 8.0.29, this option is not available, and the *`Cluster`*`.rescan()` command automatically generates and sets the system variable value in the same way as if `true` was set, with a cluster reboot required afterwards to implement the changes. When you have rebooted the cluster, you can retry the InnoDB ClusterSet creation process.

- `upgradeCommProtocol`: Boolean value used to indicate if the Group Replication communication protocol version should be upgraded (true) or not (false) to the version supported by the instance in the cluster that is at the lowest MySQL release. By default, the communication protocol version is not upgraded (false). AdminAPI operations before MySQL Shell 8.0.26 upgraded automatically where possible, but the process can cause delays in the cluster. From MySQL Shell 8.0.26, AdminAPI operations that cause a topology change return a message if the communication protocol version can be upgraded, and you can use this option to carry out the upgrade at a suitable time. It is advisable to upgrade to the highest available version of the Group Replication communication protocol to support the latest features, such as message fragmentation for large transactions. For more information, see Setting a Group's Communication Protocol Version.

  - If the value is `true` then the Group Replication communication protocol version is upgraded to the version supported by the instance in the cluster that is at the lowest MySQL release.
  - If the value is `false` then the Group Replication communication protocol version is not upgraded.

7.8.5 Fencing a Cluster

Following an emergency failover, and there is a risk of the transaction sets differing between parts of the ClusterSet, you have to fence the cluster either from write traffic or all traffic. Even though you primarily use fencing on clusters belonging to a clusterset, it is also possible to fence standalone clusters from all traffic.

From MySQL Shell 8.0.28, three fencing operations are available:

- `<Cluster>.fenceWrites()`: Stops write traffic to a primary cluster of a ClusterSet.
- `<Cluster>.unfenceWrites()`: Resumes write traffic.
- `<Cluster>.fenceAllTraffic()`: Fences a cluster from all traffic.

For more details, see Section 8.9.1, “Fencing Clusters in an InnoDB ClusterSet”.

7.9 Modifying or Dissolving an InnoDB Cluster

This section explains how to change an InnoDB Cluster from single-primary to multi-primary mode or the other way around, how to remove server instances from an InnoDB Cluster, and how to dissolve an InnoDB Cluster that you no longer need.

- Changing a Cluster's Topology
- Removing Instances from an InnoDB Cluster
- Dissolving an InnoDB Cluster

Changing a Cluster's Topology

By default, an InnoDB Cluster runs in single-primary mode, where the cluster has one primary server that accepts read and write queries (R/W), and all of the remaining instances in the cluster accept only read queries (R/O). When you configure a cluster to run in multi-primary mode, all of the instances in the cluster are primaries, which means that they accept both read and write queries (R/W). If a cluster has all of its instances running MySQL server version 8.0.15 or later, you can make changes to the topology of the cluster while the cluster is online. In previous versions it was necessary to completely dissolve and re-create the cluster to make the configuration changes. This uses the group action coordinator exposed through the functions described at Configuring an Online Group, and as such you should observe the rules for configuring online groups.

Note

Multi-primary mode is considered an advanced mode.

Usually a single-primary cluster elects a new primary when the current primary leaves the cluster unexpectedly, for example due to an unexpected halt. The election process is normally used to choose which of the current secondaries becomes the new primary. To override the election process and force a specific server instance in the underlying Group Replication group to become the new primary, use the `Cluster.setPrimaryInstance(instance[, options)` function, where `instance` specifies the connection to the instance which should become the new primary. From MySQL Shell 8.0.29, you can use the `runningTransactionsTimeout` option to add a timeout between 0 and 3600 seconds for transactions that are running when you use the function. When you set a timeout, incoming transactions after the command is issued are rejected.

You can change the mode (sometimes described as the topology) which a cluster is running in between single-primary and multi-primary using the following operations:

- `Cluster.switchToMultiPrimaryMode()`, which switches the cluster to multi-primary mode. All instances become primaries.
- `Cluster.switchToSinglePrimaryMode([instance])`, which switches the cluster to single-primary mode. If `instance` is specified, it becomes the primary and all the other instances become secondaries. If `instance` is not specified, the new primary is the instance with the highest member weight (and the lowest UUID in case of a tie on member weight).

Removing Instances from an InnoDB Cluster

You can remove an instance from a cluster at any time should you wish to do so. This can be done with the `Cluster.removeInstance(instance)` method, as in the following example:

```
mysql-js> cluster.removeInstance('root@localhost:3310')

The instance will be removed from the InnoDB cluster. Depending on the instance
being the Seed or not, the Metadata session might become invalid. If so, please
start a new session to the Metadata Storage R/W instance.

Attempting to leave from the Group Replication group...

The instance 'localhost:3310' was successfully removed from the cluster.
```

You can optionally pass in the `interactive` option to control whether you are prompted to confirm the removal of the instance from the cluster. In interactive mode, you are prompted to continue with the removal of the instance (or not) in case it is not reachable. The `cluster.removeInstance()` operation ensures that the instance is removed from the metadata of all the cluster members which are `ONLINE`, and the instance itself. The last instance that remains in `ONLINE` status in an InnoDB Cluster cannot be removed using this operation.

When the instance being removed has transactions which still need to be applied, AdminAPI waits for up to the number of seconds configured by the MySQL Shell `dba.gtidWaitTimeout` option for transactions (GTIDs) to be applied. The MySQL Shell `dba.gtidWaitTimeout` option has a default

value of 60 seconds, see Section 13.4, “Configuring MySQL Shell Options” for information on changing the default. If the timeout value defined by `dba.gtidWaitTimeout` is reached when waiting for transactions to be applied and the `force` option is `false` (or not defined) then an error is issued and the remove operation is aborted. If the timeout value defined by `dba.gtidWaitTimeout` is reached when waiting for transactions to be applied and the `force` option is set to `true` then the operation continues without an error and removes the instance from the cluster.

Note

The `force` option of `Cluster.removeInstance(instance)` forces removal of the instance from the Cluster's metadata. This is useful if the instance is no longer a member, but is still registered as part of the Cluster. This option has no effect on healthy, contactable instances, and affects only unreachable instances or instances which are otherwise unable to synchronize with the Cluster.

Dissolving an InnoDB Cluster

To dissolve an InnoDB Cluster you connect to a read-write instance, for example the primary in a single-primary cluster, and use the `Cluster.dissolve()` command. This removes all metadata and configuration associated with the cluster, and disables Group Replication on the instances. Any data that was replicated between the instances is not removed.

Important

There is no way to undo the dissolving of a cluster. To create it again use `dba.createCluster()`.

The `Cluster.dissolve()` operation can only configure instances which are `ONLINE` or reachable. If members of a cluster cannot be reached by the member where you issued the `Cluster.dissolve()` command you have to decide how the dissolve operation should proceed. If there is any chance you want to rejoin any instances that are identified as missing from the cluster, it is strongly recommended to cancel the dissolve operation and first bring the missing instances back online, before proceeding with a dissolve operation. This ensures that all instances can have their metadata updated correctly, and that there is no chance of a split-brain situation. However, if the instances from the cluster which cannot be reached have permanently left the cluster there could be no choice but to force the dissolve operation, which means that the missing instances are ignored and only online instances are affected by the operation.

Warning

Forcing the dissolve operation to ignore cluster instances can result in instances which could not be reached during the dissolve operation continuing to operate, creating the risk of a split-brain situation. Only ever force a dissolve operation to ignore missing instances if you are sure there is no chance of the instance coming online again.

In interactive mode, if members of a cluster are not reachable during a dissolve operation then an interactive prompt is displayed, for example:

```
mysql-js> Cluster.dissolve()
The cluster still has the following registered instances:
{
    "clusterName": "testCluster",
    "defaultReplicaSet": {
        "name": "default",
        "topology": [
            {
                "address": "ic-1:3306",
                "label": "ic-1:3306",
                "role": "HA"
            },
```

```
            {
                "address": "ic-2:3306",
                "label": "ic-2:3306",
                "role": "HA"
            },
            {
                "address": "ic-3:3306",
                "label": "ic-3:3306",
                "role": "HA"
            }
        ]
    }
}
WARNING: You are about to dissolve the whole cluster and lose the high
availability features provided by it. This operation cannot be reverted. All
members will be removed from the cluster and replication will be stopped,
internal recovery user accounts and the cluster metadata will be dropped. User
data will be maintained intact in all instances.

Are you sure you want to dissolve the cluster? [y/N]: y

ERROR: The instance 'ic-2:3306' cannot be removed because it is on a '(MISSING)'
state. Please bring the instance back ONLINE and try to dissolve the cluster
again. If the instance is permanently not reachable, then you can choose to
proceed with the operation and only remove the instance from the Cluster
Metadata.

Do you want to continue anyway (only the instance metadata will be removed)?
[y/N]: y

Instance 'ic-3:3306' is attempting to leave the cluster...  Instance 'ic-1:3306'
is attempting to leave the cluster...

WARNING: The cluster was successfully dissolved, but the following instance was
skipped: 'ic-2:3306'. Please make sure this instance is permanently unavailable
or take any necessary manual action to ensure the cluster is fully dissolved.
```

In this example, the cluster consisted of three instances, one of which was offline when dissolve was issued. The error is caught, and you are given the choice how to proceed. In this case the missing `ic-2` instance is ignored and the reachable members have their metadata updated.

When MySQL Shell is running in non-interactive mode, for example when running a batch file, you can configure the behavior of the *`Cluster`*`.dissolve()` operation using the `force` option. To force the dissolve operation to ignore any instances which are unreachable, issue:

```
mysql-js> Cluster.dissolve({force: true})
```

Any instances which can be reached are removed from the cluster, and any unreachable instances are ignored. The warnings in this section about forcing the removal of missing instances from a cluster apply equally to this technique of forcing the dissolve operation.

You can also use the `interactive` option with the *`Cluster`*`.dissolve()` operation to override the mode which MySQL Shell is running in, for example to make the interactive prompt appear when running a batch script. For example:

```
mysql-js> Cluster.dissolve({interactive: true})
```

The `dba.gtidWaitTimeout` MySQL Shell option configures how long the *`Cluster`*`.dissolve()` operation waits for cluster transactions to be applied before removing a target instance from the cluster, but only if the target instance is `ONLINE`. An error is issued if the timeout is reached when waiting for cluster transactions to be applied on any of the instances being removed, except if force: true is used, which skips the error in that case.

Note

After issuing `cluster.dissolve()`, any variable assigned to the *`Cluster`* object is no longer valid.

7.10 Upgrade InnoDB Cluster

This section explains how to upgrade your cluster. Much of the process of upgrading an InnoDB Cluster consists of upgrading the instances in the same way as documented at Upgrading Group Replication. This section focuses on the additional considerations for upgrading InnoDB Cluster. Before starting an upgrade, you can use the MySQL Shell Section 11.1, “Upgrade Checker Utility” to verify instances are ready for the upgrade.

Important

Due to an error in a metadata query, MySQL Shell 8.0.27 cannot be used to manage an InnoDB Cluster running MySQL Server 8.0.25. To work around this issue, upgrade MySQL Server to the 8.0.26 or 8.0.27 release on the InnoDB Cluster member instances before using MySQL Shell 8.0.27 with the cluster. The issue will be fixed in MySQL Shell 8.0.28.

From version 8.0.19, if you try to bootstrap MySQL Router against a cluster and it discovers that the metadata version is 0.0.0, this indicates that a metadata upgrade is in progress, and the bootstrap fails. Wait for the metadata upgrade to complete before you try to bootstrap again. When MySQL Router is operating normally (not bootstrapping) and it discovers the metadata version is 0.0.0 (upgrade in progress), then MySQL Router does not proceed with refreshing the metadata that it was about to begin. Instead, MySQL Router continues using the last metadata it has cached. All the existing user connections are maintained, and any new connections are routed according to the cached metadata. The Metadata refresh restarts when the Metadata version is no longer 0.0.0. In the regular (not bootstrapping) mode, MySQL Router works with the metadata of version 1.x.x and 2.x.x. The version can change between TTL refreshes. This change ensures routing continues while you upgrade the cluster.

Although it is possible to have instances in a cluster that run multiple MySQL versions, for example, version 5.7 and 8.0. Such a mix is not recommended for prolonged use. For example, in a cluster using different versions, an instance running version 5.7 leaves the cluster, and then MySQL Clone is used for a recovery operation, the instance running the lower version can no longer join the cluster because the `BACKUP_ADMIN` privilege becomes a requirement. Running a cluster with multiple versions is intended as a temporary situation to aid in migration from one version to another and is not suitable for long term use.

7.10.1 InnoDB Cluster Upgrade

To upgrade the server instances in an InnoDB Cluster, complete the following steps:

1. Upgrade MySQL Router.
2. Upgrade MySQL Shell.
3. Upgrade MySQL Server.
4. Post Upgrade Status Check.

Check the versions of the installed binaries:

- **mysqlrouter --version**: Checks the version of MySQL Router installed.
- **mysqlsh --version**: Checks the version of MySQL Shell installed.
- **mysqld --version**: Checks the version of MySQL Server installed.

Upgrade MySQL Router.

To upgrade MySQL Router, complete the following steps:

1. Stop MySQL Router.

   On a Unix system, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected when you bootstrapped the router. These files include `stop.sh`. Navigate to this directory and issue the following command:

   ```
   ./stop.sh
   ```

   On Microsoft Windows, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected when you bootstrapped the router. These files include `stop.ps1`. Navigate to this directory and issue the following command:

   ```
   .\stop.ps1
   ```

   Or on a Linux system using `systemd`, stop the MySQL Router service by issuing the following command:

   ```
   systemctl stop mysqlrouter.service
   ```

   Otherwise, kill the *process ID (PID)* of the associated *mysqlrouter* process.

2. Obtain and install the latest version of MySQL Router.

3. Start MySQL Router.

   On a Unix system, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected. These files include `start.sh`. Navigate to the directory and issue the following command:

   ```
   ./start.sh
   ```

   If the path to the new router has changed, you must update the `start.sh` shell script to reflect the path.

   ```
   #!/bin/bash
   basedir=/tmp/myrouter
   ROUTER_PID=$basedir/mysqlrouter.pid /usr/bin/mysqlrouter -c $basedir/mysqlrouter.conf &
   disown %-
   ```

   If you upgrade MySQL Router manually, opposed to using package management, you can update the `basedir=`. If you bootstrap the router again the `start.sh` shell script is regenerated.

   Or on a Linux system using `systemd`, start the MySQL Router service by issuing:

   ```
   systemctl start mysqlrouter.service
   ```

   On Microsoft Windows, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected. These files include `start.ps1`. Navigate to the directory and issue the following command:

   ```
   .\start.ps1
   ```

   On starting MySQL Router using the new router binaries, check the version of the router is upgraded by issuing the following command:

   ```
   mysqlrouter --version
   ```

Upgrade MySQL Shell

Upgrade MySQL Shell by installing the new binaries, and stopping and starting MySQL Shell:

1. Obtain and install the latest version of MySQL Shell.

2. Stop and quit MySQL Shell by issuing:

```
\q
```

3. Restart MySQL Shell from the command line, by issuing:

```
mysqlsh
```

4. Upgrade the InnoDB Cluster Metadata:

   - To upgrade an InnoDB Cluster's metadata, connect MySQL Shell's global session to your cluster and use the `dba.upgradeMetadata()` operation to upgrade the cluster's metadata to the new metadata.

Metadata Upgrade

The Metadata Upgrade may do nothing if the Cluster already uses the latest version.

Upgrade MySQL Server

Upgrade MySQL Server by upgrading all secondary instances before upgrading the primary instance.

Upgrading MySQL Server is optional

Upgrading MySQL Server is optional. Server upgrades can have a greater impact than upgrading MySQL Shell and MySQL Router. Also, you should always keep MySQL Shell and MySQL Router at the latest version, even if the server is not; this is true for InnoDB Clusters and ReplicaSets.

1. Stop MySQL Server by issuing one of the following commands:

   - If MySQL Server is using *systemd* issue:

```
systemctl stop mysqld
```

   - If MySQL Server is using *init.d* issue:

```
/etc/init.d/mysql stop
```

   - If MySQL Server is using *service* issue:

```
service mysql stop
```

   - If you deployed MySQL Server on Microsoft Windows issue:

```
mysqladmin -u root -p shutdown
```

2. Obtain and install the latest version of MySQL Server.

3. Start MySQL Server by issuing one of the following commands:

   - If MySQL Server is using *systemd* issue:

```
systemctl start mysqld
```

   - If MySQL Server is using *init.d* issue:

```
/etc/init.d/mysql start
```

   - If MySQL Server is using *service* issue:

```
service mysql start
```

   - If you deployed MySQL Server on Microsoft Windows issue:

```
mysqld
```

4. When all the secondary instances are upgraded, upgrade the primary instance to complete the upgrade process.

Post Upgrade Status Check

After upgrading MySQL Router, MySQL Shell, and MySQL Servers are upgraded:

1. Check the Cluster by issuing `<Cluster>.status()`. For more information about `<Cluster>.status()`, see Checking a Cluster's Status with `Cluster.status()`.

2. Resolve any `clusterErrors` and `statusText` returned by the `<Cluster>.status()` operation.

These commands allow you to check that the upgrade has been successful or if you need to complete any additional steps.

Note

The additional steps depend; on how many versions you are skipping, what version you are upgrading, and from what version you are coming.

1. Check the status of each InnoDB Cluster, by issuing `<Cluster>.status()`.

   In the following example, `<Cluster>.status({extended: true})`, used to provide more detailed information about the status of the Cluster, returns two issues:

```
mysqlsh> <Cluster>.status({extended: true});
{
    "clusterName": "MyCluster",
    "defaultReplicaSet": {
        "GRProtocolVersion": "8.0.16",
        "groupName": "459ec434-8926-11ec-b8c3-02001707f44a",
        "groupViewChangeUuid": "AUTOMATIC",
        "groupViewId": "16443558036060755:13",
        "name": "default",
        "ssl": "REQUIRED",
        "status": "OK",
        "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
        "topology": {
            "example-el7-1644251369:33311": {
                "address": "example-el7-1644251369:33311",
                "applierWorkerThreads": 4,
                "fenceSysVars": [],
                "instanceErrors": [
                    "NOTE: instance server_id is not registered in the metadata.
                           Use cluster.rescan() to update the metadata.",
                    "NOTE: The required parallel-appliers settings are not enabled on the instance.
                           Use dba.configureInstance() to fix it."
                ],
                "memberId": "247131ab-8926-11ec-850b-02001707f44a",
                "memberRole": "PRIMARY",
                "memberState": "ONLINE",
                "mode": "R/W",
                "readReplicas": {},
                "replicationLag": null,
                "role": "HA",
                "status": "ONLINE",
                "version": "8.0.28"
            },
            "example-el7-1644251369:33314": {
                "address": "example-el7-1644251369:33314",
                "applierWorkerThreads": 4,
                "fenceSysVars": [],
                "instanceErrors": [
```

```
                        "NOTE: instance server_id is not registered in the metadata.
                               Use cluster.rescan() to update the metadata.",
                        "NOTE: The required parallel-appliers settings are not enabled on the instar
                               Use dba.configureInstance() to fix it."
                    ],
                    "memberId": "303dcfa7-8926-11ec-a6e5-02001707f44a",
                    "memberRole": "PRIMARY",
                    "memberState": "ONLINE",
                    "mode": "R/W",
                    "readReplicas": {},
                    "replicationLag": null,
                    "role": "HA",
                    "status": "ONLINE",
                    "version": "8.0.28"
                },
                "example-el7-1644251369:33317": {
                    "address": "example-el7-1644251369:33317",
                    "applierWorkerThreads": 4,
                    "fenceSysVars": [],
                    "instanceErrors": [
                        "NOTE: instance server_id is not registered in the metadata.
                               Use cluster.rescan() to update the metadata.",
                        "NOTE: The required parallel-appliers settings are not enabled on the instar
                               Use dba.configureInstance() to fix it."
                    ],
                    "memberId": "3bb2592e-8926-11ec-8b6f-02001707f44a",
                    "memberRole": "PRIMARY",
                    "memberState": "ONLINE",
                    "mode": "R/W",
                    "readReplicas": {},
                    "replicationLag": null,
                    "role": "HA",
                    "status": "ONLINE",
                    "version": "8.0.28"
                }
            },
            "topologyMode": "Multi-Primary"
        },
        "groupInformationSourceMember": "example-el7-1644251369:33311",
        "metadataVersion": "2.1.0"
    }
```

`<Cluster>.status({extended: true})` displays more detailed information about the Cluster. In this example, we use the Boolean value `true`, which is equivalent to `<Cluster>.status({'extended':1})`. For more information, see Checking a Cluster's Status with `Cluster.status()`.

2. Resolve any errors returned by the `<Cluster>.status({extended:1})` operation.

   In this example, the `instanceErrors` suggest that in this upgrade, we should issue `<Cluster>.rescan()` and `dba.configureInstance()` on each member in the Cluster:

```
...
"NOTE: instance server_id is not registered in the
metadata. Use <Cluster>.rescan() to update the metadata.",
"NOTE: The required parallel-appliers settings are not
enabled on the instance. Use dba.configureInstance() to fix it."
...
```

   The `<Cluster>.rescan()` operation enables you to rescan the Cluster for new and obsolete Group Replication instances, as well as changes in the used topology mode. For more information, see Rescanning a Cluster.

```
mysqlsh> <Cluster>.rescan();
Rescanning the cluster...

Result of the rescanning operation for the 'MyCluster1' cluster:
{
    "name": "MyCluster1",
    "newTopologyMode": null,
```

```
    "newlyDiscoveredInstances": [],
    "unavailableInstances": [],
    "updatedInstances": []
}
```

The `dba.configureInstance()` function checks all of the settings required to enable the instance to be used for InnoDB Cluster usage. For more information, see Configuring Production Instances for InnoDB Cluster Usage.

In this example, we issue `dba.configureInstance()` on each member in the InnoDB Cluster, to ensure required the parallel-appliers settings are enabled on the instance:

```
 mysqlsh> dba.configureInstance('cladmin:cladminpw@localhost:33311')
   The instance 'example-el7-1644251369:33311' belongs to an InnoDB Cluster.
   Configuring local MySQL instance listening at port 33311 for use in an InnoDB cluster...

   This instance reports its own address as ^[[1mexample-el7-1644251369:33311^[[0m
   Clients and other cluster members will communicate with it through this address by default.
   If this is not correct, the report_host MySQL system variable should be changed.

   applierWorkerThreads will be set to the default value of 4.

   ^[[36mNOTE: ^[[0mSome configuration options need to be fixed:
   +----------------------------------------+---------------+----------------+-----------------------
   | Variable                               | Current Value | Required Value | Note
   +----------------------------------------+---------------+----------------+-----------------------
   | binlog_transaction_dependency_tracking | COMMIT_ORDER  | WRITESET       | Update the server varia
   +----------------------------------------+---------------+----------------+-----------------------

   Configuring instance...
   The instance 'example-el7-1644251369:33311' was configured to be used in an InnoDB cluster.
```

For information on troubleshooting Cluster Upgrades, see Troubleshooting InnoDB Cluster Upgrades.

7.10.2 Troubleshooting InnoDB Cluster Upgrades

This section covers trouble shooting the upgrade process.

Handling Host Name Changes

MySQL Shell uses the host value of the provided connection parameters as the target hostname used for AdminAPI operations, namely to register the instance in the metadata (for the `dba.createCluster()` and *`Cluster`*`.addInstance()` operations). However, the actual host used for the connection parameters might not match the `hostname` that is used or reported by Group Replication, which uses the value of the `report_host` system variable when it is defined (in other words it is not `NULL`), otherwise the value of `hostname` is used. Therefore, AdminAPI now follows the same logic to register the target instance in the metadata and as the default value for the `group_replication_local_address` variable on instances, instead of using the host value from the instance connection parameters. When the `report_host` variable is set to empty, Group Replication uses an empty value for the host but AdminAPI (for example in commands such as `dba.checkInstanceConfiguration()`, `dba.configureInstance()`, `dba.createCluster()`, and so on) reports the hostname as the value used which is inconsistent with the value reported by Group Replication. If an empty value is set for the `report_host` system variable, an error is generated. (Bug #28285389)

For a cluster created using a MySQL Shell version earlier than 8.0.16, an attempt to reboot the cluster from a complete outage performed using version 8.0.16 or higher results in this error. This is caused by a mismatch of the Metadata values with the `report_host` or `hostname` values reported by the instances. The workaround is to:

1. Identify which of the instances is the “seed”, in other words the one with the most recent GTID set. The `dba.rebootClusterFromCompleteOutage()` operation detects whether the instance is a seed and the operation generates an error if the current session is not connected to the most up-to-date instance.

2. Set the `report_host` system variable to the value that is stored in the Metadata schema for the target instance. This value is the `hostname:port` pair used in the instance definition upon cluster creation. The value can be consulted by querying the `mysql_innodb_cluster_metadata.instances` table.

   For example, suppose a cluster was created using the following sequence of commands:

   ```
   mysql-js> \c clusterAdmin@localhost:3306
   mysql-js> dba.createCluster("myCluster")
   ```

   Therefore the hostname value stored in the metadata is “localhost” and for that reason, `report_host` must be set to “localhost” on the seed.

3. Reboot the cluster using only the seed instance. At the interactive prompts do not add the remaining instances to the cluster.

4. Use *`Cluster`*`.rescan()` to add the other instances back to the cluster.

5. Remove the seed instance from the cluster

6. Stop mysqld on the seed instance and either remove the forced `report_host` setting (step 2), or replace it with the value previously stored in the Metadata value.

7. Restart the seed instance and add it back to the cluster using *`Cluster`*`.addInstance()`

This allows a smooth and complete upgrade of the cluster to the latest MySQL Shell version. Another possibility, that depends on the use-case, is to simply set the value of `report_host` on all cluster members to match what has been registered in the Metadata schema upon cluster creation.

Chapter 8 MySQL InnoDB ClusterSet

Table of Contents

8.1 InnoDB ClusterSet Requirements 177
8.2 InnoDB ClusterSet Limitations 180
8.3 User Accounts for InnoDB ClusterSet 181
8.4 Deploying InnoDB ClusterSet 183
8.5 Integrating MySQL Router With InnoDB ClusterSet 195
8.6 InnoDB ClusterSet Status and Topology 200
8.7 InnoDB ClusterSet Controlled Switchover 208
8.8 InnoDB ClusterSet Emergency Failover 213
8.9 InnoDB ClusterSet Repair and Rejoin 218
    8.9.1 Fencing Clusters in an InnoDB ClusterSet 220
    8.9.2 Inconsistent Transaction Sets (GTID Sets) in InnoDB ClusterSet Clusters 223
    8.9.3 Repairing Member Servers and Clusters in an InnoDB ClusterSet 225
    8.9.4 Removing a Cluster from an InnoDB ClusterSet 227
    8.9.5 Rejoining a Cluster to an InnoDB ClusterSet 229
8.10 Upgrade InnoDB ClusterSet 231

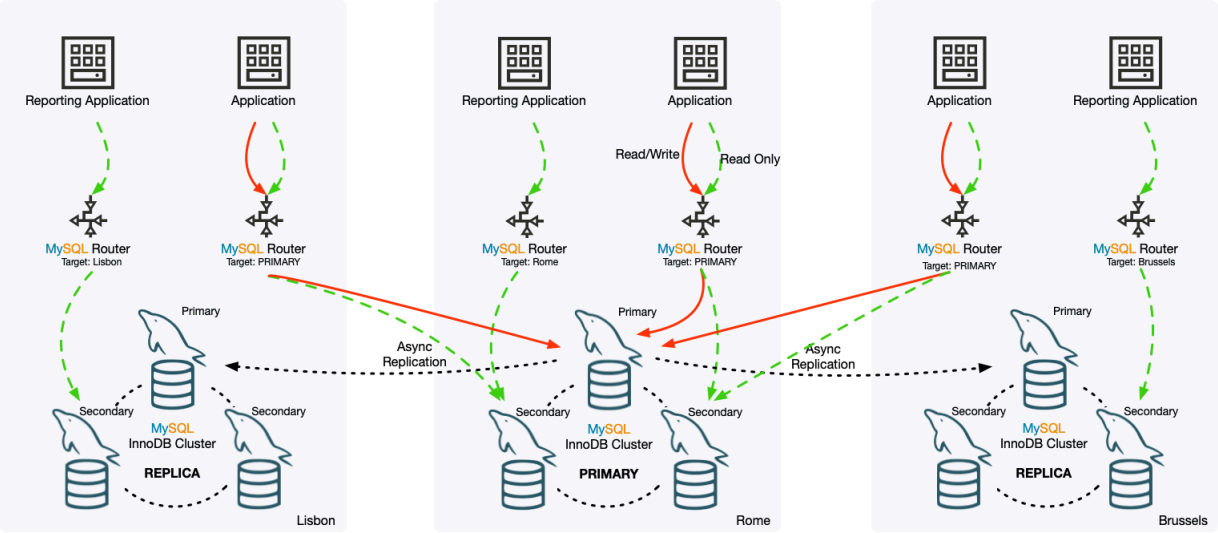
MySQL InnoDB ClusterSet provides disaster tolerance for InnoDB Cluster deployments by linking a primary InnoDB Cluster with one or more replicas of itself in alternate locations, such as different datacenters. InnoDB ClusterSet automatically manages replication from the primary cluster to the replica clusters using a dedicated ClusterSet replication channel. If the primary cluster becomes unavailable due to the loss of the data center or the loss of network connectivity to it, you can make a replica cluster active instead to restore the availability of the service. See Chapter 7, *MySQL InnoDB Cluster* for information on deploying InnoDB Cluster.

Emergency failover between the primary InnoDB Cluster and a replica cluster in an InnoDB ClusterSet deployment can be triggered by an administrator through MySQL Shell (see MySQL Shell 8.0), using AdminAPI (see Section 6.1, “Using MySQL AdminAPI”), which is included with MySQL Shell. You can also carry out a controlled switchover from the primary cluster to a replica cluster while the primary cluster is still available, for example if a configuration change or maintenance is required on the primary cluster. MySQL Router (see MySQL Router 8.0) automatically routes client applications to the right clusters in an InnoDB ClusterSet deployment.

A replica cluster in an InnoDB ClusterSet deployment cannot diverge from the primary cluster while it remains a passive replica, because it does not accept writes. It can be read by applications, although a typical amount of replication lag for asynchronous replication should be expected, so the data might not be complete yet. The minimum size of a replica cluster is a single member server instance, but a minimum of three members is recommended for fault tolerance. If more members are needed, for example because the replica cluster has become a primary cluster through a switchover or failover, you can add further instances at any time through MySQL Shell using AdminAPI. There is no defined limit on the number of replica clusters that you can have in an InnoDB ClusterSet deployment.

The example InnoDB ClusterSet deployment in the following diagram consists of a primary InnoDB Cluster in the Rome datacenter, with replica clusters in the Lisbon and Brussels datacenters. The primary cluster and its replica clusters each consist of three member server instances, one primary and two secondaries.

Figure 8.1 InnoDB ClusterSet Overview



Asynchronous replication channels replicate transactions from the primary cluster to the replica clusters. A ClusterSet replication channel named `clusterset_replication` is set up on each cluster during the InnoDB ClusterSet creation process, and when a cluster is a replica, it uses the channel to replicate transactions from the primary. The underlying Group Replication technology manages the channel and ensures that replication is always taking place between the primary server of the primary cluster (as the sender), and the primary server of the replica cluster (as the receiver). If a new primary is elected for either the primary cluster or the replica cluster, the ClusterSet replication channel is automatically re-established between them.

Although each cluster in the example InnoDB ClusterSet deployment has a primary MySQL server, only the primary server of the primary InnoDB Cluster accepts write traffic from client applications. The replica clusters do not. MySQL Router instances route all write traffic to the primary cluster in the Rome datacenter, where it is handled by the primary server. Most of the read traffic is also routed to the primary cluster, but the reporting applications that only make read requests are specifically routed to the replica cluster in their local datacenter instead, to save on networking resources. Notice that the MySQL Router instances that handle read and write traffic are set to route traffic to the primary InnoDB Cluster in the InnoDB ClusterSet whichever one that is. So if one of the other clusters becomes the primary following a controlled switchover or emergency failover, those MySQL Router instances will route traffic to that cluster instead.

It is important to know that InnoDB ClusterSet prioritizes availability over data consistency in order to maximize disaster tolerance. Consistency within each individual InnoDB Cluster is guaranteed by the underlying Group Replication technology. However, normal replication lag or network partitions can mean that some or all of the replica clusters are not fully consistent with the primary cluster at the time the primary cluster experiences an issue. In these scenarios, if you trigger an emergency failover, any unreplicated or divergent transactions are at risk of being lost, and can only be recovered and reconciled manually (if they can be accessed at all). There is no guarantee that data will be preserved in the event of an emergency failover.

You should therefore always make an attempt to repair or reconnect the primary cluster before triggering an emergency failover. AdminAPI removes the need to work directly with Group Replication to repair an InnoDB Cluster. If the primary cluster cannot be repaired quickly enough or cannot be reached, you can go ahead with the emergency failover to a replica InnoDB Cluster, to restore availability for applications. During a controlled switchover process, data consistency is assured, and the original primary cluster is demoted to a working read-only replica cluster. However, during an emergency failover process, data consistency is not assured, so for safety, the original primary cluster is marked as invalidated during the failover process. If the original primary cluster remains online, it should be shut down as soon as it can be contacted.

You can rejoin an invalidated primary cluster to the InnoDB ClusterSet topology afterwards, provided that there are no issues and the transaction set is consistent with the other clusters in the topology. Checking, restoring, and rejoining the invalidated primary cluster does not happen automatically - an administrator needs to do this using AdminAPI commands. You can choose to repair the invalidated primary cluster and bring it back online, or you can discard the original primary cluster, continue to use the new primary cluster as the primary, and create new replica clusters.

8.1 InnoDB ClusterSet Requirements

The basis of an InnoDB ClusterSet deployment is an existing InnoDB Cluster at MySQL 8.0.27 or higher, and a number of standalone MySQL Server instances that can be formed into replica clusters to provide disaster tolerance for this primary cluster. If you want to try out InnoDB ClusterSet, you can use MySQL Shell 8.0.27 or higher to set up a sandbox deployment on your local machine. You will need to install MySQL Server 8.0.27 or higher and MySQL Router 8.0.27 or higher. Using AdminAPI commands in MySQL Shell, you can create sandbox MySQL Server instances, set up an InnoDB Cluster using some of them, then follow the instructions in this section to set up an InnoDB ClusterSet deployment using the others as replica clusters. For instructions to deploy and manage sandbox instances, see Section 6.8, “AdminAPI MySQL Sandboxes”.

Important

It is recommended that you always use the most recent version of MySQL Shell available. The latest version of MySQL Shell can be used with any GA version of MySQL 5.7 or 8.0.

To set up a production deployment of InnoDB ClusterSet, ensure that you have the following components in place:

- Software components
- InnoDB Cluster
- MySQL Server instances
- MySQL Router instances

Software components

MySQL Server 8.0.27 or higher, MySQL Shell 8.0.27 or higher, and MySQL Router 8.0.27 or higher are required to set up an InnoDB ClusterSet deployment. These are the software components required by AdminAPI. See Section 6.2, “Installing AdminAPI Software Components”.

Always use the most recent versions of MySQL Shell and MySQL Router that are available to you, and ensure that their version is the same as or higher than the MySQL Server release. Both products can manage older MySQL Server releases, but older versions cannot manage features in newer MySQL Server releases.

InnoDB Cluster

An existing InnoDB Cluster that is to be the primary cluster. This is the cluster for which the InnoDB ClusterSet deployment provides disaster tolerance. A Group Replication group can be adopted as an InnoDB Cluster. For instructions to set up an InnoDB Cluster or adopt a Group Replication group, see Section 7.4, “Deploying a Production InnoDB Cluster”.

The InnoDB Cluster that is to be the primary cluster must meet these requirements:

- The cluster must not already be part of an InnoDB ClusterSet deployment. An InnoDB Cluster can only participate in one InnoDB ClusterSet deployment.
- All member server instances in the cluster must be at MySQL 8.0.27 or higher.

- The InnoDB Cluster metadata version must be 2.1.0 or higher. When you carry out any operation on a cluster (for example a `dba.getCluster()` command), AdminAPI warns you if the cluster's metadata needs updating. You can update the metadata to an appropriate version for InnoDB ClusterSet operations by issuing a `dba.upgradeMetadata()` command in MySQL Shell 8.0.27 or higher. Note that after you upgrade a cluster's metadata, it cannot be administered by older MySQL Shell versions. For more information, see Section 6.11, “Upgrade Metadata Schema”.

- The cluster must be in single-primary mode. An InnoDB Cluster can be in single-primary or multi-primary mode, but InnoDB ClusterSet does not support multi-primary mode. You can use a *`cluster`*`.switchToSinglePrimaryMode()` command in MySQL Shell to convert a cluster in multi-primary mode to single-primary mode, and choose an instance to be the primary server.

- For MySQL Server instances at release 8.0.27 and above, for an InnoDB Cluster that is part of an InnoDB ClusterSet, the `group_replication_view_change_uuid` system variable must be set with the same value on all the member servers in the cluster to supply an alternative UUID for view change events. From MySQL 8.0.27, an InnoDB Cluster that is created using the `dba.createCluster()` command gets a value generated and set for the system variable on all the member servers. InnoDB Cluster created before MySQL 8.0.27 might not have the system variable set, but the InnoDB ClusterSet creation process checks for this and fails with a warning if it is absent.

  The *`Cluster`*`.rescan()` command can be used to generate and set a value for `group_replication_view_change_uuid` on all the member servers in an InnoDB Cluster. Before MySQL Shell 8.0.29, the command does this automatically when scanning a cluster, following which you must reboot the cluster to implement the changes. From MySQL Shell 8.0.29, the command returns a warning message to let you know you must set a value for the system variable, or you can enable the option `updateViewChangeUuid` to generate and set a value automatically during the scan. When you have rebooted the cluster, you can retry the InnoDB ClusterSet creation process.

- There must be no inbound replication channels on any member server from servers outside the group. The channels created automatically by Group Replication (`group_replication_applier` and `group_replication_recovery`) are allowed.

- You need to know the InnoDB Cluster server configuration account user name and password for the cluster (see Section 8.3, “User Accounts for InnoDB ClusterSet”). This is the account that was set up using `dba.configureInstance` on the member servers in the InnoDB Cluster. You will need to create this account on the MySQL Server instances that will form the replica clusters, and use it to set them up.

  Note

  You cannot use an InnoDB Cluster administrator account (set up using *`cluster`*`.setupAdminAccount()`) to set up the standalone MySQL Server instances for the replica cluster. *`cluster`*`.setupAdminAccount()` is not available on a standalone instance, and if you create one of those accounts on the standalone instances using `dba.configureInstance` or manually, it will subsequently be replicated from the primary cluster, causing replication to stop with an error.

- At the time when you create the InnoDB ClusterSet deployment, the InnoDB Cluster must be online and healthy, and its primary member server must be reachable using MySQL Shell.

MySQL Server instances

A number of standalone MySQL Server instances which you can make into one or more replica clusters. A minimum of three member servers in each replica cluster is recommended for fault tolerance, although the InnoDB ClusterSet deployment can include replica clusters consisting of a single server instance. In a production deployment for disaster tolerance, each replica cluster would be in an alternate location, such as a different datacenter.

Each of the MySQL Server instances that you use in the replica clusters must meet these requirements:

- The server is not already part of an InnoDB ClusterSet deployment, InnoDB ClusterSet, or InnoDB ReplicaSet.
- You do not need the data on the server. If the server has previously been used for processing transactions, the data will be overwritten when it is made into a member of the replica cluster (unless the transactions happen to be a subset of those on the primary cluster).
- The server is not part of a currently running Group Replication group (even if the individual server has left the group). You cannot adopt an existing Group Replication group or a current or ex-member of it as a replica cluster. If you want to use server instances that are currently in a replication group, issue `STOP GROUP_REPLICATION` on all the members of the group, so that the group is fully offline. The separate server instances can then be made into a replica cluster using AdminAPI.

Important

Exercise caution over using former Group Replication group members as members of an InnoDB ClusterSet replica cluster, especially if you made a lot of changes to the Group Replication configuration options, or if the group was created in a much earlier release and you made configuration changes based on the situation in that release.

The InnoDB ClusterSet replica cluster creation process overwrites any existing persisted Group Replication configuration options for which you specify new settings on the command. It also always overwrites the following system variables, even if you do not specify them on the command: `group_replication_group_name`, `group_replication_group_seeds`, `group_replication_local_address`, `group_replication_view_change_uuid`, and `group_replication_enforce_update_everywhere_checks`. However, other Group Replication configuration options that you have changed are left as they were. These custom settings could potentially interfere with the running or performance of InnoDB ClusterSet, which expects the MySQL 8.0.27 defaults to be used for Group Replication configuration options that are not changed during the InnoDB ClusterSet replica cluster creation process.

If you do want to use a configured Group Replication server, check and remove any customizations if possible, in particular checking that the `group_replication_single_primary_mode` system variable is set to the default of `ON`. The safest option for an ex-Group Replication group member in this situation is to reinstall MySQL Server, rather than upgrading the installation to MySQL 8.0.27.

- The server is at MySQL 8.0.27 or higher. If you want to provision further member servers for the replica cluster by cloning, all the servers must be at the same release and on the same operating system.
- The server has a server ID (`server_id` system variable) and server UUID (`server_uuid` system variable) that are unique in the entire InnoDB ClusterSet, including any offline or unreachable member servers.
- No inbound replication channels are configured on the server. Only the Group Replication channels (`group_replication_applier` and `group_replication_recovery`) are allowed.
- The server can connect to the primary cluster in the InnoDB ClusterSet, and the primary cluster can connect to it.

- At the time when you create the InnoDB ClusterSet deployment, the server must be online and healthy, and reachable using MySQL Shell.

The required user account credentials, InnoDB ClusterSet metadata, and Group Replication configuration will be set up during the InnoDB ClusterSet replica cluster creation process.

MySQL Router instances

One or more MySQL Router instances to route client application traffic to the appropriate clusters in the InnoDB ClusterSet deployment. The recommended deployment of MySQL Router is on the same host as the client application.

Important

If you are using an existing InnoDB Cluster as the primary cluster in your InnoDB ClusterSet deployment, and you bootstrapped MySQL Router against that cluster already, bootstrap it again using the `--force` option against the InnoDB ClusterSet, then stop and restart MySQL Router. The settings in the MySQL Router instance's static configuration file need to be updated for InnoDB ClusterSet. Follow the process described in Section 8.5, “Integrating MySQL Router With InnoDB ClusterSet” to do this.

8.2 InnoDB ClusterSet Limitations

InnoDB ClusterSet uses InnoDB Cluster technology, which in turn uses Group Replication technology. The limitations for both of those technologies therefore apply to server instances used with InnoDB ClusterSet. See Section 7.2, “InnoDB Cluster Limitations” and Group Replication Limitations.

- InnoDB ClusterSet prioritizes availability over consistency in order to maximize disaster tolerance. Normal replication lag or network partitions can mean that some or all of the replica clusters are not fully consistent with the primary cluster at the time the primary cluster experiences an issue. In these scenarios, if you trigger an emergency failover, any unreplicated or divergent transactions are at risk of being lost, and can only be recovered and reconciled manually (if they can be accessed at all). There is no guarantee that data will be preserved in the event of an emergency failover.

  If you cannot tolerate any loss of transactions or data during a failover, instead of using InnoDB ClusterSet as your solution, consider using a single InnoDB Cluster with the member servers deployed across multiple datacenters. Bear in mind that this solution would have a noticeable impact on write performance, as a stable and low latency network is important for InnoDB Cluster member servers to communicate with each other for consensus on transactions.

- InnoDB ClusterSet does not fail over automatically to a replica cluster. Because a loss of transactions is possible, and data consistency is not guaranteed, an administrator must make and implement the decision to carry out an emergency failover. If the original primary cluster remains online, it should be shut down as soon as it can be contacted.

- InnoDB ClusterSet only supports asynchronous replication, and cannot use semisynchronous replication.

- InnoDB ClusterSet only supports single-primary mode for the primary and replica InnoDB Cluster instances. Multi-primary mode is not supported.

- An InnoDB ClusterSet deployment can only contain a single read-write primary cluster. All replica clusters are read-only. An active-active setup, with multiple primary clusters, is not permitted, because data consistency is not guaranteed in the event that a cluster fails.

- An InnoDB Cluster can participate in only one InnoDB ClusterSet deployment. Each InnoDB ClusterSet deployment therefore provides an availability and disaster recovery solution for a single InnoDB Cluster.

- An existing InnoDB Cluster cannot be used as a replica cluster in an InnoDB ClusterSet deployment. A replica cluster must be started from a single server instance, as a new InnoDB Cluster. It is possible to use server instances that are part of a Group Replication group as a replica cluster, but you must completely end the Group Replication group first, and be careful of any customized Group Replication settings that might affect InnoDB ClusterSet. For more information, see Section 8.1, “InnoDB ClusterSet Requirements”.
- InnoDB ClusterSet does not support the use of instances running MySQL Server 5.7. An InnoDB Cluster that includes MySQL 5.7 instances cannot be part of an InnoDB ClusterSet deployment.

8.3 User Accounts for InnoDB ClusterSet

The member servers in an InnoDB Cluster make use of three types of user accounts. One InnoDB Cluster server configuration account is used to configure the server instances for the cluster. One or more InnoDB Cluster administrator accounts can be created for administrators to manage the server instances after the cluster has been set up. One or more MySQL Router accounts can be created for MySQL Router instances to connect to the cluster. Each of the user accounts must exist on all of the member servers in the InnoDB Cluster, with the same user name and the same password.

In an InnoDB ClusterSet deployment, every member server is part of an InnoDB Cluster, so they require the same types of user accounts. The user accounts from the primary cluster are used for all of the clusters in the deployment. Each of the user accounts must exist on every member server in every cluster in the deployment - both the primary cluster and the replica clusters.

InnoDB Cluster server configuration account

This account is used to create and configure the member servers of an InnoDB Cluster and InnoDB ClusterSet deployment. Each member server has only one server configuration account. The same user account name and password must be used on every member server in the cluster. You can use the `root` account on the servers for this purpose, but if you do this, the `root` account on every member server in the cluster must have the same password. This is not recommended for security reasons.

The preferred approach is to create the InnoDB Cluster server configuration account using a `dba.configureInstance()` command with the `clusterAdmin` option. For better security, specify the password at the interactive prompt, otherwise specify it using the `clusterAdminPassword` option. Create the same account, with the same user name and password, in the same way on every server instance that will be part of the InnoDB Cluster - both the instance to which you connect to create the cluster, and the instances that will join the cluster after that.

The `dba.configureInstance()` command grants the account the required permissions automatically. You may set up the account manually if you prefer, granting it the permissions listed in Configuring InnoDB Cluster Administrator Accounts Manually. The account needs full read and write privileges on the InnoDB Cluster metadata tables, in addition to full MySQL administrator privileges.

The InnoDB Cluster server configuration account that you create using the `dba.configureInstance()` operation is *not* replicated to other servers in the InnoDB Cluster or in the InnoDB ClusterSet deployment. MySQL Shell disables binary logging for the `dba.configureInstance()` operation. This means that you must create the account on every server instance individually.

In an InnoDB ClusterSet deployment, the same InnoDB Cluster server configuration account must exist on every server instance

that is used in the deployment. When you set up a replica cluster, you therefore need to issue a `dba.configureInstance()` command with the `clusterAdmin` option to create the account on every server instance that is going to be part of the replica cluster. The command must name the InnoDB Cluster server configuration account from the primary cluster, and you must specify the same password for it. You need to do this step before joining the instances into the replica cluster, so the account is available to configure the replica InnoDB Cluster and the InnoDB ClusterSet deployment metadata and replication.

InnoDB Cluster administrator accounts

These accounts can be used to administer InnoDB Cluster and InnoDB ClusterSet after you have completed the configuration process. You can set up more than one of them. Each account must exist on every member server in an InnoDB Cluster with the same user name and password, and on every member server of every cluster in an InnoDB ClusterSet deployment.

To create an InnoDB Cluster administrator account for an InnoDB ClusterSet deployment, you issue a `cluster.setupAdminAccount()` command on one member server in the primary cluster, after you have added all the instances to that cluster. This command creates an account with the user name and password that you specify, with all the required permissions. A transaction to create an account with `cluster.setupAdminAccount()` is sent to all the other server instances in the cluster to create the account on them.

If the primary InnoDB Cluster already existed when you began to set up the InnoDB ClusterSet deployment, an InnoDB Cluster administrator account likely already exists. In that case, you do not need to issue `cluster.setupAdminAccount()` again, unless you want to create further InnoDB Cluster administrator accounts.

The replica clusters in an InnoDB ClusterSet deployment must have the same set of InnoDB Cluster administrator accounts as the primary cluster. However, when you create the replica clusters, do not attempt to set up the InnoDB Cluster administrator accounts yourself. The transactions to create accounts with `cluster.setupAdminAccount()` are written to the binary log for the primary cluster, and they are automatically replicated from the primary cluster to the replica clusters during the provisioning process. When a replica cluster applies these transactions it creates the same accounts on the member servers in the replica cluster. If the accounts already exist on a server in the replica cluster, this causes a replication error, and the server cannot join the cluster. So you need to wait for them to be replicated.

If a transaction to create an InnoDB Cluster administrator account happened a while back on the primary cluster, it might take some time for the transaction to be replicated and for the account to appear on a replica cluster. Selecting cloning as the provisioning method for the replica cluster speeds up the process.

Note

If the primary InnoDB Cluster was set up in a version before MySQL Shell 8.0.20, the `cluster.setupAdminAccount()`

command might have been used with the `update` option to update the privileges of the InnoDB Cluster server configuration account. This is a special use of the command that is not written to the binary log, and is not replicated to the replica clusters.

When the InnoDB ClusterSet deployment is complete, you may use `cluster.setupAdminAccount()` to create further InnoDB Cluster administrator accounts for the ClusterSet. You can do this while connected to any member server in the InnoDB ClusterSet deployment, either in the primary cluster or in a replica cluster. The transaction to create the account is routed to the primary cluster to be executed, then replicated to all the servers in the replica clusters, where it creates the account on all of them.

InnoDB ClusterSet Replication Accounts

`createClusterSet()` creates one replication account for each member cluster on the primary member of that cluster. These accounts are named using the format: `mysql_innodb_cs_uniqueID`.

For example, in a three-member ClusterSet, accounts similar to the following are created:

- mysql_innodb_cs_8269bdfg6
- mysql_innodb_cs_c24bef67
- mysql_innodb_cs_ed0bb30c

Only one account, the primary, is used for replication. The others are used only if their cluster is promoted to primary.

MySQL Router accounts

These accounts are used by MySQL Router to connect to server instances in an InnoDB Cluster and in an InnoDB ClusterSet deployment. You can set up more than one of them. Each account must exist on every member server in an InnoDB Cluster with the same user name and password, or SSL certificate, and on every member server of every cluster in an InnoDB ClusterSet deployment.

The process to create a MySQL Router account is the same as for an InnoDB Cluster administrator account, but using a `cluster.setupRouterAccount()` command. You create the accounts on one member server in the primary cluster, or use accounts that already exist, if the primary InnoDB Cluster already existed when you began to set up the InnoDB ClusterSet deployment. Then let the replica clusters apply the transactions to create the accounts on their member servers. For instructions to create or upgrade a MySQL Router account, see Section 6.10.2, “Configuring the MySQL Router User”.

8.4 Deploying InnoDB ClusterSet

Follow this procedure to deploy a sandbox or production InnoDB ClusterSet deployment. A sandbox deployment is where all the MySQL server instances and other software run on a single machine. For a production deployment, the server instances and other software are on separate machines.

The procedure assumes you already have the following components, as listed in Section 8.1, “InnoDB ClusterSet Requirements”:

- An existing InnoDB Cluster that meets the requirements stated in Section 8.1, “InnoDB ClusterSet Requirements”. This is the primary cluster that the InnoDB ClusterSet deployment supports.
- MySQL Shell 8.0.27 or higher, connected to the existing InnoDB Cluster. MySQL Shell's AdminAPI commands are used in the deployment procedure.
- MySQL Router 8.0.27 or higher, to bootstrap against InnoDB ClusterSet. MySQL Router instances that you had already bootstrapped against the existing InnoDB Cluster can be reused in an InnoDB ClusterSet deployment, but you need to bootstrap them again to implement the InnoDB ClusterSet configuration.
- A number of standalone MySQL Server instances (which are not part of an InnoDB Cluster or InnoDB ReplicaSet) to make into one or more replica clusters. They must meet the requirements stated in Section 8.1, “InnoDB ClusterSet Requirements”. A minimum of three member servers in each replica cluster is recommended for tolerance of failures.

The user account that you use during the InnoDB ClusterSet deployment procedure is the InnoDB Cluster server configuration account from the primary cluster. This is the account that was created on the primary cluster's member servers using a `dba.configureInstance()` command with the `clusterAdmin` option. Each member server has only one server configuration account. The same user account name and password must be used on every member server in the cluster, and you need to create it on all the servers in the InnoDB ClusterSet deployment. It is possible to use the `root` account as the InnoDB Cluster server configuration account, but this is not recommended, because it means the `root` account on every member server in the cluster must have the same password. For more information, see Section 8.3, “User Accounts for InnoDB ClusterSet”.

To set up the InnoDB ClusterSet deployment, follow this procedure:

1. Connect to any member server in the existing InnoDB Cluster with MySQL Shell, using the InnoDB Cluster server configuration account to make the connection. For example:

   ```
   mysql-js> \connect icadmin@127.0.0.1:3310

   Creating a session to 'icadmin@127.0.0.1:3310'
   Please provide the password for 'icadmin@127.0.0.1:3310': **************
   Save password for 'icadmin@127.0.0.1:3310'? [Y]es/[N]o/Ne[v]er (default No):
   Fetching schema names for autocompletion... Press ^C to stop.
   Closing old connection...
   Your MySQL connection id is 59
   Server version: 8.0.27-commercial MySQL Enterprise Server - Commercial
   No default schema selected; type \use <schema> to set one.
   <ClassicSession:icadmin@127.0.0.1:3310>
   ```

   In this example:

   - *`icadmin@127.0.0.1:3310`* is the URI-like connection string for any member server instance that is online in the InnoDB Cluster.

     The URI-like connection string is comprised of the following elements:

   - *`icadmin`* is the user name for the InnoDB Cluster server configuration account.
   - *`127.0.0.1:3310`* is the host and port for the member server instance, as displayed by the *`cluster`*`.status()` command.

2. Issue a `dba.getCluster()` command to get the `Cluster` object that represents the InnoDB Cluster, assigning it to a variable so that you can work with it. For example:

   ```
   mysql-js> cluster1 = dba.getCluster()
   <Cluster:clusterone>
   ```

   In this example, `clusterone` is the name of the existing InnoDB Cluster, as shown in the `clusterName` field returned by the *`cluster`*`.status()` command, and the returned `Cluster` object is assigned to the variable `cluster1`.

It is important to do this when you are connected to the server instance using the InnoDB Cluster server configuration account. The returned object defaults to using the account it was fetched with for operations where permissions are required. Some operations during the InnoDB ClusterSet deployment process require permissions, and the default user account stored in the object is used for this, so that the process does not need to store any other user accounts.

3. Issue a *cluster*.createClusterSet() command, using the Cluster object, to create the InnoDB ClusterSet with the existing InnoDB Cluster as the primary cluster. For example:

```
mysql-js> myclusterset = cluster1.createClusterSet('testclusterset')

A new ClusterSet will be created based on the Cluster 'clusterone'.

* Validating Cluster 'clusterone' for ClusterSet compliance.

* Creating InnoDB ClusterSet 'testclusterset' on 'clusterone'...

* Updating metadata...

ClusterSet successfully created. Use ClusterSet.createReplicaCluster() to add Replica Clusters to it

<ClusterSet:testclusterset>
```

In this example, clusterone is the name of the existing InnoDB Cluster, cluster1 is the variable to which the returned Cluster object was assigned, testclusterset is the name for the InnoDB ClusterSet that you are creating, and myclusterset is the variable to which the returned ClusterSet object is assigned.

- The *domainName* parameter is required and specifies the name of the InnoDB ClusterSet deployment that you are creating (testclusterset in the example). Only alphanumeric characters, hyphens (-), underscores (_), and periods (.) can be used, and the name must not start with a number. The maximum length is 63 characters.

- Use the dryRun option if you want to carry out validations and log the changes without actually executing them. For example:

```
mysql-js> myclusterset = cluster1.createClusterSet('testclusterset', {dryRun: true})
* Validating Cluster 'clusterone' for ClusterSet compliance.

NOTE: dryRun option was specified. Validations will be executed, but no changes will be applied.
* Creating InnoDB ClusterSet 'clusterset' on 'clusterone'...

* Updating metadata...
dryRun finished.
```

- Use the clusterSetReplicationSslMode option if you want to require or disable encryption (TLS/SSL) for the replication channels in the InnoDB ClusterSet deployment. The default setting, AUTO, enables encryption if the server instance supports it, and disables it if it does not. REQUIRED enables encryption for all the replication channels, and DISABLED disables encryption for all the replication channels. For example:

```
mysql-js> myclusterset = cluster1.createClusterSet("testclusterset", {clusterSetReplicationSslMode
```

As of MySQL Shell 8.0.33, clusterSetReplicationSslMode supports VERIFY_CA and VERIFY_IDENTITY. For example:

```
mysql-js> myclusterset = cluster.createClusterSet("testclusterset", {"clusterSetReplicationSslMode"
```

When you issue the *cluster*.createClusterSet() command, MySQL Shell checks that the target InnoDB Cluster complies with the requirements to become the primary cluster in an InnoDB

ClusterSet deployment, and returns an error if it does not. If the target InnoDB Cluster meets the requirements, MySQL Shell carries out the following setup tasks:

- Updates the metadata schema to include InnoDB ClusterSet metadata.
- Sets the `skip_replica_start` system variable to `ON` on all the member servers so that replication threads are not automatically started.
- Adds the target InnoDB Cluster to the InnoDB ClusterSet in the metadata and marks it as the primary cluster.
- Returns the `ClusterSet` object that represents the InnoDB ClusterSet.

4. Verify that the InnoDB ClusterSet deployment that you have created is healthy by issuing a *`clusterSet`*`.status()` command, using the returned `ClusterSet` object. For example:

```
mysql-js> myclusterset.status()
{
    "clusters": {
        "clusterone": {
            "clusterRole": "PRIMARY",
            "globalStatus": "OK",
            "primary": "127.0.0.1:3310"
        }
    },
    "domainName": "testclusterset",
    "globalPrimaryInstance": "127.0.0.1:3310",
    "primaryCluster": "clusterone",
    "status": "HEALTHY",
    "statusText": "All Clusters available."
}
```

You can also use a *`cluster`*`.status()` command to view the cluster itself. Alternatively, you can select the extended output for *`clusterSet`*`.status()` to see the detailed status for the clusters in the InnoDB ClusterSet topology. For example:

```
mysql-js> myclusterset.status({extended: 1})
{
    "clusters": {
        "clusterone": {
            "clusterRole": "PRIMARY",
            "globalStatus": "OK",
            "primary": "127.0.0.1:3310",
            "status": "OK",
            "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
            "topology": {
                "127.0.0.1:3310": {
                    "address": "127.0.0.1:3310",
                    "memberRole": "PRIMARY",
                    "mode": "R/W",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:3320": {
                    "address": "127.0.0.1:3320",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:3330": {
                    "address": "127.0.0.1:3330",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
```

```
                    "version": "8.0.27"
                }
            },
            "transactionSet": "953a51d5-2690-11ec-ba07-00059a3c7a00:1,c51c1b15-269e-11ec-b9ba-00059a
        }
    },
    "domainName": "testclusterset",
    "globalPrimaryInstance": "127.0.0.1:3310",
    "metadataServer": "127.0.0.1:3310",
    "primaryCluster": "clusterone",
    "status": "HEALTHY",
    "statusText": "All Clusters available."
}
```

See Section 8.6, “InnoDB ClusterSet Status and Topology” for more information and a description of the output from the *`clusterSet`*`.status()` command.

If you want to get the `ClusterSet` object representing the InnoDB ClusterSet for a connected server instance at any time, for example after restarting MySQL Shell, use a `dba.getClusterSet()` or *`cluster`*`.getClusterSet()` command. For example:

```
mysql-js> myclusterset = dba.getClusterSet()
<ClusterSet:testclusterset>
```

Assigning a returned `Cluster` or `ClusterSet` object to a variable enables you to execute further operations against the cluster or ClusterSet using the object's methods. The returned object uses a new session, independent from MySQL Shell's global session. This ensures that if you change the MySQL Shell global session, the `Cluster` or `ClusterSet` object maintains its session to the server instance. Note that when you use the object, the server instance from which you got it must still be online in the InnoDB ClusterSet. If that server instance goes offline, the object no longer works and you will need to get it again from a server that is still online.

5. Create the InnoDB Cluster server configuration account on each of the standalone server instances that will be part of the replica cluster, by issuing a `dba.configureInstance()` command with the `clusterAdmin` option. The account to create is the InnoDB Cluster server configuration account from the primary cluster, which you used to create the ClusterSet. Don't specify any of the InnoDB Cluster administrator accounts (created with *`cluster`*`.setupAdminAccount()`). These will be automatically transferred from the primary cluster to the replica clusters during the provisioning process.

   You do not need to connect to the standalone server instances beforehand, as the connection string is included in the command. In the connection string, use an account with full MySQL administrator permissions, including permissions to create accounts (`WITH GRANT OPTION`). In this example, the `root` account is used:

```
mysql-js> dba.configureInstance('root@127.0.0.1:4410', {clusterAdmin: 'icadmin'})

Please provide the password for 'root@127.0.0.1:4410': ***************
Save password for 'root@127.0.0.1:4410'? [Y]es/[N]o/Ne[v]er (default No):
Configuring local MySQL instance listening at port 4410 for use in an InnoDB cluster...
NOTE: Instance detected as a sandbox.
Please note that sandbox instances are only suitable for deploying test clusters for use within
the same host.

This instance reports its own address as 127.0.0.1:4410
Password for new account: **************
Confirm password: **************

applierWorkerThreads will be set to the default value of 4.

The instance '127.0.0.1:4410' is valid to be used in an InnoDB cluster.

Cluster admin user 'icadmin' created.
The instance '127.0.0.1:4410' is already ready to be used in an InnoDB cluster.
```

```
Successfully enabled parallel appliers.
```

In this example, *root@127.0.0.1:4410* is the URI-like connection string for the standalone server, and *icadmin* is the user name for the InnoDB Cluster server configuration account that will be created on the instance. For better security, specify the password for the InnoDB Cluster server configuration account at the interactive prompt as shown in the example, or you can provide it using the `clusterAdminPassword` option. The `dba.configureInstance()` command grants the account the required permissions automatically, although you may set up the account manually if you prefer, granting it the permissions listed in Configuring InnoDB Cluster Administrator Accounts Manually. For more details of the `dba.configureInstance()` command and its options, see Section 7.4.2, “Configuring Production Instances for InnoDB Cluster Usage”.

When you issue `dba.configureInstance()`, MySQL Shell verifies that the server instance meets the requirements for use with InnoDB Cluster. The requirements for InnoDB ClusterSet will be checked when you issue the commands to create the replica cluster and add instances to it.

6. Connect to any active instance in the primary cluster that is already in the InnoDB ClusterSet deployment, using the InnoDB Cluster server configuration account. Ensure you still have the `ClusterSet` object that was returned when you created the InnoDB ClusterSet, or fetch it again using `dba.getClusterSet()` or *`cluster`*`.getClusterSet()`. Again, it is important to do this when you are connected to the server instance using the InnoDB Cluster server configuration account. The default user account stored in the object is used for some operations during the InnoDB ClusterSet deployment process, regardless of the account that you specify on the connection.

7. Issue a *`clusterSet`*`.createReplicaCluster()` command using the `ClusterSet` object to create the replica cluster, naming one of the standalone server instances. This server instance will be the replica cluster's primary. The command returns a `Cluster` object for the replica cluster, and you can assign this to a variable if you want. For example:

```
mysql-js> cluster2 = myclusterset.createReplicaCluster("127.0.0.1:4410", "clustertwo", {recoveryProgres
Setting up replica 'clustertwo' of cluster 'clusterone' at instance '127.0.0.1:4410'.

A new InnoDB cluster will be created on instance '127.0.0.1:4410'.

Validating instance configuration at 127.0.0.1:4410...
NOTE: Instance detected as a sandbox.
Please note that sandbox instances are only suitable for deploying test clusters for use within
the same host.

This instance reports its own address as 127.0.0.1:4410

Instance configuration is suitable.
NOTE: Group Replication will communicate with other members using '127.0.0.1:44101'. Use the
localAddress option to override.


* Checking transaction state of the instance...

NOTE: The target instance '127.0.0.1:4410' has not been pre-provisioned (GTID set is empty). The
Shell is unable to decide whether replication can completely recover its state.
The safest and most convenient way to provision a new instance is through automatic clone
provisioning, which will completely overwrite the state of '127.0.0.1:4410' with a physical
snapshot from an existing clusterset member. To use this method by default, set the
'recoveryMethod' option to 'clone'.

WARNING: It should be safe to rely on replication to incrementally recover the state of the new
Replica Cluster if you are sure all updates ever executed in the ClusterSet were done with GTIDs
enabled, there are no purged transactions and the instance used to create the new Replica Cluster
contains the same GTID set as the ClusterSet or a subset of it. To use this method by default,
set the 'recoveryMethod' option to 'incremental'.


Please select a recovery method [C]lone/[I]ncremental recovery/[A]bort (default Clone):
Waiting for clone process of the new member to complete. Press ^C to abort the operation.
* Waiting for clone to finish...
```

```
NOTE: 127.0.0.1:4410 is being cloned from 127.0.0.1:3310
** Stage DROP DATA: Completed

NOTE: 127.0.0.1:4410 is shutting down...

* Waiting for server restart... ready
* 127.0.0.1:4410 has restarted, waiting for clone to finish...
** Stage FILE COPY: Completed
** Stage PAGE COPY: Completed
** Stage REDO COPY: Completed
** Stage FILE SYNC: Completed
** Stage RESTART: Completed
* Clone process has finished: 72.61 MB transferred in about 1 second (~72.61 MB/s)

Creating InnoDB cluster 'clustertwo' on '127.0.0.1:4410'...

Adding Seed Instance...
Cluster successfully created. Use Cluster.addInstance() to add MySQL instances.
At least 3 instances are needed for the cluster to be able to withstand up to
one server failure.

* Configuring ClusterSet managed replication channel...
** Changing replication source of 127.0.0.1:4410 to 127.0.0.1:3310

* Waiting for instance to synchronize with PRIMARY Cluster...
** Transactions replicated  ############################################################  100%
* Updating topology

Replica Cluster 'clustertwo' successfully created on ClusterSet 'testclusterset'.

<Cluster:clustertwo>
```

For the *`clusterSet`*`.createReplicaCluster()` command:

- The *`instance`* parameter is required and specifies the host and port number of the standalone server's MySQL Server instance. This is the server instance that is going to be the primary of the replica cluster. In the example command above, this is *`127.0.0.1:4410`*.
- The *`clusterName`* parameter is required and specifies an identifier for the replica cluster. In the example command above, *`clustertwo`* is used. The name must be unique in the InnoDB ClusterSet, and it must follow the InnoDB Cluster naming requirements. Only alphanumeric characters, hyphens (-), underscores (_), and periods (.) can be used, and the name must not start with a number. The maximum length is 63 characters. The cluster name is case sensitive.
- Use the `dryRun` option if you want to carry out validations and log the changes without actually executing them.
- Use the `interactive` option to enable or disable confirmation prompts for the provisioning method. The default is the value of MySQL Shell's `useWizards` option.
- Use the `recoveryMethod` option if you want to select a provisioning method. If you do not specify this as an option, the default setting `AUTO` is used. In that case, the function compares the GTID set on the server instance to the GTID set on the primary cluster, and attempts to determine the most appropriate provisioning method. If this cannot be determined, the function prompts you to select a provisioning method, or cancels the operation if you are not in interactive mode.

  The provisioning process, which is called distributed recovery, can use cloning, where the state of the server instance is completely overwritten by a physical snapshot taken from an existing member server in the cluster. To select this in advance, specify the `CLONE` setting. The alternative is incremental state transfer from an existing member server's binary log, in this case a member of the primary cluster. Here, the server instance receives and applies transactions from the primary cluster that it does not already have. To select this in advance, specify the `INCREMENTAL` setting.

- Use the `cloneDonor` option if you want to select a specific server to provide the snapshot that overwrites the current server, if distributed recovery is carried out by cloning. The operation chooses a secondary member of the primary cluster by default, or the primary if no secondary is available. The selected server instance must be a member of the primary cluster in the InnoDB ClusterSet. Specify a host and port number. IPv6 addresses are not supported for this option.

- Use the `recoveryProgress` option to specify the verbosity level (0, 1, or 2) for the distributed recovery process. Setting 0 shows no progress information, 1 shows detailed static progress information, and 2 shows detailed dynamic progress information using progress bars. 2 is the default if standard output is a terminal, otherwise 1 is the default.

- Use the `timeout` option if you want to set a timeout to wait for the server instance to synchronize with the primary cluster after it has been provisioned and the ClusterSet replication channel has been established. By default there is no timeout.

- Use the `manualStartOnBoot` option to specify whether Group Replication starts automatically and rejoins the cluster when the MySQL server starts up, or whether it must be started manually. The default, `false`, means Group Replication starts automatically.

- Use the `communicationStack` option to define how the members communicate with one another, using `XCOM` or `MYSQL` protocols. See Section 7.5.9, “Configuring the Group Replication Communication Stack”.

  If you are using MySQL 8.0.27, or higher, the default, and recommended, protocol is `MYSQL`.

- The options `memberSslMode`, `ipAllowlist`, `localAddress`, `exitStateAction`, `memberWeight`, `consistency`, `expelTimeout`, and `autoRejoinTries` are available if you want to configure the setup of Group Replication for the replica InnoDB Cluster. These options work in the same way as they do for an InnoDB Cluster that is not part of a ClusterSet. For details of the options, see Section 7.5, “Configuring InnoDB Cluster”. (**Note**: `ipAllowlist` and `localAddress` are available only for the `XCOM` communication stack.)

- It is possible to use the options `localAddress` and `groupName` to set a Group Replication local address and group identifier. However, this is not recommended, as incorrect values can cause errors in Group Replication. Only use these options if you already experienced an issue with the values selected by the InnoDB ClusterSet setup process for these items.

- When creating an InnoDB ClusterSet using MySQL Shell 8.0.28 and later, if you have security requirements that all accounts created automatically by AdminAPI have strict authentication requirements, you can set a value for the `replicationAllowedHost` configuration option of the ClusterSet. The `replicationAllowedHost` MySQL Shell option allows you to set internally managed replication accounts for a ClusterSet to a strict subnet based filter instead of the default wildcard value of `%`.The `replicationAllowedHost` option takes a string value. For example, to create a clusterset called `my_clusterset_domain` and set the `replicationAllowedHost` option to `192.0.2.0/24`, issue:

  ```
  mysql-js> <Cluster>.createClusterSet('my_clusterset_domain', {replicationAllowedHost:'192.0.2.0/24'})
  ```

  If you change `replicationAllowedHost` on a ClusterSet, the account used for the replication channel between clusters is changed to allow connections only from the value you specify for `replicationAllowedHost`. The host must be accessible in both the primary and replica clusters. If not, there is no replication between clusters.

  A ClusterSet can be modified after creation to set a `replicationAllowedHost`, by issuing:

  ```
  mysql-js> <Clusterset>.setOption('replicationAllowedHost','192.0.2.0/24')
  ```

When you issue the *`clusterSet`*`.createReplicaCluster()` command, MySQL Shell checks that the target server instance complies with the requirements to become the primary server in a

replica InnoDB Cluster in an InnoDB ClusterSet deployment, and returns an error if it does not. If the instance meets the requirements, MySQL Shell carries out the following setup tasks:

- Creates the ClusterSet replication channel `clusterset_replication`, and creates a replication user with a random password. This is an asynchronous replication channel between the target instance and the primary server of the primary cluster, which is managed by InnoDB ClusterSet. Encryption is configured for the channel according to the `clusterSetReplicationSslMode` option for the InnoDB ClusterSet. MySQL Shell verifies that the replication setup is working, and returns an error if it is not.
- Provisions the MySQL Server instance with the dataset from the primary InnoDB Cluster and synchronizes the GTID set, using the selected recovery method. Note that if there is a large amount of data in the ClusterSet's member servers, distributed recovery could take several hours.
- Adds the InnoDB Cluster administrator accounts and the MySQL Router administrator accounts on the server instance. If the instance is provisioned by state transfer from the binary log, the provisioning process includes the transactions that create the accounts, or else the accounts are transferred during cloning. Either way, these accounts become available on the server instance. See Section 8.3, “User Accounts for InnoDB ClusterSet” for more information.
- Configures and starts Group Replication for the replica cluster. The InnoDB ClusterSet replica cluster creation process overwrites any existing persisted Group Replication configuration options for which you specify new settings on the *`clusterSet`*`.createReplicaCluster()` command. It also always overwrites the following configuration options, even if you do not specify them on the command: `group_replication_group_name`, `group_replication_group_seeds`, `group_replication_local_address`, `group_replication_view_change_uuid`, and `group_replication_enforce_update_everywhere_checks`. However, any other Group Replication configuration options that you changed on the server instance prior to using it in the replica cluster are left as they were. See the important note about this in Section 8.1, “InnoDB ClusterSet Requirements”.
- Sets the `skip_replica_start` system variable to `ON` so that replication threads are not automatically started on the server, and sets the `super_read_only` system variable so that clients cannot write transactions to the server.
- Disables the Group Replication member action `mysql_disable_super_read_only_if_primary` so that `super_read_only` remains set on the primary of the cluster after a view change.
- Enables the Group Replication member action `mysql_start_failover_channels_if_primary` so that asynchronous connection failover for replicas is enabled for the ClusterSet replication channel. With this function enabled, if the primary that is replicating goes offline or into an error state, the new primary starts replication on the same channel when it is elected.
- Transfers the ClusterSet metadata to the server instance, creates the replica cluster in the InnoDB ClusterSet, and adds the target server instance to it as the primary.
- Returns the `Cluster` object for the replica cluster.

8. Using the `Cluster` object that was returned for the replica cluster by *`clusterSet`*`.createReplicaCluster()`, issue a *`cluster`*`.addInstance` command naming another of the standalone server instances. This server instance will be a secondary in the replica cluster. For example:

```
mysql-js> cluster2.addInstance('icadmin@127.0.0.1:4420')

NOTE: The target instance '127.0.0.1:4420' has not been pre-provisioned (GTID set is empty). The
Shell is unable to decide whether clone based recovery is safe to use.
```

```
The safest and most convenient way to provision a new instance is through automatic clone
provisioning, which will completely overwrite the state of '127.0.0.1:4420' with a physical
snapshot from an existing cluster member. To use this method by default, set the
'recoveryMethod' option to 'clone'.

Please select a recovery method [C]lone/[A]bort (default Clone): c
Validating instance configuration at localhost:4420...
NOTE: Instance detected as a sandbox.
Please note that sandbox instances are only suitable for deploying test clusters for use within
the same host.

This instance reports its own address as 127.0.0.1:4420

Instance configuration is suitable.
NOTE: Group Replication will communicate with other members using '127.0.0.1:44201'. Use the
localAddress option to override.

A new instance will be added to the InnoDB cluster. Depending on the amount of
data on the cluster this might take from a few seconds to several hours.

Adding instance to the cluster...

* Waiting for the Cluster to synchronize with the PRIMARY Cluster...
** Transactions replicated  ############################################################  100%
* Configuring ClusterSet managed replication channel...
** Changing replication source of 127.0.0.1:4420 to 127.0.0.1:3310

Monitoring recovery process of the new cluster member. Press ^C to stop monitoring and
let it continue in background.
Clone based state recovery is now in progress.

NOTE: A server restart is expected to happen as part of the clone process. If the
server does not support the RESTART command or does not come back after a
while, you may need to manually start it back.

* Waiting for clone to finish...
NOTE: 127.0.0.1:4420 is being cloned from 127.0.0.1:4410
** Stage DROP DATA: Completed
** Clone Transfer
    FILE COPY  ############################################################  100%  Completed
    PAGE COPY  ############################################################  100%  Completed
    REDO COPY  ############################################################  100%  Completed

NOTE: 127.0.0.1:4420 is shutting down...

* Waiting for server restart... ready
* 127.0.0.1:4420 has restarted, waiting for clone to finish...
** Stage RESTART: Completed
* Clone process has finished: 72.61 MB transferred in about 1 second (~72.61 MB/s)

State recovery already finished for '127.0.0.1:4420'

The instance '127.0.0.1:4420' was successfully added to the cluster.
```

For more details on the *`cluster`*`.addInstance` command, see Section 7.4.4, “Adding Instances to an InnoDB Cluster”.

If you need to get the `Cluster` object for the replica cluster again, connect to any active instance in the replica cluster using the InnoDB Cluster server configuration account and issue `dba.getCluster()`. This account is used for some of the operations in the setup process. If the setup process finds that the account is not present on the standalone server instance, an error is returned, and you will need to issue `dba.configureInstance()` to create the account.

When the command is successful, the server instance is added to the replica cluster and provisioned with the data for the InnoDB ClusterSet. The donor for a cloning operation will be from the replica cluster, not the primary cluster.

9. Repeat the *`cluster`*`.addInstance` operation to add all of the standalone server instances to the replica cluster. A minimum of three instances is recommended for tolerance to failures. You can

have up to nine member servers in a replica cluster, which is a limit built into the underlying Group Replication technology.

10. Verify that the completed replica cluster and the InnoDB ClusterSet deployment are healthy. You can do this using a *`cluster`*`.status()` command to view the replica cluster, and a *`clusterSet`*`.status()` command to view the InnoDB ClusterSet deployment. Alternatively, you can select the extended output for *`clusterSet`*`.status()` to see the detailed status for all the clusters. For example:

```
mysql-js> myclusterset.status({extended: 1})
{
    "clusters": {
        "clusterone": {
            "clusterRole": "PRIMARY",
            "globalStatus": "OK",
            "primary": "127.0.0.1:3310",
            "status": "OK",
            "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
            "topology": {
                "127.0.0.1:3310": {
                    "address": "127.0.0.1:3310",
                    "memberRole": "PRIMARY",
                    "mode": "R/W",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:3320": {
                    "address": "127.0.0.1:3320",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:3330": {
                    "address": "127.0.0.1:3330",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                }
            },
            "transactionSet": "953a51d5-2690-11ec-ba07-00059a3c7a00:1,c51c1b15-269e-11ec-b9ba-00059a
        },
        "clustertwo": {
            "clusterRole": "REPLICA",
            "clusterSetReplication": {
                "applierStatus": "APPLIED_ALL",
                "applierThreadState": "Waiting for an event from Coordinator",
                "applierWorkerThreads": 4,
                "receiver": "127.0.0.1:4410",
                "receiverStatus": "ON",
                "receiverThreadState": "Waiting for source to send event",
                "source": "127.0.0.1:3310"
            },
            "clusterSetReplicationStatus": "OK",
            "globalStatus": "OK",
            "status": "OK",
            "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
            "topology": {
                "127.0.0.1:4410": {
                    "address": "127.0.0.1:4410",
                    "memberRole": "PRIMARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
```

```
                },
                "127.0.0.1:4420": {
                    "address": "127.0.0.1:4420",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:4430": {
                    "address": "127.0.0.1:4430",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                }
            },
            "transactionSet": "0f6ff279-2764-11ec-ba06-00059a3c7a00:1-5,953a51d5-2690-11ec-ba07-00059a3
            "transactionSetConsistencyStatus": "OK",
            "transactionSetErrantGtidSet": "",
            "transactionSetMissingGtidSet": ""
        }
    },
    "domainName": "testclusterset",
    "globalPrimaryInstance": "127.0.0.1:3310",
    "metadataServer": "127.0.0.1:3310",
    "primaryCluster": "clusterone",
    "status": "HEALTHY",
    "statusText": "All Clusters available."
}
```


See Section 8.6, “InnoDB ClusterSet Status and Topology” for more information on the output of the *clusterSet*.status() command.

11. Add further replica clusters as required, by repeating the above steps with a different set of standalone instances. There is no defined limit on the number of replica clusters that you can have in an InnoDB ClusterSet deployment. The process is the same in each case, as summarized here:

    - Create the InnoDB Cluster server configuration account on each of the standalone server instances by issuing a dba.configureInstance() command with the clusterAdmin option.
    - Fetch the ClusterSet object using dba.getClusterSet() or *cluster*.getClusterSet(), when you are connected to a member of the InnoDB ClusterSet using the InnoDB Cluster server configuration account. You can get the object from any member server in the primary cluster or in one of the replica clusters that you created already.
    - Issue a *clusterSet*.createReplicaCluster() command using the ClusterSet object to create the replica cluster, naming one of the standalone server instances.
    - Using the Cluster object that was returned for the replica cluster by *clusterSet*.createReplicaCluster(), issue a *cluster*.addInstance command naming another of the standalone server instances.
    - Repeat the *cluster*.addInstance operation to add all of the standalone server instances to the replica cluster.
    - Verify that the completed replica cluster and the InnoDB ClusterSet deployment are healthy, for example by using a *clusterSet*.status() command with extended output.

12. Bootstrap MySQL Router instances against the InnoDB ClusterSet to manage application traffic, and configure them as appropriate. By default, MySQL Router directs all read and write requests to whichever cluster is currently the primary cluster in an InnoDB ClusterSet deployment, but you can

configure a MySQL Router instance to route traffic only to a specific cluster. For instructions, see Section 8.5, “Integrating MySQL Router With InnoDB ClusterSet”.

8.5 Integrating MySQL Router With InnoDB ClusterSet

MySQL Router routes client application traffic to the appropriate clusters in an InnoDB ClusterSet deployment. You can set a global policy for MySQL Router instances that are used with the InnoDB ClusterSet deployment, and override this with settings for individual MySQL Router instances.

When you bootstrap a MySQL Router instance against an InnoDB ClusterSet deployment, it is aware of the complete topology of the ClusterSet, and can manage write and read traffic appropriately. If a controlled switchover or emergency failover takes place, the MySQL Router instances connected with the InnoDB ClusterSet are aware of this and route traffic to the new primary cluster, except for any instances that you have configured to send traffic to a specific cluster. If a cluster is invalidated, MySQL Router instances stop read and write traffic to it, except for any instances that you have configured to continue sending read traffic in that situation.

For each MySQL Router instance that you are using with InnoDB ClusterSet, you can choose to configure it to follow the primary cluster, or to connect only to a specific target InnoDB Cluster. You can change between these modes online using MySQL Shell.

Follow the primary	In this mode, MySQL Router directs application traffic, both writes and reads, to the cluster in the InnoDB ClusterSet deployment that is currently the primary cluster. This mode is the default.
Named target cluster	In this mode, MySQL Router directs application traffic to the InnoDB Cluster that you specify. This can be the primary cluster in the InnoDB ClusterSet deployment, or it can be a replica cluster. If the target cluster is currently the primary cluster, MySQL Router opens the write port and applications can write to the instance. If the target cluster is currently a read-only replica cluster, MySQL Router allows only read traffic, and denies write traffic. If this situation changes due to a switchover or failover to or from the target cluster, MySQL Router changes the permitted request types accordingly. This mode is useful if an application makes only read requests, which can be made on a replica cluster, and you want to keep that traffic routed to a local cluster.

You can also configure MySQL Router to allow or disallow read traffic to a cluster that has been marked as `INVALIDATED`. A cluster in this state is not currently functioning at all as part of the InnoDB ClusterSet deployment, and cannot receive writes. Although the cluster does not necessarily have any technical issues, its data is becoming stale. The default is that MySQL Router disallows reads as well as writes to an invalidated cluster (the `drop_all` setting), but you can choose to allow reads (the `accept_ro` setting).

To bootstrap MySQL Router against InnoDB ClusterSet, you need to use an InnoDB Cluster administrator account, or the InnoDB Cluster server configuration account, which also has the required permissions. MySQL Router then uses the MySQL Router administrator account to connect to the instances in the InnoDB ClusterSet deployment. You need to specify the user name and password for both these accounts during the bootstrap operation. See Section 8.3, “User Accounts for InnoDB ClusterSet” for more information.

Important

If you are using an existing InnoDB Cluster as the primary cluster in your InnoDB ClusterSet deployment, and you bootstrapped MySQL Router against that cluster already, follow the relevant parts of this process to bootstrap it again using the `--force` option against the InnoDB ClusterSet, then stop and restart MySQL Router. The settings in the MySQL Router instance's static configuration file need to be updated for InnoDB ClusterSet.

To integrate MySQL Router with an InnoDB ClusterSet deployment, follow this process:

1. If you haven't already done so, install MySQL Router instances as appropriate for your topology. The recommended deployment of MySQL Router is on the same host as the client application. When using a sandbox deployment, everything is running on a single host, therefore you deploy MySQL Router to the same host. When using a production deployment, we recommend deploying one MySQL Router instance to each machine used to host one of your client applications. It is also possible to deploy MySQL Router to a common machine through which your application instances connect. For instructions, see Installing MySQL Router.

2. Connect to any active member server instance in the InnoDB ClusterSet deployment, using an InnoDB Cluster administrator account. You may also use the InnoDB Cluster server configuration account, which also has the required permissions. Get the `ClusterSet` object using a `dba.getClusterSet()` or *`cluster`*`.getClusterSet()` command. It is important to get the `ClusterSet` object when you are connected to the server instance using an appropriate account. The default user account stored in the object is used for some operations, regardless of the account that you specify on the connection. For example:

   ```
   mysql-js> \connect admin2@127.0.0.1:3310
   ...
   mysql-js> myclusterset = dba.getClusterSet()
   <ClusterSet:testclusterset>
   ```

   In this example:

   - *`admin2@127.0.0.1:3310`* is the URI-like connection string for any member server instance that is online in the cluster.

     The URI-like connection string is comprised of the following elements:

   - *`admin2`* is the user name for the InnoDB Cluster administrator account.
   - *`127.0.0.1:3310`* is the host and port for the member server instance, as displayed by the *`cluster`*`.status()` command.
   - The returned `ClusterSet` object is assigned to the variable `myclusterset`.

3. Verify that the InnoDB ClusterSet deployment is healthy, by issuing *`clusterSet`*`.status()` in MySQL Shell while connected to any member server in the cluster. For example:

   ```
   mysql-js> myclusterset.status({extended: 1})
   ```

   Select the extended output to see the detailed status for the clusters in the InnoDB ClusterSet topology. This gives you the host and port for each member server, so you can choose one to bootstrap MySQL Router against. See Section 8.6, “InnoDB ClusterSet Status and Topology” for more information.

4. For each MySQL Router instance, run the `mysqlrouter` command in a suitable shell on the instance where MySQL Router is installed, to bootstrap MySQL Router against InnoDB ClusterSet. In this example, the `force` option is used because MySQL Router has previously been bootstrapped against the primary InnoDB Cluster:

   ```
   $> mysqlrouter --bootstrap icadmin@127.0.0.1:3310 --account=myRouter1 --name='Rome1' --force
   Please enter MySQL password for icadmin:
   # Bootstrapping system MySQL Router instance...

   Please enter MySQL password for myRouter1:
   - Creating account(s) (only those that are needed, if any)
   - Verifying account (using it to run SQL queries that would be run by Router)
   - Storing account in keyring
   - Creating configuration C:/Program Files/MySQL/MySQL Router 8.0/mysqlrouter.conf

   # MySQL Router configured for the ClusterSet 'testclusterset'
   ```

```
After this MySQL Router has been started with the generated configuration

    > net start mysqlrouter
or
    > C:\Program Files\MySQL\MySQL Router 8.0\bin\mysqlrouter.exe -c C:/Program Files/MySQL/MySQL Ro

ClusterSet 'testclusterset' can be reached by connecting to:

## MySQL Classic protocol

- Read/Write Connections: localhost:6446
- Read/Only Connections:  localhost:6447

## MySQL X protocol

- Read/Write Connections: localhost:6448
- Read/Only Connections:  localhost:6449
```

In this example:

- *icadmin@127.0.0.1:3310* is the URI-like connection string for any member server instance that is online in the InnoDB ClusterSet deployment. The instance can be in the primary cluster or in a replica cluster. If the instance is not the primary server in the primary cluster, InnoDB ClusterSet will route the transaction to that server, provided that the InnoDB ClusterSet deployment is healthy.

  The URI-like connection string is comprised of the following elements:

- *icadmin* is the user name for an InnoDB Cluster administrator account that was set up using the *cluster*`.setupAdminAccount()` command on the primary cluster, then replicated to the replica clusters. The bootstrap operation prompts you for the password for the account. The password for an InnoDB Cluster administrator account is the same on all the server instances in the InnoDB ClusterSet deployment.

- *127.0.0.1:3310* is the host and port for the member server instance, as displayed by the *clusterSet*`.status()` command.

- *myRouter1* is the user name for a MySQL Router administrator account that was set up using the *cluster*`.setupRouterAccount()` command on the primary cluster. The account is the same on all the server instances in the InnoDB ClusterSet deployment. The bootstrap operation prompts you for the password for the account.

- `--name` can be used to assign a non-default name to the MySQL Router instance, to make it easily identifiable in the output from InnoDB ClusterSet status commands.

- `--force` is required if you are bootstrapping MySQL Router again for an existing InnoDB Cluster where it was previously bootstrapped.

MySQL Router connects to the server instance and retrieves the InnoDB ClusterSet metadata. The process is the same as when you bootstrap MySQL Router against an individual InnoDB Cluster. For more details about the process, see Section 6.10.3, “Deploying MySQL Router”.

5. After you bootstrap each MySQL Router instance, verify that it is now correctly bootstrapped against the InnoDB ClusterSet deployment, by issuing *clusterSet*`.listRouters()` in MySQL Shell while connected to any member server in the InnoDB ClusterSet. The command returns details of all the registered MySQL Router instances, or a router instance that you specify. For example:

```
mysql-js> myclusterset.listRouters()
{
    "domainName": "testclusterset",
    "routers": {
       "mymachine::Rome1": {
```

```
            "hostname": "mymachine",
            "lastCheckIn": 2021-10-15 11:58:37,
            "roPort": 6447,
            "roXPort": 6449,
            "rwPort": 6446,
            "rwXPort": 6448,
            "targetCluster": "primary",
            "version": "8.0.27"
        },
        "mymachine2::Rome2": {
            "hostname": "mymachine2",
            "lastCheckIn": 2021-10-15 11:58:37,
            "roPort": 6447,
            "roXPort": 6449,
            "rwPort": 6446,
            "rwXPort": 6448,
            "targetCluster": "primary",
            "version": "8.0.27"
        }
    }
}
```

See MySQL Router Status for InnoDB ClusterSet for more information.

6. To see the routing options that are set for each MySQL Router instance, and the global policy for the InnoDB ClusterSet deployment, issue *`clusterSet`*`.routingOptions()` in MySQL Shell while connected to any member server in the InnoDB ClusterSet deployment. For example:

```
mysql-js> myclusterset.routingOptions()
{
    "domainName": "testclusterset",
    "global": {
        "invalidated_cluster_policy": "drop_all",
        "target_cluster": "primary"
    },
    "routers": {
        "mymachine::Rome1":  {
            "target_cluster": "primary"
            "invalidated_cluster_policy": "accept_ro"
        },
        "mymachine2::Rome2": {}
    }
}
```

By default, a MySQL Router instance sends traffic to the primary cluster, and disallows both read and write traffic to a cluster that is marked as `INVALIDATED`. See MySQL Router Status for InnoDB ClusterSet for more information and an explanation of the output of the *`clusterSet`*`.routingOptions()` command.

7. If you want to change the global routing policy or the routing policy for an individual MySQL Router instance, issue *`clusterSet`*`.setRoutingOption()` in MySQL Shell while connected to any member server in the InnoDB ClusterSet deployment. You can only set one routing option at a time. It takes a few seconds for a MySQL Router instance to pick up changes to a routing policy.

   For example, this command issued for the InnoDB ClusterSet `myclusterset` changes the target cluster for a MySQL Router instance to the cluster `clustertwo`:

```
mysql-js> myclusterset.setRoutingOption('mymachine::Rome1', 'target_cluster', 'clustertwo')
Routing option 'target_cluster' successfully updated in router 'Rome1'.
```

   In this example, *`myclusterset`* is the variable for the `ClusterSet` object, *`Rome1`* is the name of the MySQL Router instance, and *`clustertwo`* is the name of the specific cluster to target.

   To set the routing policy for the instance back to following the primary, issue this command:

```
mysql-js> myclusterset.setRoutingOption('mymachine::Rome1', 'target_cluster', 'primary')
```

```
Routing option 'target_cluster' successfully updated in router 'Rome1'.
```

To clear a routing policy for an instance, use the *clusterSet*.setRoutingOption() command to set the relevant policy to null. For example:

```
mysql-js> myclusterset.setRoutingOption('mymachine::Rome1', 'target_cluster', null)
Routing option 'target_cluster' successfully updated in router 'Rome1'.
```

See setRoutingOption() for more information on the available routing options.

To set the global routing policy, do not specify a MySQL Router instance, just the policy name and the setting. See MySQL Router Status for InnoDB ClusterSet for more information and an explanation of the available routing options.

8. When you are ready to start accepting connections, configure the applications to use the ports where MySQL Router is listening for traffic to the InnoDB ClusterSet deployment. Then start the MySQL Router instances using a suitable shell or script in the servers where MySQL Router is installed. See Starting MySQL Router.

setRoutingOption()

The routing options are as follows:

"target_cluster": "primary"	With this setting, MySQL Router directs traffic from client applications to the cluster in the InnoDB ClusterSet deployment that is currently the primary cluster. A primary cluster can accept both read and write traffic. Follow the primary mode is the default for the global policy and for individual MySQL Router instances.
"target_cluster": "*clusterName*"	With this setting, MySQL Router directs traffic from applications to the specified cluster in the InnoDB ClusterSet deployment, whether it is currently in the role of the primary cluster or a replica cluster. If the target cluster is currently the primary cluster, MySQL Router opens the write port and applications can write to the instance. If the target cluster is currently a read-only replica cluster, MySQL Router allows only read traffic, and denies write traffic. If this situation changes due to a switchover or failover to or from the target cluster, MySQL Router changes the permitted request types accordingly. This mode is useful if an application makes only read requests, which can be made on a replica cluster, and you want to keep that traffic routed to a local cluster. Note that the cluster name is case sensitive.
"invalidated_cluster_policy": "drop_all"	With this setting, when a cluster is marked as INVALIDATED, MySQL Router disallows both read and write traffic to it from applications. A cluster in this state is not currently functioning at all as part of the InnoDB ClusterSet deployment, and cannot receive writes. It might be a former primary cluster that was marked as invalidated during an emergency failover process, or a replica cluster that was marked as invalidated because it was unreachable or unavailable at the time of a failover or during a controlled switchover. This setting is the default for the global policy and for individual MySQL Router instances.
"invalidated_cluster_policy": "accept_ro"	With this setting, when a cluster is marked as INVALIDATED, MySQL Router allows read traffic to it from applications but drops write traffic. Although an invalidated cluster does not necessarily have any technical issues, the data is becoming stale, so this setting means that stale reads will take place unless the issue is resolved soon. However, this setting can provide higher availability in cases where stale reads are not a high priority.

`"stats_updates_frequency": "numberOfSeconds"`: This option defines, in seconds, the interval between MySQL Router check-in updates.

If set to 0 (default), no periodic updates are done. MySQL Router rounds up the value to a multiple of its TTL. For example:

- If lower than TTL it gets rounded up to TTL. For example: if `TTL=30` and `stats_updates_frequency=1`, the effective frequency is 30 seconds.
- If not a multiple of TTL, it is rounded up and adjusted according to the TTL. For example, if `TTL=5` and `stats_updates_frequency=11`, the effective frequency is 15 seconds, or if `TTL=5` and `stats_updates_frequency=13`, the effective frequency is 15 seconds.

If the value is null, the option value is cleared and the default value takes effect.

`use_replica_primary_as_rw: [true | false]`: This option instructs MySQL Router enables it to open or close a read-write (R/W) port on a router targeting a specific Cluster (where `target_cluster` is not set to `primary`), enabling you to use a R/W port on a ReplicaCluster. The ReplicaCluster continues to only accept R/O traffic. In the event of a switchover or failover, the R/W port remains unchanged.

If set to `true`, MySQL Router R/W port in ReplicaClusters.

If set to false (default), the router's behaviour is unchanged and the R/W port is closed in ReplicaClusters.

You can change the routing options for MySQL Router instances in an InnoDB ClusterSet deployment using the *`clusterSet`*`.setRoutingOption()` command. For instructions to do this, see Section 8.5, “Integrating MySQL Router With InnoDB ClusterSet”.

To clear a routing option, set it to `null`.

8.6 InnoDB ClusterSet Status and Topology

This section describes the following:

- InnoDB ClusterSet Status
- InnoDB ClusterSet Topology
- MySQL Router Status for InnoDB ClusterSet

InnoDB ClusterSet Status

AdminAPI's *`clusterSet`*`.status()` command returns a JSON object describing the status of an InnoDB ClusterSet deployment. The output includes the status of the InnoDB ClusterSet deployment itself and the global and cluster status of each InnoDB Cluster in the ClusterSet. The extended output adds the status of each member server in each cluster, information about the asynchronous replication channels managed by InnoDB ClusterSet, and other configuration and status information. The command reports the status of ClusterSet replication as well as of the servers themselves. If there are any issues, warning and error messages are included to explain the problem in more detail.

The MySQL Shell instance where you use *`clusterSet`*`.status()` can be connected to any active member of the InnoDB ClusterSet. The metadata can be retrieved from the primary cluster by way of any other cluster that is active in the InnoDB ClusterSet.

If there is an issue with any of the clusters in the InnoDB ClusterSet, Section 8.9, “InnoDB ClusterSet Repair and Rejoin” explains the procedure for fixing it and rejoining the cluster to the ClusterSet (or removing it if the issue cannot be fixed). If the cluster with the issue is the primary cluster, you first need to carry out a controlled switchover if it is still functioning (as described in Section 8.7, “InnoDB ClusterSet Controlled Switchover”), or an emergency failover if it is not functioning or cannot be contacted (as described in Section 8.8, “InnoDB ClusterSet Emergency Failover”).

You can use the `extended` option, which defaults to 0, to increase the verbosity level of the output as follows:

- `extended: 0` or omitting the option returns basic information about the availability status of the InnoDB ClusterSet deployment, each InnoDB Cluster in the ClusterSet, and the ClusterSet replication status for each replica cluster.
- `extended: 1` adds the topology for each InnoDB Cluster in the ClusterSet, the status of each individual member server in each cluster, and more detailed information about the ClusterSet replication channel's status for each replica cluster.
- `extended: 2` adds further details about each individual member server in each cluster and about the ClusterSet replication channel, including the GTID set.
- `extended: 3` adds important configuration settings for the ClusterSet replication channel, such as the connection retry settings.

For example:

```
mysql-js> myclusterset.status({extended: 1})
{
    "clusters": {
        "clusterone": {
            "clusterRole": "PRIMARY",
            "globalStatus": "OK",
            "primary": "127.0.0.1:3310",
            "status": "OK",
            "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
            "topology": {
                "127.0.0.1:3310": {
                    "address": "127.0.0.1:3310",
                    "memberRole": "PRIMARY",
                    "mode": "R/W",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:3320": {
                    "address": "127.0.0.1:3320",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:3330": {
                    "address": "127.0.0.1:3330",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                }
            },
            "transactionSet": "953a51d5-2690-11ec-ba07-00059a3c7a00:1,c51c1b15-269e-11ec-b9ba-00059a3c7
        },
        "clustertwo": {
            "clusterRole": "REPLICA",
            "clusterSetReplication": {
                "applierStatus": "APPLIED_ALL",
                "applierThreadState": "Waiting for an event from Coordinator",
```

```
                "applierWorkerThreads": 4,
                "receiver": "127.0.0.1:4410",
                "receiverStatus": "ON",
                "receiverThreadState": "Waiting for source to send event",
                "source": "127.0.0.1:3310"
            },
            "clusterSetReplicationStatus": "OK",
            "globalStatus": "OK",
            "status": "OK",
            "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
            "topology": {
                "127.0.0.1:4410": {
                    "address": "127.0.0.1:4410",
                    "memberRole": "PRIMARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:4420": {
                    "address": "127.0.0.1:4420",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:4430": {
                    "address": "127.0.0.1:4430",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                }
            },
            "transactionSet": "0f6ff279-2764-11ec-ba06-00059a3c7a00:1-5,953a51d5-2690-11ec-ba07-00059a3c7a0
            "transactionSetConsistencyStatus": "OK",
            "transactionSetErrantGtidSet": "",
            "transactionSetMissingGtidSet": ""
        }
    },
    "domainName": "testclusterset",
    "globalPrimaryInstance": "127.0.0.1:3310",
    "metadataServer": "127.0.0.1:3310",
    "primaryCluster": "clusterone",
    "status": "HEALTHY",
    "statusText": "All Clusters available."
}
```


To get a handle to a `ClusterSet` object representing the InnoDB ClusterSet for a target server instance, use a `dba.getClusterSet()` or *`cluster`*`.getClusterSet()` command. These commands work if the target server instance is a member of an InnoDB Cluster that is part of an InnoDB ClusterSet deployment, even if the primary cluster for the InnoDB ClusterSet deployment is not currently reachable. The target server instance itself must be reachable when you use the object. If the target instance is a member of a cluster that has been marked as invalidated, the command returns a warning, but still returns the `ClusterSet` object. If the target instance is not currently a member of an InnoDB ClusterSet deployment, the command returns an error. The `ClusterSet` object contains the connection details of the server that you retrieved it from, so a `ClusterSet` object that you previously retrieved from a member server that is now offline will not work any more, and you would need to get it again from a server that is online in the InnoDB ClusterSet deployment.

The `ClusterSet` object defaults to using the account it was fetched with for operations where permissions are required. It is important to get the object when you are connected to the server instance using an appropriate user account for the operations you want to perform using it. Some operations during the InnoDB ClusterSet deployment process require permissions, and the default

user account stored in the object is used for this, so that the process does not need to store any other user accounts. For monitoring and troubleshooting an InnoDB ClusterSet that you already set up, an InnoDB Cluster administrator account is appropriate. For the initial cluster deployment process, the InnoDB Cluster server configuration account is appropriate. For more information, see Section 8.3, “User Accounts for InnoDB ClusterSet”.

When you use the `clusterSet.status()` function, the overall ClusterSet status (`status` field) reported for an InnoDB ClusterSet deployment can be one of the following:

`HEALTHY`	The primary cluster in the InnoDB ClusterSet is functioning acceptably, and all of the replica clusters are functioning acceptably.
`AVAILABLE`	The primary cluster in the InnoDB ClusterSet is functioning acceptably, but one or more of the replica clusters has impaired functioning or is not functioning.
`UNAVAILABLE`	The primary cluster in the InnoDB ClusterSet is not functioning, because it is offline or has lost quorum, or MySQL Shell cannot contact the primary cluster to determine its status.

The overall ClusterSet status reported for an InnoDB ClusterSet deployment depends on the overall status of each InnoDB Cluster. An InnoDB Cluster in a ClusterSet reports three statuses:

- The global status (`globalStatus` field) is the status of the InnoDB Cluster with regards to its role in the InnoDB ClusterSet. This status shows whether the cluster can still function acceptably in the InnoDB ClusterSet deployment, even if it has some issues, such as a member server being currently offline. An InnoDB Cluster can be marked as invalidated during a failover, regardless of the status of the member servers, and if so this is shown as the global status.
- The cluster status (`status` field) is the status of the InnoDB Cluster with regards to its own functioning. This status shows whether the cluster has any technical issues, such as one or more members being offline, a loss of quorum, or a Group Replication error state. A cluster can tolerate certain issues but still function acceptably as part of an InnoDB ClusterSet deployment. For this reason, with the default verbosity level, the `clusterSet.status()` function only reports the cluster status for those clusters where it is causing a global status issue. To view the cluster status for all clusters in the InnoDB ClusterSet whether or not it is causing a global status issue, use the `extended` option to specify a higher verbosity level.
- The ClusterSet replication status (`clusterSetReplicationStatus` field) is the status of the ClusterSet replication channel for a replica InnoDB Cluster. This status shows whether the replica cluster has any issues with replicating from the primary cluster, so that these can be considered separately from any technical issues with the member servers in the cluster. A replica InnoDB Cluster reports the ClusterSet replication status whether or not it is causing a global status issue. A primary InnoDB Cluster does not have this status field, because the ClusterSet replication channel is not operating on the primary cluster.

At higher verbosity levels, the extended output for the `clusterSet.status()` function shows the status of each member server in each InnoDB Cluster. The output includes the member's Group Replication state (`memberState` field) and for a server in a replica cluster, the state of replication on the member. For information on the Group Replication states, see Group Replication Server States.

The global status (`globalStatus` field) reported for an InnoDB Cluster can be one of the following:

`OK`	The cluster is functioning acceptably in the InnoDB ClusterSet deployment. At least one of the member servers in the cluster is in Group Replication's `ONLINE` state, and the replication group has quorum. If the cluster is a replica cluster, the ClusterSet replication status is also `OK`. This global status does not necessarily mean there are no technical issues with the cluster. Some members might be offline, or the cluster might have too few members to provide tolerance for failures. However, the cluster is functioning well

	enough to continue as part of the InnoDB ClusterSet deployment. A primary cluster or a replica cluster can have this global status.
`OK_NOT_REPLICATING`	The cluster is functioning acceptably, but replication has stopped on the ClusterSet replication channel, either as a controlled stop or due to a replication error. Only a replica cluster can have this global status.
`OK_NOT_CONSISTENT`	The cluster is functioning acceptably, but the set of transactions on the cluster (the GTID set) has diverged from that on the primary cluster, such that there are extra transactions on the replica cluster that the primary cluster does not have. Replication might have stopped on the ClusterSet replication channel, either as a controlled stop or due to a replication error, or the channel might still be replicating. Only a replica cluster can have this global status. A replica cluster with this status is not available for a planned switchover, although a forced failover is possible.
`OK_MISCONFIGURED`	The cluster is functioning acceptably, but an incorrect configuration has been detected for the ClusterSet replication channel. For example, the channel might be replicating from the wrong source. The replication channel might be still running, or replication might have stopped. Only a replica cluster can have this global status.
`NOT_OK`	The cluster is not functioning at all as part of the InnoDB ClusterSet deployment due to a technical issue. It has lost quorum or all member servers are in Group Replication's `OFFLINE` status. A primary cluster or a replica cluster can have this global status. If a primary cluster has this global status, the InnoDB ClusterSet deployment is given the status `UNAVAILABLE`.
`UNKNOWN`	The cluster is the primary cluster for the InnoDB ClusterSet deployment but MySQL Shell currently cannot contact it to determine its status. While the primary cluster cannot be contacted, the InnoDB ClusterSet deployment is given the status `UNAVAILABLE`.
`INVALIDATED`	The cluster was invalidated during a failover process. During a controlled switchover process, data consistency is assured, and the original primary cluster is demoted to a working read-only replica cluster. However, during an emergency failover process, data consistency is not assured, so for safety, the original primary cluster is marked as invalidated during the failover process. Replica clusters are also marked as invalidated if they are unreachable or unavailable at the time of the failover, or during a controlled switchover. A cluster with this global status is not functioning at all as part of the InnoDB ClusterSet deployment. The cluster does not necessarily have any technical issues, and might be capable of rejoining the InnoDB ClusterSet deployment after manual validation. If the cluster can be contacted, you should verify that it has been shut down, so that it is not accepting new transactions.

The cluster status (`status` field) reported for an InnoDB Cluster can be one of the following, which can all be reported for a primary cluster or a replica cluster:

`OK`	All the member servers in the cluster are in Group Replication's `ONLINE` state, and there are three or more members in the cluster.
`OK_PARTIAL`	At least three of the member servers in the cluster are in Group Replication's `ONLINE` state. However, one or more member servers are in Group Replication's `OFFLINE`, `RECOVERING`, `ERROR`, or

	`UNREACHABLE` state, so they are not currently participating as active members of the cluster. A cluster in this situation is functioning well enough to continue as part of the InnoDB ClusterSet deployment, but to bring it up to `OK` status, resolve the issues with the member servers.
`OK_NO_TOLERANCE`	All the member servers in the cluster are in Group Replication's `ONLINE` state, but there are less than three members in the cluster, so it does not have sufficient tolerance for failures. A cluster in this situation is functioning well enough to continue as part of the InnoDB ClusterSet deployment, but to bring it up to `OK` status, add more member servers.
`OK_NO_TOLERANCE_PARTIAL`	One or two member servers in the cluster are in Group Replication's `ONLINE` state, but one or more are in Group Replication's `OFFLINE`, `RECOVERING`, `ERROR`, or `UNREACHABLE` state. The cluster therefore does not have sufficient tolerance for failures because of the unavailability of some members. A cluster in this situation is functioning well enough to continue as part of the InnoDB ClusterSet deployment, but to bring it up to `OK` status, resolve the issues with the member servers.
`NO_QUORUM`	The cluster does not have quorum, meaning that a majority of the replication group's member servers are unavailable for agreeing on a decision. Group Replication is able to reconfigure itself to the new group number if members leave voluntarily or are expelled by a group decision, so a loss of quorum means that the missing member servers have either failed or been cut off from the others by a network partition. A cluster in this situation cannot function as part of the InnoDB ClusterSet deployment. To bring a cluster in this state up to `OK` status, see Section 8.9, “InnoDB ClusterSet Repair and Rejoin”.
`OFFLINE`	All the member servers in the cluster are in Group Replication's `OFFLINE` state. A cluster in this situation cannot function as part of the InnoDB ClusterSet deployment. To bring a cluster in this state up to `OK` status if it is not currently supposed to be offline, see Section 8.9, “InnoDB ClusterSet Repair and Rejoin”.
`ERROR`	All the member servers in the cluster are in Group Replication's `ERROR` state. A cluster in this situation cannot function as part of the InnoDB ClusterSet deployment. To bring a cluster in this state up to `OK` status, see Section 8.9, “InnoDB ClusterSet Repair and Rejoin”.
`UNKNOWN`	MySQL Shell cannot currently contact any member servers to determine the cluster's status. If this is the primary cluster, the InnoDB ClusterSet deployment is given the status `UNAVAILABLE`.
`INVALIDATED`	The cluster was invalidated during a failover process. During a controlled switchover process, data consistency is assured, and the original primary cluster is demoted to a working read-only replica cluster. However, during an emergency failover process, data consistency is not assured, so for safety, the original primary cluster is marked as invalidated during the failover process. Replica clusters are also marked as invalidated if they are unreachable or unavailable at the time of the failover, or during a controlled switchover. A cluster with this global status is not functioning at all as part of the InnoDB ClusterSet deployment. The cluster does not necessarily have any technical issues, and might be capable of rejoining the InnoDB ClusterSet deployment after manual validation.

If the cluster can be contacted, you should verify that it has been shut down, so that it is not accepting new transactions. To handle this situation, see Section 8.9, “InnoDB ClusterSet Repair and Rejoin”.

The cluster status relates to technical issues with the InnoDB Cluster as a Group Replication group, rather than to the process of replication. For a replica cluster, the ClusterSet replication status (`clusterSetReplicationStatus` field) is also reported as follows:

`OK` The ClusterSet replication channel is running.

`STOPPED` The ClusterSet replication channel has been stopped in a controlled manner. This status is shown when the receiver thread, applier thread, or both threads have been stopped.

`CONNECTING` The replication channel is connecting. If an error occurs during connection, it is ignored until the channel state updated to either ON or OFF.

`ERROR` The ClusterSet replication channel has stopped due to a replication error, such as an incorrect configuration or a set of transactions that differs from the set on the primary cluster.

`MISCONFIGURED` An incorrect configuration has been detected for the ClusterSet replication channel, such as replicating from the wrong source. The channel might be still running, or replication might have stopped.

`MISSING` The ClusterSet replication channel does not exist on the servers in this cluster.

`UNKNOWN` MySQL Shell cannot currently contact the replica cluster to determine the replication channel's status.

If a cluster's only issue is with the ClusterSet replication channel, issuing the *`clusterSet`*`.rejoinCluster()` command for the cluster automatically corrects the channel's configuration if necessary and restarts the channel. This might be sufficient to fix the issue. For instructions to do this, see Section 8.9.5, “Rejoining a Cluster to an InnoDB ClusterSet”.

InnoDB ClusterSet Topology

If you just want to view the topology of the InnoDB ClusterSet, and do not need status information, you can use the *`clusterSet`*`.describe()` function instead. This function returns a JSON object describing the topology of an InnoDB ClusterSet deployment, and giving the IP address and identifier of each member server in each InnoDB Cluster. For example:

```
mysql-js> myclusterset.describe()
{
    "clusters": {
        "clusterone": {
            "clusterRole": "PRIMARY",
            "topology": [
                {
                    "address": "127.0.0.1:3310",
                    "label": "127.0.0.1:3310"
                },
                {
                    "address": "127.0.0.1:3320",
                    "label": "127.0.0.1:3320"
                },
                {
                    "address": "127.0.0.1:3330",
                    "label": "127.0.0.1:3330"
                }
            ]
        },
```

```
        "clustertwo": {
            "clusterRole": "REPLICA",
            "topology": [
                {
                    "address": "127.0.0.1:4410",
                    "label": "127.0.0.1:4410"
                },
                {
                    "address": "127.0.0.1:4420",
                    "label": "127.0.0.1:4420"
                },
                {
                    "address": "127.0.0.1:4430",
                    "label": "127.0.0.1:4430"
                }
            ]
        }
    },
    "domainName": "testclusterset",
    "primaryCluster": "clusterone"
}
```

This information is also provided by the extended output for the `clusterSet.status()` function.

For information on `clusterSet.setRoutingOption()`, see setRoutingOption().

MySQL Router Status for InnoDB ClusterSet

To see the MySQL Router instances that are registered for the InnoDB ClusterSet, issue the `clusterSet.listRouters()` command in MySQL Shell while connected to any member server in the InnoDB ClusterSet deployment. The command returns details of all the registered MySQL Router instances, or a single router instance that you specify using its router instance definition. For example:

```
mysql-js> myclusterset.listRouters()
{
    "domainName": "testclusterset",
    "routers": {
       "mymachine::Rome1": {
            "hostname": "mymachine",
            "lastCheckIn": 2021-10-15 11:58:37,
            "roPort": 6447,
            "roXPort": 6449,
            "rwPort": 6446,
            "rwXPort": 6448,
            "targetCluster": "primary",
            "version": "8.0.27"
        },
        "mymachine2::Rome2": {
            "hostname": "mymachine2",
            "lastCheckIn": 2021-10-15 11:58:37,
            "roPort": 6447,
            "roXPort": 6449,
            "rwPort": 6446,
            "rwXPort": 6448,
            "targetCluster": "primary",
            "version": "8.0.27"
        }
    }
}
```

The instance information includes the name of the MySQL Router instance, the port numbers for read and write traffic using MySQL classic protocol and X Protocol, the target cluster, and the time the instance last checked in with the target cluster. If MySQL Router is at a lower version than that required to work with this InnoDB ClusterSet deployment, the instance information states this.

To see the routing options that are set for each MySQL Router instance, and the global policy for the InnoDB ClusterSet deployment, issue `clusterSet.routingOptions()` in MySQL Shell while connected to any member server in the InnoDB ClusterSet deployment. A setting for a specific MySQL Router instance overrides a global policy. For example:

```
mysql-js> myclusterset.routingOptions()
{
    "domainName": "testclusterset",
    "global": {
        "invalidated_cluster_policy": "drop_all",
        "target_cluster": "primary"
    },
    "routers": {
        "mymachine::Rome1":  {
            "target_cluster": "primary"
            "invalidated_cluster_policy": "accept_ro"
        },
        "mymachine2::Rome2": {}
    }
}
```

If a particular routing option is not displayed for a MySQL Router instance, as in the example above for `Rome2`, it means the instance does not have that policy set, and it follows the global policy. The output for `Rome1` shows `"target_cluster": "primary"`, which is the same as the global policy. This is because `Rome1` has had the routing option explicitly set to `"primary"` by a *`clusterSet`*`.setRoutingOption()` command, in which case it is displayed. To clear a routing option, set it to `null`.

8.7 InnoDB ClusterSet Controlled Switchover

A controlled switchover makes a selected replica cluster into the primary cluster for the InnoDB ClusterSet deployment. During a controlled switchover process, data consistency is assured. The process verifies that the selected replica cluster is synchronized with the primary cluster (which might mean a short wait if there is replication lag), then makes that cluster into the primary of the InnoDB ClusterSet. The original primary cluster is demoted to a working read-only replica cluster. You can then take the original primary offline if necessary, repair any issues, and bring it back into operation in the InnoDB ClusterSet deployment.

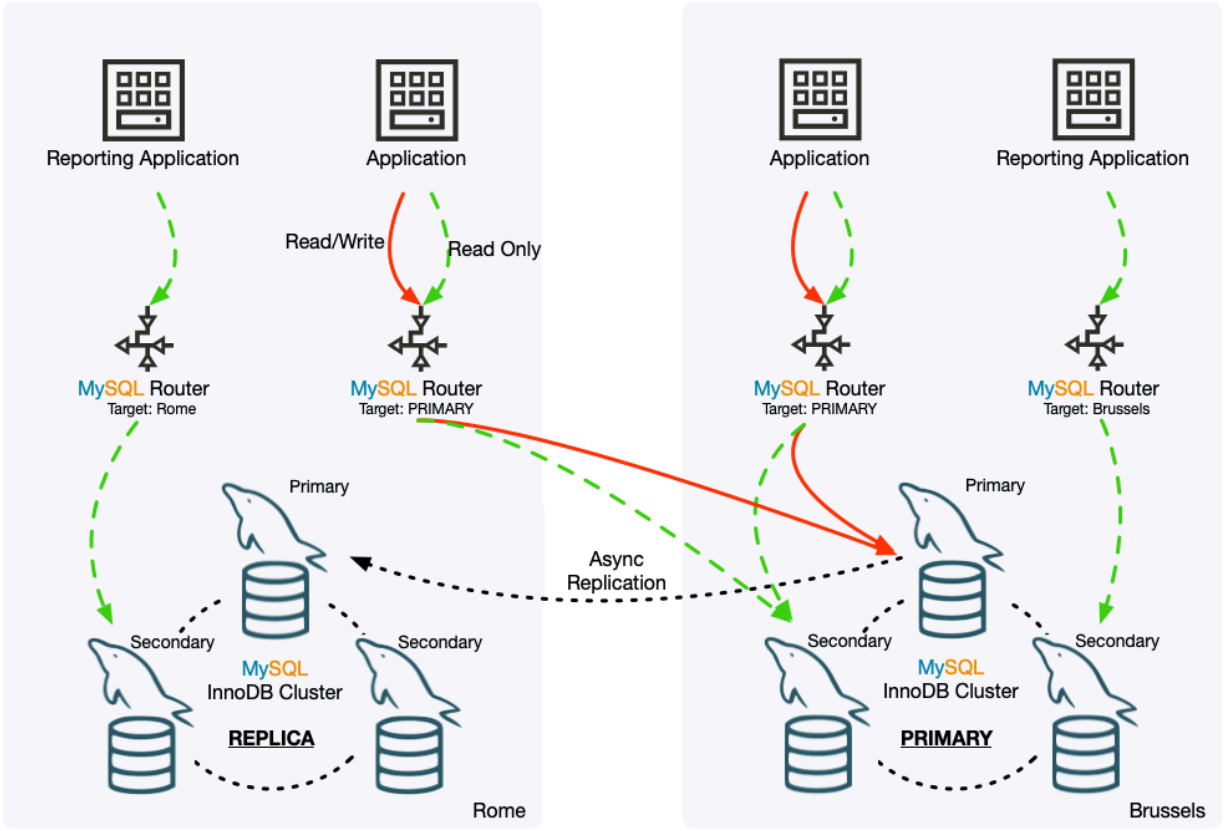
Follow the controlled switchover procedure if the primary cluster in an InnoDB ClusterSet deployment is functioning acceptably, but you need to carry out maintenance or fix some minor issues to improve the primary cluster's function. A primary cluster that is functioning acceptably has the global status `OK` when you check it using AdminAPI's *`clusterSet`*`.status()` command in MySQL Shell.

If the primary cluster is not functioning acceptably (with the global status `NOT_OK`) in the InnoDB ClusterSet deployment, first try to repair any issues using AdminAPI through MySQL Shell. For example, if the primary cluster has lost quorum, it can be restored using a *`cluster`*`.forceQuorumUsingPartitionOf` command. For instructions to do this, see Section 8.9, “InnoDB ClusterSet Repair and Rejoin”.

If you cannot fix the issue by working with the primary cluster (for example, because you cannot contact it), you need to perform an emergency failover. An emergency failover is designed for disaster recovery when the primary cluster is suddenly unavailable. That procedure carries the risk of losing transactions and creating a split-brain situation for the InnoDB ClusterSet. If you do need to carry out an emergency failover, follow the procedure in Section 8.8, “InnoDB ClusterSet Emergency Failover” to ensure that the risk is managed.

The diagram shows the effects of a controlled switchover in an example InnoDB ClusterSet deployment. The primary cluster in the Rome datacenter requires maintenance, so a controlled switchover has been carried out to make the replica cluster in the Brussels datacenter into the primary of the InnoDB ClusterSet deployment, and demote the Rome cluster to a replica. The ClusterSet replication channel on the Rome cluster has been activated by the controlled switchover process, and it is replicating transactions from the Brussels cluster. Now that the Rome cluster is a replica cluster, the member servers or the complete cluster can safely be taken offline if required to carry out the maintenance work.

Figure 8.2 InnoDB ClusterSet Switchover



The MySQL Router instances in the example InnoDB Cluster deployment that were set to follow the primary have routed read and write traffic to the Brussels cluster which is now the primary. The MySQL Router instance that was routing read traffic to the Brussels cluster by name when it was a replica cluster, continues to route traffic to it, and is not affected by the fact that the cluster is now the primary rather than a replica cluster. Similarly, the MySQL Router instance that was routing read traffic to the Rome cluster by name can continue to do this, because the replica cluster still accepts read traffic.

To carry out a controlled switchover for the primary InnoDB Cluster, follow this procedure:

1. Using MySQL Shell, connect to any member server in the primary cluster or in one of the replica clusters, using an InnoDB Cluster administrator account (created with `cluster.setupAdminAccount()`). You may also use the InnoDB Cluster server configuration account, which also has the required permissions. Get the `ClusterSet` object using `dba.getClusterSet()` or `cluster.getClusterSet()` command. It is important to use an InnoDB Cluster administrator account or server configuration account so that the default user account stored in the `ClusterSet` object has the correct permissions. For example:

```
mysql-js> \connect admin2@127.0.0.1:3310
Creating a session to 'admin2@127.0.0.1:3310'
Please provide the password for 'admin2@127.0.0.1:3310': ********
Save password for 'admin2@127.0.0.1:3310'? [Y]es/[N]o/Ne[v]er (default No):
Fetching schema names for autocompletion... Press ^C to stop.
Closing old connection...
Your MySQL connection id is 52
Server version: 8.0.27-commercial MySQL Enterprise Server - Commercial
No default schema selected; type \use <schema> to set one.
<ClassicSession:admin2@127.0.0.1:3310>
mysql-js> myclusterset = dba.getClusterSet()
<ClusterSet:testclusterset>
```

   In this example:

- *admin2@127.0.0.1:3310* is the URI-like connection string for any member server instance that is online in the cluster.

  The URI-like connection string is comprised of the following elements:

- *admin2* is the user name for an InnoDB Cluster administrator account.
- *127.0.0.1:3310* is the host and port for the member server instance, as displayed by the *cluster*.status() command.
- The returned ClusterSet object is assigned to the variable myclusterset.

2. Check the status of the whole InnoDB ClusterSet deployment using AdminAPI's *clusterSet*.status() command in MySQL Shell. Use the extended option to view detailed information for all the clusters in the deployment, and check for any issues. For example:

```
mysql-js> myclusterset.status({extended: 1})
```

   For an explanation of the output, see Section 8.6, “InnoDB ClusterSet Status and Topology”.

3. Identify a suitable replica cluster that can take over as the primary cluster. A replica cluster's eligibility for a controlled switchover depends on its global status, as reported by the *clusterSet*.status() command:

   Table 8.1 Permitted Cluster Operations By Status

InnoDB Cluster Global Status in ClusterSet	Routable	Controlled Switchover	Emergency Failover
OK	Yes	Yes	Yes
OK_NOT_REPLICATING	Yes, if specified as target cluster by name	Yes	Yes
OK_NOT_CONSISTENT	Yes, if specified as target cluster by name	No	Yes
OK_MISCONFIGURED	Yes	Yes	Yes
NOT_OK	No	No	No
INVALIDATED	Yes, if specified as target cluster by name and accept_ro routing policy is set	No	No
UNKNOWN	Connected MySQL Router instances might still be routing traffic to the cluster	No	No

   A replica cluster with the global status OK_NOT_CONSISTENT has a set of transactions on the cluster (the GTID set) that is inconsistent with the GTID set on the primary cluster. InnoDB ClusterSet does not permit a controlled switchover to a cluster in this state, because clients would access incorrect data. An emergency failover is possible, if the cluster has the most up to date set of transactions among the available options.

4. Check the routing options that are set for each MySQL Router instance, and the global policy for the InnoDB ClusterSet deployment, by issuing a *clusterSet*.routingOptions() command in MySQL Shell while connected to any member server in the InnoDB ClusterSet deployment. For example:

```
mysql-js> myclusterset.routingOptions()
{
    "domainName": "testclusterset",
    "global": {
```

```
        "invalidated_cluster_policy": "drop_all",
        "target_cluster": "primary"
    },
    "routers": {
        "Rome1":  {
            "target_cluster": "primary"
        },
        "Rome2": {}
    }
}
```

By default, a MySQL Router instance sends traffic to whichever cluster is currently the primary in the InnoDB ClusterSet deployment. If all the MySQL Router instances are set to follow the primary (`"target_cluster": "primary"`), traffic will be automatically redirected to the new primary cluster within a few seconds of the switchover. If a routing option is not displayed for a MySQL Router instance, as in the example above for `Rome2`, it means the instance does not have that policy set, and it follows the global policy.

If any of the instances are set to target the current primary cluster by name (`"target_cluster": "name_of_primary_cluster"`), they will not redirect traffic to the new primary. In that situation, if it is appropriate for the application, you can use the `clusterSet.setRoutingOption()` command to change the routing policy for those instances. You could change those instances to follow the primary (`"target_cluster": "primary"`), in which case that option can be set now. For example:

```
mysql-js> myclusterset.setRoutingOption('Rome1', 'target_cluster', 'primary')
Routing option 'target_cluster' successfully updated in router 'Rome1'.
```

In this example, *`myclusterset`* is the variable for the `ClusterSet` object, and *`Rome1`* is the name of the MySQL Router instance.

Or you could specify the replica cluster that will take over as the primary, in which case set the option (`"target_cluster": "name_of_new_primary_cluster"`) after the switchover has taken place, when you have verified that it has worked.

5. Issue a `clusterSet.setPrimaryCluster()` command, naming the replica cluster that will take over as the new primary cluster. Use the `ClusterSet` object that you retrieved using an InnoDB Cluster administrator account, with the `dba.getClusterSet()` or `cluster.getClusterSet()` command. For example:

```
mysql-js> myclusterset.setPrimaryCluster('clustertwo')
Switching the primary cluster of the clusterset to 'clustertwo'
* Verifying clusterset status
** Checking cluster clustertwo
  Cluster 'clustertwo' is available
** Checking cluster clusterone
  Cluster 'clusterone' is available

* Refreshing replication account of demoted cluster
* Synchronizing transaction backlog at 127.0.0.1:4410
** Transactions replicated  ############################################################  100%
* Updating metadata

* Updating topology
** Changing replication source of 127.0.0.1:3330 to 127.0.0.1:4410
* Acquiring locks in replicaset instances
** Pre-synchronizing SECONDARIES
** Acquiring global lock at PRIMARY
** Acquiring global lock at SECONDARIES

* Synchronizing remaining transactions at promoted primary
** Transactions replicated  ############################################################  100%
* Updating replica clusters
```

```
Cluster 'clustertwo' was promoted to PRIMARY of the clusterset. The PRIMARY instance is '127.0.0.1:4410
```

For the *clusterSet*.setPrimaryCluster() command:

- The *clusterName* parameter is required and specifies the identifier used for the replica cluster in the InnoDB ClusterSet, as given in the output from the *clusterSet*.status() command. In the example, *clustertwo* is the cluster that is to become the new primary.
- Use the dryRun option if you want to carry out validations and log the changes without actually executing them.
- Use the timeout option to set the maximum number of seconds to wait for the replica cluster to synchronize with the primary cluster before the switchover takes place. If the timeout expires, the switchover is canceled.
- Use the invalidateReplicaClusters option to name any replica clusters that are unreachable or unavailable. These will be marked as invalidated during the switchover process. The switchover is canceled if any unreachable or unavailable replica clusters that you do not name are discovered during the process. In this situation you must either repair and rejoin the replica clusters then retry the command, or name them on this option when you retry the command, and fix them later.

When you issue the *clusterSet*.setPrimaryCluster() command, MySQL Shell checks that the target replica cluster complies with the requirements to take over as the primary cluster, and returns an error if it does not. If the target replica cluster meets the requirements, MySQL Shell carries out the following tasks:

- Checks for any unreachable or unavailable replica clusters that have not been specified using invalidateReplicaClusters.
- Waits for the target replica cluster to synchronize with the current primary cluster by applying any outstanding transactions from the primary. If the timeout set by the timeout option expires before the replica cluster has finished applying transactions, the switchover is canceled.
- Locks the current primary cluster by issuing a FLUSH TABLES WITH READ LOCK statement and setting the super_read_only system variable on all member servers, to prevent further changes during the switchover. The Group Replication member action mysql_disable_super_read_only_if_primary is disabled so that super_read_only remains set after the failover.
- Reconciles the differences in view change events between the current primary cluster and the replica clusters so that the GTID sets are identical. These Group Replication internal transactions are identified by the UUID specified by the group_replication_view_change_uuid system variable. MySQL Shell injects empty transactions on all the replica clusters to match the view change events on the primary cluster.
- Updates the ClusterSet replication channel on all replica clusters to replicate from the target cluster as the new primary cluster.
- Disables super_read_only on the primary server of the target cluster, and enables the Group Replication member action mysql_disable_super_read_only_if_primary to handle any changes to the primary server in that cluster.
- Disables the Group Replication member action mysql_disable_super_read_only_if_primary on the primary server of the old primary cluster, so that it remains read-only, and enables the Group Replication member action mysql_start_failover_channels_if_primary on that server to enable asynchronous connection failover for replicas on the ClusterSet replication channel.

- Sets the target cluster as the primary cluster in the ClusterSet metadata, and changes the old primary cluster into a replica cluster.

6. Issue a *clusterSet*.status() command again using the extended option, to verify the status of the InnoDB ClusterSet deployment.

7. If you have any MySQL Router instances to switch over to targeting the new primary cluster, do that now. For example:

```
mysql-js> myclusterset.setRoutingOption('Rome1', 'target_cluster', 'clustertwo')
Routing option 'target_cluster' successfully updated in router 'Rome1'.
```

   In this example, *myclusterset* is the variable for the ClusterSet object, *Rome1* is the name of the MySQL Router instance, and *clustertwo* is the name of the specific cluster to target. When you have finished, issue a *clusterSet*.routingOptions() command to check that all the MySQL Router instances are now routing correctly.

8. Now you can work with the old primary cluster to fix issues or carry out maintenance. If you had to invalidate any replica clusters during the switchover process, you can repair these as well and add them back into the InnoDB ClusterSet. Section 8.9, “InnoDB ClusterSet Repair and Rejoin” explains how to repair issues with a cluster, how to rejoin a cluster to the InnoDB ClusterSet, and how to make a cluster into the primary cluster again.

8.8 InnoDB ClusterSet Emergency Failover

An emergency failover makes a selected replica cluster into the primary InnoDB Cluster for the InnoDB ClusterSet deployment. This procedure can be used when the current primary cluster is not working or cannot be contacted. During an emergency failover process, data consistency is not assured, so for safety, the original primary cluster is marked as invalidated during the failover process. If the original primary cluster remains online, it should be shut down as soon as it can be contacted. You can repair and rejoin an invalidated primary cluster to the InnoDB ClusterSet topology afterwards, provided that you can fix the issues.

When the primary InnoDB Cluster in an InnoDB ClusterSet deployment has an issue or you cannot access it, do not immediately implement an emergency failover to a replica cluster. Instead, you should always start by attempting to repair the currently active primary cluster.

Important

Why Not Just Fail Over? The replica clusters in the InnoDB ClusterSet topology are doing their best to keep themselves synchronized with the primary cluster. However, depending on the volume of transactions and the speed and capacity of the network connections between the primary cluster and the replica clusters, replica clusters can fall behind the primary cluster in receiving transactions and applying the changes to their data. This is called replication lag. Some replication lag is to be expected in most replication topologies, and is quite likely in an InnoDB ClusterSet deployment where the clusters are geographically dispersed and in different data centers.

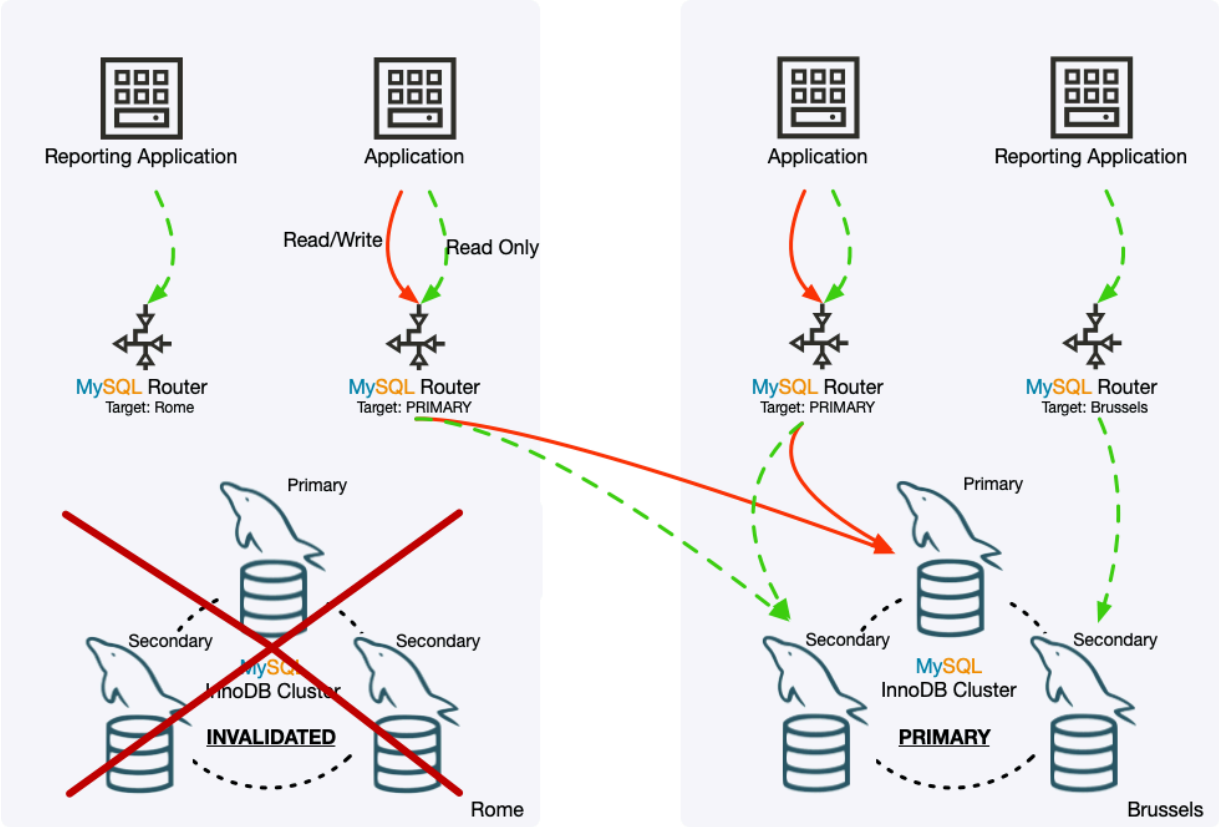
Also, it is possible for the primary cluster to become disconnected from other elements of the InnoDB ClusterSet topology by a network partition, but remain online. If that happens, some replica clusters might stay with the primary cluster, and some instances and client applications might continue to connect to the primary cluster and apply transactions. In this situation, the partitioned areas of the InnoDB ClusterSet topology begin to diverge from each other, with a different transaction set on each group of servers.

When there is replication lag or a network partition, if you trigger an emergency failover to a replica cluster, any unreplicated or divergent transactions on the primary cluster are at risk of being lost. In the case of a network partition,

> the failover can create a split-brain situation, where the different parts of the topology have divergent transaction sets. You should therefore always make an attempt to repair or reconnect the primary cluster before triggering an emergency failover. If the primary cluster cannot be repaired quickly enough or cannot be reached, you can go ahead with the emergency failover.

The diagram shows the effects of an emergency failover in an example InnoDB ClusterSet deployment. The primary cluster in the Rome datacenter has gone offline, so an emergency failover has been carried out to make the replica cluster in the Brussels datacenter into the primary InnoDB Cluster of the InnoDB ClusterSet deployment. The Rome cluster has been marked as invalidated, and its status in the InnoDB ClusterSet deployment has been demoted to a replica cluster, although it is not currently able to replicate transactions from the Brussels cluster.

Figure 8.3 InnoDB ClusterSet Failover



The MySQL Router instances that were set to follow the primary have routed read and write traffic to the Brussels cluster which is now the primary. The MySQL Router instance that was routing read traffic to the Brussels cluster by name when it was a replica cluster, continues to route traffic to it, and is not affected by the fact that the cluster is now the primary rather than a replica cluster. However, the MySQL Router instance that was routing read traffic to the Rome cluster by name cannot currently send any traffic there. The reporting application in this example does not need to report when the local datacenter is offline, but if the application did still need to function, the MySQL Router instance should have its routing options changed either to follow the primary or to send traffic to the Brussels cluster.

To carry out an emergency failover for the primary InnoDB Cluster, follow this procedure:

1. Using MySQL Shell, connect to any member server that is still active in the InnoDB ClusterSet deployment, using an InnoDB Cluster administrator account (created with `cluster.setupAdminAccount()`). You may also use the InnoDB Cluster server configuration account, which also has the required permissions.

   When the connection is established, get the `ClusterSet` object from that member server using a `dba.getClusterSet()` or `cluster.getClusterSet()` command. A `ClusterSet` object

that you previously retrieved from a member server that is now offline will not work any more, so you need to get it again from a server that is online. It is important to use an InnoDB Cluster administrator account or server configuration account so that the default user account stored in the `ClusterSet` object has the correct permissions. For example:

```
mysql-js> \connect admin2@127.0.0.1:4410
Creating a session to 'admin2@127.0.0.1:4410'
Please provide the password for 'admin2@127.0.0.1:4410': ********
Save password for 'admin2@127.0.0.1:4410'? [Y]es/[N]o/Ne[v]er (default No):
Fetching schema names for autocompletion... Press ^C to stop.
Closing old connection...
Your MySQL connection id is 71
Server version: 8.0.27-commercial MySQL Enterprise Server - Commercial
No default schema selected; type \use <schema> to set one.
<ClassicSession:admin2@127.0.0.1:4410>

mysql-js> myclusterset = dba.getClusterSet()
<ClusterSet:testclusterset>
```

2. Check the status of the whole deployment using AdminAPI's *`clusterSet`*`.status()` function in MySQL Shell. Use the `extended` option to see exactly where and what the issues are. For example:

   ```
   mysql-js> myclusterset.status({extended: 1})
   ```

   For an explanation of the output, see Section 8.6, “InnoDB ClusterSet Status and Topology”.

3. An InnoDB Cluster can tolerate some issues and be functioning well enough to continue as part of the InnoDB ClusterSet deployment. A primary cluster that is functioning acceptably has the global status `OK` when you check it using the *`clusterSet`*`.status()` command. For example, if one of the member servers in a cluster goes offline, even if that server is the primary, the underlying Group Replication technology can handle this situation and reconfigure itself.

   If the primary cluster is still functioning acceptably in the InnoDB ClusterSet deployment according to the reported status, but you need to carry out maintenance or fix some minor issues to improve the primary cluster's function, you can carry out a controlled switchover to a replica cluster. You can then take the primary cluster offline if necessary, repair any issues, and bring it back into operation in the InnoDB ClusterSet deployment. For instructions to do this, see Section 8.7, “InnoDB ClusterSet Controlled Switchover”.

4. If the primary cluster is not functioning acceptably (with the global status `NOT_OK`) in the InnoDB ClusterSet deployment, but you can contact it, first try to repair any issues using AdminAPI through MySQL Shell. For example, if the primary cluster has lost quorum, it can be restored using a *`cluster`*`.forceQuorumUsingPartitionOf` command. For instructions to do this, see Section 8.9, “InnoDB ClusterSet Repair and Rejoin”.

5. If you cannot carry out a controlled switchover, and you cannot fix the issue quickly enough by working with the primary cluster (for example, because you cannot contact it), proceed with the emergency failover. First identify a suitable replica cluster that can take over as the primary cluster. A replica cluster's eligibility for an emergency failover depends on its global status, as reported by the *`clusterSet`*`.status()` command:

   Table 8.2 Permitted Cluster Operations By Status

InnoDB Cluster Global Status in ClusterSet	Routable	Controlled Switchover	Emergency Failover
`OK`	Yes	Yes	Yes
`OK_NOT_REPLICATING`	Yes, if specified as target cluster by name	Yes	Yes
`OK_NOT_CONSISTENT`	Yes, if specified as target cluster by name	No	Yes

InnoDB Cluster Global Status in ClusterSet	Routable	Controlled Switchover	Emergency Failover
OK_MISCONFIGURED	Yes	Yes	Yes
NOT_OK	No	No	No
INVALIDATED	Yes, if specified as target cluster by name and accept_ro routing policy is set	No	No
UNKNOWN	Connected Router instances might still be routing traffic to the cluster	No	No

The replica cluster you select must have the most up to date set of transactions (GTID set) among all of the replica clusters that are reachable. If more than one replica cluster is eligible for the emergency failover, check the replication lag for each cluster (which is shown in the extended output for the *clusterSet*.status() command). Select the replica cluster with the least replication lag, which should therefore have the most transactions. The emergency failover process checks the GTID sets for all the replica clusters that are currently reachable, and tells you if another cluster is more up to date, so you can try again with that cluster.

6. Check the routing options that are set for each MySQL Router instance, and the global policy for the InnoDB ClusterSet deployment, by issuing a *clusterSet*.routingOptions() command in MySQL Shell while connected to any member server in the InnoDB ClusterSet deployment. For example:

```
mysql-js> myclusterset.routingOptions()
{
    "domainName": "testclusterset",
    "global": {
        "invalidated_cluster_policy": "drop_all",
        "target_cluster": "primary"
    },
    "routers": {
        "Rome1":  {
            "target_cluster": "primary"
        },
        "Rome2": {}
    }
}
```

If all the MySQL Router instances are set to follow the primary ("target_cluster": "primary"), traffic will be automatically redirected to the new primary cluster within a few seconds of the failover. If a routing option is not displayed for a MySQL Router instance, as in the example above with "target_cluster" for Rome2, it means the instance does not have that policy set, and it follows the global policy.

If any of the instances are set to target the current primary cluster by name ("target_cluster": "*name_of_primary_cluster*"), they will not redirect traffic to the new primary. When the primary cluster is not functioning, the *clusterSet*.setRoutingOption() command cannot be used to change the routing options, so you cannot redirect the traffic handled by that MySQL Router instance until failover to the new primary cluster is complete.

7. If you can, try to verify that the original primary cluster is offline, and if it is online, attempt to shut it down. If it remains online and continues to receive traffic from clients, a split-brain situation can be created where the separated parts of the InnoDB ClusterSet diverge.

8. To proceed with the emergency failover, issue a *clusterSet*.forcePrimaryCluster() command, naming the replica cluster that will take over as the new primary cluster. For example:

```
mysql-js> myclusterset.forcePrimaryCluster("clustertwo")
Failing-over primary cluster of the clusterset to 'clustertwo'
```

```
* Verifying primary cluster status
None of the instances of the PRIMARY cluster 'clusterone' could be reached.
* Verifying clusterset status
** Checking cluster clustertwo
  Cluster 'clustertwo' is available
** Checking whether target cluster has the most recent GTID set
* Promoting cluster 'clustertwo'
* Updating metadata

PRIMARY cluster failed-over to 'clustertwo'. The PRIMARY instance is '127.0.0.1:4410'
Former PRIMARY cluster was INVALIDATED, transactions that were not yet replicated may be lost.
```

In the *clusterSet*.forcePrimaryCluster() command:

- The *clusterName* parameter is required and specifies the identifier used for the replica cluster in the InnoDB ClusterSet, as given in the output from the *clusterSet*.status() command. In the example, *clustertwo* is the cluster that is to become the new primary.
- Use the dryRun option if you want to carry out validations and log the changes without actually executing them.
- Use the invalidateReplicaClusters option to name any replica clusters that are unreachable or unavailable. These will be marked as invalidated during the failover process. The failover is canceled if any unreachable or unavailable replica clusters that you do not name are discovered during the process. In this situation you must either repair and rejoin the replica clusters then retry the command, or name them on this option when you retry the command, and fix them later.
- Use the timeout option to define the maximum number of seconds to wait for pending transactions to be applied in each instance of the cluster. Ensuring GTID_EXECUTED has the most up-to-date GTID set. The default value is retrieved from the dba.gtidWaitTimeout option.

When you issue the *clusterSet*.forcePrimaryCluster() command, MySQL Shell checks that the target replica cluster complies with the requirements to take over as the primary cluster, and returns an error if it does not.

If the target replica cluster meets the requirements, MySQL Shell carries out the following tasks:

- Attempts to contact the current primary cluster, and stops the failover if it actually can be reached.
- Checks for any unreachable or unavailable replica clusters that have not been specified using invalidateReplicaClusters, and stops the failover if any are found.
- Marks all replica clusters listed in invalidateReplicaClusters as invalidated, and marks the old primary cluster as invalidated.
- Checks that the target replica cluster has the most up to date GTID set among the available replica clusters. This involves stopping the ClusterSet replication channel in all of the replica clusters.
- Updates the ClusterSet replication channel on all replica clusters to replicate from the target cluster as the new primary cluster.
- Sets the target cluster as the primary cluster in the ClusterSet metadata, and changes the old primary cluster into a replica cluster, although it is not currently functioning as a replica cluster because it is marked as invalidated.

During an emergency failover, MySQL Shell does not attempt to synchronize the target replica cluster with the current primary cluster, and does not lock the current primary cluster. If the original primary cluster remains online, it should be shut down as soon as it can be contacted.

9. If you have any MySQL Router instances to switch over to targeting the new primary cluster, do that now. You can change them to follow the primary (`"target_cluster": "primary"`), or specify the replica cluster that has taken over as the primary (`"target_cluster": "name_of_new_primary_cluster"`). For example:

   ```
   mysql-js> myclusterset.setRoutingOption('Rome1', 'target_cluster', 'primary')
   or
   mysql-js> myclusterset.setRoutingOption('Rome1', 'target_cluster', 'clustertwo')
   Routing option 'target_cluster' successfully updated in router 'Rome1'.
   ```

   Issue a `clusterSet.routingOptions()` command to check that all the MySQL Router instances are now routing correctly.

10. Issue a `clusterSet.status()` command again using the `extended` option, to verify the status of the InnoDB ClusterSet deployment.

11. If and when you are able to contact the old primary cluster again, first ensure that no application traffic is being routed to it, and take it offline. Then follow the process in Section 8.9, “InnoDB ClusterSet Repair and Rejoin” to check the transactions and decide how to arrange the InnoDB ClusterSet topology going forward.

    Following an emergency failover, and there is a risk of the transaction sets differing between parts of the ClusterSet, you have to fence the cluster either from write traffic or all traffic. For more details, see Fencing Clusters in an InnoDB ClusterSet.

    If you had to invalidate any replica clusters during the switchover process, if and when you are able to contact them again, you can use the process in Section 8.9, “InnoDB ClusterSet Repair and Rejoin” to repair them and add them back into the InnoDB ClusterSet.

8.9 InnoDB ClusterSet Repair and Rejoin

Use this information if you need to repair a cluster in an InnoDB ClusterSet deployment. You can use the information here in any of the following situations:

- A cluster in the InnoDB ClusterSet requires maintenance but has no issues with its functioning.
- A cluster is functioning acceptably in the InnoDB ClusterSet deployment but has some issues, such as member servers that are offline.
- A cluster is not functioning acceptably and needs to be repaired.
- A cluster has been marked as invalidated during an emergency failover or controlled switchover procedure.

Section 8.6, “InnoDB ClusterSet Status and Topology” explains how to check the status of an InnoDB Cluster and of the whole InnoDB ClusterSet deployment, and the situations in which a cluster might need repair. You can identify the following situations from the output of the `clusterSet.status()` command:

- A cluster does not have quorum (that is, not enough members are online to have a majority).
- No members of a cluster can be reached.
- A cluster's ClusterSet replication channel is stopped.
- A cluster's ClusterSet replication channel is configured incorrectly.
- A cluster's GTID set is inconsistent with the GTID set on the primary cluster in the InnoDB ClusterSet.
- A cluster has been marked as invalidated. If the cluster is still online, the command warns that a split-brain situation might result.

If the cluster is the primary cluster in the InnoDB ClusterSet deployment, before repairing it, you might need to carry out a controlled switchover or an emergency failover to demote it to a replica cluster. After that, you can take the cluster offline if necessary to repair it, and the InnoDB ClusterSet will remain available during that time.

- A controlled switchover is suitable if the primary cluster is functioning acceptably but requires maintenance or has minor issues. A primary cluster that is functioning acceptably has the global status `OK` when you check it using the *`clusterSet`*`.status()` command. Section 8.7, “InnoDB ClusterSet Controlled Switchover” explains how to perform this operation.
- An emergency failover is suitable if you cannot contact the primary cluster at all. Section 8.8, “InnoDB ClusterSet Emergency Failover” explains how to perform this operation.
- If the primary cluster is not functioning acceptably (with the global status `NOT_OK`) but it can be contacted, make an attempt to repair any issues using the information in this section. An emergency failover carries the risk of losing transactions and creating a split-brain situation for the InnoDB ClusterSet. If you cannot repair the primary cluster quickly enough to restore availability, proceed with an emergency failover and then repair it if possible.

Follow this procedure to repair an InnoDB Cluster that is part of an InnoDB ClusterSet deployment:

1. Using MySQL Shell, connect to any member server in the primary cluster or in one of the replica clusters, using an InnoDB Cluster administrator account (created with *`cluster`*`.setupAdminAccount()`). You may also use the InnoDB Cluster server configuration account, which also has the required permissions. When the connection is established, get the `ClusterSet` object using a `dba.getClusterSet()` or *`cluster`*`.getClusterSet()` command. It is important to use an InnoDB Cluster administrator account or server configuration account so that the default user account stored in the `ClusterSet` object has the correct permissions. For example:

   ```
   mysql-js> \connect admin2@127.0.0.1:4410
   Creating a session to 'admin2@127.0.0.1:4410'
   Please provide the password for 'admin2@127.0.0.1:4410': ********
   Save password for 'admin2@127.0.0.1:4410'? [Y]es/[N]o/Ne[v]er (default No):
   Fetching schema names for autocompletion... Press ^C to stop.
   Closing old connection...
   Your MySQL connection id is 42
   Server version: 8.0.27-commercial MySQL Enterprise Server - Commercial
   No default schema selected; type \use <schema> to set one.
   <ClassicSession:admin2@127.0.0.1:4410>
   mysql-js> myclusterset = dba.getClusterSet()
   <ClusterSet:testclusterset>
   ```

2. Check the status of the whole deployment using AdminAPI's *`clusterSet`*`.status()` command in MySQL Shell. Use the `extended` option to see exactly where and what the issues are. For example:

   ```
   mysql-js> myclusterset.status({extended: 1})
   ```

   For an explanation of the output, see Section 8.6, “InnoDB ClusterSet Status and Topology”.

3. Still using an InnoDB Cluster administrator account (created with *`cluster`*`.setupAdminAccount()`) or InnoDB Cluster server configuration account, get the `Cluster` object using `dba.getCluster()`. You can either connect to any member server in the cluster you are repairing, or connect to any member of the InnoDB ClusterSet and use the `name` parameter on `dba.getCluster()` to specify the cluster you want. For example:

   ```
   mysql-js> cluster2 = dba.getClusterSet()
   <Cluster:clustertwo>
   ```

4. Check the status of the cluster using AdminAPI's *`cluster`*`.status()` command in MySQL Shell. Use the `extended` option to get the most details about the cluster. For example:

   ```
   mysql-js> cluster2.status({extended: 2})
   ```

For an explanation of the output, see Checking a cluster's Status with `Cluster.status()`.

5. Following an emergency failover, and there is a risk of the transaction sets differing between parts of the ClusterSet, you have to fence the cluster either from write traffic or all traffic. Section 8.9.1, “Fencing Clusters in an InnoDB ClusterSet” explains how, to fence and unfence a cluster, from MySQL Shell 8.0.28.

6. If the set of transactions (the GTID set) on the cluster is inconsistent, fix this first. The `clusterSet.status()` command warns you if a replica cluster's GTID set is inconsistent with the GTID set on the primary cluster in the InnoDB ClusterSet. A replica cluster in this state has the global status `OK_NOT_CONSISTENT`. You also need to check the GTID set on a former primary cluster, or a replica cluster, that has been marked as invalidated during a controlled switchover or emergency failover procedure. A cluster with extra transactions compared to the other clusters in the ClusterSet can continue to function acceptably in the ClusterSet while it stays active. However, a cluster with extra transactions cannot rejoin the ClusterSet. Section 8.9.2, “Inconsistent Transaction Sets (GTID Sets) in InnoDB ClusterSet Clusters” explains how to check for and resolve issues with the transactions on a server.

7. If there is a technical issue with a member server in the cluster, or with the overall membership of the cluster (such as insufficient fault tolerance or a loss of quorum), you can work with individual member servers or adjust the cluster membership to resolve this. Section 8.9.3, “Repairing Member Servers and Clusters in an InnoDB ClusterSet” explains what operations are available to work with the member servers in a cluster.

8. If you cannot repair a cluster, you can remove it from the InnoDB ClusterSet using a `clusterSet.removeCluster()` command. For instructions to do this, see Section 8.9.4, “Removing a Cluster from an InnoDB ClusterSet”. A removed InnoDB Cluster cannot be added back into an InnoDB ClusterSet deployment. If you want to use the server instances in the deployment again, you will need to set up a new cluster using them.

9. When you have repaired a cluster or carried out the required maintenance, you can rejoin it to the InnoDB ClusterSet using a `clusterSet.rejoin()` command. This command validates that the cluster is able to rejoin, updates and starts the ClusterSet replication channel, and removes any invalidated status from the cluster. For instructions to do this, see Section 8.9.5, “Rejoining a Cluster to an InnoDB ClusterSet”.

8.9.1 Fencing Clusters in an InnoDB ClusterSet

Following an emergency failover, and there is a risk of the transaction sets differing between parts of the ClusterSet, you have to fence the cluster either from write traffic or all traffic.

If a network partition happens, then there is the possibility of a split-brain situation, where instances lose synchronization and cannot communicate correctly to define the synchronization state. A split-brain can occur in situations such as when a DBA decides to forcibly elect a replica cluster to become the primary cluster creating more than one master, leading to the split-brain situation.

In this situation, a DBA can choose to fence the original primary cluster from:

- Writes.
- All traffic.

Three fencing operations are available:

- `<Cluster>.fenceWrites()`: Stops write traffic to a primary cluster of a ClusterSet. Replica clusters do not accept writes, so this operation has no effect on them.

  As of 8.0.31, it is possible to use on INVALIDATED Replica clusters. Also, if run against a Replica cluster with `super_read_only` disabled, it will enable it.

- `<Cluster>.unfenceWrites()`: Resumes write traffic. This operation can be run on a cluster that was previously fenced from write traffic using the `<Cluster>.fenceWrites()` operation.

  It is not possible to use *`cluster`*`.unfenceWrites()` on a Replica Cluster.

- `<Cluster>.fenceAllTraffic()`: Fences a cluster from all traffic. If you have fenced a cluster from all traffic using `<Cluster>.fenceAllTraffic()`, you have to reboot the cluster using the `dba.rebootClusterFromCompleteOutage()` MySQL Shell command.

  For more information on `dba.rebootClusterFromCompleteOutage()`, see Section 7.8.3, “Rebooting a Cluster from a Major Outage”.

fenceWrites()

Issuing `.fenceWrites()` on a replica cluster returns an error:

```
ERROR: Unable to fence Cluster from write traffic:
operation not permitted on REPLICA Clusters
Cluster.fenceWrites: The Cluster '<Cluster>' is a REPLICA Cluster
of the ClusterSet '<ClusterSet>' (MYSQLSH 51616)
```

Even though you primarily use fencing on clusters belonging to a clusterset, it is also possible to fence standalone clusters using `<Cluster>.fenceAllTraffic()`.

1. To fence a primary cluster from write traffic, use the Cluster.fenceWrites command as follows:

   ```
   <Cluster>.fenceWrites()
   ```

   After running the command:

   - The automatic `super_read_only` management is disabled on the cluster.
   - `super_read_only` is enabled on all the instances in the cluster.
   - All applications are blocked from performing writes on the cluster.

   ```
   cluster.fenceWrites()
       The Cluster 'primary' will be fenced from write traffic

     * Disabling automatic super_read_only management on the Cluster...
     * Enabling super_read_only on '127.0.0.1:3311'...
     * Enabling super_read_only on '127.0.0.1:3312'...
     * Enabling super_read_only on '127.0.0.1:3313'...

     NOTE: Applications will now be blocked from performing writes on Cluster 'primary'.
      Use <Cluster>.unfenceWrites() to resume writes if you are certain a split-brain is not in effect

     Cluster successfully fenced from write traffic
   ```

2. To check that you have fenced a primary cluster from write traffic, use the `<Cluster>.status` command as follows:

   ```
   <Cluster>.clusterset.status()
   ```

   The output is as follows:

   ```
   clusterset.status()
           {
           "clusters": {
           "primary": {
           "clusterErrors": [
           "WARNING: Cluster is fenced from Write traffic.
            Use cluster.unfenceWrites() to unfence the Cluster."
           ],
           "clusterRole": "PRIMARY",
           "globalStatus": "OK_FENCED_WRITES",
   ```

```
        "primary": null,
        "status": "FENCED_WRITES",
        "statusText": "Cluster is fenced from Write Traffic."
        },
        "replica": {
        "clusterRole": "REPLICA",
        "clusterSetReplicationStatus": "OK",
        "globalStatus": "OK"
        }
        },
        "domainName": "primary",
        "globalPrimaryInstance": null,
        "primaryCluster": "primary",
        "status": "UNAVAILABLE",
        "statusText": "Primary Cluster is fenced from write traffic."
```

3. To unfence a cluster and resume write traffic to a primary cluster, use the Cluster.fenceWrites command as follows:

```
        <Cluster>.unfenceWrites()
```

   The automatic `super_read_only` management on the primary cluster is enabled, and the `super_read_only` status on the primary cluster instance.

```
        cluster.unfenceWrites()
        The Cluster 'primary' will be unfenced from write traffic

        * Enabling automatic super_read_only management on the Cluster...
        * Disabling super_read_only on the primary '127.0.0.1:3311'...

        Cluster successfully unfenced from write traffic
```

4. To fence a cluster from all traffic, use the Cluster.fenceAllTraffic command as follows:

```
      <Cluster>.fenceAllTraffic()
```

   The `super_read_only` status is enabled on the primary instance of the cluster instance. Before enabling `offline_mode` on all the instances in the cluster:

```
      cluster.fenceAllTraffic()
        The Cluster 'primary' will be fenced from all traffic

        * Enabling super_read_only on the primary '127.0.0.1:3311'...
        * Enabling offline_mode on the primary '127.0.0.1:3311'...
        * Enabling offline_mode on '127.0.0.1:3312'...
        * Stopping Group Replication on '127.0.0.1:3312'...
        * Enabling offline_mode on '127.0.0.1:3313'...
        * Stopping Group Replication on '127.0.0.1:3313'...
        * Stopping Group Replication on the primary '127.0.0.1:3311'...

        Cluster successfully fenced from all traffic
```

5. To unfence a cluster from all traffic, use the `dba.rebootClusterFromCompleteOutage()` MySQL Shell command. When you have restored the cluster, you rejoin the instances to the cluster by selecting **Y** when asked if you want to rejoin the instance to the cluster:

```
cluster = dba.rebootClusterFromCompleteOutage()
  Restoring the cluster 'primary' from complete outage...

  The instance '127.0.0.1:3312' was part of the cluster configuration.
  Would you like to rejoin it to the cluster? [y/N]: Y

  The instance '127.0.0.1:3313' was part of the cluster configuration.
  Would you like to rejoin it to the cluster? [y/N]: Y

  * Waiting for seed instance to become ONLINE...
```

```
127.0.0.1:3311 was restored.
Rejoining '127.0.0.1:3312' to the cluster.
Rejoining instance '127.0.0.1:3312' to cluster 'primary'...

The instance '127.0.0.1:3312' was successfully rejoined to the cluster.

Rejoining '127.0.0.1:3313' to the cluster.
Rejoining instance '127.0.0.1:3313' to cluster 'primary'...

The instance '127.0.0.1:3313' was successfully rejoined to the cluster.

The cluster was successfully rebooted.

<Cluster:primary>
```

8.9.2 Inconsistent Transaction Sets (GTID Sets) in InnoDB ClusterSet Clusters

AdminAPI's *`clusterSet`*`.status()` command warns you if an InnoDB Cluster's GTID set is inconsistent with the GTID set on the primary cluster in the InnoDB ClusterSet. A cluster in this state has extra transactions compared to the other clusters in the InnoDB ClusterSet, and has the global status `OK_NOT_CONSISTENT`. The cluster continues to function in the InnoDB ClusterSet with this status, and you can carry out an emergency failover to it if its GTID set is the most up to date of the available replica clusters. However, it is not eligible for a controlled switchover, because the difference in transactions might result in clients accessing incorrect data. The cluster also cannot rejoin the InnoDB ClusterSet with extra transactions if it goes offline.

A replica cluster in an InnoDB ClusterSet is read-only, so if it has always been a replica cluster, it should not contain extra transactions unless changes were made on the cluster without using AdminAPI commands. If you need to carry out administrative transactions on an instance while Group Replication is stopped, always set the value of the `sql_log_bin` system variable to `OFF` before issuing administrative statements, and back to `ON` afterwards:

```
SET SQL_LOG_BIN=0;
<administrator action>
SET SQL_LOG_BIN=1;
```

Setting this system variable to `OFF` means that the transactions that occur from that point until you set it back to `ON` are not written to the binary log and do not have GTIDs assigned to them.

A situation that can create a diverged set of transactions with no outside changes is when the primary cluster becomes unreachable and an emergency failover procedure is used. If the primary cluster remains online after the failover, it could continue to accept transactions from clients through any MySQL Router instances that are still connected to it, and pass these to any replica clusters that are still connected to it. Alternatively, significant replication lag might cause the replica cluster selected as the replacement primary cluster to be missing some transactions from the primary cluster. In that case, when the old primary cluster initially comes back online as an invalidated replica cluster, the transactions that were never transferred to the replica are identified as extra transactions.

The extended output for the *`clusterSet`*`.status()` command identifies any clusters that have extra transactions, and assigns them the `OK_NOT_CONSISTENT` global status. For example:

```
mysql-js> myclusterset.status({extended: 1})
{
    "clusters": {
        "clusterone": {
            "clusterErrors": [
                "ERROR: Errant transactions detected"
            ],
            "clusterRole": "REPLICA",
            "clusterSetReplication": {
                "applierStatus": "APPLIED_ALL",
                "applierThreadState": "Waiting for an event from Coordinator",
                "applierWorkerThreads": 4,
                "receiver": "127.0.0.1:3310",
                "receiverStatus": "ON",
```

```
                "receiverThreadState": "Waiting for source to send event",
                "source": "127.0.0.1:4410"
            },
            "clusterSetReplicationStatus": "OK",
            "globalStatus": "OK_NOT_CONSISTENT",
            "status": "OK",
            "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
            "topology": {
                "127.0.0.1:3310": {
                    "address": "127.0.0.1:3310",
                    "memberRole": "PRIMARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:3320": {
                    "address": "127.0.0.1:3320",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:3330": {
                    "address": "127.0.0.1:3330",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                }
            },
            "transactionSet": "54ff337b-2ccf-11ec-95da-3c6aa7197deb:1-131,54ff3ed7-2ccf-11ec-95da-3c6aa7197
            "transactionSetConsistencyStatus": "INCONSISTENT",
            "transactionSetConsistencyStatusText": "There are 1 transactions that were executed in this ins
            "transactionSetErrantGtidSet": "c06527d6-2ce3-11ec-a55e-3c6aa7197deb:1",
            "transactionSetMissingGtidSet": ""
        },
        "clustertwo": {
            "clusterRole": "PRIMARY",
            "globalStatus": "OK",
            "primary": "127.0.0.1:4410",
            "status": "OK",
            "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
            "topology": {
                "127.0.0.1:4410": {
                    "address": "127.0.0.1:4410",
                    "memberRole": "PRIMARY",
                    "mode": "R/W",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:4420": {
                    "address": "127.0.0.1:4420",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.27"
                },
                "127.0.0.1:4430": {
                    "address": "127.0.0.1:4430",
                    "memberRole": "SECONDARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
```

```
                    "version": "8.0.27"
                }
            },
            "transactionSet": "54ff337b-2ccf-11ec-95da-3c6aa7197deb:1-131,54ff3ed7-2ccf-11ec-95da-3c6aa
        }
    },
    "domainName": "testclusterset",
    "globalPrimaryInstance": "127.0.0.1:4410",
    "metadataServer": "127.0.0.1:4410",
    "primaryCluster": "clustertwo",
    "status": "AVAILABLE",
    "statusText": "Primary Cluster available, there are issues with a Replica cluster."
}
```


The safest method to reconcile an individual server's data with the rest of the InnoDB Cluster is to identify the server in the InnoDB ClusterSet deployment that has the best data (the most transactions, the most recent transactions, or the most important transactions) and use MySQL's cloning functionality to transfer the content from that server to the affected server. For instructions to do this, see Cloning Remote Data. Then use the *`cluster`*`.rejoinInstance()` command to have the instance rejoin the InnoDB Cluster. For details of this operation, see Section 7.8.1, “Rejoining an Instance to a Cluster”.

If the whole InnoDB Cluster is affected, remove the affected cluster from the InnoDB ClusterSet deployment following the procedure in Section 8.9.4, “Removing a Cluster from an InnoDB ClusterSet”, and set up a new InnoDB Cluster in its place. The server instances in the new InnoDB Cluster will receive the correct transaction set as part of the setup process.

If you want to keep the extra transactions, an emergency failover can be carried out to make the InnoDB Cluster with those transactions into the primary cluster, following the procedure in Section 8.8, “InnoDB ClusterSet Emergency Failover”.

If you are able to deal with the problem transactions, use a *`clusterSet`*`.rejoinCluster()` operation to rejoin the InnoDB Cluster to the InnoDB ClusterSet deployment. For instructions to do that, see Section 8.9.5, “Rejoining a Cluster to an InnoDB ClusterSet”.

8.9.3 Repairing Member Servers and Clusters in an InnoDB ClusterSet

Depending on the issues or maintenance requirements for the cluster, the following operations are available for you to work with its member servers. Unless otherwise stated, use `Cluster` and `ClusterSet` objects that you fetched with an InnoDB Cluster administrator account or server configuration account, so that the default user account stored in the `ClusterSet` object has the correct permissions.

- Add further server instances to the cluster, using the *`cluster`*`.addInstance()` command, as described in the procedure at Section 8.4, “Deploying InnoDB ClusterSet”. For more details of the command, see Section 7.4.4, “Adding Instances to an InnoDB Cluster”.

  Note that for this operation, you need to use the InnoDB Cluster server configuration account and a `Cluster` object that was fetched using that account. The account must also exist on the server instance, as explained in Section 8.3, “User Accounts for InnoDB ClusterSet”.

  When you use this command to add a member server to an InnoDB Cluster that is part of an InnoDB ClusterSet deployment, the server instance is added to the cluster and provisioned with the data for the InnoDB ClusterSet. The ClusterSet replication channel is set up on the instance, and the required configuration to operate in an InnoDB ClusterSet deployment is applied.

- Rejoin a server instance that was previously part of the cluster but could not automatically rejoin the cluster, using the *`cluster`*`.rejoinInstance()` command. For details of this operation, see Section 7.8.1, “Rejoining an Instance to a Cluster”.

  When you use this command to rejoin a member server to an InnoDB Cluster that is part of an InnoDB ClusterSet deployment, the server instance is rejoined to the cluster and provisioned with the data for the InnoDB ClusterSet. The ClusterSet replication channel is set up on the instance, and the required configuration to operate in an InnoDB ClusterSet deployment is applied.

- Remove a server instance from the cluster, using the `cluster.removeInstance()` command. Specify the host name and port number of the server instance that is to be removed. For details of this operation, see Removing Instances from an InnoDB Cluster. A `force` option is available, but this should only be used as a last resort.

  When you use this command with a member server in an InnoDB ClusterSet deployment, MySQL Shell resets all configuration that was applied for InnoDB ClusterSet and resets the ClusterSet replication channel settings.

- Change the primary of the cluster to another member server, using the `cluster.setPrimaryInstance(instance)` command. Changing the primary allows you to carry out maintenance and upgrades on the current primary server, or to select a primary if Group Replication's own election process does not automatically elect the primary server that you want.

  Specify the host name and port number of the server instance that is to be the primary. From MySQL Shell 8.0.29, you can use the `runningTransactionsTimeout` option to specify a timeout between 0 and 3600 seconds for transactions that are running when you use the function, which also stops new incoming transactions. There is no default setting for the timeout, so if you do not set it, there is no upper limit to the wait time for the operation, and new transactions can start during that time.

  When you use this command with a member server in an InnoDB ClusterSet deployment, MySQL Shell stops the ClusterSet replication channel on the server beforehand, and restarts it afterwards. Also, if the cluster is a replica cluster, MySQL Shell keeps the primary as read-only, rather than making it read-write as would normally be the case with the primary of an InnoDB Cluster.

- Restore a cluster that has lost quorum by forcing quorum with the remaining instances, using the `cluster.forceQuorumUsingPartitionOf(instance)` command. Specify the host name and port number of an online server instance with the correct metadata. The operation makes the cluster consist of this and the other reachable instances, and excludes the partitioned instances. This operation can create a split-brain scenario, so it should be considered a last resort. For details of this operation, see Section 7.8.2, “Restoring a Cluster from Quorum Loss”.

  When you use this command with a member server in an InnoDB ClusterSet deployment, MySQL Shell checks whether the target cluster is still a valid part of the ClusterSet, and warns you if it has been invalidated. It also automatically restarts the ClusterSet replication channel afterwards. If the cluster is a replica cluster, MySQL Shell keeps the primary as read-only, rather than making it read-write as would normally be the case with the primary of an InnoDB Cluster.

- Reboot a cluster that is completely offline, using the `dba.rebootClusterFromCompleteOutage()` command. For details of this operation, see Section 7.8.3, “Rebooting a Cluster from a Major Outage”.

  When you use this command with a member server in an InnoDB ClusterSet deployment, MySQL Shell checks whether the target cluster is still a valid part of the ClusterSet, and warns you if it has been invalidated.

  If the cluster was not invalidated, MySQL Shell rejoins it to the InnoDB ClusterSet deployment immediately after the reboot. If the cluster was invalidated, you must use a `clusterSet.rejoinCluster()` operation to rejoin it to the InnoDB ClusterSet deployment. For instructions to do that, see Section 8.9.5, “Rejoining a Cluster to an InnoDB ClusterSet”.

  MySQL Shell also automatically restarts the ClusterSet replication channel after this operation. If the cluster is a replica cluster, MySQL Shell keeps the primary as read-only, rather than making it read-write as would normally be the case with the primary of an InnoDB Cluster.

You cannot dissolve an InnoDB Cluster that is currently part of an InnoDB ClusterSet deployment unless it is the only cluster in the ClusterSet or the cluster is invalidated. In all other configurations, you must remove it from the InnoDB ClusterSet as described in Section 8.9.4, “Removing a Cluster from an InnoDB ClusterSet”.

As of MySQL Shell 8.0.32, if the cluster is the only cluster in the ClusterSet or the cluster is invalidated, you can use `dba.dropMetadataSchema()` or *`cluster`*`.dissolve()` on the cluster.

8.9.4 Removing a Cluster from an InnoDB ClusterSet

If you cannot repair a cluster, you can remove it from the InnoDB ClusterSet using a *`clusterSet`*`.removeCluster()` command. A `force` option is available if the cluster cannot be contacted at all.

Important

The primary cluster in an InnoDB ClusterSet cannot be removed using this command. If you do need to remove the primary cluster, you must first carry out a controlled switchover (see Section 8.7, “InnoDB ClusterSet Controlled Switchover”) or an emergency failover (see Section 8.8, “InnoDB ClusterSet Emergency Failover”) to demote the primary cluster to a replica cluster, and promote one of the replica clusters to be the primary cluster. After that, the former primary cluster can be removed using this procedure.

A removed InnoDB Cluster cannot be added back into an InnoDB ClusterSet deployment. If you want to use the server instances in the deployment again, you will need to set up a new cluster using them.

To remove a cluster from the InnoDB ClusterSet, follow this procedure:

1. Using MySQL Shell, connect to any member server in the primary cluster or in one of the replica clusters, using an InnoDB Cluster administrator account (created with *`cluster`*`.setupAdminAccount()`). You may also use the InnoDB Cluster server configuration account, which also has the required permissions. When the connection is established, get the `ClusterSet` object using `dba.getClusterSet()` or *`cluster`*`.getClusterSet()` command. It is important to use an InnoDB Cluster administrator account or server configuration account so that the default user account stored in the `ClusterSet` object has the correct permissions. For example:

   ```
   mysql-js> \connect admin2@127.0.0.1:4410
   Creating a session to 'admin2@127.0.0.1:4410'
   Please provide the password for 'admin2@127.0.0.1:4410': ********
   Save password for 'admin2@127.0.0.1:4410'? [Y]es/[N]o/Ne[v]er (default No):
   Fetching schema names for autocompletion... Press ^C to stop.
   Closing old connection...
   Your MySQL connection id is 33
   Server version: 8.0.27-commercial MySQL Enterprise Server - Commercial
   No default schema selected; type \use <schema> to set one.
   <ClassicSession:admin2@127.0.0.1:4410>
   mysql-js> myclusterset = dba.getClusterSet()
   <ClusterSet:testclusterset>
   ```

2. Check the status of the whole deployment using AdminAPI's *`clusterSet`*`.status()` function in MySQL Shell. For example:

   ```
   mysql-js> myclusterset.status({extended: 1})
   ```

   For an explanation of the output, see Section 8.6, “InnoDB ClusterSet Status and Topology”.

   When you issue the *`clusterSet`*`.removeCluster()` command, there must be an active and reachable primary cluster in the InnoDB ClusterSet deployment, and this must not be the cluster you are removing. The cluster you are removing must currently have the status of a replica cluster. It can be invalidated, and does not have to be reachable.

3. Check the routing options that are set for each MySQL Router instance, and the global policy for the InnoDB ClusterSet deployment, by issuing *`clusterSet`*`.routingOptions()` in MySQL Shell while connected to any member server in the InnoDB ClusterSet deployment. Verify that no MySQL Router instances are routing traffic to the cluster that you are going to

remove. If any are, you must change their settings to route traffic to another cluster using a *clusterSet*.setRoutingOption() command, as described in Section 8.5, “Integrating MySQL Router With InnoDB ClusterSet”. A cluster cannot be removed if any MySQL Router instances known to the InnoDB ClusterSet deployment are routing traffic to it.

4. Issue a *clusterSet*.removeCluster() command, naming the cluster that you want to remove from the InnoDB ClusterSet. For example:

```
mysql-js> myclusterset.removeCluster('clusterone')
The Cluster 'clusterone' will be removed from the InnoDB ClusterSet.

* Waiting for the Cluster to synchronize with the PRIMARY Cluster...
** Transactions replicated  ############################################################  100%
* Updating topology
** Transactions replicated  ############################################################  100%
* Stopping and deleting ClusterSet managed replication channel...

The Cluster 'clusterone' was removed from the ClusterSet.
```

   - The *clusterName* parameter is required and specifies the identifier used for the cluster in the InnoDB ClusterSet, as given in the output from the *clusterSet*.status() command. In the example, *clusterone* is the cluster that is to be removed.
   - Use the dryRun option if you want to carry out validations and log the changes without actually executing them.
   - Use the timeout option to specify the maximum number of seconds to wait for the cluster to synchronize with the primary cluster in the InnoDB ClusterSet.
   - Use the force option to remove the cluster from the ClusterSet when the cluster's primary instance is not reachable.

   When you issue the *clusterSet*.removeCluster() command, MySQL Shell checks that the primary cluster in the InnoDB ClusterSet deployment is reachable, that the target cluster is not the primary cluster, and that no MySQL Router instances are routing traffic to the target cluster. If any of these conditions are not met, an error is returned. If they are met, MySQL Shell carries out the following tasks to remove the target cluster from the InnoDB ClusterSet:

   - Drops the replication user that was created for the ClusterSet replication channel on the target cluster.
   - Synchronizes the primary server of the target cluster with the primary cluster of the InnoDB ClusterSet, and waits for all transactions to be applied locally. If the timeout expires before this is completed, the operation fails. If synchronization does not work, try again with the force option.
   - Stops the ClusterSet replication channel, then removes the channel and resets its configuration to the default values.
   - Removes the target cluster's metadata and member information from the InnoDB ClusterSet metadata.
   - Leaves the super_read_only system variable set on all the member servers, to ensure that no updates are performed on them.

5. Issue a *clusterSet*.status() command again using the extended option, to verify the status of the InnoDB ClusterSet deployment.

6. A removed InnoDB Cluster cannot be added back into an InnoDB ClusterSet deployment, so if you want to use the server instances in the deployment again, you will need to set up a new cluster using the standalone instances. From MySQL Shell 8.0.28, the InnoDB Cluster is implicitly dissolved during the removal process, so that all the members become standalone instances.

In MySQL Shell 8.0.27, dissolve the cluster manually following the instructions in Dissolving an InnoDB Cluster.

Note that the Group Replication configuration is not removed from the server instances, so you should exercise caution when reusing these in an InnoDB ClusterSet deployment, as explained in Section 8.1, “InnoDB ClusterSet Requirements”. As the instances were configured for an InnoDB ClusterSet deployment, the possibility of issues is lower, but you should be aware of the potential for configuration differences especially if the instances are reused in a different InnoDB ClusterSet deployment.

8.9.5 Rejoining a Cluster to an InnoDB ClusterSet

If an InnoDB Cluster is part of an InnoDB ClusterSet deployment, MySQL Shell automatically restores it to its role in the topology immediately after a reboot, provided that it is functioning acceptably and has not been marked as invalidated. However, if a cluster has been marked as invalidated or its ClusterSet replication channel has stopped, you must use a `clusterSet.rejoinCluster()` operation to rejoin it to the InnoDB ClusterSet deployment.

The `clusterSet.rejoinCluster()` operation verifies that the target cluster meets these requirements:

- The cluster has previously been a member of the ClusterSet.
- The cluster has quorum (sufficient members are online to form a majority).
- The cluster's primary server is reachable.
- The cluster is not holding any metadata locks or InnoDB transaction locks.
- The cluster's GTID set (`gtid_executed`) contains no extra transactions compared to the active members of the ClusterSet, with the exception of view change events. These Group Replication internal transactions are identified by the UUID specified by the `group_replication_view_change_uuid` system variable, and the cluster rejoin process can reconcile them.

If the cluster meets these requirements, the operation restarts the ClusterSet replication channel and removes the `INVALIDATED` status. If it does not, you will need to fix any issues that were identified and retry the command.

Follow this procedure to rejoin an InnoDB Cluster to the InnoDB ClusterSet:

1. Using MySQL Shell, connect to any member server in the primary cluster or in one of the replica clusters, using an InnoDB Cluster administrator account (created with `cluster.setupAdminAccount()`). You may also use the InnoDB Cluster server configuration account, which also has the required permissions. When the connection is established, get the `ClusterSet` object using `dba.getClusterSet()` or `cluster.getClusterSet()` command. It is important to use an InnoDB Cluster administrator account or server configuration account so that the default user account stored in the `ClusterSet` object has the correct permissions. For example:

```
mysql-js> \connect admin2@127.0.0.1:3310
Creating a session to 'admin2@127.0.0.1:3310'
Please provide the password for 'admin2@127.0.0.1:3310': ********
Save password for 'admin2@127.0.0.1:3310'? [Y]es/[N]o/Ne[v]er (default No):
Fetching schema names for autocompletion... Press ^C to stop.
Closing old connection...
Your MySQL connection id is 28
Server version: 8.0.27-commercial MySQL Enterprise Server - Commercial
No default schema selected; type \use <schema> to set one.
<ClassicSession:admin2@127.0.0.1:3310>
mysql-js> myclusterset = dba.getClusterSet()
<ClusterSet:testclusterset>
```

2. Check the status of the whole deployment using AdminAPI's *clusterSet*.status() function in MySQL Shell. For example:

```
mysql-js> myclusterset.status({extended: 1})
```

   For an explanation of the output, see Section 8.6, “InnoDB ClusterSet Status and Topology”.

3. Issue a *clusterSet*.rejoinCluster() command, naming the cluster that you want to rejoin to the InnoDB ClusterSet. For example:

```
mysql-js> myclusterset.rejoinCluster('clustertwo')
Rejoining cluster 'clustertwo' to the clusterset
NOTE: Cluster 'clustertwo' is invalidated
* Updating metadata

* Rejoining cluster
** Changing replication source of 127.0.0.1:4420 to 127.0.0.1:3310
** Changing replication source of 127.0.0.1:4430 to 127.0.0.1:3310
** Changing replication source of 127.0.0.1:4410 to 127.0.0.1:3310

Cluster 'clustertwo' was rejoined to the clusterset
```

   For the *clusterSet*.rejoinCluster() command:

   - The *clusterName* parameter is required and specifies the identifier used for the cluster in the InnoDB ClusterSet, as given in the output from the *clusterSet*.status() command. In the example, *clustertwo* is the name of the cluster that is being rejoined.
   - Use the dryRun option if you want to carry out validations and log the changes without actually executing them.

   When you issue the *clusterSet*.rejoinCluster() command, MySQL Shell checks that the target cluster meets the requirements to rejoin the ClusterSet, and returns an error if it does not. If the target cluster meets the requirements, MySQL Shell carries out the following tasks:

   - Checks whether the ClusterSet replication channel is replicating from the current primary cluster, and reconfigures it to do that if it isn't already.
   - Resets the replication user's password.
   - Restarts the ClusterSet replication channel.
   - Clears the INVALIDATED status for the cluster.

   The target cluster rejoins the InnoDB ClusterSet as a replica cluster, even if it was previously a primary cluster. A controlled switchover is required if you want to make the target cluster into the primary cluster.

   Note that if the target cluster has members that are not online or not reachable when you issue the *clusterSet*.rejoinCluster() command, these members are not correctly configured by the command. If you no longer require these instances, you can remove them using the *cluster*.removeInstance() command. If you repair these instances or bring them online again, issue the *clusterSet*.rejoinCluster() command again after those members return to the cluster.

4. Issue a *clusterSet*.status() command again using the extended option, to verify the status of the InnoDB ClusterSet deployment.

5. If you do want to make the rejoined cluster into the primary cluster, issue a *clusterSet*.setPrimaryCluster() command, naming the rejoined cluster. Section 8.7, “InnoDB ClusterSet Controlled Switchover” has instructions for the procedure, including how to direct MySQL Router instances to send traffic to the new primary cluster.

8.10 Upgrade InnoDB ClusterSet

To upgrade the server instances in an InnoDB ClusterSet, complete the following steps:

1. Upgrade MySQL Router.
2. Upgrade MySQL Shell.
3. Upgrade MySQL Server:
4. Post Upgrade Status Check.

Check the versions of the installed binaries:

- **mysqlrouter --version**: Checks the version of MySQL Router installed.
- **mysqlsh --version**: Checks the version of MySQL Shell installed.
- **mysqld --version**: Checks the version of MySQL Server installed.

Upgrade MySQL Router.

To upgrade MySQL Router, complete the following steps:

1. Stop MySQL Router.

   On a Unix system, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected when you bootstrapped the router. These files include `stop.sh`. Navigate to this directory and issue this command:

   ```
   ./stop.sh
   ```

   On Microsoft Windows, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected when you bootstrapped the router. These files include `stop.ps1`. Navigate to this directory and issue this command:

   ```
   .\stop.ps1
   ```

   Or on a Linux system using `systemd`, stop the MySQL Router service by issuing:

   ```
   systemctl stop mysqlrouter.service
   ```

   Otherwise, kill the *process ID (PID)* of the associated *mysqlrouter* process.

2. Obtain and install the latest version of MySQL Router.
3. Start MySQL Router.

   On a Unix system, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected. These files include `start.sh`. Navigate to the directory and issue this command:

   ```
   ./start.sh
   ```

   If the path to the new router has changed, you must update the `start.sh` shell script to reflect the path.

   ```
   #!/bin/bash
   basedir=/tmp/myrouter
   ROUTER_PID=$basedir/mysqlrouter.pid /usr/bin/mysqlrouter -c $basedir/mysqlrouter.conf &
   disown %-
   ```

If you upgrade MySQL Router manually, opposed to using package management, you can update the `basedir=`. Bootstrapping the router again also regenerates the `start.sh` shell script.

Or on a Linux system using `systemd`, start the MySQL Router service by issuing:

```
systemctl start mysqlrouter.service
```

On Microsoft Windows, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected. These files include `start.ps1`. Navigate to the directory and issue this command:

```
.\start.ps1
```

On starting MySQL Router using the new router binaries, the version of the router is upgraded:

```
mysqlrouter --version
```

Upgrade MySQL Shell

Upgrade MySQL Shell by installing the new binaries, and stopping and starting MySQL Shell:

1. Obtain and install the latest version of MySQL Shell.
2. Stop and quit MySQL Shell by issuing:

   ```
   \q
   ```

3. Restart MySQL Shell from the command line, by issuing:

   ```
   mysqlsh
   ```

4. Upgrade the InnoDB ClusterSet Metadata:

   - To upgrade a ClusterSet, connect MySQL Shell's global session to your ClusterSet and use the `dba.upgradeMetadata()` operation to upgrade the ClusterSet's metadata to the new metadata.

     If a ClusterSet's metadata schema needs to be upgraded, then the upgrade process itself should be performed in the ClusterSet's primary Cluster.

Metadata Upgrade

The Metadata Upgrade may do nothing if the InnoDB ClusterSet already uses the latest version.

Upgrade MySQL Server

Upgrade MySQL Server by upgrading all the replica clusters' instances first and then upgrading the primary cluster's instances.

When upgrading each cluster, primary or replica clusters, upgrade all secondary instances before upgrading the primary instance.

Upgrading MySQL Server is optional

Upgrading MySQL Server is optional. Server upgrades can have a greater impact than upgrading MySQL Shell and MySQL Router. Also, you should always keep MySQL Shell and MySQL Router at the latest version, even if the server is not; this is true for InnoDB Clusters and ReplicaSets.

For details on upgrading with Group Replication, see Upgrading a Group Replication Member.

1. Stop MySQL Server by issuing one of the following commands:

- If MySQL Server is using *systemd* issue:

```
systemctl stop mysqld
```

- If MySQL Server is using *init.d* issue:

```
/etc/init.d/mysql stop
```

- If MySQL Server is using *service* issue:

```
service mysql stop
```

- If you deployed MySQL Server on Microsoft Windows issue:

```
mysqladmin -u root -p shutdown
```

2. Obtain and install the latest version of MySQL Server.
3. Start MySQL Server by issuing one of the following commands:

- If MySQL Server is using *systemd* issue:

```
systemctl start mysqld
```

- If MySQL Server is using *init.d* issue:

```
/etc/init.d/mysql start
```

- If MySQL Server is using *service* issue:

```
service mysql start
```

- If you deployed MySQL Server on Microsoft Windows issue:

```
mysqld
```

4. When all the secondary instances are upgraded, upgrade the primary instance to complete the upgrade process.

Post Upgrade Status Check

After upgrading MySQL Router, MySQL Shell, and MySQL Servers are upgraded:

1. Check the status of the ClusterSet by issuing `<ClusterSet>.status()`. For more information about `<ClusterSet>.status()`, see Section 8.6, “InnoDB ClusterSet Status and Topology”.
2. Resolve any `clusterErrors` and `statusText` returned by the `<ClusterSet>.status()` operation.
3. Check each Cluster in the ClusterSet by issuing `<Cluster>.status()` and resolve any issues. For more information about `<Cluster>.status()`, see Checking a cluster's Status with `Cluster.status()`.
4. Check the details of all the registered MySQL Router instances by issuing `<ClusterSet>.listRouters()`. For more information, see Integrating MySQL Router With InnoDB ClusterSet.

These commands allow you to check that the upgrade has been successful or if you need to complete any additional steps.

Note

The additional steps depend; on how many versions you are skipping, what version you are upgrading, and from what version you are coming.

1. Begin your post upgrade check by checking the status of the InnoDB ClusterSet. This check uses the `<ClusterSet>.status({extended:1}) operation`.

   In this example, we issue `<ClusterSet>.status({extended:1})`:

```
 mysql-js><ClusterSet>.status({extended:1})
{
    "clusters": {
        "cluster1": {
            "clusterRole": "PRIMARY",
            "globalStatus": "OK",
            "primary": "127.0.0.1:3310",
            "status": "OK_NO_TOLERANCE",
            "statusText": "Cluster is NOT tolerant to any failures.",
            "topology": {
                "127.0.0.1:3310": {
                    "address": "127.0.0.1:3310",
                    "memberRole": "PRIMARY",
                    "mode": "R/W",
                    "status": "ONLINE",
                    "version": "8.0.28"
                }
            },
            "transactionSet": "c0361cad-9093-11ec-94ce-0a0027000010:1-90,c0362acf-9093-11ec-94ce-0a0027
        },
        "replicacluster1": {
            "clusterErrors": [
                "ERROR: Cluster members are reachable but they're all OFFLINE.",
                "WARNING: Replication from the Primary Cluster not in expected state"
            ],
            "clusterRole": "REPLICA",
            "clusterSetReplication": {
                "applierStatus": "OFF",
                "applierThreadState": "",
                "applierWorkerThreads": 4,
                "receiver": "127.0.0.1:3320",
                "receiverStatus": "OFF",
                "receiverThreadState": "",
                "source": "127.0.0.1:3310"
            },
            "clusterSetReplicationStatus": "STOPPED",
            "globalStatus": "NOT_OK",
            "status": "OFFLINE",
            "statusText": "All members of the group are OFFLINE",
            "topology": {
                "127.0.0.1:3320": {
                    "address": "127.0.0.1:3320",
                    "instanceErrors": [
                        "NOTE: group_replication is stopped."
                    ],
                    "memberRole": "SECONDARY",
                    "memberState": "OFFLINE",
                    "mode": "R/O",
                    "status": "(MISSING)",
                    "version": "8.0.28"
                }
            },
            "transactionSet": "1ec95a0b-9094-11ec-9bc5-0a0027000010:1,c0361cad-9093-11ec-94ce
                               -0a0027000010:1-90,c0362acf-9093-11ec-94ce-0a0027000010:1",
            "transactionSetConsistencyStatus": "OK",
            "transactionSetErrantGtidSet": "",
            "transactionSetMissingGtidSet": ""
        },
        "replicacluster2": {
            "clusterRole": "REPLICA",
            "clusterSetReplication": {
                "applierStatus": "APPLIED_ALL",
                "applierThreadState": "Waiting for an event from Coordinator",
                "applierWorkerThreads": 4,
                "receiver": "127.0.0.1:3330",
                "receiverStatus": "ON",
```

```
                "receiverThreadState": "Waiting for source to send event",
                "source": "127.0.0.1:3310"
            },
            "clusterSetReplicationStatus": "OK",
            "globalStatus": "OK",
            "status": "OK_NO_TOLERANCE",
            "statusText": "Cluster is NOT tolerant to any failures.",
            "topology": {
                "127.0.0.1:3330": {
                    "address": "127.0.0.1:3330",
                    "memberRole": "PRIMARY",
                    "mode": "R/O",
                    "replicationLagFromImmediateSource": "",
                    "replicationLagFromOriginalSource": "",
                    "status": "ONLINE",
                    "version": "8.0.28"
                }
            },
            "transactionSet": "329dc243-9094-11ec-b9dd-0a0027000010:1,c0361cad-9093-11ec
                               -94ce-0a0027000010:1-90,c0362acf-9093-11ec-94ce-0a0027000010:1",
            "transactionSetConsistencyStatus": "OK",
            "transactionSetErrantGtidSet": "",
            "transactionSetMissingGtidSet": ""
        }
    },
    "domainName": "clusterset1",
    "globalPrimaryInstance": "127.0.0.1:3310",
    "metadataServer": "127.0.0.1:3310",
    "primaryCluster": "cluster1",
    "status": "AVAILABLE",
    "statusText": "Primary Cluster available, there are issues with a Replica cluster."
}
```

For more information about the `<ClusterSet>.status()` operation, see ClusterSet.status().

2. Resolve any errors returned by the `<ClusterSet>.status({extended:1})` operation.

   In this example, we have an error returned in `clusterErrors` informing us that `<ClusterSet>.status({extended:1})` operation was unable to connect to any online members, and with the `statusText` that the Primary Cluster is available, but there are issues with a replica cluster in the InnoDB ClusterSet.

```
  ...
  "replicacluster1": {
    "clusterErrors": [
                "ERROR: Could not connect to any ONLINE members but there are unreachable instance
                        that could still be ONLINE."
            ],
  ...
   "statusText": "Primary Cluster available, there are issues with a Replica cluster."
}
```

   In this example, we need to check the status of the InnoDB Cluster `replicacluster1` and ensure it is brought back online.

3. Once we have resolved the issues returned by the `<ClusterSet>.status({extended:1})` operation, we check the status of each InnoDB Cluster in the ClusterSet.

   Check the status of each InnoDB Cluster, by issuing `<Cluster>.status()`.

   In the following example, `<Cluster>.status({extended: true})`, used to provide more detailed information about the status of the InnoDB Cluster, returns two issues:

```
    mysqlsh> cluster.status({extended: true});
    {
        "clusterName": "MyCluster",
        "defaultReplicaSet": {
            "GRProtocolVersion": "8.0.16",
```

```
            "groupName": "459ec434-8926-11ec-b8c3-02001707f44a",
            "groupViewChangeUuid": "AUTOMATIC",
            "groupViewId": "16443558036060755:13",
            "name": "default",
            "ssl": "REQUIRED",
            "status": "OK",
            "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
            "topology": {
                "example-el7-1644251369:33311": {
                    "address": "example-el7-1644251369:33311",
                    "applierWorkerThreads": 4,
                    "fenceSysVars": [],
                    "instanceErrors": [
                        "NOTE: instance server_id is not registered in the metadata.
                               Use cluster.rescan() to update the metadata.",
                        "NOTE: The required parallel-appliers settings are not enabled on the instance.
                               Use dba.configureInstance() to fix it."
                    ],
                    "memberId": "247131ab-8926-11ec-850b-02001707f44a",
                    "memberRole": "PRIMARY",
                    "memberState": "ONLINE",
                    "mode": "R/W",
                    "readReplicas": {},
                    "replicationLag": null,
                    "role": "HA",
                    "status": "ONLINE",
                    "version": "8.0.28"
                },
                "example-el7-1644251369:33314": {
                    "address": "example-el7-1644251369:33314",
                    "applierWorkerThreads": 4,
                    "fenceSysVars": [],
                    "instanceErrors": [
                        "NOTE: instance server_id is not registered in the metadata.
                               Use cluster.rescan() to update the metadata.",
                        "NOTE: The required parallel-appliers settings are not enabled on the instance.
                               Use dba.configureInstance() to fix it."
                    ],
                    "memberId": "303dcfa7-8926-11ec-a6e5-02001707f44a",
                    "memberRole": "PRIMARY",
                    "memberState": "ONLINE",
                    "mode": "R/W",
                    "readReplicas": {},
                    "replicationLag": null,
                    "role": "HA",
                    "status": "ONLINE",
                    "version": "8.0.28"
                },
                "example-el7-1644251369:33317": {
                    "address": "example-el7-1644251369:33317",
                    "applierWorkerThreads": 4,
                    "fenceSysVars": [],
                    "instanceErrors": [
                        "NOTE: instance server_id is not registered in the metadata.
                               Use cluster.rescan() to update the metadata.",
                        "NOTE: The required parallel-appliers settings are not enabled on the instance.
                               Use dba.configureInstance() to fix it."
                    ],
                    "memberId": "3bb2592e-8926-11ec-8b6f-02001707f44a",
                    "memberRole": "PRIMARY",
                    "memberState": "ONLINE",
                    "mode": "R/W",
                    "readReplicas": {},
                    "replicationLag": null,
                    "role": "HA",
                    "status": "ONLINE",
                    "version": "8.0.28"
                }
            },
            "topologyMode": "Multi-Primary"
        },
        "groupInformationSourceMember": "example-el7-1644251369:33311",
```

```
        "metadataVersion": "2.1.0"
    }
```

`<Cluster>.status({extended: true})` displays more detailed information about the cluster. In this example, we use the Boolean value `true`, which is equivalent to `<Cluster>.status({'extended':1})`. For more information, see Checking a Cluster's Status with `Cluster.status()`.

The `instanceErrors` suggest that in this upgrade, we should issue `<Cluster>.rescan()` and `dba.configureInstance()` on each member in the InnoDB Cluster:

```
...
"NOTE: instance server_id is not registered in the
metadata. Use cluster.rescan() to update the metadata.",
"NOTE: The required parallel-appliers settings are not
enabled on the instance. Use dba.configureInstance() to fix it."
...
```

The `<Cluster>.rescan()` operation enables you to rescan the InnoDB Cluster for new and obsolete Group Replication instances, as well as changes in the used topology mode. For more information, see Rescanning a Cluster.

```
mysqlsh> cluster1.rescan();
Rescanning the cluster...

Result of the rescanning operation for the 'MyCluster1' cluster:
{
    "name": "MyCluster1",
    "newTopologyMode": null,
    "newlyDiscoveredInstances": [],
    "unavailableInstances": [],
    "updatedInstances": []
}
```

Note

You can only run `<Cluster>.rescan()` on the individual Clusters of the ClusterSet, not the ClusterSet as a whole.

The `dba.configureInstance()` function checks all of the settings required to enable the instance to be used for InnoDB Cluster usage. For more information, see Configuring Production Instances for InnoDB Cluster Usage.

In this example, we issue `dba.configureInstance()` on each member in the Cluster, to ensure required the parallel-appliers settings are enabled on the instance:

```
 mysqlsh> dba.configureInstance('cladmin:cladminpw@localhost:33311')
   The instance 'example-el7-1644251369:33311' belongs to an InnoDB Cluster.
   Configuring local MySQL instance listening at port 33311 for use in an InnoDB cluster...

   This instance reports its own address as ^[[1mexample-el7-1644251369:33311^[[0m
   Clients and other cluster members will communicate with it through this address by default.
   If this is not correct, the report_host MySQL system variable should be changed.

   applierWorkerThreads will be set to the default value of 4.

   ^[[36mNOTE: ^[[0mSome configuration options need to be fixed:
   +----------------------------------------+---------------+----------------+----------------------
   | Variable                               | Current Value | Required Value | Note
   +----------------------------------------+---------------+----------------+----------------------
   | binlog_transaction_dependency_tracking | COMMIT_ORDER  | WRITESET       | Update the server va
   +----------------------------------------+---------------+----------------+----------------------

   Configuring instance...
   The instance 'example-el7-1644251369:33311' was configured to be used in an InnoDB cluster.
```

4. Once you have resolved the issues returned by the `<ClusterSet>.status({extended:1})` and `<Cluster>.status({extended:1})`operations, you must run `<ClusterSet>.listRouters()`.

   `<ClusterSet>.listRouters()` returns details of all the registered MySQL Router instances. The details provides information about each registered MySQL Router instance, such as its name in the metadata, the hostname, ports, and so on. For more information, see Integrating MySQL Router With InnoDB ClusterSet.

   For example, on our example ClusterSet we issue:

```
mysqlsh> <ClusterSet>.listRouters();

WARNING: The following Routers were bootstrapped before the ClusterSet was created: [EXAMPLE::R1].
         Please re-bootstrap the Routers to ensure the optimal configurations are set.

{
    "domainName": "MyClusterSet",
    "routers": {
        "EXAMPLE::R1": {
            "hostname": "EXAMPLE",
            "lastCheckIn": "2022-02-23 07:14:50",
            "roPort": 6447,
            "roXPort": 6449,
            "routerErrors": [
                "WARNING: Router needs to be re-bootstraped."
            ],
            "rwPort": 6446,
            "rwXPort": 6448,
            "targetCluster": null,
            "version": "8.0.28"
        }
    }
}
```

   The returned information shows:

   - The name of the MySQL Router instance.
   - Last check-in timestamp, which is generated by a periodic ping from the MySQL Router stored in the metadata.
   - Hostname where the MySQL Router instance is running.
   - Read-Only and Read-Write ports which the MySQL Router publishes for classic MySQL protocol connections.
   - Read-Only and Read-Write ports which the MySQL Router publishes for X Protocol connections.
   - The name of the target cluster. In this example, MySQL Router directs traffic from client applications to the cluster in the InnoDB ClusterSet deployment that is currently the primary cluster.
   - Version of this MySQL Router instance.

   In this example, there is also information on `routerErrors` returned.

   The `routerErrors` informs us that MySQL Router needs to be re-bootstraped. The reason for this error is that if you create a ClusterSet based on that standalone Cluster, then MySQL Router must be bootstrapped again to inform the Router that it is working on a ClusterSet.

   Resolve these warning to complete your post-upgrade checks. If you do not receive any warnings your post-upgrade checks are complete.

Chapter 9 MySQL InnoDB ReplicaSet

Table of Contents

9.1 Deploying InnoDB ReplicaSet 240
9.2 Configuring InnoDB ReplicaSet Instances 241
9.3 Creating an InnoDB ReplicaSet 241
9.4 Adding Instances to a ReplicaSet 244
    9.4.1 Provisioning Instances for InnoDB ReplicaSet 244
    9.4.2 Example of Adding Instances to a ReplicaSet 245
9.5 Adopting an Existing Replication Setup 246
9.6 Changing the Primary Instance 247
9.7 Forcing a New Primary Instance 248
9.8 Tagging ReplicaSets 249
9.9 Checking the Status of InnoDB ReplicaSet 249
9.10 Upgrade InnoDB ReplicaSet 250

The AdminAPI includes support for InnoDB ReplicaSet, which enables you to administer a set of MySQL instances similarly running asynchronous GTID-based replication, which is completely transaction-based, to InnoDB Cluster. An InnoDB ReplicaSet consists of single primary and multiple secondaries (traditionally referred to as the MySQL replication source and replicas).

You administer your ReplicaSets using a `ReplicaSet` object and the AdminAPI operations, for example, to check the status of the InnoDB ReplicaSet, and manually failover to a new primary in the event of a failure.

Similar to InnoDB Cluster, MySQL Router supports bootstrapping against InnoDB ReplicaSet, which means you can automatically configure MySQL Router to use your InnoDB ReplicaSet without manually configuring it. This automatic configuration makes InnoDB ReplicaSet a quick and easy way to get MySQL replication and MySQL Router up and running. It makes it suited to scaling out `reads` and providing manual failover capabilities in use cases that do not require the high availability offered by InnoDB Cluster.

In addition to deploying an InnoDB ReplicaSet using AdminAPI, you can adopt an existing replication setup. AdminAPI configures the InnoDB ReplicaSet based on the topology of the replication setup. Once you have completed the replication setup, you administer it the same way as an InnoDB ReplicaSet deployed from scratch. You can take advantage of AdminAPI and MySQL Router without creating a new ReplicaSet. For more information see Section 9.5, “Adopting an Existing Replication Setup”.

You can use InnoDB ReplicaSet over a Wide Area Network (WAN) with no impact on write performance, as the server instances are connected by asynchronous replication channels and do not need consensus on transactions. However, replication lag is larger over a WAN. This lag causes the secondary servers in the InnoDB ReplicaSet to be further behind the primary server.

InnoDB ReplicaSet Limitations.
An InnoDB ReplicaSet has several limitations compared to an InnoDB Cluster. It is recommended that you deploy InnoDB Cluster wherever possible. Generally, an InnoDB ReplicaSet on its own does not provide high availability. Among the limitations of InnoDB ReplicaSet are:

- No automatic failover. In events where the primary becomes unavailable, a failover needs to be triggered manually using AdminAPI before any changes are possible again. However, secondary instances remain available for reads.
- No protection from partial data loss due to an unexpected halt or unavailability: Transactions that are not complete at the time of the unexpected halt could be lost.

- No protection against inconsistencies after an unexpected exit or unavailability. If a manual failover promotes a secondary instance while the former primary is still available, for example, due to a network partition, the split-brain situation could introduce data inconsistencies.
- InnoDB ReplicaSet does not support a multi-primary mode. Data consistency cannot be guaranteed with classic replication topologies that allow writes to all members.
- Read scale-out is limited. InnoDB ReplicaSet is based on asynchronous replication, and therefore there is no possible tuning of flow control as there is with Group Replication.
- All secondary members replicate from a single source. For some particular use-cases, this could impact the single source, for example, numerous small updates.
- Only instances running MySQL version 8.0 and later are supported.
- Only GTID-based replication is supported, Binary log file position replication is incompatible with InnoDB ReplicaSet.
- Only Row-Based Replication (RBR) is supported, Statement-Based Replication (SBR) is unsupported.
- Replication filters are not supported.
- Unmanaged replication channels are not allowed on any instance.
- A ReplicaSet consists of a maximum of one primary instance. One or multiple secondaries are supported. Although there is no limit to the number of secondaries you can add to a ReplicaSet, each MySQL Router connected to a ReplicaSet has to monitor each instance. Therefore, the more instances added to a ReplicaSet, the more monitoring there is.
- The ReplicaSet must be managed by MySQL Shell. For example, the replication account is created and managed by MySQL Shell. Making configuration changes to the instance outside MySQL Shell, for example, using SQL statements directly to change the primary instance, is not supported. Always use MySQL Shell to work with InnoDB ReplicaSet.

The main reason to use InnoDB ReplicaSets is you have better write performance. Another reason to use InnoDB ReplicaSets is that they allows deployment on unstable or slow networks, while InnoDB Cluster does not.

9.1 Deploying InnoDB ReplicaSet

Important

It is recommended that you always use the most recent version of MySQL Shell available. The latest version of MySQL Shell can be used with any GA version of MySQL 5.7 or 8.0.

You deploy InnoDB ReplicaSet in a similar way to InnoDB Cluster.

1. Configure at least two MySQL server instances, see Section 6.1, “Using MySQL AdminAPI”:
   - One functions as the primary, in the following example, `rs-1`.
   - The other instance functions as the secondary, in this tutorial `rs-2`, which replicates the transactions applied by the primary.

   This asynchronous MySQL replication, using source and replica, is similar to InnoDB Cluster. The procedure is described at Deploying a Production InnoDB Cluster.

2. Connect to the instances using MySQL Shell, and configure each instance you will use in your ReplicaSet before creating a ReplicaSet.

   - When creating an InnoDB ReplicaSet using MySQL Shell 8.0.28 and later, if you have security requirements that want all accounts created automatically by AdminAPI to have strict

authentication requirements, you can set a value for the replicationAllowedHost configuration option of the ReplicaSet.

3. Once you have created the ReplicaSet, you can add instances to it. For more information, see Adding Instances to a ReplicaSet.

InnoDB ReplicaSet is compatible with sandbox instances, which you can use to deploy locally for testing purposes. See Section 6.8.1, “Deploying Sandbox Instances” for instructions. However, this tutorial assumes you are deploying a production InnoDB ReplicaSet, where each instance is running on a different host.

9.2 Configuring InnoDB ReplicaSet Instances

Use `dba.configureReplicaSetInstance(instance)` to configure each instance you want to use in your ReplicaSet. MySQL Shell can either connect to an instance and then configure it, or you can pass in an `instance` name to configure a specific remote instance. To use an instance in a ReplicaSet, it must support persisting settings. See Section 6.2.4, “Persisting Settings”.

When you connect to the instance for administration tasks, you require a user with suitable privileges. The preferred method to create users to administer a ReplicaSet is using the `setupAdminAccount()` operation. See. Alternatively, the `dba.configureReplicaSetInstance()` operation can optionally create an administrator account, if you provide the `clusterAdmin` option. The account is created with the correct set of privileges required to manage InnoDB ReplicaSet.

Important

The administrator account must have the same user name and password across all instances of the same cluster or replica set.

To configure the instance at `rs-1:3306`, with a cluster administrator named `rsadmin`, issue:

```
mysql-js> dba.configureReplicaSetInstance('root@rs-1:3306', {clusterAdmin: "'rsadmin'@'rs-1%'"});
```

The interactive prompt requests the password required by the specified user. To configure the instance MySQL Shell is currently connected to, you can specify a null instance definition. For example, issue:

```
mysql-js> dba.configureReplicaSetInstance('', {clusterAdmin: "'rsadmin'@'rs-1%'"});
```

The interactive prompt requests the password required by the specified user, this checks the instance which MySQL Shell is currently connected to is valid for use in an InnoDB ReplicaSet. Settings that are incompatible with InnoDB ReplicaSet are configured if possible. The cluster administrator account is created with the privileges required for InnoDB ReplicaSet.

You can define a password expiration using the `clusterAdminPasswordExpiration` option. This option can be set to a number of days, `NEVER` to never expire, or `DEFAULT`, to use the system default.

If you are using SSL certificates for authentication, you can add the certificate issuer and subject using the `clusterAdminCertIssuer` and `clusterAdminCertSubject` options, respectively.

9.3 Creating an InnoDB ReplicaSet

Once you have configured your instances, create an InnoDB ReplicaSet by completing the following steps:

1. Connect to an instance and use `dba.createReplicaSet()` to create a managed ReplicaSet that uses MySQL asynchronous replication, rather than MySQL Group Replication used by InnoDB Cluster. The MySQL instance, which MySQL Shell is connected to, is used as the initial primary of the ReplicaSet.

   The `dba.createReplicaSet()` operation performs several checks to ensure that the instance state and configuration are compatible with a managed ReplicaSet, and if so, a metadata schema is initialized on the instance.

If the ReplicaSet is created successfully, a `ReplicaSet` object is returned. Therefore, it is best practice to assign the returned `ReplicaSet` to a variable. This enables you to work with the ReplicaSet, for example by calling the `<ReplicaSet>status()` operation. To create a ReplicaSet named *`example`* on instance `rs-1` and assign it to the `rs` variable, issue:

```
mysql-js> \connect root@rs-1:3306
...
mysql-js> var rs = dba.createReplicaSet("example")
A new replicaset with instance 'rs-1:3306' will be created.

* Checking MySQL instance at rs-1:3306

This instance reports its own address as rs-1:3306
rs-1:3306: Instance configuration is suitable.

* Updating metadata...

ReplicaSet object successfully created for rs-1:3306.
Use rs.addInstance() to add more asynchronously replicated instances to this replicaset
and rs.status() to check its status.
```

2. Use the returned `ReplicaSet` object to verify that the operation was successful. For example, this provides the *`ReplicaSet`*`.status()` operation, which displays information about the ReplicaSet. The returned `ReplicaSet` is already assigned to the variable `rs`, so issue:

```
mysql-js> rs.status()
{
    "replicaSet": {
        "name": "example",
        "primary": "rs-1:3306",
        "status": "AVAILABLE",
        "statusText": "All instances available.",
        "topology": {
            "rs-1:3306": {
                "address": "rs-1:3306",
                "instanceRole": "PRIMARY",
                "mode": "R/W",
                "status": "ONLINE"
            }
        },
        "type": "ASYNC"
    }
}
```

This output shows that the ReplicaSet named `example` has been created, and that the primary is `rs-1`. Currently, there is only one instance, and the next task is to add more instances to the ReplicaSet.

ReplicaSet Encryption and Authentication

Replicas can verify the identity of the source and use client SSL certificates for authentication. The following options were added to `dba.createReplicaSet`:

- `memberAuthType`: defines the authentication type used for the internal replication accounts. This option takes one of the following values:
  - `PASSWORD`: Account authenticates with password only.
  - `CERT_ISSUER`: Account authenticates with a client certificate, which must match the expected issuer. This value is equivalent to `VERIFY_CA`.
  - `CERT_SUBJECT`: Account authenticates with a client certificate, which must match the expected issuer and subject. This value is equivalent to `VERIFY_IDENTITY`.
  - `CERT_ISSUER_PASSWORD`: Account authenticates with a combination of `PASSWORD` and `CERT_ISSUER` values.

    - `CERT_SUBJECT_PASSWORD`: Account authenticates with a combination of `PASSWORD` and `CERT_SUBJECT` values.
- `certIssuer`: Defines the certificate issuer required for authentication if `memberAuthType` contains `CERT_ISSUER` or `CERT_SUBJECT`.
- `certSubject`: Defines the certificate subject of the instance. Required if `memberAuthType` contains `CERT_SUBJECT`.
- `replicationSslMode`: Defines the authentication type of the replication channels in the replicaSet. This option takes one of the following values:
    - `DISABLED`: TLS encryption is disabled for the replication channel.
    - `REQUIRED`: TLS encryption is enabled for the replication channel.
    - `VERIFY_CA`: The same as REQUIRED, but additionally verifies the peer server TLS certificate against the configured Certificate Authority (CA) certificates.
    - `VERIFY_IDENTITY`: The same as VERIFY_CA, but additionally verifies that the peer server certificate matches the host to which the connection is attempted.
    - `AUTO`: TLS encryption is enabled if supported by the instance. Disabled if the instance does not support TLS.

For example:

```
mysql-js> myreplicaset = dba.createReplicaSet("replicaSet1",
  { "replicationSslMode": "VERIFY_IDENTITY", "memberAuthType":"CERT_SUBJECT",
  "certIssuer":"/CN=MyCertAuthority", "certSubject": "/CN=mysql-5.local"});
```

Note

As of MySQL Shell 8.0.32, all new replication channels are created with SSL enabled. This is not true for replication groups adopted with MySQL Shell 8.0.32. Their replication channels remain unencrypted.

InnoDB ReplicaSet replicationAllowedHost

When creating an InnoDB ReplicaSet using MySQL Shell 8.0.28 and later, if you have security requirements that want all accounts created automatically by AdminAPI to have strict authentication requirements, you can set a value for the `replicationAllowedHost` configuration option of the ReplicaSet. The `replicationAllowedHost` MySQL Shell option allows you to set internally managed replication accounts for a ReplicaSet to a strict subnet based filter instead of the default wildcard value of `%`.The `replicationAllowedHost` option can take a string value. For example, to set the `replicationAllowedHost` to `192.0.2.0/24`, issue:

```
mysql-js> var rs = dba.createReplicaSet('example', {replicationAllowedHost:'192.0.2.0/24'})
        A new replicaset with instance 'rs-1:3306' will be created.

* Checking MySQL instance at rs-1:3306

This instance reports its own address as rs-1:3306
rs-1:3306: Instance configuration is suitable.

* Updating metadata...

ReplicaSet object successfully created for rs-1:3306.
Use rs.addInstance() to add more asynchronously replicated instances to this replicaset
and rs.status() to check its status.
```

An InnoDB ReplicaSet can be modified after creation to set the variable `replicationAllowedHost` through the `setOption` configuration option, by issuing:

```
mysql-js> rs.setOption('replicationAllowedHost', '192.0.2.0/24')
```

9.4 Adding Instances to a ReplicaSet

When you have created a ReplicaSet, you can use the *`ReplicaSet`*`.addInstance()` operation to add an instance as a read-only secondary replica of the current primary of the ReplicaSet.

The primary of the ReplicaSet must be reachable and available during this operation. MySQL Replication is configured between the added instance and the primary, using an automatically created MySQL account with a random password. Before the instance can be an operational secondary, it must be in a synchronistic arrangement with the primary. This process is called `recovery`, and InnoDB ReplicaSet supports different methods which you configure with the `recoveryMethod` option.

For an instance to be able to join a ReplicaSet, various prerequisites must be satisfied. They are automatically checked by *`ReplicaSet`*`.addInstance()`, and the operation fails if any issues are found.

Use `dba.configureReplicaSetInstance()` to validate and configure binary log and replication related options before adding an instance. MySQL Shell connects to the target instance using the same user name and password used to obtain the `ReplicaSet` handle object. All instances of the ReplicaSet are expected to have the same administrator account with the same grants and passwords. You can create a custom administrator account with the required grants when you configure an instance with `dba.configureReplicaSetInstance()` option. See Section 9.2, “Configuring InnoDB ReplicaSet Instances”.

9.4.1 Provisioning Instances for InnoDB ReplicaSet

When you add new instances to an InnoDB ReplicaSet you need to provision the instances with the existing data that the ReplicaSet contains. You can do this provisioning automatically using one of the following methods:

- *MySQL Clone*: Takes a snapshot from an online instance and then replaces any data on the new instance with the snapshot. MySQL Clone is well suited for joining a new blank instance to an InnoDB ReplicaSet. MySQL Clone does not rely on there being a complete binary log of all transactions applied by the InnoDB ReplicaSet.

  Warning

  When you add an instance, the MySQL Clone operation destroys all the instance's previous data.

- *Incremental Recovery*: Relies on MySQL Replication to apply all missing transactions on the new instance. Incremental Recovery is the fastest method if only a few transactions are missing on the new instance. However, you can only use this method if at least one online instance of the InnoDB ReplicaSet has a complete binary log, which contains the entire transaction history of the InnoDB ReplicaSet.

  You cannot use this method:

  - If you have purged the binary logs from all members.
  - If you enabled the binary log after databases already existed in the instance.

  If you have many transactions to apply, there could be a substantial delay before the instance can join the InnoDB ReplicaSet.

When an instance is joining a ReplicaSet, recovery is used in much the same way that it is in InnoDB Cluster. MySQL Shell attempts to automatically select a suitable recovery method. If it is not possible to choose a method safely, MySQL Shell prompts for what to use. For more information, see

Section 7.4.6, “Using MySQL Clone with InnoDB Cluster”. This section covers the differences when adding instances to a ReplicaSet.

9.4.2 Example of Adding Instances to a ReplicaSet

To add instances to a ReplicaSet, complete the following steps:

1. Use the *`ReplicaSet`*`.addInstance(`*`instance`*`)` operation to add secondary instances to the *`ReplicaSet`*. You specify the *`instance`* as a URI-like connection string. The user you specify must have the privileges required and must be the same on all instances in the ReplicaSet. For more information, see Section 9.2, “Configuring InnoDB ReplicaSet Instances”.

   For example, to add the instance at `rs-2`, port number `3306`, and user `rsadmin`, issue:

```
mysql-js> rs.addInstance('rsadmin@rs-2:3306')

Adding instance to the replicaset...

* Performing validation checks

This instance reports its own address as rsadmin@rs-2
rsadmin@rs-2: Instance configuration is suitable.

* Checking async replication topology...

* Checking transaction state of the instance...

NOTE: The target instance 'rsadmin@rs-2' has not been pre-provisioned (GTID set
is empty). The Shell is unable to decide whether replication can completely
recover its state.  The safest and most convenient way to provision a new
instance is through automatic clone provisioning, which will completely
overwrite the state of 'rsadmin@rs-2' with a physical snapshot from an existing
replicaset member. To use this method by default, set the 'recoveryMethod'
option to 'clone'.

WARNING: It should be safe to rely on replication to incrementally recover the
state of the new instance if you are sure all updates ever processed in the
replicaset were done with GTIDs enabled, there are no purged transactions and
the new instance contains the same GTID set as the replicaset, or a subset of it.
To use this method by default, set the 'recoveryMethod' option to 'incremental'.
Please select a recovery method [C]lone/[I]ncremental recovery/[A]bort (default Clone):
```

2. In this case, we did not specify the recovery method, so the operation advises you on how to best proceed. In this example, we choose the `Clone` option because we do not have any existing transactions on the instance joining the ReplicaSet. Therefore, there is no risk of deleting data from the joining instance. For more information, see Recovery Methods for InnoDB ReplicaSet.

```
Please select a recovery method [C]lone/[I]ncremental recovery/[A]bort (default Clone): C
* Updating topology
Waiting for clone process of the new member to complete. Press ^C to abort the operation.
* Waiting for clone to finish...
NOTE: rsadmin@rs-2 is being cloned from rsadmin@rs-1
** Stage DROP DATA: Completed
** Clone Transfer
FILE COPY  ##################################################################  100%  Completed
PAGE COPY  ##################################################################  100%  Completed
REDO COPY  ##################################################################  100%  Completed
** Stage RECOVERY: \
NOTE: rsadmin@rs-2 is shutting down...

* Waiting for server restart... ready
* rsadmin@rs-2 has restarted, waiting for clone to finish...
* Clone process has finished: 59.63 MB transferred in about 1 second (~1.00 B/s)

** Configuring rsadmin@rs-2 to replicate from rsadmin@rs-1
** Waiting for new instance to synchronize with PRIMARY...

The instance 'rsadmin@rs-2' was added to the replicaset and is replicating from rsadmin@rs-1.
```

3. Assuming the instance is valid for InnoDB ReplicaSet usage, recovery proceeds. In this case, the newly joining instance uses MySQL Clone to copy all the transactions it has not yet applied from the primary, then it joins the ReplicaSet as an online instance. To verify, use the `rs.status()` operation:

```
mysql-js> rs.status()
{
 "replicaSet": {
  "name": "example",
  "primary": "rs-1:3306",
  "status": "AVAILABLE",
  "statusText": "All instances available.",
  "topology": {
   "rs-1:3306": {
    "address": "rs-1:3306",
    "instanceRole": "PRIMARY",
    "mode": "R/W",
    "status": "ONLINE"
   },
   "rs-2:3306": {
    "address": "rs-2:3306",
    "instanceRole": "SECONDARY",
    "mode": "R/O",
    "replication": {
     "applierStatus": "APPLIED_ALL",
     "applierThreadState": "Replica has read all relay log; waiting for more updates",
     "receiverStatus": "ON",
     "receiverThreadState": "Waiting for source to send event",
     "replicationLag": null
    },
    "status": "ONLINE"
   }
  },
  "type": "ASYNC"
 }
}
```

This output shows that the ReplicaSet named `example` now consists of two MySQL instances, and that the primary is `rs-1`. Currently, there is one secondary instance at `rs-2`, which is a replica of the primary. The ReplicaSet is online, which means that the primary and secondary are in synchrony. At this point, the ReplicaSet is ready to process transactions.

4. To override the interactive MySQL Shell mode, choose the most suitable recovery method. Use the `recoveryMethod` option to configure how the instance recovers the data required to be able to join the ReplicaSet. For more information, see Section 7.4.6, “Using MySQL Clone with InnoDB Cluster”.

9.5 Adopting an Existing Replication Setup

As an alternative to creating a ReplicaSet from scratch, you can adopt an existing replication setup using the `adoptFromAR` option with `dba.createReplicaSet()`. The replication setup is scanned, and if it is compatible with the InnoDB ReplicaSet Limitations, AdminAPI creates the necessary metadata. Once the replication setup has been adopted, you can only use AdminAPI to administer the InnoDB ReplicaSet.

To convert an existing replication setup to an InnoDB ReplicaSet connect to the primary, also referred to as the source. The replication topology is automatically scanned and validated, beginning from the connected global session of the instance MySQL Shell. The configuration of all instances is checked during the adoption to ensure they are compatible with InnoDB ReplicaSet usage:

- All replication channels must be active, and their transaction sets as verified through GTID sets must be consistent.

- Instances are assumed to have the same state or be able to converge.

Note

As of 8.0.32, all new replication channels are created with SSL enabled. This is not true for replication groups adopted with MySQL Shell 8.0.32. Their replication channels remain unencrypted.

The replication topology is automatically scanned and validated, starting from the instance MySQL Shell's global session is connected to. The only changes made by this operation to an adopted ReplicaSet are the creation of the metadata schema. Existing replication channels are not changed during adoption, although you can change them during subsequent primary switch operations.

For example, to adopt a replication topology consisting of the MySQL server instances on `example1` and `example2` to an InnoDB ReplicaSet.

Connect to the primary at `example1` and issue:

```
mysql-js> rs = dba.createReplicaSet('testadopt', {'adoptFromAR':1})
A new replicaset with the topology visible from 'example1:3306' will be created.

* Scanning replication topology...
** Scanning state of instance example1:3306
** Scanning state of instance example2:3306

* Discovering async replication topology starting with example1:3306
Discovered topology:
- example1:3306: uuid=00371d66-3c45-11ea-804b-080027337932 read_only=no
- example2:3306: uuid=59e4f26e-3c3c-11ea-8b65-080027337932 read_only=no
    - replicates from example1:3306
 source="localhost:3310" channel= status=ON receiver=ON applier=ON

* Checking configuration of discovered instances...

This instance reports its own address as example1:3306
example1:3306: Instance configuration is suitable.

This instance reports its own address as example2:3306
example2:3306: Instance configuration is suitable.

* Checking discovered replication topology...
example1:3306 detected as the PRIMARY.
Replication state of example2:3306 is OK.

Validations completed successfully.

* Updating metadata...

ReplicaSet object successfully created for example1:3306.
Use rs.addInstance() to add more asynchronously replicated instances to
this replicaset and rs.status() to check its status.
```

Once the InnoDB ReplicaSet has been adopted, you can use it in the same way that you would use a ReplicaSet which was created.

Warning

From this point, you must administer the InnoDB ReplicaSet using only AdminAPI.

9.6 Changing the Primary Instance

Use the *`ReplicaSet`*`.setPrimaryInstance()` operation to safely perform a change of the primary of a ReplicaSet to another instance. The current primary is demoted to a secondary and made read-only, while the promoted instance becomes the new primary and is made read-write. All other secondary instances are updated to replicate from the new primary. MySQL Router instances, which

have been bootstrapped against the ReplicaSet automatically start redirecting read-write clients to the new primary.

For a safe change of the primary to be possible, all ReplicaSet instances must be reachable by MySQL Shell and have consistent `GTID_EXECUTED` sets. If the primary is not available, and there is no way to restore it, a forced failover might be the only option instead, see Section 9.7, “Forcing a New Primary Instance”.

During a change of primary instance, the promoted instance is synchronized with the old primary, ensuring that all transactions present on the primary are applied before the topology change is committed. If this synchronization step takes too long or is not possible on any of the secondary instances, the operation is aborted. In such a situation, the secondary instances must be repaired or removed from the ReplicaSet for the failover to be possible.

9.7 Forcing a New Primary Instance

Unlike InnoDB Cluster, which supports automatic failover in the event of an unexpected failure of the primary, InnoDB ReplicaSet does not have automatic failure detection or a consensus-based protocol such as that provided by Group Replication. If the primary is not available, a manual failover is required. An InnoDB ReplicaSet which has lost its primary is effectively read-only, and for any write changes to be possible a new primary must be chosen. If you cannot connect to the primary, and you cannot use `ReplicaSet.setPrimaryInstance()` to safely perform a switchover to a new primary as described at Section 9.6, “Changing the Primary Instance”, use the `ReplicaSet.forcePrimaryInstance()` operation to perform a forced failover of the primary. This is a last resort operation that must only be used in a disaster type scenario where the current primary is unavailable and cannot be restored in any way.

Warning

A forced failover is a potentially destructive action and must be used with caution.

If a target instance is not reachable (or is null), the most up-to-date instance is automatically selected and promoted to be the new primary. If a target instance is reachable, it is promoted to be the new primary. Other reachable secondary instances replicate from this new primary. The target instance must have the most up-to-date `GTID_EXECUTED` set among reachable instances, otherwise the operation fails.

A failover is different from a planned primary change because it promotes a secondary instance without synchronizing with or updating the old primary. That has the following major consequences:

- Any transactions that had not yet been applied by a secondary at the time the old primary failed are lost.
- If the old primary is still running and processing transactions, there is a split-brain, and the datasets of the old and new primaries diverge.

If the last known primary is still reachable, the `ReplicaSet.forcePrimaryInstance()` operation fails, to reduce the risk of split-brain situations. But it is the administrator's responsibility to ensure that the old primary is not reachable by the other instances to prevent or minimize such scenarios.

After a forced failover, the old primary is considered invalid by the new primary and can no longer be part of the ReplicaSet. If you later find an instance that can be recovered, you must remove it from the ReplicaSet and add it as a new instance. A secondary instance is considered invalid if it cannot be switched to the new primary during the failover.

Data loss is possible after a failover because the old primary might have had transactions that were not yet replicated to the secondary being promoted. Moreover, if the instance that was presumed to have failed can still process transactions, for example because the network where it is located is still functioning but unreachable from MySQL Shell, it continues diverging from the promoted instances.

Recovering once transaction sets on instances have diverged requires manual intervention and could not be possible in some situations, even if the failed instances can be recovered. Often, the fastest and simplest way to recover from a disaster that required a forced failover is by discarding such diverged transactions and re-provisioning a new instance from the newly promoted primary.

9.8 Tagging ReplicaSets

Tagging is supported by ReplicaSets, and their instances. For the purpose of tagging, ReplicaSets support the `setOption()`, `setInstanceOption()` and `options()` operations. These operations function in generally the same way as their `Cluster` equivalents. For more information, see Section 6.9, “Tagging Metadata”. This section documents the differences in working with tags for ReplicaSets.

Important

There are no other options which can be configured for ReplicaSets and their instances. For ReplicaSets, the options documented at Section 7.5.1, “Setting Options for InnoDB Cluster” are not supported. The only supported option is the tagging described here.

The *`ReplicaSet`*`.options()` operation shows information about the tags assigned to individual ReplicaSet instances as well as to the ReplicaSet itself.

The `option` argument of *`ReplicaSet`*`.setOption()` and *`ReplicaSet`*`.setInstanceOption()` only support options with the `tag` namespace and throw an error otherwise.

The *`ReplicaSet`*`.setInstanceOption(`*`instance, option, value`*`)` and *`ReplicaSet`*`.setOption(`*`option, value`*`)` operations behave in the same way as the `Cluster` equivalent operations.

There are no differences in hiding instances as described at Removing Instances from Routing. For example, to hide the ReplicaSet instance `rs-1`, issue:

```
mysql-js> myReplicaSet.setInstanceOption("icadmin@rs-1:3306", "tag:_hidden", true);
```

A MySQL Router that has been bootstrapped against the ReplicaSet detects the change and removes the `rs-1` instance from the routing destinations.

9.9 Checking the Status of InnoDB ReplicaSet

Check information about a ReplicaSet using the `ReplicaSet.status()` operation. The `ReplicaSet.status()` operation supports the extended option to get different levels of detail. For example:

- `ReplicaSet.status({extended:0})`: Provides a regular level of details. Only basic information about the status of the instance and replication is included, in addition to non-default or unexpected replication settings and status.
- `ReplicaSet.status({extended:1})`: Setting extended to 1 includes Metadata Version, server UUID, replication information such as lag and worker threads, the raw information used to derive the status of the instance, size of the applier queue, value of system variables that protect against unexpected writes and so on.
- `ReplicaSet.status({extended:2})`: Setting extended to 2 includes important replication related configuration settings, such as encrypted connections, and so on.

The output of ReplicaSet.status(extended=1) is similar to Cluster.status(extended=1), but the main difference is that the replication field is always available because InnoDB ReplicaSet relies on MySQL Replication all the time, unlike InnoDB Cluster which uses it during incremental recovery. For more information on the fields, see Checking a cluster's Status with Cluster.status().

9.10 Upgrade InnoDB ReplicaSet

To upgrade the instances in an InnoDB ReplicaSet, complete the following steps:

1. Upgrade MySQL Router.
2. Upgrade MySQL Shell.
3. Upgrade MySQL Server.
4. Post Upgrade Status Check.

Check the versions of the installed binaries:

- **mysqlrouter --version**: Checks the version of MySQL Router installed.
- **mysqlsh --version**: Checks the version of MySQL Shell installed.
- **mysqld --version**: Checks the version of MySQL Server installed.

Upgrade MySQL Router.

To upgrade MySQL Router, complete the following steps:

1. Stop MySQL Router.

   On a Unix system, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected when you bootstrapped the router. These files include `stop.sh`. Navigate to this directory and issue this command:

   ```
   ./stop.sh
   ```

   On Microsoft Windows, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected when you bootstrapped the router. These files include `stop.ps1`. Navigate to this directory and issue this command:

   ```
   .\stop.ps1
   ```

   Or on a Linux system using `systemd`, stop the MySQL Router service by issuing:

   ```
   systemctl stop mysqlrouter.service
   ```

   Otherwise, kill the *process ID (PID)* of the associated *mysqlrouter* process.

2. Obtain and install the latest version of MySQL Router.
3. Start MySQL Router.

   On a Unix system, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected. These files include `start.sh`. Navigate to the directory and issue this command:

   ```
   ./start.sh
   ```

   If the path to the new router has changed, you must update the `start.sh` shell script to reflect the path.

   ```
   #!/bin/bash
   basedir=/tmp/myrouter
   ROUTER_PID=$basedir/mysqlrouter.pid /usr/bin/mysqlrouter -c $basedir/mysqlrouter.conf &
   disown %-
   ```

If you upgrade MySQL Router manually, opposed to using package management, you can update the `basedir=`. Bootstrapping the router again also regenerates the `start.sh` shell script.

Or on a Linux system using `systemd`, start the MySQL Router service by issuing:

```
systemctl start mysqlrouter.service
```

On Microsoft Windows, if you used the optional `--directory` bootstrap option, a self-contained installation is created with all generated directories and files at the location you selected. These files include `start.ps1`. Navigate to the directory and issue this command:

```
.\start.ps1
```

On starting MySQL Router using the new router binaries, the version of the router is upgraded:

```
mysqlrouter --version
```

Upgrade MySQL Shell

Upgrade MySQL Shell by installing the new binaries, and stopping and starting MySQL Shell:

1. Obtain and install the latest version of MySQL Shell.
2. Stop and quit MySQL Shell by issuing:

   ```
   \q
   ```

3. Restart MySQL Shell from the command line, by issuing:

   ```
   mysqlsh
   ```

4. Upgrade the InnoDB ReplicaSet Metadata:

   - To upgrade an InnoDB ReplicaSet, connect MySQL Shell's global session to your ReplicaSet and use the `dba.upgradeMetadata()` operation to upgrade the ReplicaSet's metadata to the new metadata.

     The `dba.upgradeMetadata()` function compares the version of the installed metadata schema with the version of the metadata schema supported by this Shell. If the installed metadata version is lower, an upgrade process is started.

Metadata Upgrade

The Metadata Upgrade may do nothing if the ReplicaSet already uses the latest version.

Upgrade MySQL Server

Upgrade MySQL Server by upgrading all secondary instances before upgrading the primary instance.

Upgrading MySQL Server is optional

Upgrading MySQL Server is optional. Server upgrades can have a greater impact than upgrading MySQL Shell and MySQL Router. Also, you should always keep MySQL Shell and MySQL Router at the latest version, even if the server is not; this is true for InnoDB Clusters and ReplicaSets.

1. Stop MySQL Server by issuing one of the following commands:

   - If MySQL Server is using *systemd* issue:

     ```
     systemctl stop mysqld
     ```

- If MySQL Server is using *init.d* issue:

```
/etc/init.d/mysql stop
```

- If MySQL Server is using *service* issue:

```
service mysql stop
```

- If you deployed MySQL Server on Microsoft Windows issue:

```
mysqladmin -u root -p shutdown
```

2. Obtain and install the latest version of MySQL Server.
3. Start MySQL Server by issuing one of the following commands:
   - If MySQL Server is using *systemd* issue:

```
systemctl start mysqld
```

   - If MySQL Server is using *init.d* issue:

```
/etc/init.d/mysql start
```

   - If MySQL Server is using *service* issue:

```
service mysql start
```

   - If you deployed MySQL Server on Microsoft Windows issue:

```
mysqld
```

4. When you have upgraded all the secondary instances, upgrade the primary instance to complete the upgrade process.

   There is no automatic primary switching in InnoDB ReplicaSet. You need to set the primary instance to a member you have upgraded already before upgrading the primary instance.

   Set an upgraded secondary instance to be the primary instance:

```
<ReplicaSet>.setPrimaryInstance('<host>:<port>')
```

   Use the `<ReplicaSet>.setPrimaryInstance()` operation to safely perform a change of the primary of a ReplicaSet to another instance. The current primary is demoted to a secondary and made read-only, while the promoted instance becomes the new primary and is made read-write. All other secondary instances are updated to replicate from the new primary. MySQL Router instances that you have bootstrapped against the ReplicaSet automatically start redirecting read-write clients to the new primary.

   Upgrade the `old` primary instance. Once upgraded, you can use `<ReplicaSet>.setPrimaryInstance()` to restore this upgraded instance back to being primary. For more information, see Section 9.6, “Changing the Primary Instance”.

Post Upgrade Status Check

After upgrading MySQL Router, MySQL Shell, and MySQL Servers are upgraded:

- Check the status of the ReplicaSet by issuing `<ReplicaSet>.status()`. In the following example, `<ReplicaSet>.status()` returns an `instanceErrors`:

```
 mysqlsh> <ReplicaSet>.status();
{
    "replicaSet": {
        "name": "myReplicaSet",
```

```
        "primary": "example-el7-1644251369:30014",
        "status": "AVAILABLE",
        "statusText": "All instances available.",
        "topology": {
            "example-el7-1644251369:30011": {
                "address": "example-el7-1644251369:30011",
                "instanceErrors": [
                    "NOTE: The required parallel-appliers settings are not enabled on the instance.
                           Use dba.configureReplicaSetInstance() to fix it."
                ],
                "instanceRole": "SECONDARY",
                "mode": "R/O",
                "replication": {
                    "applierStatus": "APPLIED_ALL",
                    "applierThreadState": "Waiting for an event from Coordinator",
                    "applierWorkerThreads": 4,
                    "receiverStatus": "ON",
                    "receiverThreadState": "Waiting for source to send event",
                    "replicationLag": null
                },
                "status": "ONLINE"
            },
            "example-el7-1644251369:30014": {
                "address": "example-el7-1644251369:30014",
                "instanceErrors": [
                    "NOTE: The required parallel-appliers settings are not enabled on the instance.
                           Use dba.configureReplicaSetInstance() to fix it."
                ],
                "instanceRole": "PRIMARY",
                "mode": "R/W",
                "status": "ONLINE"
            },
            "example-el7-1644251369:30017": {
                "address": "example-el7-1644251369:30017",
                "instanceErrors": [
                    "NOTE: The required parallel-appliers settings are not enabled on the instance.
                           Use dba.configureReplicaSetInstance() to fix it."
                ],
                "instanceRole": "SECONDARY",
                "mode": "R/O",
                "replication": {
                    "applierStatus": "APPLIED_ALL",
                    "applierThreadState": "Waiting for an event from Coordinator",
                    "applierWorkerThreads": 4,
                    "receiverStatus": "ON",
                    "receiverThreadState": "Waiting for source to send event",
                    "replicationLag": null
                },
                "status": "ONLINE"
            },
            "example-el7-1644251369:30021": {
                "address": "example-el7-1644251369:30021",
                "instanceErrors": [
                    "NOTE: The required parallel-appliers settings are not enabled on the instance.
                           Use dba.configureReplicaSetInstance() to fix it."
                ],
                "instanceRole": "SECONDARY",
                "mode": "R/O",
                "replication": {
                    "applierStatus": "APPLIED_ALL",
                    "applierThreadState": "Waiting for an event from Coordinator",
                    "applierWorkerThreads": 4,
                    "receiverStatus": "ON",
                    "receiverThreadState": "Waiting for source to send event",
                    "replicationLag": null
                },
                "status": "ONLINE"
            }
        },
        "type": "ASYNC"
    }
```

```
}
```

The `instanceError` tells us to issue `dba.configureReplicaSetInstance()` to fix the error.

`dba.configureReplicaSetInstance(`*`instance`*`)` configures each instance you want to use in the ReplicaSet. MySQL Shell can either connect to an instance and then configure it, or you can pass in an `instance` name to configure a specific remote instance. For more information, see Section 6.2.4, “Persisting Settings”.

The following example shows the output of `<ReplicaSet>.status()` if the `PRIMARY` member has `read_only` or `super_read_only` set to `ON`:

```
mysqlsh > <ReplicaSet>.status();
replicaset.status();
{
    "replicaSet": {
        "name": "myReplicaSet",
        "primary": "example-el7-1644251369:30014",
        "status": "UNAVAILABLE",
        "statusText": "PRIMARY instance is not available, but there is at least one SECONDARY
                      that could be force-promoted.",
        "topology": {
            "example-el7-1644251369:30011": {
                "address": "example-el7-1644251369:30011",
                "instanceErrors": [
                    "NOTE: The required parallel-appliers settings are not enabled on the instance.
                           Use dba.configureReplicaSetInstance() to fix it."
                ],
                "instanceRole": "SECONDARY",
                "mode": "R/O",
                "replication": {
                    "applierStatus": "APPLIED_ALL",
                    "applierThreadState": "Waiting for an event from Coordinator",
                    "applierWorkerThreads": 4,
                    "receiverStatus": "ON",
                    "receiverThreadState": "Waiting for source to send event",
                    "replicationLag": null
                },
                "status": "ONLINE"
            },
            "example-el7-1644251369:30014": {
                "address": "example-el7-1644251369:30014",
                "fenced": true,
                "instanceErrors": [
                    "ERROR: Instance is a PRIMARY but is READ-ONLY: read_only=ON, super_read_only=ON",
                    "NOTE: The required parallel-appliers settings are not enabled on the instance.
                           Use dba.configureReplicaSetInstance() to fix it."
                ],
                "instanceRole": "PRIMARY",
                "mode": "R/O",
                "status": "ERROR"
            },
            "example-el7-1644251369:30017": {
                "address": "example-el7-1644251369:30017",
                "instanceErrors": [
                    "NOTE: The required parallel-appliers settings are not enabled on the instance.
                           Use dba.configureReplicaSetInstance() to fix it."
                ],
                "instanceRole": "SECONDARY",
                "mode": "R/O",
                "replication": {
                    "applierStatus": "APPLIED_ALL",
                    "applierThreadState": "Waiting for an event from Coordinator",
                    "applierWorkerThreads": 4,
                    "receiverStatus": "ON",
                    "receiverThreadState": "Waiting for source to send event",
                    "replicationLag": null
                },
                "status": "ONLINE"
            },
```

```
            "example-el7-1644251369:30021": {
                "address": "example-el7-1644251369:30021",
                "instanceErrors": [
                    "NOTE: The required parallel-appliers settings are not enabled on the instance.
                           Use dba.configureReplicaSetInstance() to fix it."
                ],
                "instanceRole": "SECONDARY",
                "mode": "R/O",
                "replication": {
                    "applierStatus": "APPLIED_ALL",
                    "applierThreadState": "Waiting for an event from Coordinator",
                    "applierWorkerThreads": 4,
                    "receiverStatus": "ON",
                    "receiverThreadState": "Waiting for source to send event",
                    "replicationLag": null
                },
                "status": "ONLINE"
            }
        },
        "type": "ASYNC"
    }
}
```

For more information, see Section 9.9, “Checking the Status of InnoDB ReplicaSet”. .

Chapter 10 Extending MySQL Shell

Table of Contents

10.1 Reporting with MySQL Shell 257
10.1.1 Creating MySQL Shell Reports 258
10.1.2 Registering MySQL Shell Reports 258
10.1.3 Persisting MySQL Shell Reports 260
10.1.4 Example MySQL Shell Report 260
10.1.5 Running MySQL Shell Reports 261
10.1.6 Built-in MySQL Shell Reports 262
10.2 Adding Extension Objects to MySQL Shell 265
10.2.1 Creating User-Defined MySQL Shell Global Objects 265
10.2.2 Creating Extension Objects 266
10.2.3 Persisting Extension Objects 268
10.2.4 Example MySQL Shell Extension Objects 268
10.3 MySQL Shell Plugins 270
10.3.1 Creating MySQL Shell Plugins 270
10.3.2 Creating Plugin Groups 271
10.3.3 Example MySQL Shell Plugins 271

You can define extensions to the base functionality of MySQL Shell in the form of reports and extension objects. Reports and extension objects can be created using JavaScript or Python, and can be used regardless of the active MySQL Shell language. You can persist reports and extension objects in plugins that are loaded automatically when MySQL Shell starts.

- MySQL Shell reports are available from MySQL Shell 8.0.16. See Section 10.1, “Reporting with MySQL Shell”.
- Extension objects are available from MySQL Shell 8.0.17. See Section 10.2, “Adding Extension Objects to MySQL Shell”.
- Reports and extension objects can be stored as MySQL Shell plugins from MySQL Shell 8.0.17. See Section 10.3, “MySQL Shell Plugins”.

10.1 Reporting with MySQL Shell

MySQL Shell enables you to set up and run reports to display live information from a MySQL server, such as status and performance information. MySQL Shell's reporting facility supports both built-in reports and user-defined reports. The reporting facility is available from MySQL Shell 8.0.16. Reports can be created directly at the MySQL Shell interactive prompt, or defined in scripts that are automatically loaded when MySQL Shell starts.

A report is a plain JavaScript or Python function that performs operations to generate the desired output. You register the function as a MySQL Shell report through the `shell.registerReport()` method in JavaScript or the `shell.register_report()` method in Python. Section 10.1.1, “Creating MySQL Shell Reports” has instructions to create, register, and store your reports. You can store your report as part of a MySQL Shell plugin (see Section 10.3, “MySQL Shell Plugins”).

Reports written in any of the supported languages (JavaScript, Python, or SQL) can be run regardless of the active MySQL Shell language. Reports can be run once using the MySQL Shell `\show` command, or run and then refreshed continuously in a MySQL Shell session using the `\watch` command. They can also be accessed as API functions using the `shell.reports` object. Section 10.1.5, “Running MySQL Shell Reports” explains how to run reports in each of these ways.

MySQL Shell includes a number of built-in reports, described in Section 10.1.6, “Built-in MySQL Shell Reports”.

10.1.1 Creating MySQL Shell Reports

You can create and register a user-defined report for MySQL Shell in either of the supported scripting languages, JavaScript and Python. The reporting facility handles built-in reports and user-defined reports using the same API frontend scheme.

Reports can specify a list of report-specific options that they accept, and can also accept a specified number of additional arguments. Your report can support both, one, or neither of these inputs. When you request help for a report, MySQL Shell provides a listing of options and arguments, and any available descriptions of these that are provided when the report is registered.

Signature

The signature for the Python or JavaScript function to be registered as a MySQL Shell report must be as follows:

```
Dict report(Session session, List argv, Dict options);
```

Where:

- `session` is a MySQL Shell session object that is to be used to execute the report.
- `argv` is an optional list containing string values of additional arguments that are passed to the report.
- `options` is an optional dictionary with key names and values that correspond to any report-specific options and their values.

Report types

A report function is expected to return data in a specific format, depending on the type you use when registering it:

List type	Returns output as a list of lists, with the first list consisting of the names of columns, and the remainder being the content of rows. MySQL Shell displays the output in table format by default, or in vertical format if the `--vertical` or `--E` option was specified on the `\show` or `\watch` command. The values for the rows are converted to string representations of the items. If a row has fewer elements than the number of column names, the missing elements are considered to be NULL. If a row has more elements than the number of column names, the extra elements are ignored. When you register this report, use the type “list”.
Report type	Returns free-form output as a list containing a single item. MySQL Shell displays this output using YAML. When you register this report, use the type “report”.
Print type	Prints the output directly to screen, and return an empty list to MySQL Shell to show that the output has already been displayed. When you register this report, use the type “print”.

To provide the output, the API function for the report must return a dictionary with the key `report`, and a list of JSON objects, one for each of the items in your returned list. For the List type, use one element for each list, for the Report type use a single element, and for the Print type use no elements.

10.1.2 Registering MySQL Shell Reports

To register your user-defined report with MySQL Shell, call the `shell.registerReport()` method in JavaScript or `shell.register_report()` in Python. The syntax for the method is as follows:

```
shell.registerReport(name, type, report[, description])
```

Where:

- `name` is a string giving the unique name of the report.
- `type` is a string giving the report type which determines the output format, either “list”, “report”, or “print”.
- `report` is the function to be called when the report is invoked.
- `description` is a dictionary with options that you can use to specify the options that the report supports, additional arguments that the report accepts, and help information that is provided in the MySQL Shell help system.

The `name`, `type`, and `report` parameters are all required. The report name must meet the following requirements:

- It must be unique in your MySQL Shell installation.
- It must be a valid scripting identifier, so the first character must be a letter or underscore character, followed by any number of letters, numbers, or underscore characters.
- It can be in mixed case, but it must still be unique in your MySQL Shell installation when converted to lower case.

The report name is not case-sensitive during the registration process and when running the report using the `\show` and `\watch` commands. The report name is case-sensitive when calling the corresponding API function at the `shell.reports` object. There you must call the function using the exact name that was used to register the report, whether you are in Python or JavaScript mode.

The optional dictionary contains the following keys, which are all optional:

`brief` — A brief description of the report.

`details` — A detailed description of the report, provided as an array of strings. This is provided when you use the `\help` command or the `--help` option with the `\show` command.

`options` — Any report-specific options that the report can accept. Each dictionary in the array describes one option, and must contain the following keys:

- `name` (string, required): The name of the option in the long form, which must be a valid scripting identifier.
- `brief` (string, optional): A brief description of the option.
- `shortcut` (string, optional): An alternate name for the option as a single alphanumeric character.
- `details` (array of strings, optional): A detailed description of the option. This is provided when you use the `\help` command or the `--help` option with the `\show` command.
- `type` (string, optional): The value type of the option. The permitted values are “string”, “bool”, “integer”, and “float”, with a default of “string” if `type` is not specified. If “bool” is specified, the option acts as a switch: it defaults to `false` if not specified, defaults to `true` (and accepts no value) when you run the report using the `\show` or `\watch` command, and must have a valid value when you run the report using the `shell.reports` object.

- `required` (bool, optional): Whether the option is required. If `required` is not specified, it defaults to `false`. If the option type is “bool” then `required` cannot be true.
- `values` (array of strings, optional): A list of allowed values for the option. Only options with type “string” can have this key. If `values` is not specified, the option accepts any values.

`argc`

A string specifying the number of additional arguments that the report expects, which can be one of the following:

- An exact number of arguments, which is specified as a single number.
- Zero or more arguments, which is specified as an asterisk.
- A range of argument numbers, which is specified as two numbers separated by a dash (for example, “1-5”).
- A range of argument numbers with a minimum but no maximum, which is specified as a number and an asterisk separated by a dash (for example, “1-*”).

10.1.3 Persisting MySQL Shell Reports

A MySQL Shell report must be saved with a file extension of `.js` for JavaScript code, or `.py` for Python code, to match the scripting language used for the report. The file extension is not case-sensitive.

The preferred way to persist a report is by adding it into a MySQL Shell plugin. Plugins and plugin groups are loaded automatically when MySQL Shell starts, and the functions that they define and register are available immediately. In a MySQL Shell plugin, the file containing the initialization script must be named `init.js` or `init.py` as appropriate for the language. For instructions to use MySQL Shell plugins, see Section 10.3, “MySQL Shell Plugins”.

As an alternative, scripts containing reports can be stored directly in the `init.d` folder in the MySQL Shell user configuration path. When MySQL Shell starts, all files found in the `init.d` folder with a `.js` or `.py` file extension are processed automatically and the functions in them are made available. (In this location, the file name does not matter to MySQL Shell.) The default MySQL Shell user configuration path is `~/.mysqlsh/` on Unix and `%AppData%\MySQL\mysqlsh\` on Windows. The user configuration path can be overridden on all platforms by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`.

10.1.4 Example MySQL Shell Report

This example user-defined report `sessions` shows which sessions currently exist.

```
def sessions(session, args, options):
    sys = session.get_schema('sys')
    session_view = sys.get_table('session')
    query = session_view.select(
        'thd_id', 'conn_id', 'user', 'db', 'current_statement',
        'statement_latency AS latency', 'current_memory AS memory')
    if (options.has_key('limit')):
        limit = int(options['limit'])
        query.limit(limit)

    result = query.execute()
    report = [result.get_column_names()]
    for row in result.fetch_all():
        report.append(list(row))
```

```
    return {'report': report}

shell.register_report(
    'sessions',
    'list',
    sessions,
    {
        'brief': 'Shows which sessions exist.',
        'details': ['You need the SELECT privilege on sys.session view and the underlying tables and fu
        'options': [
            {
                'name': 'limit',
                'brief': 'The maximum number of rows to return.',
                'shortcut': 'l',
                'type': 'integer'
            }
        ],
        'argc': '0'
    }
)
```

10.1.5 Running MySQL Shell Reports

Built-in reports and user-defined reports that have been registered with MySQL Shell can be run in any interactive MySQL Shell mode (JavaScript, Python, or SQL) using the `\show` or `\watch` command, or called using the `shell.reports` object from JavaScript or Python scripts. The `\show` command or `\watch` command with no parameters list all the available built-in and user-defined reports.

Using the Show and Watch Commands

To use the `\show` and `\watch` commands, an active MySQL session must be available.

The `\show` command runs the named report, which can be either a built-in MySQL Shell report or a user-defined report that has been registered with MySQL Shell. You can specify any options or additional arguments that the report supports. For example, the following command runs the built-in report `query`, which takes as an argument a single SQL statement:

```
\show query show session status
```

The report name is case-insensitive, and the dash and underscore characters are treated as the same.

The `\show` command also provides the following standard options:

- `--vertical` (or `-E`) displays the results from a report that returns a list in vertical format, instead of table format.
- `--help` displays any provided help for the named report. (Alternatively, you can use the `\help` command with the name of the report, which displays help for the report function.)

Standard options and report-specific options are given before the arguments. For example, the following command runs the built-in report `query` and returns the results in vertical format:

```
\show query --vertical show session status
```

The `\watch` command runs a report in the same way as the `\show` command, but then refreshes the results at regular intervals until you cancel the command using **Ctrl + C**. The `\watch` command has additional standard options to control the refresh behavior, as follows:

- `--interval=`*`float`* (or `-i` *`float`*) specifies a number of seconds to wait between refreshes. The default is 2 seconds. Fractional seconds can be specified, with a minimum interval of 0.1 second, and the interval can be set up to a maximum of 86400 seconds (24 hours).

  `--nocls` specifies that the screen is not cleared before refreshes, so previous results can still be seen.

For example, the following command uses the built-in report `query` to display the statement counter variables and refresh the results every 0.5 seconds:

```
\watch query --interval=0.5 show global status like 'Com%'
```

Quotes are interpreted by the command handler rather than directly by the server, so if they are used in a query, they must be escaped by preceding them with a backslash (`\`).

Using the `shell.reports` Object

Built-in MySQL Shell reports and user-defined reports that have been registered with MySQL Shell can also be accessed as API functions in the `shell.reports` object. The `shell.reports` object is available in JavaScript and Python mode, and uses the report name supplied during the registration as the function name. The function has the following signature:

```
Dict report(Session session, List argv, Dict options);
```

Where:

- `session` is a MySQL Shell session object that is to be used to execute the report.
- `argv` is a list containing string values of additional arguments that are passed to the report.
- `options` is a dictionary with key names and values that correspond to any report-specific options and their values. The short form of the options cannot be used with the `shell.reports` object.

The return value is a dictionary with the key `report`, and a list of JSON objects containing the report. For the List type of report, there is an element for each list, for the Report type there is a single element, and for the Print type there are no elements.

With the `shell.reports` object, if a dictionary of options is present, the `argv` list is required even if there are no additional arguments. Use the `\help` *`report_name`* command to display the help for the report function and check whether the report requires any arguments or options.

For example, the following code runs a user-defined report named `sessions` which shows the sessions that currently exist. A MySQL Shell session object is created to execute the report. A report-specific option is used to limit the number of rows returned to 10. There are no additional arguments, so the `argv` list is present but empty.

```
report = shell.reports.sessions(shell.getSession(), [], {'limit':10});
```

10.1.6 Built-in MySQL Shell Reports

MySQL Shell includes built-in reports to display the following information:

- The results of any specified SQL query (`query`, available from MySQL Shell 8.0.16).
- A listing of the current threads in the connected MySQL server (`threads`, available from MySQL Shell 8.0.18).
- Detailed information about a specified thread (`thread`, available from MySQL Shell 8.0.18).

As with user-defined reports, the built-in reports can be run once using the MySQL Shell `\show` command, or run and then refreshed continuously in a MySQL Shell session using the `\watch` command. The built-in reports support the standard options for the `\show` and `\watch` commands in addition to their report-specific options, unless noted otherwise in their descriptions. They can also be accessed as API functions using the `shell.reports` object. Section 10.1.5, “Running MySQL Shell Reports” explains how to run reports in each of these ways.

10.1.6.1 Built-in MySQL Shell Report: Query

The built-in MySQL Shell report `query` is available from MySQL Shell 8.0.16. It executes the single SQL statement that is provided as an argument, and returns the results using MySQL Shell's reporting

facility. You can use the `query` report as a convenient way to generate simple reports for your immediate use.

The `query` report has no report-specific options, but the standard options for the `\show` and `\watch` commands may be used, as described in Section 10.1.5, “Running MySQL Shell Reports”.

For example, the following command uses the `query` report to display the statement counter variables and refresh the results every 0.5 seconds:

```
\watch query --interval=0.5 show global status like 'Com%'
```

10.1.6.2 Built-in MySQL Shell Report: Threads

The built-in MySQL Shell report `threads` is available from MySQL Shell 8.0.18. It lists the current threads in the connected MySQL server which belong to the user account that is used to run the report. The report works with servers running all supported MySQL 5.7 and MySQL 8.0 versions. If any item of information is not available in the MySQL Server version of the target server, the report leaves it out.

The `threads` report provides information for each thread drawn from various sources including MySQL's Performance Schema. Using the report-specific options, you can choose to show foreground threads, background threads, or all threads. You can report a default set of information for each thread, or select specific information to include in the report from a larger number of available choices. You can filter, sort, and limit the output. For details of the report-specific options and the full listing of information that you can include in the report, issue one of the following MySQL Shell commands to view the report help:

```
\help threads
\show threads --help
```

In addition to the report-specific options, the `threads` report accepts the standard options for the `\show` and `\watch` commands, as described in Section 10.1.5, “Running MySQL Shell Reports”. The `threads` report is of the list type, and by default the results are returned as a table, but you can use the `--vertical` (or `-E`) option to display them in vertical format.

The `threads` report uses MySQL Server's `format_statement()` function (see The format_statement() Function). Any truncated statements displayed in the report are truncated according to the setting for the `statement_truncate_len` option in MySQL Server's `sys_config` table, which defaults to 64 characters.

The following list summarizes the capabilities provided by the report-specific options for the `threads` report. See the report help for full details and the short forms of the options:

`--foreground`, `--background`, `--all`
: List foreground threads only, background threads only, or all threads. The report displays a default set of appropriate fields for your thread type selection, unless you use the `--format` option to specify your own choice of fields instead.

`--format`
: Define your own custom set of information to display for each thread, specified as a comma-separated list of columns (and display names, if you want). The report help lists all of the columns that you can include to customize your report.

`--where`, `--order-by`, `--desc`, `--limit`
: Filter the returned results using logical expressions (`--where`), sort on selected columns (`--order-by`), sort in descending instead of ascending order `--desc`), or limit the number of returned threads (`--limit`).

For example, the following command runs the `threads` report to display all foreground threads, with a custom set of information comprising the thread ID, ID of any spawning thread, connection ID, user name and host name, client program name, type of command that the thread is executing, and memory allocated by the thread:

```
mysql-js> \show threads --foreground -o tid,ptid,cid,user,host,progname,command,memory
```

10.1.6.3 Built-in MySQL Shell Report: Thread

The built-in MySQL Shell report `thread` is available from MySQL Shell 8.0.18. It provides detailed information about a specific thread in the connected MySQL server. The report works with servers running all supported MySQL 5.7 and MySQL 8.0 versions. If any item of information is not available in the MySQL Server version of the target server, the report leaves it out.

The `thread` report provides information for the selected thread and its activity, drawn from various sources including MySQL's Performance Schema. By default, the report shows information on the thread used by the current connection, or you can identify a thread by its ID or by the connection ID. You can select one or more categories of information, or view all of the available information about the thread. For details of the report-specific options and the information that you can include in the report, issue one of the following MySQL Shell commands to view the report help:

```
\help thread
\show thread --help
```

In addition to the report-specific options, the `thread` report accepts most of the standard options for the `\show` and `\watch` commands, as described in Section 10.1.5, “Running MySQL Shell Reports”. The exception is the `--vertical` (or `-E`) option for the `\show` command, which is not accepted. The `thread` report has a custom output format that includes vertical listings and tables presented in different sections, and you cannot change this output format.

The `threads` report uses MySQL Server's `format_statement()` function (see The format_statement() Function). Any truncated statements displayed in the report are truncated according to the setting for the `statement_truncate_len` option in MySQL Server's `sys_config` table, which defaults to 64 characters.

The following list summarizes the capabilities provided by the report-specific options for the `threads` report. See the report help for full details and the short forms of the options:

`--tid`, `--cid`	Identify the thread ID or connection ID on which you want to report.
`--general`	Show basic information about the thread. This information is returned by default if you do not use any of the following options.
`--brief`	Show a brief description of the thread on one line.
`--client`	Show information about the client connection and client session.
`--innodb`	Show information about the current InnoDB transaction using the thread, if any.
`--locks`	Show information about locks blocking and blocked by the thread.
`--prep-stmts`	Show information about the prepared statements allocated for the thread.
`--status`	Show information about the session status variables for the thread. You can specify a list of prefixes to match, in which case only matching variables are displayed.
`--vars`	Show information about the session system variables for the thread. You can specify a list of prefixes to match, in which case only matching variables are displayed.
`--user-vars`	Show information about the user-defined variables for the thread. You can specify a list of prefixes to match, in which case only matching variables are displayed.

`--all`	Show all of the above information, except for the brief description.

For example, the following command runs the `thread` report for the thread with thread ID 53, and returns general information about the thread, details of the client connection, and information about any locks that the thread is blocking or is blocked by:

```
mysql-py>  \show thread --tid 53 --general --client --locks
```

10.2 Adding Extension Objects to MySQL Shell

From MySQL Shell 8.0.17, you can define extension objects and make them available as part of user-defined MySQL Shell global objects. When you create and register an extension object, it is available in both JavaScript and Python modes.

An extension object comprises one or more members. A member can be a basic data type value, a function written in native JavaScript or Python, or another extension object. You construct and register extension objects using functions provided by the built-in global object `shell`. You can continue to extend the object by adding further members to it after it has been registered with MySQL Shell.

Note

You can register an extension object containing functions directly as a MySQL Shell global object. However, for good management of your extension objects, it can be helpful to create one or a small number of top-level extension objects to act as entry points for all your extension objects, and to register these top-level extension objects as MySQL Shell global objects. You can then add your current and future extension objects as members of an appropriate top-level extension object. With this structure, a top-level extension object that is registered as a MySQL Shell global object provides a place for developers to add various extension objects created at different times and stored in different MySQL Shell plugins.

10.2.1 Creating User-Defined MySQL Shell Global Objects

To create a new MySQL Shell global object to act as an entry point for your extension objects, first create a new top-level extension object using the built-in `shell.createExtensionObject()` function in JavaScript or `shell.create_extension_object()` in Python:

```
shell.createExtensionObject()
```

Then register this top-level extension object as a MySQL Shell global object by calling the `shell.registerGlobal()` method in JavaScript or `shell.register_global()` in Python. The syntax for the method is as follows:

```
shell.registerGlobal(name, object[, definition])
```

Where:

- `name` is a string giving the name (and class) of the global object. The name must be a valid scripting identifier, so the first character must be a letter or underscore character, followed by any number of letters, numbers, or underscore characters. The name must be unique in your MySQL Shell installation, so it must not be the name of a built-in MySQL Shell global object (for example, `db`, `dba`, `cluster`, `session`, `shell`, `util`) and it must not be a name you have already used for a user-defined MySQL Shell global object. The examples below show how to check whether the name already exists before registering the global object.

Important

The name that you use to register the global object is used as-is when you access the object in both JavaScript and Python modes. It is therefore good practice to use a simple one-word name for the global object (for example,

> `ext`). If you register the global object with a complex name in camel case or snake case (for example, `myCustomObject`), when you use the global object, you must specify the name as it was registered. Only the names used for members are handled in a language-appropriate way.

- `object` is the extension object that you are registering as a MySQL Shell global object. You can only register an extension object once.
- `definition` is an optional dictionary with help information for the global object that is provided in the MySQL Shell help system. The dictionary contains the following keys:
  - `brief` (string, optional): A short description of the global object to be provided as help information.
  - `details` (list of strings, optional): A detailed description of the global object to be provided as help information.

10.2.2 Creating Extension Objects

To create a new extension object to provide one or more functions, data types, or further extension objects, use the built-in `shell.createExtensionObject()` function in JavaScript or `shell.create_extension_object()` in Python:

```
shell.createExtensionObject()
```

To add members to the extension object, use the built-in `shell.addExtensionObjectMember()` function in JavaScript or `shell.add_extension_object_member()` in Python:

```
shell.addExtensionObjectMember(object, name, member[, definition])
```

Where:

- `object` is the extension object where the new member is to be added.
- `name` is the name of the new member. The name must be a valid scripting identifier, so the first character must be a letter or underscore character, followed by any number of letters, numbers, or underscore characters. The name must be unique among the members that have already been added to the same extension object, and if the member is a function, the name does not have to match the name of the defined function. The name should preferably be specified in camel case, even if you are using Python to define and add the member. Specifying the member name in camel case enables MySQL Shell to automatically enforce naming conventions. MySQL Shell makes the member available in JavaScript mode using camel case, and in Python mode using snake case.
- `member` is the value of the new member, which can be any of the following:
  - A supported basic data type. The supported data types are “none” or “null”, “bool”, “number” (integer or floating point), “string”, “array”, and “dictionary”.
  - A JavaScript or Python function. You can use native code in the body of functions that are added as members to an extension object, provided that the interface (parameters and return values) is limited to the supported data types in Table 10.1, “Supported data type pairs for extension objects”. The use of other data types in the interface can lead to undefined behavior.
  - Another extension object.
- `definition` is an optional dictionary that can contain help information for the member, and also if the member is a function, a list of parameters that the function receives. Help information is defined using the following attributes:
  - `brief` is a brief description of the member.
  - `details` is a detailed description of the member, provided as a list of strings. This is provided when you use the MySQL Shell `\help` command.

Parameters for a function are defined using the following attribute:

- `parameters` is a list of dictionaries describing each parameter that the function receives. Each dictionary describes one parameter, and can contain the following keys:
  - `name` (string, required): The name of the parameter.
  - `type` (string, required): The data type of the parameter, one of “string”, “integer”, “bool”, “float”, “array”, “dictionary”, or “object”. If the type is “object”, the `class` or `classes` key can also be used. If the type is “string”, the `values` key can also be used. If the type is “dictionary”, the `options` key can also be used.
  - `class` (string, optional, allowed when data type is “object”): Defines the object type that is allowed as a parameter.
  - `classes` (list of strings, optional, allowed when data type is “object”): A list of classes defining the object types that are allowed as a parameter. The supported object types for `class` and `classes` are those that are exposed by the MySQL Shell APIs, for example `Session`, `ClassicSession`, `Table`, or `Collection`. An error is raised if an object type is passed to the function that is not in this list.
  - `values` (list of strings, optional, allowed when data type is “string”): A list of values that are valid for the parameter. An error is raised if a value is passed to the function that is not in this list.
  - `options` (list of options, optional, allowed when data type is “dictionary”): A list of options that are allowed for the parameter. Options use the same definition structure as the parameters, with the exception that if `required` is not specified for an option, it defaults to `false`. MySQL Shell validates the options specified by the end user and raises an error if an option is passed to the function that is not in this list. In MySQL Shell 8.0.17 through 8.0.19, this parameter is required when the data type is “dictionary”, but from MySQL Shell 8.0.20 it is optional. If you create a dictionary with no list of options, any options that the end user specifies for the dictionary are passed directly through to the function by MySQL Shell with no validation.
  - `required` (bool, optional): Whether the parameter is required. If `required` is not specified for a parameter, it defaults to `true`.
  - `brief` (string, optional): A short description of the parameter to be provided as help information.
  - `details` (list of strings, optional): A detailed description of the parameter to be provided as help information.

An extension object is considered to be under construction until it has been registered as a MySQL Shell global object, or added as a member to another extension object that is registered as a MySQL Shell global object. An error is returned if you attempt to use an extension object in MySQL Shell when it has not yet been registered.

Cross Language Considerations

An extension object can contain a mix of members defined in Python and members defined in JavaScript. MySQL Shell manages the transfer of data from one language to the other as parameters and return values. Table 10.1, “Supported data type pairs for extension objects” shows the data types that MySQL Shell supports when transferring data between languages, and the pairs that are used as representations of each other:

Table 10.1 Supported data type pairs for extension objects

JavaScript	Python
Boolean	Boolean
String	String

JavaScript	Python
Integer	Long
Number	Float
Null	None
Array	List
Map	Dictionary

An extension object is literally the same object in both languages.

10.2.3 Persisting Extension Objects

A script to define and register extension objects must have a file extension of `.js` for JavaScript code, or `.py` for Python code, to match the language used for the script. The file extension is not case-sensitive.

The preferred way to persist an extension object is by adding it into a MySQL Shell plugin. Plugins and plugin groups are loaded automatically when MySQL Shell starts, and the functions that they define and register are available immediately. In a MySQL Shell plugin, the file containing the initialization script must be named `init.js` or `init.py` as appropriate for the language. A plugin can only contain code in one language, so if you are creating an extension object with a mix of members defined in Python and members defined in JavaScript, you must store the members as separate language-appropriate plugins. For instructions to use MySQL Shell plugins, see Section 10.3, “MySQL Shell Plugins”.

As an alternative, scripts containing extension objects can be stored directly in the `init.d` folder in the MySQL Shell user configuration path. When MySQL Shell starts, all files found in the `init.d` folder with a `.js` or `.py` file extension are processed automatically and the functions that they register are made available. (In this location, the file name does not matter to MySQL Shell.) The default MySQL Shell user configuration path is `~/.mysqlsh/` on Unix and `%AppData%\MySQL\mysqlsh\` on Windows. The user configuration path can be overridden on all platforms by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`.

10.2.4 Example MySQL Shell Extension Objects

Example 10.1 Creating and Registering Extension Objects - Python

This example creates a function `hello_world()` which is made available through the user-defined MySQL Shell global object `demo`. The code creates a new extension object and adds the `hello_world()` function to it as a member, then registers the extension object as the MySQL Shell global object `demo`.

```
# Define a hello_world function that will be exposed by the global object 'demo'
def hello_world():
    print("Hello world!")

# Create an extension object where the hello_world function will be registered
plugin_obj = shell.create_extension_object()

shell.add_extension_object_member(plugin_obj, "helloWorld", hello_world,
                                  {"brief": "Prints 'Hello world!'", "parameters": []})

# Registering the 'demo' global object
shell.register_global("demo", plugin_obj,
                      {"brief": "A demo plugin that showcases MySQL Shell's plugin feature."})
```

Note that the member name is specified in camel case in the `shell.add_extension_object_member()` function. When you call the member in Python mode, use snake case for the member name, and MySQL Shell automatically handles the conversion. In JavaScript mode, the function is called like this:

```
mysql-js> demo.helloWorld()
```

In Python mode, the function is called like this:

```
mysql-py> demo.hello_world()
```

Example 10.2 Creating and Registering Extension Objects - JavaScript

This example creates an extension object with the function `listTables()` as a member, and registers it directly as the MySQL Shell global object `tools`:

```
// Define a listTables function that will be exposed by the global object tools

function listTables(session, schemaName, options) {
...
}

// Create an extension object and add the listTables function to it as a member

var object = shell.createExtensionObject()

shell.addExtensionObjectMember(object, "listTables", listTables,

                      {
                        brief:"Retrieves the tables from a given schema.",
                        details: ["Retrieves the tables of the schema named schemaName.",
                                  "If excludeCollections is true, the collection tables will not be ret
                        parameters:
                        [
                          {
                            name: "session",
                            type: "object",
                            class: "Session",
                            brief: "An X Protocol session object."
                          },
                          {
                            name: "schemaName",
                            type: "string",
                            brief: "The name of the schema from which the table list will be pulled."
                          },
                          {
                            name: "options",
                            type: "dictionary",
                            brief: "Additional options that affect the function behavior.",
                            options: [
                              {
                                name: "excludeViews",
                                type: "bool",
                                brief: "If set to true, the views will not be included on the list, def
                              },
                              {
                                name: "excludeCollections",
                                type: "bool",
                                brief: "If set to true, the collections will not be included on the lis
                              }
                            ]
                          },
                        ]
                      });


// Register the extension object as the global object "tools"

shell.registerGlobal("tools", object, {brief:"Global object for ExampleCom administrator tools",
                     details:[
                        "Global object to access homegrown ExampleCom administrator tools.",
                        "Add new tools to this global object as members with shell.addExtensionObjectMem
```

In JavaScript mode, the function is called like this:

```
mysql-js> tools.listTables(session, "world_x", {excludeViews: true})
```

In Python mode, the function is called like this:

```
mysql-py> tools.list_tables(session, "world_x", {"excludeViews": True})
```

10.3 MySQL Shell Plugins

From MySQL Shell 8.0.17, you can extend MySQL Shell with user-defined plugins that are loaded at startup. Plugins can be written in either JavaScript or Python, and the functions they contain are available in MySQL Shell in both JavaScript and Python modes.

10.3.1 Creating MySQL Shell Plugins

MySQL Shell plugins can be used to contain functions that are registered as MySQL Shell reports (see Section 10.1, “Reporting with MySQL Shell”), and functions that are members of extension objects that are made available by user-defined MySQL Shell global objects (see Section 10.2, “Adding Extension Objects to MySQL Shell”). A single plugin can contain and register more than one function, and can contain a mix of reports and members of extension objects. Functions that are registered as reports or members of extension objects by a MySQL Shell plugin are available immediately when MySQL has completed startup.

A MySQL Shell plugin is a folder containing an initialization script appropriate for the language (an `init.js` or `init.py` file). The initialization script is the entry point for the plugin. A plugin can only contain code in one language, so if you are creating an extension object with a mix of members defined in Python and members defined in JavaScript, you must store the members as separate language-appropriate plugins.

For a MySQL Shell plugin to be loaded automatically at startup, its folder must be located under the `plugins` folder in the MySQL Shell user configuration path. MySQL Shell searches for any initialization scripts in this location. MySQL Shell ignores any folders in the `plugins` location whose name begins with a dot (.) but otherwise the name you use for a plugin's folder is not important.

The default path for the `plugins` folder is `~/.mysqlsh/plugins` on Unix and `%AppData%\MySQL\mysqlsh\plugins` in Windows. The user configuration path can be overridden on all platforms by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`. The value of this variable replaces `%AppData%\MySQL\mysqlsh\` on Windows or `~/.mysqlsh/` on Unix.

When an error is found while loading plugins, a warning is shown and the error details are available in the MySQL Shell application log. To see more details on the loading process use the `--log-level=debug` option when starting MySQL Shell.

When a MySQL Shell plugin is loaded, the following objects are available as global variables:

- The built in global objects `shell`, `dba`, and `util`.
- The Shell API main module `mysql`.
- The X DevAPI main module `mysqlx`.
- The AdminAPI main module `dba`.

10.3.1.1 Common Code and Packages

If you use common code or inner packages in Python code that is part of a MySQL Shell plugin or plugin group, you must follow these requirements for naming and importing to avoid potential clashes between package names:

- The plugin or plugin group's top-level folder, and each inner folder that is to be recognized as a package, must be a valid regular package name according to Python's PEP 8 style guide, using only letters, numbers, and underscores.

- Each inner folder that is to be recognized as a package must contain a file named `__init__.py`.
- When importing, the full path for the package name must be specified. For example, if a plugin group named `ext` contains a plugin named `demo`, which has an inner package named `src` containing a module named `sample`, the module must be imported as follows:

```
from ext.demo.src import sample
```

10.3.2 Creating Plugin Groups

You can create a plugin group by placing the folders for multiple MySQL Shell plugins in a containing folder under the `plugins` folder. A plugin group can contain a mix of plugins defined using JavaScript and plugins defined using Python. Plugin groups can be used to organize plugins that have something in common, for example:

- Plugins that provide reports on a particular theme.
- Plugins that reuse the same common code.
- Plugins that add functions to the same extension object.

If a subdirectory of the `plugins` folder does not contain an initialization script (an `init.js` or `init.py` file), MySQL Shell treats it as a plugin group and searches its subfolders for the initialization scripts for the plugins. The containing folder can contain other files with code that is shared by the plugins in the plugin group. As for a plugin's subfolder, the containing folder is ignored if its name begins with a dot (.) but otherwise the name is not important to MySQL Shell.

For example, a plugin group comprising all the functions provided by the user-defined MySQL Shell global object `ext` can be structured like this:

- The folder `C:\Users\exampleuser\AppData\Roaming\MySQL\mysqlsh\plugins\ext` is the containing folder for the plugin group.
- Common code for the plugins is stored in this folder at `C:\Users\exampleuser\AppData\Roaming\MySQL\mysqlsh\plugins\ext\common.py`
- The plugins in the plugin group are stored in subfolders of the `ext` folder, each with an `init.py` file, for example `C:\Users\exampleuser\AppData\Roaming\MySQL\mysqlsh\plugins\ext\helloWorld\init.py`.
- The plugins import the common code from `ext.common` and use its functions.

10.3.3 Example MySQL Shell Plugins

Example 10.3 MySQL Shell plugin containing a report and an extension object

This example defines a function `show_processes()` to display the currently running processes, and a function `kill_process()` to kill a process with a specified ID. `show_processes()` is going to be a MySQL Shell report, and `kill_process()` is going to be a function provided by an extension object.

The code registers `show_processes()` as a MySQL Shell report `proc` using the `shell.register_report()` method. To register `kill_process()` as `ext.process.kill()`, the code checks whether the global object `ext` and the extension object `process` already exist, and creates and registers them if not. The `kill_process()` function is then added as a member to the `process` extension object.

The plugin code is saved as the file `~/.mysqlsh/plugins/ext/process/init.py`. At startup, MySQL Shell traverses the folders in the plugins folder, locates this `init.py` file, and executes the

code. The report `proc` and the function `kill()` are registered and made available for use. The global object `ext` and the extension object `process` are created and registered if they have not yet been registered by another plugin, otherwise the existing objects are used.

```
# Define a show_processes function that generates a MySQL Shell report

def show_processes(session, args, options):
  query = "SELECT ID, USER, HOST, COMMAND, INFO FROM INFORMATION_SCHEMA.PROCESSLIST"
  if (options.has_key('command')):
    query += " WHERE COMMAND = '%s'" % options['command']

  result = session.runSql(query).execute();
  report = []
  if (result.has_data()):
    report = [result.get_column_names()]
    for row in result.fetch_all():
        report.append(list(row))

  return {"report": report}


# Define a kill_process function that will be exposed by the global object 'ext'

def kill_process(session, id):
  result = session.runSql("KILL CONNECTION %d" % id).execute()


# Register the show_processes function as a MySQL Shell report


shell.register_report("proc", "list", show_processes, {"brief":"Lists the processes on the target server.",
                                                        "options": [{
                                                           "name": "command",
                                                           "shortcut": "c",
                                                           "brief": "Use this option to list processes over
                                                        }]})


# Register the kill_process function as ext.process.kill()

# Check if global object 'ext' has already been registered
if 'ext' in globals():
    global_obj = ext
else:
    # Otherwise register new global object named 'ext'
    global_obj = shell.create_extension_object()
    shell.register_global("ext", global_obj,
        {"brief":"MySQL Shell extension plugins."})

# Add the 'process' extension object as a member of the 'ext' global object
try:
    plugin_obj = global_obj.process
except IndexError:
    # If the 'process' extension object has not been registered yet, do it now
    plugin_obj = shell.create_extension_object()
    shell.add_extension_object_member(global_obj, "process", plugin_obj,
        {"brief": "Utility object for process operations."})

# Add the kill_process function to the 'process' extension object as member 'kill'
try:
    shell.add_extension_object_member(plugin_obj, "kill", kill_process, {"brief": "Kills the process with t
                                                                          "parameters": [
                                                                            {
                                                                              "name":"session",
                                                                              "type":"object",
                                                                              "class":"Session",
                                                                              "brief": "The session to be used on the c
                                                                            },
                                                                            {
                                                                              "name":"id",
                                                                              "type":"integer",
                                                                              "brief": "The ID of the process to be kil
```

```
                                                                    }
                                                                  ]
                                                                })
except Exception as e:
    shell.log("ERROR", "Failed to register ext.process.kill ({0}).".
       format(str(e).rstrip()))
```

Here, the user runs the report `proc` using the MySQL Shell `\show` command, then uses the `ext.process.kill()` function to stop one of the listed processes:

```
mysql-py> \show proc
+----+-----------------+-----------------+---------+-----------------------------------------------
| ID | USER            | HOST            | COMMAND | INFO
+----+-----------------+-----------------+---------+-----------------------------------------------
| 66 | root            | localhost:53998 | Query   | PLUGIN: SELECT ID, USER, HOST, COMMAND, INFO FROM
| 67 | root            | localhost:34022 | Sleep   | NULL
| 4  | event_scheduler | localhost       | Daemon  | NULL
+----+-----------------+-----------------+---------+-----------------------------------------------

mysql-py> ext.process.kill(session, 67)
mysql-py> \show proc
+----+-----------------+-----------------+---------+-----------------------------------------------
| ID | USER            | HOST            | COMMAND | INFO
+----+-----------------+-----------------+---------+-----------------------------------------------
| 66 | root            | localhost:53998 | Query   | PLUGIN: SELECT ID, USER, HOST, COMMAND, INFO FROM
| 4  | event_scheduler | localhost       | Daemon  | NULL
+----+-----------------+-----------------+---------+-----------------------------------------------
```

Chapter 11 MySQL Shell Utilities

Table of Contents

11.1 Upgrade Checker Utility 276
11.2 JSON Import Utility 281
11.2.1 Running the Utility 281
11.2.2 Importing JSON Documents With the Mysqlsh Command Interface 283
11.2.3 Importing JSON Documents With the `--import` Command 284
11.2.4 Conversions for Representations of BSON Data Types 285
11.3 Table Export Utility 286
11.4 Parallel Table Import Utility 293
11.5 Instance Dump Utility, Schema Dump Utility, and Table Dump Utility 302
11.6 Dump Loading Utility 324
11.7 Diagnostics Utilities 343
11.7.1 collectDiagnostics Utility 343
11.7.2 collectHighLoadDiagnostics Utility 346
11.7.3 collectSlowQueryDiagnostics Utility 347

MySQL Shell includes utilities for working with MySQL. To access the utilities from within MySQL Shell, use the `util` global object, which is available in JavaScript and Python modes, but not SQL mode. The `util` global object provides the following functions:

`checkForServerUpgrade()`	An upgrade checker utility that enables you to verify whether MySQL server instances are ready for upgrade. See Section 11.1, “Upgrade Checker Utility”.
`importJSON()`	A JSON import utility that enables you to import JSON documents to a MySQL Server collection or table. See Section 11.2, “JSON Import Utility”.
`exportTable()`	A table export utility that exports a MySQL relational table into a data file, which can then be uploaded into a table on a target MySQL server using MySQL Shell's parallel table import utility, or to import data to a different application, or as a light-weight logical backup for a single data table. See Section 11.3, “Table Export Utility”.
`importTable()`	A parallel table import utility that splits up a single data file and uses multiple threads to load the chunks into a MySQL table. See Section 11.4, “Parallel Table Import Utility”.
`dumpInstance()`, `dumpSchemas()`, `dumpTables()`	An instance dump utility, schema dump utility, and table dump utility that can export all schemas, a selected schema, or selected tables and views, from a MySQL instance into an Oracle Cloud Infrastructure Object Storage bucket or a set of local files. See Section 11.5, “Instance Dump Utility, Schema Dump Utility, and Table Dump Utility”.
`loadDump()`	A dump loading utility that can import schemas dumped using MySQL Shell's instance dump utility and schema dump utility into a MySQL instance. See Section 11.6, “Dump Loading Utility”.
`collectDiagnostics()`	A diagnostics generation utility that gathers information from the connected MySQL server, generates reports in TSV and YAML formats, and presents them in a zip archive in the location of your choice.

This utility enables you to retrieve diagnostic information from standalone servers, members of replication topologies, and MySQL HeatWave Service DB Systems.

See Section 11.7.1, “collectDiagnostics Utility”.

Important

As of MySQL Shell 8.0.32, MySQL Shell reads MySQL Server option files and login paths by default. As a result, if you connect to a MySQL Server which uses an option file, it will be used, by default, and attempt to create a global session using that configuration. If you do not want to use the options file, you must add `--no-defaults` to your command line.

11.1 Upgrade Checker Utility

The `util.checkForServerUpgrade()` function is an upgrade checker utility that enables you to verify whether MySQL server instances are ready for upgrade. You can select a target MySQL Server release to which you plan to upgrade, ranging from the first MySQL Server 8.0 General Availability (GA) release (8.0.11), up to the MySQL Server release number that matches the current MySQL Shell release number. The upgrade checker utility carries out the automated checks that are relevant for the specified target release, and advises you of further relevant checks that you should make manually.

- About the Utility
- Running the Utility
- JSON Output from the Upgrade Checker Utility

About the Utility

You can use the upgrade checker utility to check MySQL 5.7 server instances, and MySQL 8.0 server instances at another GA status release within the MySQL 8.0 release series, for compatibility errors and issues for upgrading. If you invoke `checkForServerUpgrade()` without specifying a MySQL Server instance, the instance currently connected to the global session is checked. To see the currently connected instance, issue the `\status` command.

Note

1. The upgrade checker utility does not support checking MySQL Server instances at a version earlier than MySQL 5.7.
2. MySQL Server only supports upgrade between GA releases. Upgrades from non-GA releases of MySQL 5.7 or 8.0 are not supported. For more information on supported upgrade paths, see Upgrade Paths.

The upgrade checker utility can check the configuration file (`my.cnf` or `my.ini`) for the server instance. The utility checks for any system variables that are defined in the configuration file but have been removed in the target MySQL Server release, and also for any system variables that are not defined in the configuration file and will have a different default value in the target MySQL Server release. For these checks, when you invoke `checkForServerUpgrade()`, you must provide the file path to the configuration file.

The upgrade checker utility can generate its output in text format, which is the default, or in JSON format, which might be simpler to parse and process for use in devops automation.

Running the Utility

The upgrade checker utility can operate over either an X Protocol connection or a classic MySQL protocol connection, using either TCP or Unix sockets. You can create the connection beforehand,

or specify it as arguments to the function. The utility always creates a new session to connect to the server, so the MySQL Shell global session is not affected.

Up to MySQL Shell 8.0.20, the user account that is used to run the upgrade checker utility must have `ALL` privileges. From MySQL Shell 8.0.21, the user account requires `RELOAD`, `PROCESS`, and `SELECT` privileges.

The upgrade checker utility has the following signature:

```
checkForServerUpgrade (ConnectionData connectionData, Dictionary options)
```

Both arguments are optional. The first provides connection data if the connection does not already exist, and the second is a dictionary that you can use to specify the following options:

`password` The password for the user account that is used to run the upgrade checker utility. You can provide the password using this dictionary option or as part of the connection details. If you do not provide the password, the utility prompts for it when connecting to the server.

`targetVersion` The target MySQL Server version to which you plan to upgrade. You can specify any release from 8.0.11 (the first MySQL Server 8.0 GA release) up to the MySQL Server release with the same version number as the MySQL Shell release that you are using. If you specify the short form version number 8.0, or omit the `targetVersion` option, the utility checks for upgrade to the MySQL Server release number that matches the release number for the MySQL Shell release that you are using.

`configPath` The local path to the `my.cnf` or `my.ini` configuration file for the MySQL server instance that you are checking, for example, `C:\ProgramData\MySQL\MySQL Server 8.0\my.ini`. If you omit the file path and the upgrade checker utility needs to run a check that requires the configuration file, that check fails with a message informing you that you must specify the file path.

`outputFormat` The format in which the output from the upgrade checker utility is returned. The default if you omit the option is text format (`TEXT`). If you specify `JSON`, well-formatted JSON output is returned instead, in the format listed in JSON Output from the Upgrade Checker Utility.

For example, the following commands verify then check the MySQL server instance currently connected to the global session, with output in text format:

```
mysqlsh> \status
MySQL Shell version 8.0.29
...
Server version:               5.7.33-log MySQL Community Server (GPL)
...
mysqlsh> util.checkForServerUpgrade()
```

The following command checks the MySQL server at URI `user@example.com:3306` for upgrade to MySQL Server release 8.0.27. The user password and the configuration file path are supplied as part of the options dictionary, and the output is returned in the default text format:

```
mysqlsh> util.checkForServerUpgrade('user@example.com:3306',
           {"password":"password", "targetVersion":"8.0.27", "configPath":"C:/ProgramData/MySQL/MySQL S
```

The following command checks the same MySQL server for upgrade to the MySQL Server release number that matches the current MySQL Shell release number (the default), and returns JSON output for further processing:

```
mysqlsh> util.checkForServerUpgrade('user@example.com:3306',
           {"password":"password", "outputFormat":"JSON", "configPath":"C:/ProgramData/MySQL/MySQL Serv
```

You can start the upgrade checker utility from the command line using the `mysqlsh` command interface. For information on this syntax, see Section 5.8, “API Command Line Integration”. The following example checks a MySQL server for upgrade to release 8.0.27, and returns JSON output:

```
mysqlsh -- util checkForServerUpgrade user@localhost:3306
                   --target-version=8.0.27 --output-format=JSON --config-path=/etc/mysql/my.cnf
```

The connection data can also be specified as named options grouped together by using curly brackets, as in the following example, which also shows that lower case and hyphens can be used for the method name rather than camelCase:

```
mysqlsh -- util check-for-server-upgrade { --user=user --host=localhost --port=3306 }
  --target-version=8.0.27 --output-format=JSON --config-path=/etc/mysql/my.cnf
```

The following example uses a Unix socket connection and shows the older format for invoking the utility from the command line, which is still valid:

```
./bin/mysqlsh --socket=/tmp/mysql.sock --user=user -e "util.checkForServerUpgrade()"
```

To get help for the upgrade checker utility, issue:

```
mysqlsh> util.help("checkForServerUpgrade")
```

`util.checkForServerUpgrade()` does not return a value (before MySQL Shell 8.0.13, the value 0, 1, or 2 was returned).

When you invoke the upgrade checker utility, MySQL Shell connects to the server instance and tests the settings described at Preparing Your Installation for Upgrade. For example:

```
The MySQL server at example.com:3306, version
5.7.33-enterprise-commercial-advanced - MySQL Enterprise Server - Advanced Edition (Commercial),
will now be checked for compatibility issues for upgrade to MySQL 8.0.37...

1) Usage of old temporal type
  No issues found

2) Usage of db objects with names conflicting with new reserved keywords
  Warning: The following objects have names that conflict with new reserved keywords.
  Ensure queries sent by your applications use `quotes` when referring to them or they will result in error
  More information: https://dev.mysql.com/doc/refman/en/keywords.html

  dbtest.System - Table name
  dbtest.System.JSON_TABLE - Column name
  dbtest.System.cube - Column name

3) Usage of utf8mb3 charset
  Warning: The following objects use the utf8mb3 character set. It is recommended to convert them to use
  utf8mb4 instead, for improved Unicode support.
  More information: https://dev.mysql.com/doc/refman/8.0/en/charset-unicode-utf8mb3.html

  dbtest.view1.col1 - column's default character set: utf8

4) Table names in the mysql schema conflicting with new tables in 8.0
  No issues found

5) Partitioned tables using engines with non native partitioning
  Error: In MySQL 8.0 storage engine is responsible for providing its own
  partitioning handler, and the MySQL server no longer provides generic
  partitioning support. InnoDB and NDB are the only storage engines that
  provide a native partitioning handler that is supported in MySQL 8.0. A
  partitioned table using any other storage engine must be altered—either to
  convert it to InnoDB or NDB, or to remove its partitioning—before upgrading
  the server, else it cannot be used afterwards.
  More information:
    https://dev.mysql.com/doc/refman/8.0/en/upgrading-from-previous-series.html#upgrade-configuration-chang

  dbtest.part1_hash - MyISAM engine does not support native partitioning

.....
```

```
Errors:   7
Warnings: 36
Notices:  0

7 errors were found. Please correct these issues before upgrading to avoid compatibility issues.
```

- In this example, the checks carried out on the server instance returned some errors for the upgrade scenario that were found on the checked server, so changes are required before the server instance can be upgraded to the target MySQL 8.0 release.
- When you have made the required changes to clear the error count for the report, you should also consider making further changes to remove the warnings. Those configuration improvements would make the server instance more compatible with the target release. The server instance can, however, be successfully upgraded without removing the warnings.
- The upgrade checker utility also provides advice and instructions for further relevant checks that cannot be automated and that you should make manually, which are rated as either warning or notice (informational) level.

JSON Output from the Upgrade Checker Utility

When you select JSON output using the `outputFormat` dictionary option, the JSON object returned by the upgrade checker utility has the following key-value pairs:

serverAddress	Host name and port number for MySQL Shell's connection to the MySQL server instance that was checked.
serverVersion	Detected MySQL version of the server instance that was checked.
targetVersion	Target MySQL version for the upgrade checks.
errorCount	Number of errors found by the utility.
warningCount	Number of warnings found by the utility.
noticeCount	Number of notices found by the utility.
summary	Text of the summary statement that would be provided at the end of the text output (for example, "No known compatibility errors or issues were found.").
checksPerformed	An array of JSON objects, one for each individual upgrade issue that was automatically checked (for example, usage of removed functions). Each JSON object has the following key-value pairs:

id	The ID of the check, which is a unique string.
title	A short description of the check.
status	"OK" if the check ran successfully, "ERROR" otherwise.
description	A long description of the check (if available) incorporating advice, or an error message if the check failed to run.
documentationLink	If available, a link to documentation with further information or advice.

	detectedProblems	An array (which might be empty) of JSON objects representing the errors, warnings, or notices that were found as a result of the check. Each JSON object has the following key-value pairs:	
		level	The message level, one of Error, Warning, or Notice.
		dbObject	A string identifying the database object to which the message relates.
		description	If available, a string with a specific description of the issue with the database object.
manualChecks	An array of JSON objects, one for each individual upgrade issue that is relevant to your upgrade path and needs to be checked manually (for example, the change of default authentication plugin in MySQL 8.0). Each JSON object has the following key-value pairs:		
	id	The ID of the manual check, which is a unique string.	
	title	A short description of the manual check.	

description	A long description of the manual check, with information and advice.
documentationLink	If available, a link to documentation with further information or advice.

11.2 JSON Import Utility

MySQL Shell's JSON import utility `util.importJSON()` enables you to import JSON documents from a file (or FIFO special file) or standard input to a MySQL Server collection or relational table. The utility checks that the supplied JSON documents are well-formed and inserts them into the target database, removing the need to use multiple `INSERT` statements or write scripts to achieve this task.

You can import the JSON documents to an existing table or collection or to a new one created for the import. If the target table or collection does not exist in the specified database, it is automatically created by the utility, using a default collection or table structure. The default collection is created by calling the `createCollection()` function from a `schema` object. The default table is created as follows:

```
CREATE TABLE `dbname`.`tablename` (
   target_column JSON,
   id INTEGER AUTO_INCREMENT PRIMARY KEY
) CHARSET utf8mb4 ENGINE=InnoDB;
```

The default collection name or table name is the name of the supplied import file (without the file extension), and the default `target_column` name is `doc`.

The JSON import utility can process BSON (binary JSON) data types that are represented in JSON documents. The data types used in BSON documents are not all natively supported by JSON, but can be represented using extensions to the JSON format. The import utility can process documents that use JSON extensions to represent BSON data types, convert them to an identical or compatible MySQL representation, and import the data value using that representation. The resulting converted data values can be used in expressions and indexes, and manipulated by SQL statements and X DevAPI functions.

To convert JSON extensions for BSON types into MySQL types, you must specify the `convertBsonTypes` option when you run the import utility. Additional options are available to control the mapping and conversion for specific BSON data types. If you import documents with JSON extensions for BSON types and do not use this option, the documents are imported in the same way as they are represented in the input file.

11.2.1 Running the Utility

The JSON import utility requires an existing X Protocol connection to the server. The utility cannot operate over a classic MySQL protocol connection.

In the MySQL Shell API, the JSON import utility is a function of the `util` global object, and has the following signature:

```
importJSON (path, options)
```

`path` is a string specifying the file path for the file containing the JSON documents to be imported. This can be a file written to disk, or a FIFO special file (named pipe). Standard input can only be imported with the `--import` command line invocation of the utility.

`options` is a dictionary of import options that can be omitted if it is empty. (Before MySQL 8.0.14, the dictionary was required.) The following options are available to specify where and how the JSON documents are imported:

`schema: "db_name"`	The name of the target database. If you omit this option, MySQL Shell attempts to identify and use the schema name in use for the current session, as specified in a URI-like connection string, `\use` command, or MySQL Shell option. If the schema name is not specified and cannot be identified from the session, an error is returned.
`collection: "collection_name"`	The name of the target collection. This is an alternative to specifying a table and column. If the collection does not exist, the utility creates it. If you specify none of the `collection`, `table`, or `tableColumn` options, the utility defaults to using or creating a target collection with the name of the supplied import file (without the file extension).
`table: "table_name"`	The name of the target table. This is an alternative to specifying a collection. If the table does not exist, the utility creates it.
`tableColumn: "column_name"`	The name of the column in the target table to which the JSON documents are imported. The specified column must be present in the table if the table already exists. If you specify the `table` option but omit the `tableColumn` option, the default column name `doc` is used. If you specify the `tableColumn` option but omit the `table` option, the name of the supplied import file (without the file extension) is used as the table name.
`convertBsonTypes: true`	Recognizes and converts BSON data types that are represented using extensions to the JSON format. The default for this option is `false`. When you specify `convertBsonTypes: true`, each represented BSON type is converted to an identical or compatible MySQL representation, and the data value is imported using that representation. Additional options are available to control the mapping and conversion for specific BSON data types; for a list of these control options and the default type conversions, see Section 11.2.4, “Conversions for Representations of BSON Data Types”. The `convertBsonOid` option must also be set to `true`, which is that option's default setting when you specify `convertBsonTypes: true`. If you import documents with JSON extensions for BSON types and do not use `convertBsonTypes: true`, the documents are imported in the same way as they are represented in the input file, as embedded JSON documents.
`convertBsonOid: true`	Recognizes and converts MongoDB ObjectIDs, which are a 12-byte BSON type used as an `_id` value for documents, represented in MongoDB Extended JSON strict mode. The default for this option is the value of the `convertBsonTypes` option, so if that option is set to `true`, MongoDB ObjectIDs are automatically also converted. When importing data from MongoDB, `convertBsonOid` must always be set to `true` if you do not convert the BSON types, because MySQL Server requires the `_id` value to be converted to the `varbinary(32)` type.
`extractOidTime: "field_name"`	Recognizes and extracts the timestamp value that is contained in a MongoDB ObjectID in the `_id` field for a document, and places it into a separate field in the imported data. `extractOidTime` names the field in the document that contains the timestamp. The timestamp is the first 4 bytes of the ObjectID, which remains unchanged. `convertBsonOid: true` must be set to use this option, which is the default when `convertBsonTypes` is set to true.

The following examples, the first in MySQL Shell's JavaScript mode and the second in MySQL Shell's Python mode, import the JSON documents in the file `/tmp/products.json` to the `products` collection in the `mydb` database:

```
mysql-js> util.importJson("/tmp/products.json", {schema: "mydb", collection: "products"})
```

```
mysql-py> util.import_json("/tmp/products.json", {"schema": "mydb", "collection": "products"})
```

The following example in MySQL Shell's JavaScript mode has no options specified, so the dictionary is omitted. `mydb` is the active schema for the MySQL Shell session. The utility therefore imports the JSON documents in the file `/tmp/stores.json` to a collection named `stores` in the `mydb` database:

```
mysql-js> \use mydb
mysql-js> util.importJson("/tmp/stores.json")
```

The following example in MySQL Shell's JavaScript mode imports the JSON documents in the file `/europe/regions.json` to the column `jsondata` in a relational table named `regions` in the `mydb` database. BSON data types that are represented in the documents by JSON extensions are converted to a MySQL representation:

```
mysql-js> util.importJson("/europe/regions.json", {schema: "mydb", table: "regions", tableColumn: "json
```

The following example in MySQL Shell's JavaScript mode carries out the same import but without converting the JSON representations of the BSON data types to MySQL representations. However, the MongoDB ObjectIDs in the documents are converted as required by MySQL, and their timestamps are also extracted:

```
mysql-js> util.importJson("/europe/regions.json", {schema: "mydb", table: "regions", tableColumn: "json
```

When the import is complete, or if the import is stopped partway by the user with **Ctrl+C** or by an error, a message is returned to the user showing the number of successfully imported JSON documents, and any applicable error message. The function itself returns void, or an exception in case of an error.

The JSON import utility can also be invoked from the command line. Two alternative formats are available for the command line invocation. You can use the `mysqlsh` command interface, which accepts input only from a file (or FIFO special file), or the `--import` command, which accepts input from standard input or a file. For instructions, see Section 11.2.2, “Importing JSON Documents With the Mysqlsh Command Interface” or Section 11.2.3, “Importing JSON Documents With the `--import` Command”.

11.2.2 Importing JSON Documents With the Mysqlsh Command Interface

With the `mysqlsh` command interface, you invoke the JSON import utility as follows:

```
mysqlsh user@host:port/mydb -- util importJson <path> [options]
or
mysqlsh user@host:port/mydb -- util import-json <path> [options]
```

For information on this syntax, see Section 5.8, “API Command Line Integration”. For the JSON import utility, specify the parameters as follows:

`user`	The user name for the user account that is used to run the JSON import utility.
`host`	The host name for the MySQL server.
`port`	The port number for MySQL Shell's connection to the MySQL server. The default port for this connection is 33060.
`mydb`	The name of the target database. When invoking the JSON import utility from the command line, you must specify the target database.

You can either specify it in the URI-like connection string, or using an additional --schema command line option.

path — The file path for the file (or FIFO special file) containing the JSON documents to be imported.

options — The --collection, --table, and --tableColumn options specify a target collection or a target table and column. The relationships and defaults when the JSON import utility is invoked using the mysqlsh command interface are the same as when the corresponding options are used in a MySQL Shell session. If you specify none of these options, the utility defaults to using or creating a target collection with the name of the supplied import file (without the file extension).

The --convertBsonTypes option converts BSON data types that are represented using extensions to the JSON format. The additional control options for specific BSON data types can also be specified; for a list of these control options and the default type conversions, see Section 11.2.4, "Conversions for Representations of BSON Data Types". The --convertBsonOid option is automatically set on when you specify --convertBsonTypes. When importing data from MongoDB, --convertBsonOid must be specified if you do not convert the BSON types, because MySQL Server requires the _id value to be converted to the varbinary(32) type. --extractOidTime=*field_name* can be used to extract the timestamp from the _id value into a separate field.

The following example imports the JSON documents in the file products.json to the products collection in the mydb database:

```
mysqlsh user@localhost/mydb -- util importJson products.json --collection=products
```

11.2.3 Importing JSON Documents With the --import Command

The --import command is available as an alternative to the mysqlsh command interface for command line invocation of the JSON import utility. This command provides a short form syntax without using option names, and it accepts JSON documents from standard input. The syntax is as follows:

```
mysqlsh user@host:port/mydb --import <path> [target] [tableColumn] [options]
```

As with the mysqlsh command interface, you must specify the target database, either in the URI-like connection string, or using an additional --schema command line option. The first parameter for the --import command is the file path for the file containing the JSON documents to be imported. To read JSON documents from standard input, specify a dash (-) instead of the file path. The end of the input stream is the end-of-file indicator, which is **Ctrl+D** on Unix systems and **Ctrl+Z** on Windows systems.

After specifying the path (or - for standard input), the next parameter is the name of the target collection or table. If standard input is used, you must specify a target.

- If you use standard input and the specified target is a relational table that exists in the specified schema, the documents are imported to it. You can specify a further parameter giving a column name, in which case the specified column is used for the import destination. Otherwise the default column name doc is used, which must be present in the existing table. If the target is not an existing table, the utility searches for any collection with the specified target name, and imports the documents to it. If no such collection is found, the utility creates a collection with the specified target name and imports the documents to it. To create and import to a table, you must also specify

a column name as a further parameter, in which case the utility creates a relational table with the specified table name and imports the data to the specified column.

- If you specify a file path and a target, the utility searches for any collection with the specified target name. If none is found, the utility by default creates a collection with that name and imports the documents to it. To import the file to a table, you must also specify a column name as a further parameter, in which case the utility searches for an existing relational table and imports to it, or creates a relational table with the specified table name and imports the data to the specified column.
- If you specify a file path but do not specify a target, the utility searches for any existing collection in the specified schema that has the name of the supplied import file (without the file extension). If one is found, the documents are imported to it. If no collection with the name of the supplied import file is found in the specified schema, the utility creates a collection with that name and imports the documents to it.

If you are importing documents containing representations of BSON (binary JSON) data types, you can also specify the options `--convertBsonOid`, `--extractOidTime=field_name`, `--convertBsonTypes`, and the control options listed in Section 11.2.4, “Conversions for Representations of BSON Data Types”.

The following example reads JSON documents from standard input and imports them to a target named `territories` in the `mydb` database. If no collection or table named `territories` is found, the utility creates a collection named `territories` and imports the documents to it. If you want to create and import the documents to a relational table named `territories`, you must specify a column name as a further parameter.

```
mysqlsh user@localhost/mydb --import - territories
```

The following example with a file path and a target imports the JSON documents in the file `/europe/regions.json` to the column `jsondata` in a relational table named `regions` in the `mydb` database. The schema name is specified using the `--schema` command line option instead of in the URI-like connection string:

```
mysqlsh user@localhost:33062 --import /europe/regions.json regions jsondata --schema=mydb
```

The following example with a file path but no target specified imports the JSON documents in the file `/europe/regions.json`. If no collection or table named `regions` (the name of the supplied import file without the extension) is found in the specified `mydb` database, the utility creates a collection named `regions` and imports the documents to it. If there is already a collection named `regions`, the utility imports the documents to it.

```
mysqlsh user@localhost/mydb --import /europe/regions.json
```

MySQL Shell returns a message confirming the parameters for the import, for example, `Importing from file "/europe/regions.json" to table `mydb`.`regions` in MySQL Server at 127.0.0.1:33062`.

When an import is complete, or if the import is stopped partway by the user with **Ctrl+C** or by an error, a message is returned to the user showing the number of successfully imported JSON documents, and any applicable error message. The process returns zero if the import finished successfully, or a nonzero exit code if there was an error.

11.2.4 Conversions for Representations of BSON Data Types

When you specify the `convertBsonTypes: true` (`--convertBsonTypes`) option to convert BSON data types that are represented by JSON extensions, by default, the BSON types are imported as follows:

Date (“date”)	Simple value containing the value of the field.
Timestamp (“timestamp”)	MySQL timestamp created using the `time_t` value.

Decimal (“decimal”)	Simple value containing a string representation of the decimal value.
Integer (“int” or “long”)	Integer value.
Regular expression (“regex” plus options)	String containing the regular expression only, and ignoring the options. A warning is printed if options are present.
Binary data (“binData”)	Base64 string.
ObjectID (“objectId”)	Simple value containing the value of the field.

The following control options can be specified to adjust the mapping and conversion of these BSON types. `convertBsonTypes: true` (`--convertBsonTypes`) must be specified to use any of these control options:

`ignoreDate: true` (`--ignoreDate`)	Disable conversion of the BSON “date” type. The data is imported as an embedded JSON document exactly as in the input file.
`ignoreTimestamp: true` (`--ignoreTimestamp`)	Disable conversion of the BSON “timestamp” type. The data is imported as an embedded JSON document exactly as in the input file.
`decimalAsDouble: true` (`--decimalAsDouble`)	Convert the value of the BSON “decimal” type to the MySQL `DOUBLE` type, rather than a string.
`ignoreRegex: true` (`--ignoreRegex`)	Disable conversion of regular expressions (the BSON “regex” type). The data is imported as an embedded JSON document exactly as in the input file.
`ignoreRegexOptions: false` (`--ignoreRegexOptions=false`)	Include the options associated with a regular expression in the string, as well as the regular expression itself (in the format `/<regular expression>/<options>`). By default, the options are ignored (`ignoreRegexOptions: true`), but a warning is printed if any options were present. `ignoreRegex` must be set to the default of `false` to specify `ignoreRegexOptions`.
`ignoreBinary: true` (`--ignoreBinary`)	Disable conversion of the BSON “binData” type. The data is imported as an embedded JSON document exactly as in the input file.

The following example imports documents from the file `/europe/regions.json` to the column `jsondata` in a relational table named `regions` in the `mydb` database. BSON data types that are represented by JSON extensions are converted to MySQL representations, with the exception of regular expressions, which are imported as embedded JSON documents:

```
mysqlsh user@localhost/mydb --import /europe/regions.json regions jsondata --convertBsonTypes --ignoreRegex
```

11.3 Table Export Utility

MySQL Shell's table export utility `util.exportTable()`, introduced in MySQL Shell 8.0.22, exports a MySQL relational table into a data file, either on the local server or in an Oracle Cloud Infrastructure Object Storage bucket. The data can then be uploaded into a table on a target MySQL server using MySQL Shell's parallel table import utility `util.importTable()` (see Section 11.4, “Parallel Table Import Utility”), which uses parallel connections to provide rapid data import for large data files. The data file can also be used to import data to a different application, or as a lightweight logical backup for a single data table.

- About the Utility
- Requirements and Restrictions

- Running the Utility
- Options
- Options for OCI Cloud Infrastructure
- Options for S3-compatible Services
- Options for Microsoft Azure Blob Storage

About the Utility

By default, the table export utility produces a data file in the default format for MySQL Shell's parallel table import utility. Preset options are available to export CSV files for either DOS or UNIX systems, and TSV files. The table export utility cannot produce JSON data. You can also set field- and line-handling options as for the `SELECT...INTO OUTFILE` statement to create data files in arbitrary formats.

`util.exportTable()` can be used with partitioned and subpartitioned tables, but does not perform any special handling of these. One file is always created per table by this utility, regardless of release version.

When choosing a destination for the table export file, note that for import into a MySQL HeatWave Service DB System, the MySQL Shell instance where you run the parallel table import utility must be installed on an Oracle Cloud Infrastructure Compute instance that has access to the MySQL HeatWave Service DB System. If you export the table to a file in an Object Storage bucket, you can access the Object Storage bucket from the Compute instance. If you create the table export file on your local system, you need to transfer it to the Oracle Cloud Infrastructure Compute instance using the copy utility of your choice, depending on the operating system you chose for your Compute instance.

Requirements and Restrictions

The following requirements apply to exports using the table export utility:

- MySQL 5.7 or later is required for the source MySQL instance and the destination MySQL instance.
- The upload method used to transfer files to an Oracle Cloud Infrastructure Object Storage bucket has a file size limit of 1.2 TiB.

Running the Utility

The table export utility uses the MySQL Shell global session to obtain the connection details of the target MySQL server from which the export is carried out. You must open the global session (which can have an X Protocol connection or a classic MySQL protocol connection) before running the utility. The utility opens its own session for each thread, copying options such as connection compression and SSL options from the global session, and does not make any further use of the global session. You can limit the maximum rate of data transfer to balance the load on the network.

In the MySQL Shell API, the table export utility is a function of the `util` global object, and has the following signature:

```
util.exportTable(table, outputUrl[, options])
```

`table` is the name of the relational data table to be exported to the data file. The table name can be qualified with a valid schema name, and quoted with the backtick character if needed. If the schema is omitted, the active schema for the MySQL Shell global session is used.

`options` is a dictionary of options that can be omitted if it is empty. The options are listed in the final section of this topic.

If you are exporting the data to the local filesystem, `outputUrl` is a string specifying the path to the exported data file, and the file name itself, with an appropriate extension. You can specify an absolute path or a path relative to the current working directory. You can prefix a local directory path with the `file://` schema. In this example in MySQL Shell's JavaScript mode, the user exports the `employees` table from the `hr` schema using the default dialect. The file is written to the `exports` directory in the user's home directory, and is given a `.txt` extension that is appropriate for a file in this format:

```
shell-js> util.exportTable("hr.employees", "file:///home/hanna/exports/employees.txt")
```

The target directory must exist before the export takes place, but it does not have to be empty. If the exported data file already exists there, it is overwritten. For an export to a local directory, the data file is created with the access permissions `rw-r-----` (on operating systems where these are supported). The owner of the file is the user account that is running MySQL Shell.

If you are exporting the data to an Oracle Cloud Infrastructure Object Storage bucket, or to S3-compatible storage,`outputUrl` is the name for the data file in the bucket, including a suitable file extension. You can include directory separators to simulate a directory structure. Use the `osBucketName` option to provide the name of the Object Storage bucket, and the `osNamespace` option to identify the namespace for the bucket. In this example in MySQL Shell's Python mode, the user exports the `employees` table from the `hr` schema as a file in TSV format to the Object Storage bucket `hanna-bucket`:

```
shell-py> util.export_table("hr.employees", "dump/employees.tsv", {
        > dialect: "tsv", "osBucketName": "hanna-bucket", "osNamespace": "idx28w1ckztq" })
```

The namespace for an Object Storage bucket is displayed in the **Bucket Information** tab of the bucket details page in the Oracle Cloud Infrastructure console, or can be obtained using the Oracle Cloud Infrastructure command line interface. A connection is established to the Object Storage bucket using the default profile in the default Oracle Cloud Infrastructure CLI configuration file, or alternative details that you specify using the `ociConfigFile` and `ociProfile` options. For instructions to set up a CLI configuration file, see SDK and CLI Configuration File.

Options

`where: "string"`

A valid SQL condition expression used to filter the data being exported.

Note

The SQL is validated only when it is executed. If you are exporting many tables, any SQL-syntax-related issues will only be seen late in the process. As such, it is recommended you test your SQL condition before using it in a long-running export process.

In the following example, `where` exports only those rows of the table `sakila.actor` where the value of `actor_id` is greater than 150, to a file named `dump.csv`:

```
util.exportTable("sakila.actor", "dump.csv", {"where" : "acto
```

`partitions: ["string","string",..]`

A list of valid partition names which limits the export to the specified partitions.

The following example exports the partitions p1 and p2 from `schema.table` to a file named `dump.csv`:

```
util.exportTable("schema.table", "dump.csv", {"partitions
```

`dialect: [default|csv|csv-unix|tsv]`

Specify a set of field- and line-handling options for the format of the exported data file. You can use the selected dialect as a base for further customization, by also specifying one or more of the `linesTerminatedBy`, `fieldsTerminatedBy`, `fieldsEnclosedBy`, `fieldsOptionallyEnclosed`, and `fieldsEscapedBy` options to change the settings.

The default dialect produces a data file matching what would be created using a `SELECT...INTO OUTFILE` statement with the default settings for that statement. `.txt` is an appropriate file extension to assign to these output files. Other dialects are available to export CSV files for either DOS or UNIX systems (`.csv`), and TSV files (`.tsv`).

The settings applied for each dialect are as follows:

Table 11.1 Dialect settings for table export utility

dialect	linesTerminatedBy	fieldsTerminatedBy	fieldsEnclosedBy	fieldsOptionallyEnclosed	fieldsEscapedBy
default	[LF]	[TAB]	[empty]	false	\
csv	[CR][LF]	,	"	true	\
csv-unix	[LF]	,	"	false	\
tsv	[CR][LF]	[TAB]	"	true	\

Note

1. The carriage return and line feed values for the dialects are operating system independent.
2. If you use the `linesTerminatedBy`, `fieldsTerminatedBy`, `fieldsEnclosedBy`, `fieldsOptionallyEnclosed`, and `fieldsEscapedBy` options, depending on the escaping conventions of your command interpreter, the backslash character (\) might need to be doubled if you use it in the option values.
3. Like the MySQL server with the `SELECT...INTO OUTFILE` statement, MySQL Shell does not validate the field- and line-handling options that you specify. Inaccurate selections for these options can cause data to be exported partially or incorrectly. Always verify your settings before starting the export, and verify the results afterwards.

`linesTerminatedBy: "characters"`

One or more characters (or an empty string) with which the utility terminates each of the lines in the exported data file. The default is as for the specified dialect, or a linefeed character (`\n`) if the dialect option is omitted. This option is equivalent to the `LINES`

	`TERMINATED BY` option for the `SELECT...INTO OUTFILE` statement. Note that the utility does not provide an equivalent for the `LINES STARTING BY` option for the `SELECT...INTO OUTFILE` statement, which is set to the empty string.
`fieldsTerminatedBy: "`*`characters`*`"`	One or more characters (or an empty string) with which the utility terminates each of the fields in the exported data file. The default is as for the specified dialect, or a tab character (`\t`) if the dialect option is omitted. This option is equivalent to the `FIELDS TERMINATED BY` option for the `SELECT...INTO OUTFILE` statement.
`fieldsEnclosedBy: "`*`character`*`"`	A single character (or an empty string) with which the utility encloses each of the fields in the exported data file. The default is as for the specified dialect, or the empty string if the dialect option is omitted. This option is equivalent to the `FIELDS ENCLOSED BY` option for the `SELECT...INTO OUTFILE` statement.
`fieldsOptionallyEnclosed: [ true | false ]`	Whether the character given for `fieldsEnclosedBy` is to enclose all of the fields in the exported data file (`false`), or to enclose a field only if it has a string data type such as `CHAR`, `BINARY`, `TEXT`, or `ENUM` (`true`). The default is as for the specified dialect, or `false` if the dialect option is omitted. This option makes the `fieldsEnclosedBy` option equivalent to the `FIELDS OPTIONALLY ENCLOSED BY` option for the `SELECT...INTO OUTFILE` statement.
`fieldsEscapedBy: "`*`character`*`"`	The character that is to begin escape sequences in the exported data file. The default is as for the specified dialect, or a backslash (\) if the dialect option is omitted. This option is equivalent to the `FIELDS ESCAPED BY` option for the `SELECT...INTO OUTFILE` statement. If you set this option to the empty string, no characters are escaped, which is not recommended because special characters used by `SELECT...INTO OUTFILE` must be escaped.
`maxRate: "`*`string`*`"`	The maximum number of bytes per second per thread for data read throughput during the export. The unit suffixes `k` for kilobytes, `M` for megabytes, and `G` for gigabytes can be used (for example, setting `100M` limits throughput to 100 megabytes per second per thread). Setting `0` (which is the default value), or setting the option to an empty string, means no limit is set.
`showProgress: [ true | false ]`	Display (`true`) or hide (`false`) progress information for the export. The default is `true` if `stdout` is a terminal (`tty`), such as when MySQL Shell is in interactive mode, and `false` otherwise. The progress information includes the estimated total number of rows to be exported, the number of rows exported so far, the percentage complete, and the throughput in rows and bytes per second.
`compression: "`*`string`*`"`	The compression type to use when writing the exported data file. The default is to use no compression (`none`). The alternatives are to use gzip compression (`gzip`) or zstd compression (`zstd`).
`defaultCharacterSet: "`*`string`*`"`	The character set to be used during the session connections that are opened by MySQL Shell to the server for the export. The default is `utf8mb4`. The session value of the system variables `character_set_client`, `character_set_connection`, and `character_set_results` are set to this value for each connection. The character set must be permitted by the

`character_set_client` system variable and supported by the MySQL instance.

Options for OCI Cloud Infrastructure

`osBucketName: "string"` The name of the Oracle Cloud Infrastructure Object Storage bucket to which the exported data file is to be written. By default, the `[DEFAULT]` profile in the Oracle Cloud Infrastructure CLI configuration file located at `~/.oci/config` is used to establish a connection to the bucket. You can substitute an alternative profile to be used for the connection with the `ociConfigFile` and `ociProfile` options. For instructions to set up a CLI configuration file, see SDK and CLI Configuration File.

`osNamespace: "string"` The Oracle Cloud Infrastructure namespace where the Object Storage bucket named by `osBucketName` is located. The namespace for an Object Storage bucket is displayed in the Bucket Information tab of the bucket details page in the Oracle Cloud Infrastructure console, or can be obtained using the Oracle Cloud Infrastructure command line interface.

`ociConfigFile: "string"` An Oracle Cloud Infrastructure CLI configuration file that contains the profile to use for the connection, instead of the one in the default location `~/.oci/config`.

`ociProfile: "string"` The profile name of the Oracle Cloud Infrastructure profile to use for the connection, instead of the `[DEFAULT]` profile in the Oracle Cloud Infrastructure CLI configuration file used for the connection.

Options for S3-compatible Services

MySQL Shell supports exporting tables to S3-compatible buckets, such as Amazon Web Services (AWS) S3.

Note

MySQL Shell supports AWS S3 configuration in command line options, environment variables, and configuration files. Command line options override environment variables, configuration files, and default options.

For information on configuration requirements, see Section 4.7, “Cloud Service Configuration”.

`s3BucketName: "string"` The name of the S3 bucket to which the export is to be written. By default, the `default` profile of the `config` and `credentials` files located at `~/.aws/` are used to establish a connection to the S3 bucket. You can substitute alternative configurations and credentials for the connection with the `s3ConfigFile` and `s3CredentialsFile` options. For instructions on installing and configuring the AWS CLI, see Getting started with the AWS CLI.

`s3CredentialsFile:"string"` A credentials file that contains the user's credentials to use for the connection, instead of the one in the default location, `~/.aws/credentials`. Typically, the credentials file contains the `aws_access_key_id` and `aws_secret_access_key` to use for the connection.

`s3ConfigFile: "string"` A configuration file that contains the profile to use for the connection, instead of the one in the default location, such as

`~/.aws/config`. Typically, the config file contains the region and output type to use for the connection.

`s3Profile: "string"` — The profile name of the s3 CLI profile to use for the connection, instead of the `default` profile.

`s3Region: "string"` — The name of the region to use for the connection.

`s3EndpointOverride: "string"` — The URL of the endpoint to use instead of the default.

When connecting to the Oracle Cloud Infrastructure S3 compatbility API, the endpoint takes the following format: `https://namespace.compat.objectstorage.region.oraclecloud.co` Replace *`namespace`* with the Object Storage namespace and *`region`* with your region identifier. For example, the region identifier for the US East (Ashburn) region is `us-ashburn-1`.

For a namespace named axaxnpcrorw5 in the US East (Ashburn) region:

`https://axaxnpcrorw5.compat.objectstorage.us-ashburn-1.oraclecloud.com`.

Options for Microsoft Azure Blob Storage

MySQL Shell supports exporting to Microsoft Azure Blob Storage.

Note

MySQL Shell supports Microsoft Azure Blob Storage configuration in command line options, environment variables, and configuration files. Command line options override environment variables, and configuration files.

For information on configuration requirements and the order of precedence of the configuration types, see Section 4.7, “Cloud Service Configuration”.

`azureContainerName: "string"` — Mandatory. The name of the Azure container to which the export is to be written. The container must exist.

`azureConfigFile: "string"` — Optional. A configuration file that contains the storage connection parameters, instead of the one in the default location, such as `~/.azure/config`. If this is not defined, the default configuration file is used.

`azureContainerName` must be defined, and not be empty.

`azureStorageAccount: "string"` — Optional. The name of the Azure storage account to use for the operation.

`azureStorageSasToken: "string"` — Optional. Azure Shared Access Signature (SAS) token to be used for the authentication of the operation, instead of a key.

In the following example, the configuration uses a configuration string for the connection parameters, which means the `exportTable` command requires only the `azureContainerName`.

Example `config` file:

```
[cloud]
 name = AzureCloud

[storage]
 connection_string=alphanumericConnectionString
```

Example `exportTable` command, which exports the `sakila.actor` table, as a TSV file, to a container named *`mysqlshellazure`*:

```
        util.exportTable("sakila.actor", "actor.tsv", {dialect: "tsv", azureContainerName: "mysqlshella
```

11.4 Parallel Table Import Utility

MySQL Shell's parallel table import utility `util.importTable()` provides rapid data import to a MySQL relational table for large data files. The utility analyzes an input data file, distributes it into chunks, and uploads the chunks to the target MySQL server using parallel connections. The utility is capable of completing a large data import many times faster than a standard single-threaded upload using a `LOAD DATA` statement.

- About the Utility
- Requirements and Restrictions
- Running the Utility
- Options for Importing Tables
- Options for Oracle Cloud Infrastructure
- Options for S3-Compatible Services
- Options for Microsoft Azure Blob Storage

About the Utility

MySQL Shell's parallel table import utility supports the output from MySQL Shell's table export utility, which can compress the data file it produces as output, and can export it to a local folder or an Object Storage bucket. The default dialect for the parallel table import utility is the default for the output file produced by the table export utility. The parallel table import utility can also be used to upload files from other sources.

The data file or files to be imported can be in any of the following locations:

- A location that is accessible to the client host as a local disk.
- A remote location that is accessible to the client host through HTTP or HTTPS, specified with a URL. Pattern matching is not supported for files accessed in this way.
- An Oracle Cloud Infrastructure Object Storage bucket (from MySQL Shell 8.0.21).

The data is imported to a single relational table in the MySQL server to which the active MySQL session is connected.

When you run the parallel table import utility, you specify the mapping between the fields in the data file or files, and the columns in the MySQL table. You can set field- and line-handling options as for the `LOAD DATA` statement to handle data files in arbitrary formats. For multiple files, all the files must be in the same format. The default dialect for the utility maps to a file created using a `SELECT...INTO OUTFILE` statement with the default settings for that statement. The utility also has preset dialects that map to the standard data formats for CSV files (created on DOS or UNIX systems), TSV files, and JSON, and you can customize these using the field- and line-handling options as necessary. Note that JSON data must be in document-per-line format.

A number of functions have been added to the parallel table import utility since it was introduced, so use the most recent version of MySQL Shell to get the utility's full functionality.

Input preprocessing

From MySQL Shell 8.0.22, the parallel table import utility can capture columns from the data file or files for input preprocessing, in the same way as with a `LOAD DATA` statement. The selected data

	can be discarded, or you can transform the data and assign it to a column in the target table.
Oracle Cloud Infrastructure Object Storage import	Up to MySQL Shell 8.0.20, the data must be imported from a location that is accessible to the client host as a local disk. From MySQL Shell 8.0.21, the data can also be imported from an Oracle Cloud Infrastructure Object Storage bucket, specified by the `osBucketName` option.
Multiple data file import	Up to MySQL Shell 8.0.22, the parallel table import utility can import a single input data file to a single relational table. From MySQL Shell 8.0.23, the utility is also capable of importing a specified list of files, and it supports wildcard pattern matching to include all relevant files from a location. Multiple files uploaded by a single run of the utility are placed into a single relational table, so for example, data that has been exported from multiple hosts could be merged into a single table to be used for analytics.
Compressed file handling	Up to MySQL Shell 8.0.21, the parallel table import utility only accepts an uncompressed input data file. The utility analyzes the data file, distributes it into chunks, and uploads the chunks to the relational table in the target MySQL server, dividing the chunks up between the parallel connections. From MySQL Shell 8.0.22, the utility can also accept data files compressed in the `gzip` (`.gz`) and `zstd` (`.zst`) formats, detecting the format automatically based on the file extension. The utility uploads a compressed file from storage in the compressed format, saving bandwidth for that part of the transfer. Compressed files cannot be distributed into chunks, so instead the utility uses its parallel connections to decompress and upload multiple files simultaneously to the target server. If there is only one input data file, the upload of a compressed file can only use a single connection.

MySQL Shell's dump loading utility `util.loadDump()` is designed to import the combination of chunked output files and metadata produced by MySQL Shell's instance dump utility `util.dumpInstance()`, schema dump utility `util.dumpSchemas()`, and table dump utility `util.dumpTables()`. The parallel table import utility can be used in combination with the dump loading utility if you want to modify any of the data in the chunked output files before uploading it to the target server. To do this, first use the dump loading utility to load only the DDL for the selected table, to create the table on the target server. Then use the parallel table import utility to capture and transform data from the output files for the table, and import it to the target table. Repeat that process as necessary for any other tables where you want to modify the data. Finally, use the dump loading utility to load the DDL and data for any remaining tables that you do not want to modify, excluding the tables that you did modify. For a description of the procedure, see Modifying Dumped Data.

Requirements and Restrictions

Note

The parallel table import utility requires a classic connection to the target server. It does not currently support X Protocol connections.

The parallel table import utility uses `LOAD DATA LOCAL INFILE` statements to upload data, so the `local_infile` system variable must be set to `ON` on the target server. You can do this by issuing the following statement in SQL mode before running the parallel table import utility:

```
SET GLOBAL local_infile = 1;
```

To avoid a known potential security issue with `LOAD DATA LOCAL`, when the MySQL server replies to the parallel table import utility's `LOAD DATA` requests with file transfer requests, the utility only sends

the predetermined data chunks, and ignores any specific requests attempted by the server. For more information, see Security Considerations for LOAD DATA LOCAL.

Running the Utility

The parallel table import utility requires an existing classic MySQL protocol connection to the target MySQL server. Each thread opens its own session to send chunks of the data to the MySQL server, or in the case of compressed files, to send multiple files in parallel. You can adjust the number of threads, number of bytes sent in each chunk, and maximum rate of data transfer per thread, to balance the load on the network and the speed of data transfer. The utility cannot operate over X Protocol connections, which do not support `LOAD DATA` statements.

In the MySQL Shell API, the parallel table import utility is a function of the `util` global object, and has the following signature:

```
importTable ({file_name | file_list}, options)
```

`options` is a dictionary of import options that can be omitted if it is empty. The options are listed in the final section of this topic.

`file_name` is a string specifying the name and path for a single file containing the data to be imported. Alternatively, `file_list` is an array of file paths specifying multiple data files. On Windows, backslashes must be escaped in file paths, or you can use forward slashes instead.

- For files that are accessible to the client host on a local disk, you can prefix the directory path with the `file://` schema, or allow it to default to that. For files accessed in this way, file paths can contain the wildcards `*` (multiple characters) and `?` (single character) for pattern matching. Note that if these wildcard characters are present in file paths, the utility treats them as wildcards and might therefore attempt an incorrect strategy for file transfer.
- For files that are accessible to the client host through HTTP or HTTPS, provide a URL or a list of URLs, prefixed with the `http://` or `https://` schema as appropriate, in the format `http[s]://host.domain[:port]/path`. For files accessed in this way, pattern matching is not available. The HTTP server must support the Range request header, and must return the Content-Range response header to the client.
- For files in an Oracle Cloud Infrastructure Object Storage bucket, specify a path to the file in the bucket, and use the `osBucketName` option to specify the bucket name.

The function returns void, or an exception in case of an error. If the import is stopped partway by the user with **Ctrl+C** or by an error, the utility stops sending data. When the server finishes processing the data it received, messages are returned showing the chunk that was being imported by each thread at the time, the percentage complete, and the number of records that were updated in the target table.

The following examples, the first in MySQL Shell's JavaScript mode and the second in MySQL Shell's Python mode, import the data in a single CSV file `/tmp/productrange.csv` to the `products` table in the `mydb` database, skipping a header row in the file:

```
mysql-js> util.importTable("/tmp/productrange.csv", {schema: "mydb", table: "products", dialect: "csv-u
```

```
mysql-py> util.import_table("/tmp/productrange.csv", {"schema": "mydb", "table": "products", "dialect":
```

The following example in MySQL Shell's Python mode only specifies the dialect for the CSV file. `mydb` is the active schema for the MySQL Shell session. The utility therefore imports the data in the file `/tmp/productrange.csv` to the `productrange` table in the `mydb` database:

```
mysql-py> \use mydb
mysql-py> util.import_table("/tmp/productrange.csv", {"dialect": "csv-unix"})
```

The following example in MySQL Shell's Python mode imports the data from multiple files, including a mix of individually named files, ranges of files specified using wildcard pattern matching, and compressed files:

```
mysql-py> util.import_table(
```

```
    [
        "data_a.csv",
        "data_b*",
        "data_c*",
        "data_d.tsv.zst",
        "data_e.tsv.zst",
        "data_f.tsv.gz",
        "/backup/replica3/2021_01_12/data_g.tsv",
        "/backup/replica3/2021_01_13/*.tsv",
    ],
    {"schema": "mydb", "table": "productrange"}
)
```

The parallel table import utility can also be invoked from the command line using the `mysqlsh` command interface. With this interface, you invoke the utility as in the following examples:

```
mysqlsh mysql://root:@127.0.0.1:3366 --ssl-mode=DISABLED -- util import-table /r/mytable.dump --schema=mydb
```

When you import multiple data files, ranges of files specified using wildcard pattern matching are expanded by MySQL Shell's glob pattern matching logic if they are quoted, as in the following example. Otherwise they are expanded by the pattern matching logic for the user shell where you entered the `mysqlsh` command.

```
mysqlsh mysql://root:@127.0.0.1:3366 -- util import-table data_a.csv "data_b*" data_d.tsv.zst --schema=mydb
```

When you use the `mysqlsh` command's API reference argument to directly invoke the parallel table import utility (the dash-dash-space sequence "`--` "), before MySQL Shell 8.0.24, the `columns` option is not supported because array values are not accepted, so the input lines in your data file must contain a matching field for every column in the target table. From MySQL Shell 8.0.24, the option is supported, and you can specify columns using a dictionary argument . Also note that as shown in the above example, line feed characters must be passed using ANSI-C quoting in shells that support this function (such as `bash`, `ksh`, `mksh`, and `zsh`). For information on the `mysqlsh` command-line integration, see Section 5.8, “API Command Line Integration”.

Options for Importing Tables

The following import options are available for the parallel table import utility to specify how the data is imported:

`schema: "db_name"`
: The name of the target database on the connected MySQL server. If you omit this option, the utility attempts to identify and use the schema name in use for the current MySQL Shell session, as specified in a connection URI string, `\use` command, or MySQL Shell option. If the schema name is not specified and cannot be identified from the session, an error is returned.

`table: "table_name"`
: The name of the target relational table. If you omit this option, the utility assumes the table name is the name of the data file without the extension. The target table must exist in the target database.

`columns: array of column names`
: An array of strings containing column names from the import file or files, given in the order that they map to columns in the target relational table. Use this option if the imported data does not contain all the columns of the target table, or if the order of the fields in the imported data differs from the order of the columns in the table. If you omit this option, input lines are expected to contain a matching field for each column in the target table.

  From MySQL Shell 8.0.22, you can use this option to capture columns from the import file or files for input preprocessing, in the same way as with a `LOAD DATA` statement. When you use an integer value in place of a column name in the array, that column in the import file or files is captured as a user variable `@int`, for

example `@1`. The selected data can be discarded, or you can use the `decodeColumns` option to transform the data and assign it to a column in the target table.

In this example in MySQL Shell's JavaScript mode, the second and fourth columns from the import file are assigned to the user variables `@1` and `@2`, and no `decodeColumns` option is present to assign them to any column in the target table, so they are discarded.

```
mysql-js> util.importTable('file.txt', {
            table: 't1',
            columns: ['column1', 1, 'column2', 2, 'column3']
          });
```

`decodeColumns: dictionary`

A dictionary of key-value pairs that assigns import file columns captured as user variables by the `columns` option to columns in the target table, and specifies preprocessing transformations for them in the same way as the `SET` clause of a `LOAD DATA` statement. This option is available from MySQL Shell 8.0.22.

In this example in MySQL Shell's JavaScript mode, the first input column from the data file is used as the first column in the target table. The second input column, which has been assigned to the variable `@1` by the `columns` option, is subjected to a division operation before being used as the value of the second column in the target table.

```
mysql-js> util.importTable('file.txt', {
            columns: ['column1', 1],
            decodeColumns: {'column2': '@1 / 100'}
          });
```

In this example in MySQL Shell's JavaScript mode, the input columns from the data file are both assigned to variables, then transformed in various ways and used to populate the columns of the target table:

```
mysql-js> util.importTable('file.txt', {
            table: 't1',
            columns: [1, 2],
            decodeColumns: {
              'a': '@1',
              'b': '@2',
              'sum': '@1 + @2',
              'multiple': '@1 * @2',
              'power': 'POW(@1, @2)'
            }
          });
```

`skipRows: number`

Skip this number of rows at the beginning of the import file, or in the case of multiple import files, at the beginning of every file included in the file list. You can use this option to omit an initial header line containing column names from the upload to the table. The default is that no rows are skipped.

`replaceDuplicates: [true|false]`

Whether input rows that have the same value for a primary key or unique index as an existing row should be replaced (`true`) or skipped (`false`). The default is `false`.

`dialect: [default|csv|csv-unix|tsv|json]`

Use a set of field- and line-handling options appropriate for the specified file format. You can use the selected dialect as a base for further customization, by also specifying one or more of the `linesTerminatedBy`, `fieldsTerminatedBy`, `fieldsEnclosedBy`, `fieldsOptionallyEnclosed`, and

`fieldsEscapedBy` options to change the settings. The default dialect maps to a file created using a `SELECT...INTO OUTFILE` statement with the default settings for that statement. This is the default for the output file produced by MySQL Shell's table export utility. Other dialects are available to suit CSV files (created on either DOS or UNIX systems), TSV files, and JSON data. The settings applied for each dialect are as follows:

Table 11.2 Dialect settings for parallel table import utility

dialect	linesTerminatedBy	fieldsTerminatedBy	fieldsEnclosedBy	fieldsOptionallyEnclosed	fieldsEscapedBy
default	[LF]	[TAB]	[empty]	false	\
csv	[CR][LF]	,	"	true	\
csv-unix	[LF]	,	"	false	\
tsv	[CR][LF]	[TAB]	"	true	\
json	[LF]	[LF]	[empty]	false	[empty]

Note

1. The carriage return and line feed values for the dialects are operating system independent.
2. If you use the `linesTerminatedBy`, `fieldsTerminatedBy`, `fieldsEnclosedBy`, `fieldsOptionallyEnclosed`, and `fieldsEscapedBy` options, depending on the escaping conventions of your command interpreter, the backslash character (\) might need to be doubled if you use it in the option values.
3. Like the MySQL server with the `LOAD DATA` statement, MySQL Shell does not validate the field- and line-handling options that you specify. Inaccurate selections for these options can cause data to be imported into the wrong fields, partially, and/or incorrectly. Always verify your settings before starting the import, and verify the results afterwards.

`linesTerminatedBy: "characters"`

One or more characters (or an empty string) that terminates each of the lines in the input data file or files. The default is as for the specified dialect, or a linefeed character (`\n`) if the dialect option is omitted. This option is equivalent to the `LINES TERMINATED BY` option for the `LOAD DATA` statement. Note that the utility does not provide an equivalent for the `LINES STARTING BY` option for the `LOAD DATA` statement, which is set to the empty string.

`fieldsTerminatedBy: "characters"`

One or more characters (or an empty string) that terminates each of the fields in the input data file or files. The default is as for the specified dialect, or a tab character (`\t`) if the dialect option is omitted. This option is equivalent to the `FIELDS TERMINATED BY` option for the `LOAD DATA` statement.

`fieldsEnclosedBy: "character"`
: A single character (or an empty string) that encloses each of the fields in the input data file or files. The default is as for the specified dialect, or the empty string if the dialect option is omitted. This option is equivalent to the `FIELDS ENCLOSED BY` option for the `LOAD DATA` statement.

`fieldsOptionallyEnclosed: [ true | false ]`
: Whether the character given for `fieldsEnclosedBy` encloses all of the fields in the input data file or files (`false`), or encloses the fields only in some cases (`true`). The default is as for the specified dialect, or `false` if the dialect option is omitted. This option makes the `fieldsEnclosedBy` option equivalent to the `FIELDS OPTIONALLY ENCLOSED BY` option for the `LOAD DATA` statement.

`fieldsEscapedBy: "character"`
: The character that begins escape sequences in the input data file or files. If this is not provided, escape sequence interpretation does not occur. The default is as for the specified dialect, or a backslash (\) if the dialect option is omitted. This option is equivalent to the `FIELDS ESCAPED BY` option for the `LOAD DATA` statement.

`characterSet: "charset"`
: Added in MySQL Shell 8.0.21. This option specifies a character set encoding with which the input data is interpreted during the import. Setting the option to `binary` means that no conversion is done during the import. When you omit this option, the import uses the character set specified by the `character_set_database` system variable to interpret the input data.

`bytesPerChunk: "size"`
: For a list of multiple input data files, this option is not available. For a single input data file, this option specifies the number of bytes (plus any additional bytes required to reach the end of the row) that threads send for each `LOAD DATA` call to the target server. The utility distributes the data into chunks of this size for threads to pick up and send to the target server. The chunk size can be specified as a number of bytes, or using the suffixes k (kilobytes), M (megabytes), G (gigabytes). For example, `bytesPerChunk="2k"` makes threads send chunks of approximately 2 kilobytes. The minimum chunk size is 131072 bytes, and the default chunk size is 50M.

`threads: number`
: The maximum number of parallel threads to use to send the data in the input file or files to the target server. If you do not specify a number of threads, the default maximum is 8. For a list of multiple input data files, the utility creates the specified or maximum number of threads. For a single input data file, the utility calculates an appropriate number of threads to create up to this maximum, using the following formula:

  ```
  min{max{1, threads}, chunks}}
  ```

  where `threads` is the maximum number of threads, and `chunks` is the number of chunks that the data will be split into, which is calculated by dividing the file size by the `bytesPerChunk` size then adding 1. The calculation ensures that if the maximum number of threads exceeds the number of chunks that will actually be sent, the utility does not create more threads than necessary.

  Compressed files cannot be distributed into chunks, so instead the utility uses its parallel connections to upload multiple files at a time. If there is only one input data file, the upload of a compressed file can only use a single connection.

`maxRate: "rate"`
: The maximum limit on data throughput in bytes per second per thread. Use this option if you need to avoid saturating the network or the I/O or CPU for the client host or target server. The maximum rate can be specified as a number of bytes, or using the suffixes k (kilobytes), M (megabytes), G (gigabytes). For example, `maxRate="5M"` limits each thread to 5MB of data per second, which for eight threads gives a transfer rate of 40MB/second. The default is 0, meaning that there is no limit.

`showProgress: [ true | false ]`
: Display (`true`) or hide (`false`) progress information for the import. The default is `true` if stdout is a terminal (tty), and `false` otherwise.

`sessionInitSql: list of strings`
: A list of SQL statements to run at the start of each client session used for loading data into the target MySQL instance. You can use this option to change session variables. This option is available from MySQL Shell 8.0.30. For example, the following statements skip binary logging on the target MySQL instance for the sessions used by the utility during the course of the import, and increase the number of threads available for index creation:

```
sessionInitSQL: ["SET SESSION sql_log_bin=0;", "SET SESSION innodb_ddl_thre
```

If an error occurs while running the SQL statements, the import stops and returns an error message.

Options for Oracle Cloud Infrastructure

MySQL Shell supports importing input data files stored in Oracle Cloud Infrastructure Object Storage buckets.

`osBucketName: "string"`
: Added in MySQL Shell 8.0.21. The name of the Oracle Cloud Infrastructure Object Storage bucket where the input data file is located. By default, the `[DEFAULT]` profile in the Oracle Cloud Infrastructure CLI configuration file located at `~/.oci/config` is used to establish a connection to the bucket. You can substitute an alternative profile to be used for the connection with the `ociConfigFile` and `ociProfile` options. For instructions to set up a CLI configuration file, see SDK and CLI Configuration File.

`osNamespace: "string"`
: Added in MySQL Shell 8.0.21. The Oracle Cloud Infrastructure namespace where the Object Storage bucket named by `osBucketName` is located. The namespace for an Object Storage bucket is displayed in the Bucket Information tab of the bucket details page in the Oracle Cloud Infrastructure console, or can be obtained using the Oracle Cloud Infrastructure command line interface.

`ociConfigFile: "string"`
: Added in MySQL Shell 8.0.21. An Oracle Cloud Infrastructure CLI configuration file that contains the profile to use for the connection, instead of the one in the default location `~/.oci/config`.

`ociProfile: "string"`
: Added in MySQL Shell 8.0.21. The profile name of the Oracle Cloud Infrastructure profile to use for the connection, instead of the `[DEFAULT]` profile in the Oracle Cloud Infrastructure CLI configuration file used for the connection.

Options for S3-Compatible Services

MySQL Shell supports importing input data files stored in S3-compatible buckets, such as Amazon Web Services (AWS) S3.

Note

MySQL Shell supports AWS S3 configuration in command line options, environment variables, and configuration files. Command line options override environment variables, configuration files, and default options.

For information on configuration requirements, see Section 4.7, “Cloud Service Configuration”.

`s3BucketName: "string"`
: Added in MySQL Shell 8.0.30. The name of the S3 bucket where the dump files are located. By default, the `default` profile in the Amazon Web Services (AWS) CLI `config` and `credentials` files located at `~/.aws/` are used to establish a connection to the S3 bucket. You can substitute alternative configurations and credentials for the connection with the `s3ConfigFile` and `s3CredentialsFile` options. For instructions on installing and configuring the AWS CLI, see Getting started with the AWS CLI.

`s3CredentialsFile: "string"`
: Added in MySQL Shell 8.0.30. A credentials file that contains the user's credentials to use for the connection, instead of the one in the default location, `~/.aws/credentials`. Typically, the credentials file contains the `aws_access_key_id` and `aws_secret_access_key` to use for the connection.

`s3ConfigFile: "string"`
: Added in MySQL Shell 8.0.30. An AWS CLI configuration file that contains the profile to use for the connection, instead of the one in the default location `~/.aws/config`. Typically, the config file contains the region and output type to use for the connection.

`s3Profile: "string"`
: Added in MySQL Shell 8.0.30. The profile name of the s3 CLI profile to use for the connection, instead of the `default` profile in the AWS CLI configuration file used for the connection.

`s3Region: "string"`
: The name of the region to use for the connection.

`s3EndpointOverride: "string"`
: The URL of the endpoint to use instead of the default.

  Added in MySQL Shell 8.0.30. When connecting to the Oracle Cloud Infrastructure S3 compatbility API, the endpoint takes the following format: `https://namespace.compat.objectstorage.region.oraclecloud`. Replace *`namespace`* with the Object Storage namespace and *`region`* with your region identifier. For example, the region identifier for the US East (Ashburn) region is `us-ashburn-1`.

  For a namespace named axaxnpcrorw5 in the US East (Ashburn) region:

  `https://axaxnpcrorw5.compat.objectstorage.us-ashburn-1.oraclecloud.com`.

Options for Microsoft Azure Blob Storage

MySQL Shell supports importing from Microsoft Azure Blob Storage.

Note

MySQL Shell supports Microsoft Azure Blob Storage configuration in command line options, environment variables, and configuration files. Command line options override environment variables and configuration files.

For information on configuration requirements and the order of precedence of the configuration types, see Section 4.7, “Cloud Service Configuration”.

`azureContainerName: "string"` — Mandatory. The name of the Azure container from which the table is to be imported. The container must exist.

`azureConfigFile: "string"` — Optional. A configuration file that contains the storage connection parameters, instead of the one in the default location, such as `~/.azure/config`. If this is not defined, the default configuration file is used.

`azureContainerName` must be defined, and not be empty.

`azureStorageAccount: "string"` — Optional. The name of the Azure storage account to use for the operation.

`azureStorageSasToken: "string"` — Optional. Azure Shared Access Signature (SAS) token to be used for the authentication of the operation, instead of a key.

11.5 Instance Dump Utility, Schema Dump Utility, and Table Dump Utility

MySQL Shell's instance dump utility `util.dumpInstance()` and schema dump utility `util.dumpSchemas()`, introduced in MySQL Shell 8.0.21, support the export of all schemas or a selected schema from an on-premise MySQL instance into an Oracle Cloud Infrastructure Object Storage bucket or a set of local files. The table dump utility `util.dumpTables()`, introduced in MySQL Shell 8.0.22, supports the same operations for a selection of tables or views from a schema. The exported items can then be imported into a MySQL HeatWave Service DB System or a MySQL Server instance using the `util.loadDump()` utility (see Section 11.6, “Dump Loading Utility”). To get the best functionality, always use the most recent version available of MySQL Shell's dump and dump loading utilities.

- About the Utilities
- Requirements and Restrictions
- Running the Utilities
- Options for Dump Control
- Options for Dump Output
- Options for Filtering
- Options for MySQL HeatWave Service and Oracle Cloud Infrastructure
- Options for S3-compatible Services
- Options for Microsoft Azure Blob Storage
- Utility Error Messages

About the Utilities

MySQL Shell's instance dump utility, schema dump utility, and table dump utility provide Oracle Cloud Infrastructure Object Storage streaming, MySQL HeatWave Service compatibility checks and modifications, parallel dumping with multiple threads, and file compression, which are not provided by `mysqldump`. Progress information is displayed during the dump. You can carry out a dry run with your chosen set of dump options to show information about what actions would be performed, what items

would be dumped, and (for the instance dump utility and schema dump utility) what MySQL HeatWave Service compatibility issues would need to be fixed, when you run the utility for real with those options.

When choosing a destination for the dump files, note that for import into a MySQL HeatWave Service DB System, the MySQL Shell instance where you run the dump loading utility must be installed on an Oracle Cloud Infrastructure Compute instance that has access to the MySQL HeatWave Service DB System. If you dump the instance, schema, or tables to an Object Storage bucket, you can access the Object Storage bucket from the Compute instance. If you create the dump files on your local system, you need to transfer them to the Oracle Cloud Infrastructure Compute instance using the copy utility of your choice, depending on the operating system you chose for your Compute instance.

The dumps created by MySQL Shell's instance dump utility, schema dump utility, and table dump utility comprise DDL files specifying the schema structure, and tab-separated `.tsv` files containing the data. You can also choose to produce the DDL files only or the data files only, if you want to set up the exported schema as a separate exercise from populating it with the exported data. You can choose whether or not to lock the instance for backup during the dump for data consistency. By default, the dump utilities chunk table data into multiple data files and compress the files.

You can use options for the utilities to include or exclude specified schemas and tables, users and their roles and grants, events, routines, and triggers. If you specify conflicting include and exclude options or name an object that is not included in the dump anyway, before MySQL Shell 8.0.28 the situation is ignored and the dump proceeds without the object. From MySQL Shell 8.0.28 an error is reported and the dump stops so you can correct the options. If you need to dump the majority of the schemas in a MySQL instance, as an alternative strategy, you can use the instance dump utility rather than the schema dump utility, and specify the `excludeSchemas` option to list those schemas that are not to be dumped. Similarly, if you need to dump the majority of the tables in a schema, you can use the schema dump utility with the `excludeTables` option rather than the table dump utility.

The data for the `mysql.apply_status`, `mysql.general_log`, `mysql.schema`, and `mysql.slow_log tables` is always excluded from a dump created by MySQL Shell's schema dump utility, although their DDL statements are included. The `information_schema`, `mysql`, `ndbinfo`, `performance_schema`, and `sys` schemas are always excluded from an instance dump.

By default, the time zone is standardized to UTC in all the timestamp data in the dump output, which facilitates moving data between servers with different time zones and handling data that has multiple time zones. You can use the `tzUtc: false` option to keep the original timestamps if preferred.

The MySQL Shell dump loading utility `util.loadDump()` supports loading exported instances and schemas from an Object Storage bucket using a pre-authenticated request (PAR). From MySQL Shell 8.0.22 to 8.0.26, instances and schemas must be exported with the `ociParManifest` enabled to permit a load operation from Object Storage using a PAR. For details, see the description for the `ociParManifest` option. From MySQL Shell 8.0.27, with the introduction of support for PARs for all objects in a bucket or objects in a bucket with a specific prefix, enabling the `ociParManifest` option when exporting instances and schemas is no longer strictly necessary. For information about loading dumps using a PAR, see Section 11.6, “Dump Loading Utility”.

Beginning with MySQL Shell 8.0.27, MySQL Shell's instance dump utility, schema dump utility, and table dump utility are partition aware (see Partitioning, in the *MySQL Manual*). When a table being dumped is partitioned, each partition is treated as an independent table; if the table has subpartitions each subpartition is treated as an independent table. This also means that, when chunking is enabled, each partition or subpartition of a partitioned or subpartitioned table is chunked independently. The base names of dump files created for partitioned tables use the format *`schema@table@partition`*, where *`schema`* and *`table`* are, respectively the names of the parent schema and table, and *`partition`* is the URL-encoded name of the partition or subpartition.

To manage additions of features that are not supported by earlier versions of the MySQL Shell utilities, beginning with MySQL Shell 8.0.27, `util.dumpInstance()`, `util.dumpSchemas()`, `util.dumpTables()`, and `util.loadDump()` all write a list of features used in creating the dump to the dump metadata file; for each such feature, an element is added to the list. When the dump loading

utility reads the metadata file and finds an unsupported feature listed, it reports an error; the error message includes a version of MySQL Shell that supports the feature. The partition awareness feature introduced in MySQL Shell 8.0.27 is the first feature to support feature management.

Requirements and Restrictions

- The instance dump utility, schema dump utility, and table dump utility only support General Availability (GA) releases of MySQL Server versions.
- MySQL 5.7 or later is required for the destination MySQL instance where the dump will be loaded.
- For the source MySQL instance, dumping from MySQL 5.7 or later is fully supported in all MySQL Shell releases where the utilities are available. From MySQL Shell 8.0.22 through MySQL Shell 8.0.25, it is possible to dump an instance, schema, or table from a MySQL 5.6 instance and load it into a MySQL 5.7 or later destination, but dumping user accounts from MySQL 5.6 is not supported. From MySQL Shell 8.0.26, dumping user accounts from MySQL 5.6 is supported.
- MySQL Shell's dump loading utility `util.loadDump()` from versions of MySQL Shell previous to 8.0.27 cannot load dumps that are created using the dump utilities in MySQL Shell 8.0.27 or later. This is because from MySQL Shell 8.0.27, information is included in the dump metadata about features used in creating the dump. This feature list is not backward compatible, but it supports backward compatibility when new features are added in future releases. To get the best functionality, always use the most recent version available of MySQL Shell's dump and dump loading utilities.
- Object names in the instance or schema must be in the `latin1` or `utf8` characterset.
- Data consistency is guaranteed only for tables that use the `InnoDB` storage engine.
- The minimum required set of privileges that the user account used to run the utility must have on all the schemas involved is as follows: `EVENT`, `RELOAD`, `SELECT`, `SHOW VIEW`, and `TRIGGER`.
  - If the `consistent` option is set to `true`, which is the default, the `LOCK TABLES` privilege on all dumped tables can substitute for the `RELOAD` privilege if the latter is not available.
  - Before MySQL Shell 8.0.29, if the `consistent` option is set to `true`, which is the default, the `BACKUP_ADMIN` privilege is also required. From MySQL Shell 8.0.29, it is not required. If the user account does not have the `BACKUP_ADMIN` privilege and `LOCK INSTANCE FOR BACKUP` cannot be executed, the utilities make an extra consistency check during the dump. If this check fails, an instance dump is stopped, but a schema dump or a table dump continues and returns an error message to alert the user that the consistency check failed.
  - If the `consistent` option is set to `false`, the `BACKUP_ADMIN` and `RELOAD` privileges are not required.
  - If the dump is from a MySQL 5.6 or MySQL 5.7 instance, the `EXECUTE` privilege is also required if a view in the dump calls a function to get its data (up until MySQL 8.0.27, when it is no longer needed).
  - If the dump is from a MySQL 5.6 instance and includes user accounts (which is possible only with the instance dump utility), the `SUPER` privilege is also required.
  - If `activate_all_roles_on_login` is enabled, the user requires `SELECT` on `mysql.role_edges`. If it is not enabled, the user requires `SELECT` on `mysql.default_roles`.
- From MySQL Shell 8.0.24, the user account used to run the utility needs the `REPLICATION CLIENT` privilege in order for the utility to be able to include the binary log file name and position in the dump metadata. If the user ID does not have that privilege, the dump continues but does not include the binary log information. The binary log information can be used after loading the dumped data into the replica server to set up replication with a non-GTID source server, using the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` option of the `CHANGE REPLICATION SOURCE TO` statement (which is available from MySQL Server 8.0.23).

- The upload method used to transfer files to an Oracle Cloud Infrastructure Object Storage bucket has a file size limit of 1.2 TiB. In MySQL Shell 8.0.21, the multipart size setting means that the numeric limit on multiple file parts applies first, creating a limit of approximately 640 GB. From MySQL Shell 8.0.22, the multipart size setting has been changed to allow the full file size limit.
- The utilities convert columns with data types that are not safe to be stored in text form (such as `BLOB`) to Base64. The size of these columns therefore must not exceed approximately 0.74 times the value of the `max_allowed_packet` system variable (in bytes) that is configured on the target MySQL instance.
- For the table dump utility, exported views and triggers must not use qualified names to reference other views or tables.
- The table dump utility does not dump routines, so any routines referenced by the dumped objects (for example, by a view that uses a function) must already exist when the dump is loaded.
- For import into a MySQL HeatWave Service DB System, set the `ocimds` option to `true`, to ensure compatibility with MySQL HeatWave Service.
- For compatibility with MySQL HeatWave Service, all tables must use the `InnoDB` storage engine. The `ocimds` option checks for any exceptions found in the dump, and the `compatibility` option alters the dump files to replace other storage engines with `InnoDB`.
- For the instance dump utility and schema dump utility, for compatibility with MySQL HeatWave Service, all tables in the instance or schema must be located in the MySQL data directory and must use the default schema encryption. The `ocimds` option alters the dump files to apply these requirements.
- MySQL HeatWave Service uses `partial_revokes=ON`, which means database-level user grants on schemas which contain wildcards, such as `_` or `%`, are reported as errors.

  The compatibility option, `ignore_wildcard_grants` was introduced in MySQL Shell 8.0.33, along with `strip_invalid_grants`

  See Options for MySQL HeatWave Service and Oracle Cloud Infrastructure for more information.
- A number of other security related restrictions and requirements apply to items such as tablespaces and privileges for compatibility with MySQL HeatWave Service. The `ocimds` option checks for any exceptions found during the dump, and the `compatibility` option automatically alters the dump files to resolve some of the compatibility issues. You might need (or prefer) to make some changes manually. For more details, see the description for the `compatibility` option.
- For MySQL HeatWave Service High Availability, which uses Group Replication, primary keys are required on every table. From MySQL Shell 8.0.24, the `ocimds` option checks and reports an error for any tables in the dump that are missing primary keys. The `compatibility` option can be set to ignore missing primary keys if you do not need them, or to notify MySQL Shell's dump loading utility to add primary keys in invisible columns where they are not present. For details, see the description for the `compatibility` option. If possible, instead of managing this in the utility, consider creating primary keys in the tables on the source server before dumping them again.
- As of MySQL Shell 8.0.30, if any of the dump utilities are run against MySQL 5.7, with `"ocimds": true`, `util.checkForServerUpgrade` is run automatically. Pre-upgrade checks are run depending on the type of objects included in the dump.

Running the Utilities

The instance dump utility, schema dump utility, and table dump utility use the MySQL Shell global session to obtain the connection details of the target MySQL server from which the export is carried out. You must open the global session (which can have an X Protocol connection or a classic MySQL protocol connection) before running one of the utilities. The utilities open their own sessions for each thread, copying options such as connection compression and SSL options from the global session, and do not make any further use of the global session.

In the MySQL Shell API, the instance dump utility, schema dump utility, and table dump utility are functions of the `util` global object, and have the following signatures:

```
util.dumpInstance(outputUrl[, options])
util.dumpSchemas(schemas, outputUrl[, options])
util.dumpTables(schema, tables, outputUrl[, options])
```

`options` is a dictionary of options that can be omitted if it is empty. The available options for the instance dump utility, schema dump utility, and table dump utility are listed in the remaining sections in this topic.

For the schema dump utility, `schemas` specifies a list of one or more schemas to be dumped from the MySQL instance.

For the table dump utility, `schema` specifies the schema that contains the items to be dumped, and `tables` is an array of strings specifying the tables or views to be dumped. From MySQL Shell 8.0.23, the table dump includes the information required to set up the specified schema in the target MySQL instance, although it can be loaded into an alternative target schema by using the dump loading utility's `schema` option. In MySQL Shell 8.0.22, schema information is not included, so the dump files produced by this utility must be loaded into an existing target schema.

The table dump utility can be used to select individual tables from a schema, for example if you want to transfer tables between schemas. In this example in MySQL Shell's JavaScript mode, the tables `employees` and `salaries` from the `hr` schema are exported to the local directory `emp`, which the utility creates in the current working directory:

```
shell-js> util.dumpTables("hr", [ "employees", "salaries" ], "emp")
```

To dump all of the views and tables from the specified schema, use the `all` option and set the `tables` parameter to an empty array, as in this example:

```
shell-js> util.dumpTables("hr", [], "emp", { "all": true })
```

If you are dumping to the local filesystem, `outputUrl` is a string specifying the path to a local directory where the dump files are to be placed. You can specify an absolute path or a path relative to the current working directory. You can prefix a local directory path with the `file://` schema. In this example, the connected MySQL instance is dumped to a local directory, with some modifications made in the dump files for compatibility with MySQL HeatWave Service. The user first carries out a dry run to inspect the schemas and view the compatibility issues, then runs the dump with the appropriate compatibility options applied to remove the issues:

```
shell-js> util.dumpInstance("C:/Users/hanna/worlddump", {dryRun: true, ocimds: true})
Checking for compatibility with MySQL HeatWave Service 8.0.21
...
Compatibility issues with MySQL HeatWave Service 8.0.21 were found. Please use the
'compatibility' option to apply compatibility adaptations to the dumped DDL.
Util.dumpInstance: Compatibility issues were found (RuntimeError)
shell-js> util.dumpInstance("C:/Users/hanna/worlddump", {
        > ocimds: true, compatibility: ["strip_definers", "strip_restricted_grants"]})
```

The target directory must be empty before the export takes place. If the directory does not yet exist in its parent directory, the utility creates it. For an export to a local directory, the directories created during the dump are created with the access permissions `rwxr-x---`, and the files are created with the access permissions `rw-r-----` (on operating systems where these are supported). The owner of the files and directories is the user account that is running MySQL Shell.

If you are dumping to an Oracle Cloud Infrastructure Object Storage bucket, `outputUrl` is a path that will be used to prefix the dump files in the bucket, to simulate a directory structure. Use the `osBucketName` option to provide the name of the Object Storage bucket, and the `osNamespace` option to identify the namespace for the bucket. In this example, the user dumps the `world` schema from the connected MySQL instance to an Object Storage bucket, with the same compatibility modifications as in the previous example:

```
shell-js> util.dumpSchemas(["world"], "worlddump", {
```

```
        > "osBucketName": "hanna-bucket", "osNamespace": "idx28w1ckztq",
        > "ocimds": "true", "compatibility": ["strip_definers", "strip_restricted_grants"]})
```

In the Object Storage bucket, the dump files all appear with the prefix `worlddump`, for example:

```
worlddump/@.done.json
worlddump/@.json
worlddump/@.post.sql
worlddump/@.sql
worlddump/world.json
worlddump/world.sql
worlddump/world@city.json
worlddump/world@city.sql
worlddump/world@city@@0.tsv.zst
worlddump/world@city@@0.tsv.zst.idx
...
```

The namespace for an Object Storage bucket is displayed in the **Bucket Information** tab of the bucket details page in the Oracle Cloud Infrastructure console, or can be obtained using the Oracle Cloud Infrastructure command line interface. A connection is established to the Object Storage bucket using the default profile in the default Oracle Cloud Infrastructure CLI configuration file, or alternative details that you specify using the `ociConfigFile` and `ociProfile` options. For instructions to set up a CLI configuration file, see SDK and CLI Configuration File.

Options for Dump Control

`dryRun: [ true | false ]`
: Display information about what would be dumped with the specified set of options, and about the results of MySQL HeatWave Service compatibility checks (if the `ocimds` option is specified), but do not proceed with the dump. Setting this option enables you to list out all of the compatibility issues before starting the dump. The default is `false`.

`showProgress: [ true | false ]`
: Display (`true`) or hide (`false`) progress information for the dump. The default is `true` if `stdout` is a terminal (`tty`), such as when MySQL Shell is in interactive mode, and `false` otherwise. The progress information includes the estimated total number of rows to be dumped, the number of rows dumped so far, the percentage complete, and the throughput in rows and bytes per second.

`threads: int`
: The number of parallel threads to use to dump chunks of data from the MySQL instance. Each thread has its own connection to the MySQL instance. The default is 4.

`maxRate: "string"`
: The maximum number of bytes per second per thread for data read throughput during the dump. The unit suffixes `k` for kilobytes, `M` for megabytes, and `G` for gigabytes can be used (for example, setting `100M` limits throughput to 100 megabytes per second per thread). Setting `0` (which is the default value), or setting the option to an empty string, means no limit is set.

`defaultCharacterSet: "string"`
: The character set to be used during the session connections that are opened by MySQL Shell to the server for the dump. The default is `utf8mb4`. The session value of the system variables `character_set_client`, `character_set_connection`, and `character_set_results` are set to this value for each connection. The character set must be permitted by the `character_set_client` system variable and supported by the MySQL instance.

`consistent: [ true | false ]`
: Enable (`true`) or disable (`false`) consistent data dumps by locking the instance for backup during the dump. The default is `true`.

When `true` is set, the utility sets a global read lock using the `FLUSH TABLES WITH READ LOCK` statement (if the user ID used to run the utility has the `RELOAD` privilege), or a series of table locks using `LOCK TABLES` statements (if the user ID does not have the `RELOAD` privilege but does have `LOCK TABLES`). The transaction for each thread is started using the statements `SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ` and `START TRANSACTION WITH CONSISTENT SNAPSHOT`. When all threads have started their transactions, the instance is locked for backup (as described in LOCK INSTANCE FOR BACKUP and UNLOCK INSTANCE Statements) and the global read lock is released.

From MySQL Shell 8.0.29, if the user account does not have the `BACKUP_ADMIN` privilege and `LOCK INSTANCE FOR BACKUP` cannot be executed, the utilities make an extra consistency check during the dump. If this check fails, an instance dump is stopped, but a schema dump or a table dump continues and returns an error message to alert the user that the consistency check failed.

`skipConsistencyChecks: [ true | false ]`

Enable (`true`) or disable (`false`) the extra consistency check performed when `consistent: true`. Default is `false`.

This option is ignored if `consistent: false`.

Options for Dump Output

`tzUtc: [ true | false ]`

Include a statement at the start of the dump to set the time zone to UTC. All timestamp data in the dump output is converted to this time zone. The default is `true`, so timestamp data is converted by default. Setting the time zone to UTC facilitates moving data between servers with different time zones, or handling a set of data that has multiple time zones. Set this option to `false` to keep the original timestamps if preferred.

`compression: "string"`

The compression type to use when writing data files for the dump. The default is to use zstd compression (`zstd`). The alternatives are to use gzip compression (`gzip`) or no compression (`none`).

`chunking: [ true | false ]`

Enable (`true`) or disable (`false`) chunking for table data, which splits the data for each table into multiple files. The default is `true`, so chunking is enabled by default. Use `bytesPerChunk` to specify the chunk size. If you set the chunking option to `false`, chunking does not take place and the utility creates one data file for each table.

Prior to MySQL Shell 8.0.32, to chunk table data into separate files, a primary key or unique index must be defined for the table, which the utility uses to select an index column to order and chunk the data. If a table does not contain either of these, a warning is displayed and the table data is written to a single file.

As of MySQL Shell 8.0.32, if a table has no primary key or unique index, chunking is done based on the number of rows in the table, the average row length, and the `bytesPerChunk` value.

`bytesPerChunk: "string"`

Sets the approximate number of bytes to be written to each data file when chunking is enabled. The unit suffixes `k` for kilobytes, `M` for megabytes, and `G` for gigabytes can be used. The default is

64 MB (`64M`) from MySQL Shell 8.0.22 (32 MB in MySQL Shell 8.0.21), and the minimum is 128 KB (`128k`). Specifying this option sets `chunking` to `true` implicitly. The utility aims to chunk the data for each table into files each containing this amount of data before compression is applied. The chunk size is an average and is calculated based on table statistics and explain plan estimates.

`dialect: [default|csv|csv-unix|tsv]`

(Added in MySQL Shell 8.0.32) Specify a set of field- and line-handling options for the format of the exported data file. You can use the selected dialect as a base for further customization, by also specifying one or more of the `linesTerminatedBy`, `fieldsTerminatedBy`, `fieldsEnclosedBy`, `fieldsOptionallyEnclosed`, and `fieldsEscapedBy` options to change the settings.

The default dialect produces a data file matching what would be created using a `SELECT...INTO OUTFILE` statement with the default settings for that statement. `.txt` is an appropriate file extension to assign to these output files. Other dialects are available to export CSV files for either DOS or UNIX systems (`.csv`), and TSV files (`.tsv`).

The settings applied for each dialect are as follows:

Table 11.3 Dialect settings for table export utility

dialect	linesTerminatedBy	fieldsTerminatedBy	fieldsEnclosedBy	fieldsOptionallyEnclosed	fieldsEscapedBy
default	[LF]	[TAB]	[empty]	false	\
csv	[CR][LF]	,	"	true	\
csv-unix	[LF]	,	"	false	\
tsv	[CR][LF]	[TAB]	"	true	\

Note

1. The carriage return and line feed values for the dialects are operating system independent.
2. If you use the `linesTerminatedBy`, `fieldsTerminatedBy`, `fieldsEnclosedBy`, `fieldsOptionallyEnclosed`, and `fieldsEscapedBy` options, depending on the escaping conventions of your command interpreter, the backslash character (\) might need to be doubled if you use it in the option values.
3. Like the MySQL server with the `SELECT...INTO OUTFILE` statement, MySQL Shell does not validate the field- and line-handling options that you specify. Inaccurate selections for these options can cause data to be exported partially or incorrectly. Always verify your settings before starting the export, and verify the results afterwards.

`linesTerminatedBy: "characters"`

(Added in MySQL Shell 8.0.32) One or more characters (or an empty string) with which the utility terminates each of the lines in the exported data file. The default is as for the specified dialect, or a linefeed character (`\n`) if the dialect option is omitted. This option is equivalent to the `LINES TERMINATED BY` option for the `SELECT...INTO OUTFILE` statement. Note that the utility does not provide an equivalent for the `LINES STARTING BY` option for the `SELECT...INTO OUTFILE` statement, which is set to the empty string.

`fieldsTerminatedBy: "characters"`

(Added in MySQL Shell 8.0.32) One or more characters (or an empty string) with which the utility terminates each of the fields in the exported data file. The default is as for the specified dialect, or a tab character (`\t`) if the dialect option is omitted. This option is equivalent to the `FIELDS TERMINATED BY` option for the `SELECT...INTO OUTFILE` statement.

`fieldsEnclosedBy: "character"`

(Added in MySQL Shell 8.0.32) A single character (or an empty string) with which the utility encloses each of the fields in the exported data file. The default is as for the specified dialect, or the empty string if the dialect option is omitted. This option is equivalent to the `FIELDS ENCLOSED BY` option for the `SELECT...INTO OUTFILE` statement.

`fieldsOptionallyEnclosed: [ true | false ]`

(Added in MySQL Shell 8.0.32) Whether the character given for `fieldsEnclosedBy` is to enclose all of the fields in the exported data file (`false`), or to enclose a field only if it has a string data type such as `CHAR`, `BINARY`, `TEXT`, or `ENUM` (`true`). The default is as for the specified dialect, or `false` if the dialect option is omitted. This option makes the `fieldsEnclosedBy` option equivalent to the `FIELDS OPTIONALLY ENCLOSED BY` option for the `SELECT...INTO OUTFILE` statement.

`fieldsEscapedBy: "character"`

(Added in MySQL Shell 8.0.32) The character that is to begin escape sequences in the exported data file. The default is as for the specified dialect, or a backslash (\) if the dialect option is omitted. This option is equivalent to the `FIELDS ESCAPED BY` option for the `SELECT...INTO OUTFILE` statement. If you set this option to the empty string, no characters are escaped, which is not recommended because special characters used by `SELECT...INTO OUTFILE` must be escaped.

Options for Filtering

`where:`

A key-value pair comprising of a valid table identifier, of the form *`schemaName.tableName`*, and a valid SQL condition expression used to filter the data being exported.

Note

The SQL is validated only when it is executed. If you are exporting many tables, any SQL-syntax-related issues will only be seen late in the process. As such, it is recommended you test your SQL condition before using it in a long-running export process.

In the following example, `where` exports only those rows of the table`sakila.actor` and `sakila.actor_info` where the value of `actor_id` is greater than 150, to a local folder named `out`:

```
util.dumpTables("sakila", ["actor","actor_info"], "out", {"where" :
  {"sakila.actor": "actor_id > 150", "sakila.actor_info": "actor_id >
```

`partitions: ["string","string",..]`

A list of valid partition names which limits the export to the specified partitions.

For example, to export only the partitions named `p1` and `p2`: `partitions: ["p1", "p2"]`

The following example exports the partitions p1 and p2 from table1 and the partition p2 from table2:

```
util.dumpTables("schema", ["table","table2"], "out", {"partitions" :
{ "schema.table1":  ["p1", "p2"],"schema.table2":  ["p2"]}})
```

`ddlOnly: [ true | false ]`

Setting this option to `true` includes only the DDL files for the dumped items in the dump, and does not dump the data. The default is `false`.

`dataOnly: [ true | false ]`

Setting this option to `true` includes only the data files for the dumped items in the dump, and does not include DDL files. The default is `false`.

`users: [ true | false ]`

(Instance dump utility only) Include (`true`) or exclude (`false`) users and their roles and grants in the dump. The default is `true`, so users are included by default. The schema dump utility and table dump utility do not include users, roles, and grants in a dump.

From MySQL Shell 8.0.22, you can use the `excludeUsers` or `includeUsers` option to specify individual user accounts to be excluded or included in the dump files. These options can also be used with MySQL Shell's dump loading utility `util.loadDump()` to exclude or include individual user accounts at the point of import, depending on the requirements of the target MySQL instance.

Note

1. Dumping user accounts from a MySQL 5.6 instance is not supported. If you are dumping from this version, set `users: false`.
2. In MySQL Shell 8.0.21, attempting to import users to a MySQL HeatWave Service DB System causes the import to fail if the `root` user account or another restricted user account name is present in the dump files, so the import of users to a MySQL DB System is not supported in that release.

`excludeUsers: array of strings`

(Instance dump utility only) Exclude the named user accounts from the dump files. This option is available from MySQL Shell 8.0.22, and you can use it to exclude user accounts that are not accepted for import to a MySQL HeatWave Service DB

System, or that already exist or are not wanted on the target MySQL instance. Specify each user account string in the format `"'user_name'@'host_name'"` for an account that is defined with a user name and host name, or `"'user_name'"` for an account that is defined with a user name only. If you do not supply a host name, all accounts with that user name are excluded.

`includeUsers: array of strings`

(Instance dump utility only) Include only the named user accounts in the dump files. Specify each user account string as for the `excludeUsers` option. This option is available from MySQL Shell 8.0.22, and you can use it as an alternative to `excludeUsers` if only a few user accounts are required in the dump. You can also specify both options to include some accounts and exclude others.

`excludeSchemas: array of strings`

(Instance dump utility only) Exclude the named schemas from the dump. Note that the `information_schema`, `mysql`, `ndbinfo`, `performance_schema`, and `sys` schemas are always excluded from an instance dump.

`includeSchemas: array of strings`

(Instance dump utility only) Include only the named schemas in the dump. This option is available from MySQL Shell 8.0.28. You cannot include the `information_schema`, `mysql`, `ndbinfo`, `performance_schema`, or `sys` schemas by naming them on this option. If you want to dump one or more of these schemas, you can do this using the schema dump utility `util.dumpSchemas()`.

`excludeTables: array of strings`

(Instance dump utility and schema dump utility only) Exclude the named tables from the dump. Table names must be qualified with a valid schema name, and quoted with the backtick character if needed. Tables named by the `excludeTables` option do not have DDL files or data files in the dump. Note that the data for the `mysql.apply_status`, `mysql.general_log`, `mysql.schema`, and `mysql.slow_log tables` is always excluded from a schema dump, although their DDL statements are included, and you cannot include that data by naming the table in another option or utility.

Note

Schema and table names containing multi-byte characters must be surrounded with backticks.

`includeTables: array of strings`

(Instance dump utility and schema dump utility only) Include only the named tables in the dump. This option is available from MySQL Shell 8.0.28. Table names must be qualified with a valid schema name, and quoted with the backtick character if needed.

Note

Schema and table names containing multi-byte characters must be surrounded with backticks.

`events: [ true | false ]`

(Instance dump utility and schema dump utility only) Include (`true`) or exclude (`false`) events for each schema in the dump. The default is `true`.

`excludeEvents: array of strings`

(Instance dump utility and schema dump utility only) Exclude the named events from the dump. This option is available from MySQL Shell 8.0.28. Names of events must be qualified with a valid schema name, and quoted with the backtick character if needed.

`includeEvents: array of strings`
(Instance dump utility and schema dump utility only) Include only the named events in the dump. This option is available from MySQL Shell 8.0.28. Event names must be qualified with a valid schema name, and quoted with the backtick character if needed.

`routines: [ true | false ]`
(Instance dump utility and schema dump utility only) Include (`true`) or exclude (`false`) functions and stored procedures for each schema in the dump. The default is `true`. Note that user-defined functions are not included, even when `routines` is set to `true`.

`excludeRoutines: array of strings`
(Instance dump utility and schema dump utility only) Exclude the named functions and stored procedures from the dump. This option is available from MySQL Shell 8.0.28. Names of routines must be qualified with a valid schema name, and quoted with the backtick character if needed.

`includeRoutines: array of strings`
(Instance dump utility and schema dump utility only) Include only the named functions and stored procedures in the dump. This option is available from MySQL Shell 8.0.28. Names of routines must be qualified with a valid schema name, and quoted with the backtick character if needed.

`all: [ true | false ]`
(Table dump utility only) Setting this option to `true` includes all views and tables from the specified schema in the dump. The default is `false`. When you use this option, set the `tables` parameter to an empty array, for example:

```
shell-js> util.dumpTables("hr", [], "emp", { "all": true })
```

`triggers: [ true | false ]`
(All dump utilities) Include (`true`) or exclude (`false`) triggers for each table in the dump. The default is `true`.

`excludeTriggers: array of strings`
(All dump utilities) Exclude the named triggers from the dump. This option is available from MySQL Shell 8.0.28. Names of triggers must be qualified with a valid schema name and table name (`schema.table.trigger`), and quoted with the backtick character if needed. You can exclude all triggers for a specific table by specifying a schema name and table name with this option (`schema.table`).

`includeTriggers: array of strings`
(All dump utilities) Include only the named triggers in the dump. This option is available from MySQL Shell 8.0.28. Names of triggers must be qualified with a valid schema name and table name (`schema.table.trigger`), and quoted with the backtick character if needed. You can include all triggers for a specific table by specifying a schema name and table name with this option (`schema.table`).

Options for MySQL HeatWave Service and Oracle Cloud Infrastructure

`osBucketName: "string"`
The name of the Oracle Cloud Infrastructure Object Storage bucket to which the dump is to be written. By default, the `[DEFAULT]` profile in the Oracle Cloud Infrastructure CLI configuration file located at `~/.oci/config` is used to establish a connection to the bucket. You can substitute an alternative profile to be used for the connection with the `ociConfigFile` and `ociProfile` options. For instructions to set up a CLI configuration file, see SDK and CLI Configuration File.

`osNamespace: "string"`

The Oracle Cloud Infrastructure namespace where the Object Storage bucket named by `osBucketName` is located. The namespace for an Object Storage bucket is displayed in the Bucket Information tab of the bucket details page in the Oracle Cloud Infrastructure console, or can be obtained using the Oracle Cloud Infrastructure command line interface.

`ociConfigFile: "string"`

An Oracle Cloud Infrastructure CLI configuration file that contains the profile to use for the connection, instead of the one in the default location `~/.oci/config`.

`ociProfile: "string"`

The profile name of the Oracle Cloud Infrastructure profile to use for the connection, instead of the `[DEFAULT]` profile in the Oracle Cloud Infrastructure CLI configuration file used for the connection.

`ocimds: [ true | false ]`

Setting this option to `true` enables checks and modifications for compatibility with MySQL HeatWave Service. The default is `false`. From MySQL Shell 8.0.23, this option is available for all the utilities, and before that release, it is only available for the instance dump utility and schema dump utility.

When this option is set to `true`, `DATA DIRECTORY`, `INDEX DIRECTORY`, and `ENCRYPTION` options in `CREATE TABLE` statements are commented out in the DDL files, to ensure that all tables are located in the MySQL data directory and use the default schema encryption. Checks are carried out for any storage engines in `CREATE TABLE` statements other than `InnoDB`, for grants of unsuitable privileges to users or roles, and for other compatibility issues. If any non-conforming SQL statement is found, an exception is raised and the dump is halted. Use the `dryRun` option to list out all of the issues with the items in the dump before the dumping process is started. Use the `compatibility` option to automatically fix the issues in the dump output.

From MySQL Shell 8.0.22 to MySQL Shell 8.0.26, when this option is set to `true` and an Object Storage bucket name is supplied using the `osBucketName` option, the `ociParManifest` option also defaults to `true`, meaning that a manifest file is generated contains pre-authenticated requests (PARs) for all of the files in the dump, and the dump files can only be accessed using these PARs. From MySQL Shell 8.0.27, with the introduction of support for PARs for all objects in a bucket or objects in a bucket with a specific prefix, the `ociParManifest` option is set to `false` by default and is only enabled if set to `true` explicitly.

Note

As of MySQL Shell 8.0.30, if any of the dump utilities are run against MySQL 5.7, with `"ocimds": true`, `util.checkForServerUpgrade` is run automatically. Pre-upgrade checks are run depending on the type of objects included in the dump.

`compatibility: array of strings`

Apply the specified requirements for compatibility with MySQL HeatWave Service for all tables in the dump output, altering the dump files as necessary. From MySQL Shell 8.0.23, this option is available for all the utilities, and before that release, it is only available for the instance dump utility and schema dump utility.

The following modifications can be specified as an array of strings:

`force_innodb`	Change `CREATE TABLE` statements to use the `InnoDB` storage engine for any tables that do not already use it.
`skip_invalid_accounts`	Remove user accounts created with external authentication plugins that are not supported in MySQL HeatWave Service. From MySQL Shell 8.0.26, this option also removes user accounts that do not have passwords set, except where an account with no password is identified as a role, in which case it is dumped using the `CREATE ROLE` statement.
`strip_definers`	Remove the `DEFINER` clause from views, routines, events, and triggers, so these objects are created with the default definer (the user invoking the schema), and change the `SQL SECURITY` clause for views and routines to specify `INVOKER` instead of `DEFINER`. MySQL HeatWave Service requires special privileges to create these objects with a definer other than the user loading the schema. If your security model requires that views and routines have more privileges than the account querying or calling them, you must manually modify the schema before loading it.
`strip_restricted_grants`	Remove specific privileges that are restricted by MySQL HeatWave Service from `GRANT` statements, so users and their roles cannot be given these privileges (which would cause user creation to fail). From MySQL Shell 8.0.22, this option also removes `REVOKE` statements for system schemas (`mysql` and `sys`) if the administrative user account on an Oracle Cloud Infrastructure Compute instance does not itself have the relevant privileges, so cannot remove them.
`strip_tablespaces`	Remove the `TABLESPACE` clause from `CREATE TABLE` statements,

	so all tables are created in their default tablespaces. MySQL HeatWave Service has some restrictions on tablespaces.
`ignore_missing_pks`	Make the instance, schema, or table dump utility ignore any missing primary keys when the dump is carried out, so that the `ocimds` option can still be used without the dump stopping due to this check. Dumps created with this modification cannot be loaded into a MySQL HeatWave Service High Availability instance, because primary keys are required for MySQL HeatWave Service High Availability, which uses Group Replication. To add the missing primary keys instead, use the `create_invisible_pks` modification, or consider creating primary keys in the tables on the source server.
`ignore_wildcard_grants`	If enabled, ignores errors from grants on schemas with wildcards, which are interpreted differently in systems where the `partial_revokes` system variable is enabled.
`strip_invalid_grants`	If enabled, strips grant statements which would fail when users are loaded. Such as grants referring to a specific routine which does not exist.
`create_invisible_pks`	Add a flag in the dump metadata to notify MySQL Shell's dump loading utility to add primary keys in invisible columns, for each table that does not contain a primary key. This modification enables a dump where some tables lack primary keys to be loaded into a MySQL HeatWave Service High Availability instance. Primary keys are required for MySQL HeatWave Service High Availability, which uses Group Replication. The dump data is unchanged by this modification, as the tables do not contain the invisible columns until they have been processed

by the dump loading utility. The invisible columns (which are named "`my_row_id`") have no impact on applications that use the uploaded tables.

Adding primary keys in this way does not yet enable inbound replication of the modified tables to a High Availability instance, as that feature currently requires the primary keys to exist in both the source server and the replica server. If possible, instead of using this modification, consider creating primary keys in the tables on the source server, before dumping them again. From MySQL 8.0.23, you can do this with no impact to applications by using invisible columns to hold the primary keys. This is a best practice for performance and usability, and helps the dumped database to work seamlessly with MySQL HeatWave Service.

Note

MySQL Shell's dump loading utility can only be used to load dumps created with the `create_invisible_` modification onto a target MySQL instance at MySQL Server 8.0.24 or later, due to a limitation on hidden columns in MySQL 8.0.23. The dump loading

utility in versions of MySQL Shell before MySQL Shell 8.0.24 silently ignores the dump metadata flag and does not add the primary keys, so ensure that you use the latest version of the utility.

`ociParManifest: [ true | false ]`

Setting this option to `true` generates a PAR for read access (an Object Read PAR) for each item in the dump, and a manifest file listing all the PAR URLs. The PARs expire after a week by default, which you can change using the `ociParExpireTime` option.

This option is available from MySQL Shell 8.0.22 for the instance dump utility and schema dump utility, and can only be used when exporting to an Object Storage bucket (so with the `osBucketName` option set). From MySQL Shell 8.0.23, this option is available for all the dump utilities.

From MySQL Shell 8.0.22 to MySQL Shell 8.0.26, when the `ocimds` option is set to `true` and an Object Storage bucket name is supplied using the `osBucketName` option, `ociParManifest` is set to `true` by default, otherwise it is set to `false` by default. From MySQL Shell 8.0.27, with the introduction of support for PARs for all objects in a bucket or objects in a bucket with a specific prefix, `ociParManifest` is set to `false` by default and is only enabled if set to `true` explicitly.

The user named in the Oracle Cloud Infrastructure profile that is used for the connection to the Object Storage bucket (the `DEFAULT` user or another user as named by the `ociProfile` option) is the creator for the PARs. This user must have `PAR_MANAGE` permissions and appropriate permissions for interacting with the objects in the bucket, as described in Using Pre-Authenticated Requests. If there is an issue with creating the PAR for any object, the associated file is deleted and the dump is stopped.

To enable loading of dump files created with the `ociParManifest` option enabled, create a read-only PAR for the manifest file (`@.manifest.json`) following the instructions in Using Pre-Authenticated Requests. You can do this while the dump is still in progress if you want to start loading the dump before it completes. You can create this PAR using any user account that has the

required permissions. The PAR URL must then be used by the dump loading utility to access the dump files through the manifest file. The URL is only displayed at the time of creation, so copy it to durable storage.

Important

Before using this access method, assess the business requirement for and the security ramifications of pre-authenticated access to a bucket or objects. A PAR gives anyone who has the PAR access to the targets identified in the request. Carefully manage the distribution of PARs.

`ociParExpireTime: "string"`

The expiry time for the PARs that are generated when the `ociParManifest` option is set to true. The default is the current time plus one week, in UTC format.

This option is available from MySQL Shell 8.0.22 for the instance dump utility and schema dump utility. From MySQL Shell 8.0.23, this option is available for all the dump utilities.

The expiry time must be formatted as an RFC 3339 timestamp, as required by Oracle Cloud Infrastructure when creating a PAR. The format is `YYYY-MM-DDTHH-MM-SS` immediately followed by either the letter Z (for UTC time), or the UTC offset for the local time expressed as `[+|-]hh:mm`, for example `2020-10-01T00:09:51.000+02:00`. MySQL Shell does not validate the expiry time, but any formatting error causes the PAR creation to fail for the first file in the dump, which stops the dump.

Options for S3-compatible Services

MySQL Shell supports dumping MySQL data to S3-compatible buckets, such as Amazon Web Services (AWS) S3.

Note

MySQL Shell supports AWS S3 configuration in command line options, environment variables, and configuration files. Command line options override environment variables, configuration files, and default options.

For information on configuration requirements, see Section 4.7, “Cloud Service Configuration”.

`s3BucketName: "string"`

The name of the S3 bucket to which the dump is to be written. By default, the `default` profile of the `config` and `credentials` files located at `~/.aws/` are used to establish a connection to the S3 bucket. You can substitute alternative configurations and credentials for the connection with the `s3ConfigFile` and `s3CredentialsFile` options. For instructions on installing and configuring the AWS CLI, see Getting started with the AWS CLI.

`s3CredentialsFile:"string"`

A credentials file that contains the user's credentials to use for the connection, instead of the one in the default location, `~/.aws/credentials`. Typically, the credentials file contains the `aws_access_key_id` and `aws_secret_access_key` to use for the connection.

`s3ConfigFile: "string"`
: A configuration file that contains the profile to use for the connection, instead of the one in the default location, such as `~/.aws/config`. Typically, the config file contains the region and output type to use for the connection.

`s3Profile: "string"`
: The profile name of the s3 CLI profile to use for the connection, instead of the `default` profile.

`s3Region: "string"`
: The name of the region to use for the connection.

`s3EndpointOverride: "string"`
: The URL of the endpoint to use instead of the default.

  When connecting to the Oracle Cloud Infrastructure S3 compatbility API, the endpoint takes the following format: `https://namespace.compat.objectstorage.region.oraclecloud.co` Replace *`namespace`* with the Object Storage namespace and *`region`* with your region identifier. For example, the region identifier for the US East (Ashburn) region is `us-ashburn-1`.

  For a namespace named axaxnpcrorw5 in the US East (Ashburn) region:

  `https://axaxnpcrorw5.compat.objectstorage.us-ashburn-1.oraclecloud.com`.

The following example shows the dump of a MySQL instance to a folder, `test`, in an S3 bucket, `Bucket001`, with some compatibility options:

```
util.dumpInstance("test",{s3bucketName: "Bucket001", threads: 4,
compatibility: ["strip_restricted_grants", "strip_definers", "ignore_missing_pks"]})
```

The following example shows the dump of a MySQL instance to a prefix, `test`, in an object storage bucket, `Bucket001`, using a configuration profile, `oci`, the `s3EndpointOverride` to direct the connection to the OCI endpoint of the required tenancy and region, and some compatibility options:

```
util.dumpInstance("test",{s3BucketName: "Bucket001",
s3EndpointOverride: "https://axaxnpcrorw5.compat.objectstorage.us-ashburn-1.oraclecloud.com",
s3Profile: "oci", threads: 4,
compatibility: ["strip_restricted_grants", "strip_definers", "ignore_missing_pks"]})
```

Options for Microsoft Azure Blob Storage

MySQL Shell supports dumping to Microsoft Azure Blob Storage.

Note

MySQL Shell supports Microsoft Azure Blob Storage configuration in command line options, environment variables, and configuration files. Command line options override environment variables, and configuration files.

For information on configuration requirements and the order of precedence of the configuration types, see Section 4.7, “Cloud Service Configuration”.

`azureContainerName: "string"`
: Mandatory. The name of the Azure container to which the dump is to be written. The container must exist.

`azureConfigFile: "string"`
: Optional. A configuration file that contains the storage connection parameters, instead of the one in the default location, such as `~/.azure/config`. If this is not defined, the default configuration file is used.

  `azureContainerName` must be defined, and not be empty.

`azureStorageAccount: "string"`	Optional. The name of the Azure storage account to use for the operation.
`azureStorageSasToken: "string"`	Optional. Azure Shared Access Signature (SAS) token to be used for the authentication of the operation, instead of a key.

In the following example, the configuration uses a configuration string for the connection parameters, which means the dump command only requires the `azureContainerName`.

Example `config` file:

```
[cloud]
 name = AzureCloud

[storage]
 connection_string=alphanumericConnectionString
```

Example `dumpInstance` command, which exports the contents of the instance to a folder named `prefix1`, in a container named `mysqlshellazure`:

```
util.dumpInstance("prefix1", {azureContainerName: "mysqlshellazure", threads: 4})
```

Utility Error Messages

Error numbers in the range 52000-52999 are specific to MySQL Shell's instance dump utility `util.dumpInstance()`, schema dump utility `util.dumpSchemas()`, and table dump utility `util.dumpTables()`. The following errors might be returned:

- Error number: `52000`; Symbol: `SHERR_DUMP_LOCK_TABLES_MISSING_PRIVILEGES`

  Message: User %s is missing the following privilege(s) for %s: %s.

- Error number: `52001`; Symbol: `SHERR_DUMP_GLOBAL_READ_LOCK_FAILED`

  Message: Unable to acquire global read lock

- Error number: `52002`; Symbol: `SHERR_DUMP_LOCK_TABLES_FAILED`

  Message: Unable to lock tables: %s.

- Error number: `52003`; Symbol: `SHERR_DUMP_CONSISTENCY_CHECK_FAILED`

  Message: Consistency check has failed.

- Error number: `52004`; Symbol: `SHERR_DUMP_COMPATIBILITY_ISSUES_FOUND`

  Message: Compatibility issues were found

- Error number: `52005`; Symbol: `SHERR_DUMP_COMPATIBILITY_OPTIONS_FAILED`

  Message: Could not apply some of the compatibility options

- Error number: `52006`; Symbol: `SHERR_DUMP_WORKER_THREAD_FATAL_ERROR`

  Message: Fatal error during dump

- Error number: `52007`; Symbol: `SHERR_DUMP_MISSING_GLOBAL_PRIVILEGES`

  Message: User %s is missing the following global privilege(s): %s.

- Error number: `52008`; Symbol: `SHERR_DUMP_MISSING_SCHEMA_PRIVILEGES`

  Message: User %s is missing the following privilege(s) for schema %s: %s.

- Error number: 52009; Symbol: SHERR_DUMP_MISSING_TABLE_PRIVILEGES

  Message: User %s is missing the following privilege(s) for table %s: %s.
- Error number: 52010; Symbol: SHERR_DUMP_NO_SCHEMAS_SELECTED

  Message: Filters for schemas result in an empty set.
- Error number: 52011; Symbol: SHERR_DUMP_MANIFEST_PAR_CREATION_FAILED

  Message: Failed creating PAR for object '%s': %s
- Error number: 52012; Symbol: SHERR_DUMP_DW_WRITE_FAILED

  Message: Failed to write %s into file %s
- Error number: 52013; Symbol: SHERR_DUMP_IC_FAILED_TO_FETCH_VERSION

  Message: Failed to fetch version of the server.
- Error number: 52014; Symbol: SHERR_DUMP_SD_CHARSET_NOT_FOUND

  Message: Unable to find charset: %s
- Error number: 52015; Symbol: SHERR_DUMP_SD_WRITE_FAILED

  Message: Got errno %d on write
- Error number: 52016; Symbol: SHERR_DUMP_SD_QUERY_FAILED

  Message: Could not execute '%s': %s
- Error number: 52017; Symbol: SHERR_DUMP_SD_COLLATION_DATABASE_ERROR

  Message: Error processing select @@collation_database; results
- Error number: 52018; Symbol: SHERR_DUMP_SD_CHARACTER_SET_RESULTS_ERROR

  Message: Unable to set character_set_results to: %s
- Error number: 52019; Symbol: SHERR_DUMP_SD_CANNOT_CREATE_DELIMITER

  Message: Can't create delimiter for event: %s
- Error number: 52020; Symbol: SHERR_DUMP_SD_INSUFFICIENT_PRIVILEGE

  Message: %s has insufficient privileges to %s!
- Error number: 52021; Symbol: SHERR_DUMP_SD_MISSING_TABLE

  Message: %s not present in information_schema
- Error number: 52022; Symbol: SHERR_DUMP_SD_SHOW_CREATE_TABLE_FAILED

  Message: Failed running: show create table %s with error: %s
- Error number: 52023; Symbol: SHERR_DUMP_SD_SHOW_CREATE_TABLE_EMPTY

  Message: Empty create table for table: %s
- Error number: 52024; Symbol: SHERR_DUMP_SD_SHOW_FIELDS_FAILED

  Message: SHOW FIELDS FROM failed on view: %s
- Error number: 52025; Symbol: SHERR_DUMP_SD_SHOW_KEYS_FAILED

Message: Can't get keys for table %s: %s

- Error number: 52026; Symbol: SHERR_DUMP_SD_SHOW_CREATE_VIEW_FAILED

  Message: Failed: SHOW CREATE TABLE %s

- Error number: 52027; Symbol: SHERR_DUMP_SD_SHOW_CREATE_VIEW_EMPTY

  Message: No information about view: %s

- Error number: 52028; Symbol: SHERR_DUMP_SD_SCHEMA_DDL_ERROR

  Message: Error while dumping DDL for schema '%s': %s

- Error number: 52029; Symbol: SHERR_DUMP_SD_TABLE_DDL_ERROR

  Message: Error while dumping DDL for table '%s'.'%s': %s

- Error number: 52030; Symbol: SHERR_DUMP_SD_VIEW_TEMPORARY_DDL_ERROR

  Message: Error while dumping temporary DDL for view '%s'.'%s': %s

- Error number: 52031; Symbol: SHERR_DUMP_SD_VIEW_DDL_ERROR

  Message: Error while dumping DDL for view '%s'.'%s': %s

- Error number: 52032; Symbol: SHERR_DUMP_SD_TRIGGER_COUNT_ERROR

  Message: Unable to check trigger count for table: '%s'.'%s'

- Error number: 52033; Symbol: SHERR_DUMP_SD_TRIGGER_DDL_ERROR

  Message: Error while dumping triggers for table '%s'.'%s': %s

- Error number: 52034; Symbol: SHERR_DUMP_SD_EVENT_DDL_ERROR

  Message: Error while dumping events for schema '%s': %s

- Error number: 52035; Symbol: SHERR_DUMP_SD_ROUTINE_DDL_ERROR

  Message: Error while dumping routines for schema '%s': %s

- Error number: 52036; Symbol: SHERR_DUMP_ACCOUNT_WITH_APOSTROPHE

  Message: Account %s contains the ' character, which is not supported

- Error number: 52037; Symbol: SHERR_DUMP_USERS_MARIA_DB_NOT_SUPPORTED

  Message: Dumping user accounts is currently not supported in MariaDB. Set the 'users' option to false to continue.

- Error number: 52038; Symbol: SHERR_DUMP_INVALID_GRANT_STATEMENT

  Message: Dump contains an invalid grant statement. Use the 'strip_invalid_grants' compatibility option to fix this.

Error numbers in the range 54000-54999 are for connection and network errors experienced by MySQL Shell's dump loading utility `util.loadDump()`, or by MySQL Shell's instance dump utility `util.dumpInstance()`, schema dump utility `util.dumpSchemas()`, and table dump utility `util.dumpTables()`. In most cases, the error code matches the HTTP error involved – for example, error 54404 occurs when the target of a URL is not found (HTTP 404 Not Found). The following errors might be returned:

- Error number: 54000; Symbol: SHERR_DL_COMMON_CONNECTION_ERROR

Message: %sConnection error: %s.

- Error number: `54100` to `54511`; Symbol: `SHERR_NETWORK_[HTTP error name]`

  Message: Context-specific message

11.6 Dump Loading Utility

MySQL Shell's dump loading utility `util.loadDump()`, introduced in MySQL Shell 8.0.21, supports the import into a MySQL HeatWave Service DB System or a MySQL Server instance of schemas or tables dumped using MySQL Shell's Section 11.5, “Instance Dump Utility, Schema Dump Utility, and Table Dump Utility”. The dump loading utility provides data streaming from remote storage, parallel loading of tables or table chunks, progress state tracking, resume and reset capability, and the option of concurrent loading while the dump is still taking place. To get the best functionality, always use the most recent version available of MySQL Shell's dump and dump loading utilities.

- About the Utility
- Requirements and Restrictions
- Using PARs to Load Dump Files
- Running the Utility
- Options for Load Control
- Options for Load Content
- Options for MySQL HeatWave Service and Oracle Cloud Infrastructure
- Options for S3-compatible Services
- Options for Microsoft Azure Blob Storage
- Options for Filtering
- Generated Invisible Primary Key Mode
- Modifying Dumped Data
- Utility Error Messages

About the Utility

For import into a MySQL HeatWave Service DB System, MySQL Shell must be installed on an Oracle Cloud Infrastructure Compute instance that has access to the MySQL HeatWave Service DB System. If the dump files are in an Oracle Cloud Infrastructure Object Storage bucket, you can access the Object Storage bucket from the Compute instance. If the dump files are on your local system, you need to transfer them to the Oracle Cloud Infrastructure Compute instance using the copy utility of your choice, depending on the operating system you chose for your Compute instance. Ensure the dump was created with the `ocimds` option set to `true` in MySQL Shell's instance dump utility or schema dump utility, for compatibility with MySQL HeatWave Service. MySQL Shell's table dump utility does not use this option.

For output produced by the instance dump utility or schema dump utility, MySQL Shell's dump loading utility uses the DDL files and tab-separated `.tsv` data files to set up the server instance or schema in the target MySQL instance, then loads the data. Dumps containing only the DDL files or only the data files can be used to perform these tasks separately. The dump loading utility also lets you separately apply the DDL files and data files from a regular dump that contains both sorts of files.

You can use options for the utility to include or exclude specified schemas and tables, users and their roles and grants, events, routines, and triggers from the import. Note that users and their roles and

grants are excluded from the load by default. If you specify conflicting include and exclude options or name an object that is not included in the dump files anyway, before MySQL Shell 8.0.28 the situation is ignored and the load proceeds without the object. From MySQL Shell 8.0.28 an error is reported and the load stops so you can correct the options.

For output produced by MySQL Shell's table dump utility, from MySQL Shell 8.0.23, the dump contains the information required to set up the schema that originally contained the table. By default, from that release, the schema is recreated in the target MySQL instance if it does not already exist. Alternatively, you can specify the `schema` option in the dump loading utility to load the table into an alternative schema in the target MySQL instance, which must exist there. In MySQL Shell 8.0.22, the table dump utility's files do not contain the schema information, so the target schema must exist in the target MySQL instance. In that release, by default, the current schema of the global MySQL Shell session is used as the target schema, or the `schema` option can be used to name the schema.

You can carry out a dry run with your chosen set of dump loading options to show what actions would be performed when you run the utility for real with those options.

The `waitDumpTimeout` option lets you apply a dump that is still in the process of being created. Tables are loaded as they become available, and the utility waits for the specified number of seconds after new data stops arriving in the dump location. When the timeout elapses, the utility assumes the dump is complete and stops importing.

Progress state for an import is stored in a persistent progress state file, which records steps successfully completed and steps that were interrupted or failed. By default, the progress state file is named `load-progress.`*`server_uuid`*`.json` and created in the dump directory, but you can choose a different name and location. The dump loading utility references the progress state file when you resume or retry the import for a dump, and skips completed steps. De-duplication is automatically managed for tables that were partially loaded. If you interrupt a dump in progress by using **Ctrl + C**, on the first use of that key combination, no new tasks are started by the utility but existing tasks continue. Pressing **Ctrl + C** again stops existing tasks, resulting in error messages. In either case, the utility can still resume the import from where it stopped.

You can choose to reset the progress state and start the import for a dump again from the beginning, but in this case the utility does not skip objects that were already created and does not manage de-duplication. If you do this, to ensure a correct import, you must manually remove from the target MySQL instance all previously loaded objects from that dump, including schemas, tables, users, views, triggers, routines, and events. Otherwise, the import stops with an error if an object in the dump files already exists in the target MySQL instance. With appropriate caution, you may use the `ignoreExistingObjects` option to make the utility report duplicate objects but skip them and continue with the import. Note that the utility does not check whether the contents of the object in the target MySQL instance and in the dump files are different, so it is possible for the resulting import to contain incorrect or invalid data.

Important

Do not change the data in the dump files between a dump stopping and a dump resuming. Resuming a dump after changing the data has undefined behavior and can lead to data inconsistency and data loss. If you need to change the data after partially loading a dump, manually drop all objects that were created during the partial import (as listed in the progress state file), then run the dump loading utility with the `resetProgress` option to start again from the beginning.

If you need to modify any data in the dump's data files before importing it to the target MySQL instance, you can do this by combining MySQL Shell's parallel table import utility `util.importTable()` with the dump loading utility. To do this, first use the dump loading utility to load only the DDL for the selected table, to create the table on the target server. Then use the parallel table import utility to capture and transform data from the output files for the table, and import it to the target table. Repeat that process as necessary for any other tables where you want to modify the data. Finally, use the dump loading utility to load the DDL and data for any remaining tables that you do not want to modify,

excluding the tables that you did modify. For a description of the procedure, see Modifying Dumped Data.

The tables in a dump are loaded in parallel by the number of threads you specify using the `threads` option, which defaults to 4. If table data was chunked when the dump was created, multiple threads can be used for a table, otherwise each thread loads one table at a time. The dump loading utility schedules data imports across threads to maximize parallelism. From MySQL Shell 8.0.27, a pool of background threads is used to fetch the contents of files. If the dump files were compressed by MySQL Shell's dump utilities, the dump loading utility handles decompression for them.

By default, fulltext indexes for a table are created only after the table is completely loaded, which speeds up the import. You can choose to defer all index creation (except the primary index) until each table is completely loaded. You can also opt to create all indexes during the table import. You can also choose to disable index creation during the import, and create the indexes afterwards, for example if you want to make changes to the table structure after loading.

For an additional improvement to data loading performance, you can disable the `InnoDB` redo log on the target MySQL instance during the import. Note that this should only be done on a new MySQL Server instance (not a production system), and this feature is not available on MySQL DB System. For more information, see Disabling Redo Logging.

Requirements and Restrictions

- MySQL 5.7 or later is required for the destination MySQL instance where the dump is loaded.
- MySQL Shell's dump loading utility from versions of MySQL Shell previous to 8.0.27 cannot load dumps that are created using the dump utilities in MySQL Shell 8.0.27 or later. This is because from MySQL Shell 8.0.27, information is included in the dump metadata about features used in creating the dump. This feature list is not backward compatible, but it supports backward compatibility when new features are added in future releases. To get the best functionality, always use the most recent version available of MySQL Shell's dump and dump loading utilities.
- The dump loading utility uses the `LOAD DATA LOCAL INFILE` statement, so the global setting of the `local_infile` system variable on the target MySQL instance must be `ON` for the duration of the import. By default, this system variable is set to `ON` in a standard MySQL HeatWave Service DB System configuration.
- The `LOAD DATA LOCAL INFILE` statement uses nonrestrictive data interpretation, which turns errors into warnings and continues with the load operation. This process can include assigning default values and implicit default values to fields, and converting invalid values to the closest valid value for the column data type. For details of the statement's behavior, see `LOAD DATA`.
- On the target MySQL instance, the dump loading utility checks whether the `sql_require_primary_key` system variable is set to `ON`, and if it is, returns an error if there is a table in the dump files with no primary key. By default, this system variable is set to `OFF` in a standard MySQL HeatWave Service DB System configuration.
- The dump loading utility does not automatically apply the `gtid_executed` GTID set from the source MySQL instance on the target MySQL instance. The GTID set is included in the dump metadata from MySQL Shell's instance dump utility, schema dump utility, or table dump utility, as the `gtidExecuted` field in the `@.json` dump file. To apply these GTIDs on the target MySQL instance for use with replication, use the `updateGtidSet` option or import them manually, depending on the release of the target MySQL instance and the MySQL Shell release. From MySQL Shell 8.0.23, this is supported on MySQL HeatWave Service DB System instances. See the description of the `updateGtidSet` option for details.

Using PARs to Load Dump Files

MySQL Shell supports loading dump files from an Object Storage bucket using a pre-authenticated request (PAR). PARs provide a way to let users access a bucket or an object without having their own credentials.

Important

Before using this access method, assess the business requirement for and the security ramifications of pre-authenticated access to a bucket or objects in a bucket. A PAR gives anyone who has the PAR access to the targets identified in the request. Carefully manage the distribution of PARs.

From MySQL Shell 8.0.27, MySQL Shell supports using a read access PAR (an Object Read PAR) for all objects in a bucket or objects in a bucket with a specific prefix. For information about creating bucket PARs and prefix PARs, see Using Pre-Authenticated Requests. When using a bucket PAR or prefix PAR, the dump loading utility requires a local progress state file. The content of the file is in JSON format, so a text file with a `.json` extension is appropriate (for example, `progress.json`). The following example shows the syntax for loading dump files using a PAR created for all objects in a bucket:

```
JS> util.loadDump("BucketPARURL", {progressFile: "progress.json"})
```

The same syntax is used to load objects in a bucket with a specific prefix, but in this case, the PAR URL includes the prefix:

```
shell-js> util.loadDump("PrefixPARURL", progressFile: "progress.json"})
```

From MySQL Shell 8.0.22, MySQL Shell supports using a read access PAR (an Object Read PAR) created for a MySQL Shell dump manifest file (`@.manifest.json`) to load data from an Object Storage bucket. For information about creating a PAR for a specific object such as a manifest file, see Using Pre-Authenticated Requests. When the `ociParManifest` option is enabled, the MySQL Shell dump loading utility creates a manifest file when exporting data to an Object Storage bucket. The manifest file contains a PAR for each item in the dump. Prior to MySQL 8.0.27, if the `ocimds` option is enabled and a bucket name is provided by the `osBucketName` option, `ociParManifest` is enabled automatically. From MySQL Shell 8.0.27, with the introduction of support for PARs for all objects in a bucket or objects in a bucket with a specific prefix, the `ociParManifest` option is set to `false` by default and is only enabled if set to `true` explicitly.

When using a PAR created for a manifest file, a progress state file is required. The content of the file is in JSON format, so a text file with a `.json` extension is appropriate (for example, `progress.json`). The progress state file can be created in the same prefixed location as the dump files in the Object Storage bucket, or it can be created locally. If the progress state file is created in the Object Storage bucket, you must create a read-write access PAR (an Object Read Write PAR) for the progress state file. For information about creating a PAR for a specific object, see Using Pre-Authenticated Requests. You can use any user account with the required permissions to create a PAR for the progress state file. A local progress state file does not require a PAR. Consider using a local progress state file if you do not have the permissions required to create a PAR. Note that a local progress file does not permit resuming progress from a different location in the event of a failure.

Note

Creating a dump with the `ociParManifest` option enabled generates a manifest file containing a PAR for each item in the dump. Generating PARs for each item in a dump is time consuming for large datasets, and an additional PAR must be created for the manifest file and possibly for a progress state file. Also, when PARs expire, the dump must be recreated to regenerate PARs for the items in the dump. For these reasons, using a bucket or prefix PAR (supported from MySQL Shell 8.0.27) is the recommended method for loading MySQL Shell dump files from an Object Storage bucket. When using a bucket or prefix PAR, there is only a single PAR to create and manage, and PARs are not required for each item in the dump.

The following example shows the syntax for loading dump files using PARs created for the manifest file and a progress state file. If using a local progress state file, the `progressFile` option specifies the path to the local progress state file instead of a PAR URL.

```
shell-js> util.loadDump("PARURLofManifest", {osBucketName: "mds-bucket",
          osNamespace: "NamespaceID", progressFile: "RWPARUrlOfJsonProgressFile"})
```

While the dump is still in progress, the dump loading utility monitors and waits for new additions to the manifest file, rather than to the Object Storage bucket.

Running the Utility

The dump loading utility uses the MySQL Shell global session to obtain the connection details of the target MySQL instance to which the dump is to be imported. You must open the global session (which can have an X Protocol connection or a classic MySQL protocol connection) before running the utility. The utility opens its own sessions for each thread, copying options such as connection compression and SSL options from the global session, and does not make any further use of the global session.

In the MySQL Shell API, the dump loading utility is a function of the `util` global object, and has the following signature:

```
util.loadDump(url[, options])
```

`options` is a dictionary of options that can be omitted if it is empty. The options are listed in the remaining sections in this topic.

If you are importing a dump that is located in the Oracle Cloud Infrastructure Compute instance's filesystem where you are running the utility, `url` is a string specifying the path to a local directory containing the dump files. You can prefix a local directory path with the `file://` schema. In this example in MySQL Shell's JavaScript mode, a dry run is carried out to check that there will be no issues when the dump files are loaded from a local directory into the connected MySQL instance:

```
shell-js> util.loadDump("/mnt/data/worlddump", {dryRun: true})
```

If you are importing a dump from an Oracle Cloud Infrastructure Object Storage bucket, `url` is the path prefix that the dump files have in the bucket, which was assigned using the `outputUrl` parameter when the dump was created. Use the `osBucketName` option to provide the name of the Object Storage bucket, and the `osNamespace` option to identify the namespace for the bucket. In this example in MySQL Shell's JavaScript mode, the dump prefixed `worlddump` is loaded from an Object Storage bucket into the connected MySQL HeatWave Service DB System using 8 threads:

```
shell-js> util.loadDump("worlddump", {
          threads: 8, osBucketName: "hanna-bucket", osNamespace: "idx28w1ckztq"})
```

The namespace for an Object Storage bucket is displayed in the **Bucket Information** tab of the bucket details page in the Oracle Cloud Infrastructure console, or can be obtained using the Oracle Cloud Infrastructure command line interface. A connection is established to the Object Storage bucket using the default profile in the default Oracle Cloud Infrastructure CLI configuration file, or alternative details that you specify using the `ociConfigFile` and `ociProfile` options. For instructions to set up a CLI configuration file, see SDK and CLI Configuration File.

Options for Load Control

`dryRun: [ true | false ]` Display information about what actions would be performed given the specified options and dump files, including any errors that would be returned based on the dump contents, but do not proceed with the import. The default is `false`.

`waitDumpTimeout: number` Setting this option to a value greater than 0 activates concurrent loading of the dump while it is still being produced. The value is a timeout (in seconds) for which the utility waits for further data after all uploaded data chunks in the dump location have been processed. This allows the utility to import the dump while it is still in the process of being created. Data is processed as it becomes available, and the import stops when the timeout is exceeded with

no further data appearing in the dump location. The default setting, `0`, means that the utility marks the dump as complete when all uploaded data chunks have been processed and does not wait for more data. With the default setting, concurrent loading is disabled.

`schema: "string"`

The target schema into which a dump produced by MySQL Shell's dump utilities must be loaded.

From MySQL Shell 8.0.31, if the schema does not exist, it is created, and the dump is loaded to that new schema. If the new schema name differs from the schema name in the dump, the dump is loaded to the new schema, but no changes are made to the loaded data. That is, any reference to the old schema name remains in the data. All stored procedures, views, and so on, refer to the original schema, not the new one.

This load option is supported for single schema dumps, or for filtering options which result in a single schema.

From MySQL Shell 8.0.23, this option is not required, because the dump files from the table dump utility contain the information required to set up the schema that originally contained the table. By default, from that release, the schema is recreated in the target MySQL instance if it does not already exist. Alternatively, you can specify the `schema` option to load the table into an alternative schema in the target MySQL instance, which must exist there.

In MySQL Shell 8.0.22, the dump files from the table dump utility do not contain the schema information, so the target schema must exist in the target MySQL instance. In that release, by default, the current schema of the global shell session is used as the target schema, or the `schema` option can be used to name the target schema.

`threads: number`

The number of parallel threads to use to upload chunks of data to the target MySQL instance. Each thread has its own connection to the MySQL instance. The default is 4. if the dump was created with chunking enabled (which is the default), the utility can use multiple threads to load data for a table; otherwise a thread is only used for one table.

`backgroundThreads: number`

The number of threads in the pool of background threads used to fetch the contents of files. This option, and the thread pool, are available from MySQL Shell 8.0.27. The default is the value of the `threads` option for a dump loaded from the local server, or four times the value of the `threads` option for a dump loaded from a non-local server.

`progressFile: "string"`

Specifies the path to a local progress state file for tracking load progress. Other values are permitted depending on the type of load operation:

When loading a dump from local storage:

- The `progressFile` option may be omitted. In this case, a progress state file named `load-progress-server-uuid.json` is automatically created in the dump directory.
- The `progressFile` option can be set to an empty string to disable progress state tracking, which means that the dump loading utility cannot resume a partially completed import.

When loading a dump from OCI Object Storage using a pre-authenticated request (PAR), the `progressFile` option is mandatory.

- If the load operation is performed using a bucket or prefix PAR, set the `progressFile` option to the path of a local progress state file.
- If the load operation is performed using a manifest file PAR, set the `progressFile` option to the path of a local progress state file or specify a write PAR for a progress state file residing in the same location as the manifest file.

If a local progress state file or a valid write PAR is specified but the progress state file does not exist, the file will be created.

`showProgress: [ true | false ]`

Display (`true`) or hide (`false`) progress information for the import. The default is `true` if `stdout` is a terminal (`tty`), such as when MySQL Shell is in interactive mode, and `false` otherwise. The progress information includes the number of active threads and their actions, the amount of data loaded so far, the percentage complete and the rate of throughput. When the progress information is not displayed, progress state is still recorded in the dump loading utility's progress state file.

`resetProgress: [ true | false ]`

Setting this option to `true` resets the progress state and starts the import again from the beginning. The default is `false`. Note that with this option, the dump loading utility does not skip objects that were already created and does not manage de-duplication. If you want to use this option, to ensure a correct import, you must first manually remove from the target MySQL instance all previously loaded objects, including schemas, tables, users, views, triggers, routines, and events from that dump. Otherwise, the import stops with an error if an object in the dump files already exists in the target MySQL instance. With appropriate caution, you may use the `ignoreExistingObjects` option to make the utility report duplicate objects but skip them and continue with the import.

`skipBinlog: [ true | false ]`

Skips binary logging on the target MySQL instance for the sessions used by the utility during the course of the import, by issuing a `SET sql_log_bin=0` statement. The default is `false`, so binary logging is active by default. For MySQL HeatWave Service DB System, this option is not used, and the import stops with an error if you attempt to set it to `true`. For other MySQL instances, always set `skipBinlog` to `true` if you are applying the `gtid_executed` GTID set from the source MySQL instance on the target MySQL instance, either using the `updateGtidSet` option or manually. When GTIDs are in use on the target MySQL instance (`gtid_mode=ON`), setting this option to `true` prevents new GTIDs from being generated and assigned as the import is being carried out, so that the original GTID set from the source server can be used. The user account must have the required permissions to set the `sql_log_bin` system variable.

`ignoreVersion: [ true | false ]`

Import the dump even if the major version number of the MySQL instance from which the data was dumped is different to the major version number of the MySQL instance to which the data will be uploaded. The default is `false`, meaning that an error is issued and the import does not proceed if the major version number is different.

When this option is set to `true`, a warning is issued and the import proceeds. Note that the import will only be successful if the schemas in the dump files have no compatibility issues with the new major version.

Before attempting an import using the `ignoreVersion` option, use MySQL Shell's upgrade checker utility `checkForServerUpgrade()` to check the schemas on the source MySQL instance. Fix any compatibility issues identified by the utility before dumping the schemas and importing them to the target MySQL instance.

`ignoreExistingObjects: [ true | false ]`

Import the dump even if it contains objects that already exist in the target schema in the MySQL instance. The default is `false`, meaning that an error is issued and the import stops when a duplicate object is found, unless the import is being resumed from a previous attempt using a progress state file, in which case the check is skipped. When this option is set to `true`, duplicate objects are reported but no error is generated and the import proceeds. This option should be used with caution, because the utility does not check whether the contents of the object in the target MySQL instance and in the dump files are different, so it is possible for the resulting import to contain incorrect or invalid data. An alternative strategy is to use the `excludeTables` option to exclude tables that you have already loaded where you have verified the object in the dump files is identical with the imported object in the target MySQL instance. The safest choice is to remove duplicate objects from the target MySQL instance before restarting the dump.

From MySQL Shell 8.0.23, this option also permits the import of a dump created without the use of the `ocimds` option into a MySQL HeatWave Service instance.

`handleGrantErrors: [ abort | drop_account | ignore ]`

The action taken in the event of errors related to `GRANT` or `REVOKE` errors.

- `abort`: (default) stops the load process and displays an error.
- `drop_account`: deletes the account and continues the load process.
- `ignore`: ignores the error and continues the load process.

`characterSet: "string"`

The character set to be used for the import to the target MySQL instance, for example in the `CHARACTER SET` option of the `LOAD DATA` statement. The default is the character set given in the dump metadata that was used when the dump was created by MySQL Shell's instance dump utility, schema dump utility, or table dump utility, which default to using `utf8mb4`. The character set must be permitted by the `character_set_client` system variable and supported by the MySQL instance.

`maxBytesPerTransaction: number`

The maximum number of bytes that can be loaded from a data file in a single `LOAD DATA` statement. If a data file exceeds the `maxBytesPerTransaction` value, multiple `LOAD DATA` statements load data from the file in chunks less than or equal to the

`maxBytesPerTransaction` value. This option is available from MySQL Shell 8.0.27.

The unit suffixes `k` for kilobytes, `M` for megabytes, and `G` for gigabytes can be used. The minimum value is 4069 bytes. If a lesser value is specified, the 4096 byte minimum is used implicitly. If the `maxBytesPerTransaction` option is unset, the `bytesPerChunk` value used to dump the data is used as the default setting for files larger than 1.5 * the `bytesPerChunk` value. If the `maxBytesPerTransaction` option is unset and the data file is less than 1.5 * the `bytesPerChunk` value, the data is requested in a single `LOAD DATA` statement.

If a data file contains a row that is larger than the `maxBytesPerTransaction` setting, the row's data is requested in a single `LOAD DATA` statement. A warning is emitted for the first row encountered that exceeds the `maxBytesPerTransaction` setting.

If a load operation with a configured `maxBytesPerTransaction` setting is interrupted and resumes execution, chunks that were already loaded are skipped. The resumed load operation uses the current `maxBytesPerTransaction` setting. The setting used before the operation was interrupted is not saved to the progress state file.

An intended use for this option is to load data in smaller chunks when a data file is too large for the target server's limits, such as the limits defined by the server's `group_replication_transaction_size_limit` or `max_binlog_cache_size` settings. For example, If you receive the error `"MySQL Error 1197 (HY000): Multi-statement transaction required more than 'max_binlog_cache_size' bytes of storage"` when loading data, set `maxBytesPerTransaction` to a value less than or equal to the server instance's `max_binlog_cache_size` setting.

`sessionInitSql: *list of strings*`

A list of SQL statements to run at the start of each client session used for loading data into the target MySQL instance. You can use this option to change session variables. This option is available from MySQL Shell 8.0.30. For example, the following statements skip binary logging on the target MySQL instance for the sessions used by the utility during the course of the import, and increase the number of threads available for index creation:

```
sessionInitSQL: ["SET SESSION sql_log_bin=0;", "SET SESSION innodb_ddl_thre
```

If an error occurs while running the SQL statements, the import stops and returns an error message.

Options for Load Content

`loadIndexes: [ true | false ]`

Create (`true`) or do not create (`false`) secondary indexes for tables. The default is `true`. When this option is set to `false`, secondary indexes are not created during the import, and you must create them afterwards. This can be useful if you are loading the DDL files and data files separately, and if you want to make changes to the table structure after loading the DDL files. Afterwards, you can create the secondary indexes by running the

dump loading utility again with `loadIndexes` set to `true` and `deferTableIndexes` set to `all`.

As of 8.0.31, MySQL Shell utilizes MySQL Server's parallel index creation. Previously, the dump loading utilities added indexes sequentially, one at a time. As of this release, all indexes in a table are added simultaneously.

See Configuring Parallel Threads for Online DDL Operations for restrictions and configuration.

`deferTableIndexes: [ off | fulltext | all ]`

Defer the creation of secondary indexes until after the table data is loaded. This can reduce loading times. `off` means all indexes are created during the table load. The default setting `fulltext` defers full-text indexes only. `all` defers all secondary indexes and only creates primary indexes during the table load, and also (from MySQL Shell 8.0.22) indexes defined on columns containing auto-increment values. In MySQL Shell 8.0.21, do not set `all` if you have any unique key columns containing auto-increment values.

`analyzeTables: [ off | on | histogram ]`

Execute `ANALYZE TABLE` for tables when they have been loaded. `on` analyzes all tables, and `histogram` analyzes only tables that have histogram information stored in the dump. The default is `off`. You can run the dump loading utility with this option to analyze the tables even if the data has already been loaded.

`showMetadata: [ true | false ]`

Prints the `gtid_executed` GTID set and the binary log file name and position from the source instance, taken from the dump metadata included with dumps produced by MySQL Shell's instance dump utility, schema dump utility, or table dump utility. The metadata is printed in YAML format. This option is available from MySQL Shell 8.0.24.

The `gtid_executed` GTID set is always included in the dump as the `gtidExecuted` field in the `@.json` dump file. The dump loading utility does not automatically apply the `gtid_executed` GTID set from the source MySQL instance on the target MySQL instance. To apply these GTIDs on the target MySQL instance for use with replication, use the `updateGtidSet` option or import them manually, depending on the release of the target MySQL instance. From MySQL Shell 8.0.23, this is supported on MySQL HeatWave Service DB System instances. See the description of the `updateGtidSet` option for details.

The binary log file name and position are included provided that the user account used to run the dump utility had the `REPLICATION CLIENT` privilege. The binary log file name and position can be used to set up replication from a source server that does not have GTIDs enabled and does not use GTID-based replication, to a replica that has GTIDs enabled, using the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` option of the `CHANGE REPLICATION SOURCE TO` statement (which is available from MySQL Server 8.0.23).

`updateGtidSet: [ off | append | replace ]`

Apply the `gtid_executed` GTID set from the source MySQL instance, as recorded in the dump metadata, to the `gtid_purged` GTID set on the target MySQL instance. The `gtid_purged` GTID set holds the GTIDs of all transactions that have been applied on the server, but do not exist on any binary log file on the server. This option is available from MySQL Shell 8.0.22, but in that release it

is not supported on MySQL HeatWave Service DB System due to a permissions restriction. From MySQL 8.0.23, the option can also be used for a MySQL HeatWave Service DB System instance. The default is `off`, meaning that the GTID set is not applied.

Do not use this option for a dump produced by MySQL Shell's table dump utility, only for dumps produced by MySQL Shell's instance dump utility or schema dump utility. Also, do not use this option when Group Replication is running on the target MySQL instance.

For MySQL instances that are not MySQL HeatWave Service DB System instances, when you set `append` or `replace` to update the GTID set, also set the `skipBinlog` option to `true`. This ensures the GTIDs on the source server match the GTIDs on the target server. For MySQL HeatWave Service DB System instances, this option is not used.

For a target MySQL instance from MySQL 8.0, you can set the option to `append`, which appends the `gtid_executed` GTID set from the source MySQL instance to the `gtid_purged` GTID set on the target MySQL instance. The `gtid_executed` GTID set to be applied, which is shown in the `gtidExecuted` field in the `@.json` dump file, must not intersect with the `gtid_executed` set already on the target MySQL instance. For example, you can use this option when importing a schema from a different source MySQL instance to a target MySQL instance that already has schemas from other source servers.

You can also use `replace` for a target MySQL instance from MySQL 8.0, to replace the `gtid_purged` GTID set on the target MySQL instance with the `gtid_executed` GTID set from the source MySQL instance. To do this, the `gtid_executed` GTID set from the source MySQL instance must be a superset of the `gtid_purged` GTID set on the target MySQL instance, and must not intersect with the set of transactions in the target's `gtid_executed` GTID set that are not in its `gtid_purged` GTID set.

For a target MySQL instance at MySQL 5.7, set the option to `replace`, which replaces the `gtid_purged` GTID set on the target MySQL instance with the `gtid_executed` GTID set from the source MySQL instance. In MySQL 5.7, to do this the `gtid_executed` and `gtid_purged` GTID sets on the target MySQL instance must be empty, so the instance must be unused with no previously imported GTID sets.

In MySQL Shell 8.0.21, where this option is not available, you can apply the GTID set manually on a MySQL Server instance (except where Group Replication is in use). For MySQL HeatWave Service DB System, this method is not supported. To apply the GTID set, after the import, use MySQL Shell's `\sql` command (or enter SQL mode) to issue the following statement on the connected MySQL instance, copying the `gtid_executed` GTID set from

the `gtidExecuted` field in the `@.json` dump file in the dump metadata:

```
shell-js> \sql SET @@GLOBAL.gtid_purged= "+gtidExecuted_set";
```

This statement, which works from MySQL 8.0, adds the source MySQL Server instance's `gtid_executed` GTID set to the target MySQL instance's `gtid_purged` GTID set. For MySQL 5.7, the plus sign (`+`) must be omitted, and the `gtid_executed` and `gtid_purged` GTID sets on the target MySQL instance must be empty. For more details, see the description of the `gtid_purged` system variable in the release of the target MySQL instance.

`createInvisiblePKs: [ true | false ]`

Add primary keys in invisible columns for each table in the dump that does not contain a primary key. The `true` setting is applied automatically if the dump was created with the `create_invisible_pks` option by MySQL Shell's instance dump utility `util.dumpInstance()`, schema dump utility `util.dumpSchemas()`, or table dump utility `util.dumpTables()`. The primary keys are only added if the DDL for the dump is loaded (`loadDdl: true`). The invisible columns (which are named "`my_row_id`") have no impact on applications that use the uploaded tables.

`createInvisiblePKs` is present from MySQL Shell 8.0.24, and when the `true` setting is in effect, the target MySQL instance must be MySQL Server 8.0.24 or later, or the load fails. Invisible columns are available from MySQL Server 8.0.23, but a limitation on them in that release prevents the use of this function. The dump loading utility in versions of MySQL Shell before MySQL Shell 8.0.24 silently ignores the dump metadata flag and does not add the primary keys, so ensure that you use the latest version of the utility.

Adding primary keys in this way does not yet enable inbound replication of the modified tables to a MySQL HeatWave Service High Availability DB System, as that feature currently requires the primary keys to exist in both the source server and the replica server. If possible, instead of using this option, consider creating primary keys in the tables on the source server, before dumping them again. From MySQL 8.0.23, you can do this with no impact to applications by using invisible columns to hold the primary keys. This is a best practice for performance and usability, and helps the dumped database to work seamlessly with MySQL HeatWave Service.

Options for MySQL HeatWave Service and Oracle Cloud Infrastructure

`osBucketName: "string"`

The name of the Oracle Cloud Infrastructure Object Storage bucket where the dump files are located. By default, the `[DEFAULT]` profile in the Oracle Cloud Infrastructure CLI configuration file located at `~/.oci/config` is used to establish a connection to the bucket. You can substitute an alternative profile to be used for the connection with the `ociConfigFile` and `ociProfile` options. For instructions to set up a CLI configuration file, see SDK and CLI Configuration File.

`osNamespace: "string"`

The Oracle Cloud Infrastructure namespace where the Object Storage bucket named by `osBucketName` is located. The namespace for an Object Storage bucket is displayed in the Bucket

Information tab of the bucket details page in the Oracle Cloud Infrastructure console, or can be obtained using the Oracle Cloud Infrastructure command line interface.

`ociConfigFile: "string"` An Oracle Cloud Infrastructure CLI configuration file that contains the profile to use for the connection, instead of the one in the default location `~/.oci/config`.

`ociProfile: "string"` The profile name of the Oracle Cloud Infrastructure profile to use for the connection, instead of the `[DEFAULT]` profile in the Oracle Cloud Infrastructure CLI configuration file used for the connection.

Options for S3-compatible Services

MySQL Shell supports loading dumps stored in S3-compatible buckets, such as Amazon Web Services (AWS) S3.

Note

MySQL Shell supports AWS S3 configuration in command line options, environment variables, and configuration files. Command line options override environment variables, configuration files, and default options.

For information on configuration requirements, see Section 4.7, “Cloud Service Configuration”.

`s3BucketName: "string"` The name of the S3 bucket where the dump files are located. By default, the `default` profile in the Amazon Web Services (AWS) CLI `config` and `credentials` files located at `~/.aws/` are used to establish a connection to the S3 bucket. You can substitute alternative configurations and credentials for the connection with the `s3ConfigFile` and `s3CredentialsFile` options. For instructions on installing and configuring the AWS CLI, see Getting started with the AWS CLI.

`s3CredentialsFile: "string"` A credentials file that contains the user's credentials to use for the connection, instead of the one in the default location, `~/.aws/credentials`. Typically, the credentials file contains the `aws_access_key_id` and `aws_secret_access_key` to use for the connection.

`s3ConfigFile: "string"` An AWS CLI configuration file that contains the profile to use for the connection, instead of the one in the default location `~/.aws/config`. Typically, the config file contains the region and output type to use for the connection.

`s3Profile: "string"` The profile name of the s3 CLI profile to use for the connection, instead of the `default` profile in the AWS CLI configuration file used for the connection.

`s3Region: "string"` The name of the region to use for the connection.

`s3EndpointOverride: "string"` The URL of the endpoint to use instead of the default.

When connecting to the Oracle Cloud Infrastructure S3 compatbility API, the endpoint takes the following format: `https://namespace.compat.objectstorage.region.oraclecloud.co`
Replace *`namespace`* with the Object Storage namespace and *`region`* with your region identifier. For example, the region identifier for the US East (Ashburn) region is `us-ashburn-1`.

For a namespace named axaxnpcrorw5 in the US East (Ashburn) region:

`https://axaxnpcrorw5.compat.objectstorage.us-ashburn-1.oraclecloud.com`.

The following example shows the load of a MySQL dump from a folder, `test`, in an AWS S3 bucket, `Bucket001`, :

```
util.loadDump("test",{s3BucketName: "Bucket001", threads: 4})
```

The following example shows the load of a MySQL dump from a prefix, `test`, in an Object Storage bucket, `Bucket001`, using a configuration profile, `oci`, and the `s3EndpointOverride` to direct the connection to the OCI endpoint of the required tenancy and region:

```
util.loadDump("test",{s3BucketName: "Bucket001",
s3EndpointOverride: "https://axaxnpcrorw5.compat.objectstorage.us-ashburn-1.oraclecloud.com",
s3Profile: "oci", threads: 4})
```

Options for Microsoft Azure Blob Storage

MySQL Shell supports loading from Microsoft Azure Blob Storage.

Note

MySQL Shell supports Microsoft Azure Blob Storage configuration in command line options, environment variables, and configuration files. Command line options override environment variables, and configuration files.

For information on configuration requirements and the order of precedence of the configuration types, see Section 4.7, “Cloud Service Configuration”.

`azureContainerName: "string"`
Mandatory. The name of the Azure container from which the dump is to be loaded. The container must exist.

`azureConfigFile: "string"`
Optional. A configuration file that contains the storage connection parameters, instead of the one in the default location, such as `~/.azure/config`. If this is not defined, the default configuration file is used.

`azureContainerName` must be defined, and not be empty.

`azureStorageAccount: "string"`
Optional. The name of the Azure storage account to use for the operation.

`azureStorageSasToken: "string"`
Optional. Azure Shared Access Signature (SAS) token to be used for the authentication of the operation, instead of a key.

In the following example, the configuration uses a configuration string for the connection parameters, which means the dump command only requires the `azureContainerName`.

Example `config` file:

```
[cloud]
 name = AzureCloud
```

```
[storage]
 connection_string=alphanumericConnectionString
```

Example `loadDump` command, which imports the contents of a folder named `prefix1`, in a container named `mysqlshellazure`, to the connected MySQL instance:

```
util.loadDump("prefix1", {azureContainerName: "mysqlshellazure", threads: 4})
```

Options for Filtering

`loadDdl: [ true | false ]`

Setting this option to `false` excludes the DDL files in the dump from the load. The default is `true`, meaning that the DDL files are loaded.

`loadData: [ true | false ]`

Setting this option to `false` excludes the data files in the dump from the load. The default is `true`, meaning that the data files are loaded.

`loadUsers: [ true | false ]`

Import (`true`) or do not import (`false`) users and their roles and grants into the target MySQL instance. The default is `false`, so users are not imported by default. Statements for the current user are skipped. From MySQL Shell 8.0.22, if a user already exists in the target MySQL instance, an error is returned and the user's grants from the dump files are not applied. From MySQL Shell 8.0.22, you can use the `excludeUsers` or `includeUsers` option in the dump loading utility to specify user accounts to be excluded or included in the import.

Note

In MySQL Shell 8.0.21, attempting to import users to a MySQL HeatWave Service DB System causes the import to fail if the `root` user account or another restricted user account name is present in the dump files, so the import of users to a MySQL HeatWave Service DB System is not supported in that release.

MySQL Shell's schema dump utility and table dump utility do not include users, roles, and grants in a dump, but the instance dump utility can, and does by default. From MySQL Shell 8.0.22, the `excludeUsers` and `includeUsers` options can also be used in the instance dump utility to exclude or include named user accounts from the dump files.

If you specify `true` but the supplied dump files do not contain user accounts, before MySQL Shell 8.0.23, the utility returns an error and stops the import. From MySQL Shell 8.0.23, the utility instead returns a warning and continues.

`excludeUsers: array of strings`

Exclude the named user accounts from the import. This option is available from MySQL Shell 8.0.22, and you can use it to exclude user accounts that are not accepted for import to a MySQL HeatWave Service DB System, or that already exist or are not wanted on the target MySQL instance. Specify each user account string in the format `"'user_name'@'host_name'"` for an account that is defined with a user name and host name, or `"'user_name'"` for an account that is defined with a user name

	only. If you do not supply a host name, all accounts with that user name are excluded.
`includeUsers: array of strings`	Include only the named user accounts in the import. Specify each user account string as for the `excludeUsers` option. This option is available from MySQL Shell 8.0.22, and you can use it as an alternative to `excludeUsers` if only a few user accounts are required in the target MySQL instance. You can also specify both options to include some accounts and exclude others.
`excludeSchemas: array of strings`	Exclude the named schemas from the import. Note that the `information_schema`, `mysql`, `ndbinfo`, `performance_schema`, and `sys` schemas are always excluded from a dump that is created by MySQL Shell's instance dump utility.
`includeSchemas: array of strings`	Load only the named schemas from the dump files. You can specify both options to include some schemas and exclude others.
`excludeTables: array of strings`	Exclude the named tables from the import, so that they are not uploaded to the target MySQL instance. Table names must be qualified with a valid schema name, and quoted with the backtick character if needed. Note that the data for the `mysql.apply_status`, `mysql.general_log`, `mysql.schema`, and `mysql.slow_log tables` is always excluded from a dump created by MySQL Shell's schema dump utility, although their DDL statements are included.
`includeTables: array of strings`	Load only the named tables from the dump files. Table names must be qualified with a valid schema name, and quoted with the backtick character if needed. You can specify both options to include some tables and exclude others.
`excludeEvents: array of strings`	Exclude the named events from the import. This option is available from MySQL Shell 8.0.28. Names of events must be qualified with a valid schema name, and quoted with the backtick character if needed.
`includeEvents: array of strings`	Load only the named events from the dump files. This option is available from MySQL Shell 8.0.28. Event names must be qualified with a valid schema name, and quoted with the backtick character if needed.
`excludeRoutines: array of strings`	Exclude the named functions and stored procedures from the import. This option is available from MySQL Shell 8.0.28. Names of routines must be qualified with a valid schema name, and quoted with the backtick character if needed.
`includeRoutines: array of strings`	Load only the named functions and stored procedures from the dump files. This option is available from MySQL Shell 8.0.28. Names of routines must be qualified with a valid schema name, and quoted with the backtick character if needed.
`excludeTriggers: array of strings`	Exclude the named triggers from the import. This option is available from MySQL Shell 8.0.28. Names of triggers must be qualified with a valid schema name and table name (`schema.table.trigger`), and quoted with the backtick character if needed. You can exclude all triggers for a specific table by specifying a schema name and table name with this option (`schema.table`).
`includeTriggers: array of strings`	Load only the named triggers from the dump files. This option is available from MySQL Shell 8.0.28. Names of triggers

must be qualified with a valid schema name and table name (`schema.table.trigger`), and quoted with the backtick character if needed. You can include all triggers for a specific table by specifying a schema name and table name with this option (`schema.table`).

Generated Invisible Primary Key Mode

MySQL Server 8.0.30 introduced GIPK mode, Generated Invisible Primary Keys. When running in this mode, for any InnoDB table that is created without an explicit primary key, the MySQL server automatically adds a generated invisible primary key (GIPK) to the table. This mode is enabled by setting `sql_generate_invisible_primary_key` to ON.

MySQL Shell's load utility option `createInvisiblePKs` uses the server's GIPK mode to generate invisible primary keys for tables which do not have primary keys.

Under certain circumstances, if a user has insufficient privileges to use GIPK mode, MySQL Shell can fall back to the previous method of generating invisible primary keys.

If `createInvisiblePKs:false` and `sql_generate_invisible_primary_key=OFF`, primary keys are not generated for any table loaded from the dump.

If `createInvisiblePKs:false` and `sql_generate_invisible_primary_key=ON`, MySQL Shell attempts to set `sql_generate_invisible_primary_key=OFF`. If the change is successful, primary keys are not generated for any table loaded from the dump.

If `createInvisiblePKs:true` and `sql_generate_invisible_primary_key=OFF`, MySQL Shell attempts to set `sql_generate_invisible_primary_key=ON`. If the change is successful, primary keys are generated for every table without primary keys loaded from the dump.

If `createInvisiblePKs:true` and `sql_generate_invisible_primary_key=ON`, primary keys are generated for every table loaded using the MySQL Server GIPK mode.

If the user running the MySQL Shell load utility does not have the required MYSQL Server privileges, the attempt to set `sql_generate_invisible_primary_key` fails. If the attempt fails and `createInvisiblePKs:true`, MySQL Shell generates the invisible primary keys for any table which does not have them.

If the GIPK server option, `sql_generate_invisible_primary_key` is enabled and the MySQL Shell load utility option `createInvisiblePKs` is disabled, and the user does not have the required privileges to change `sql_generate_invisible_primary_key`, an error is generated and the load fails. It is possible to override this behaviour by enabling the MySQL Shell environment variable, `MYSQLSH_ALLOW_ALWAYS_GIPK`. This environment variable overrides `createInvisiblePKs:false`, enabling invisible private keys on all loaded tables, if the `sql_generate_invisible_primary_key` is enabled.

If the MySQL Shell load utility option `createInvisiblePKs` is enabled, but the GIPK server option, `sql_generate_invisible_primary_key` is disabled, MySQL Shell attempts to enable `sql_generate_invisible_primary_key`. If the user has the appropriate privileges on the target MySQL server, `sql_generate_invisible_primary_key` is enabled, and the load utility uses the server's GIPK mode to create the invisible primary keys on the loaded tables. If the user does not have the required privileges, such as on MySQL HeatWave Service, MySQL Shell falls back to the previous behaviour and creates the primary keys on the table without using the server's GIPK mode.

Modifying Dumped Data

MySQL Shell's parallel table import utility `util.importTable()` can be used in combination with the dump loading utility `util.loadDump()` to modify data in the chunked output files before uploading it

to the target MySQL instance. You can modify the data for one table at a time by this method. Follow this procedure, which works from MySQL Shell 8.0.23:

1. Use the dump loading utility with the `loadDdl: true` and `loadData: false` options, to load the DDL file only, and create the selected table on the target MySQL instance with no data.

```
shell-js> util.loadDump("/mnt/data/proddump", {
        > includeTables: ["product.pricing"],
        > loadDdl: true,
        > loadData: false});
```

2. Use the parallel table import utility to capture and transform the data for the table, and import it to the empty table on the target MySQL instance. In this example, the data for the `pricing` table is in multiple compressed files, which are specified using wildcard pattern matching. The values from the `id` and `prodname` columns in the dump files are assigned unchanged to the same columns in the target table. The values from the `price` column in the dump files are captured and assigned to the variable `@1`. The `decodeColumns` option is then used to reduce the prices by a standard amount, and the reduced prices are placed in the `price` column of the target table.

```
shell-js> util.importTable ("/mnt/data/proddump/product@pricing@*.zst", {
        > schema: "product",
        > table: "pricing",
        > columns: ["id",  "prodname",  1],
        > decodeColumns: { "price": "0.8 * @1"}});
```

3. Repeat Steps 1 and 2 as needed for any other tables in the dump files where you need to modify the data.

4. When you have finished uploading all the tables and data that needed to be modified, use the dump loading utility to load both the DDL and the data for any remaining tables that you do not need to modify. Be sure to exclude the tables that you did modify in the previous steps.

```
shell-js> util.loadDump("/mnt/data/proddump", {excludeTables: ["product.pricing"]});
```

Utility Error Messages

Error numbers in the range 53000-53999 are specific to MySQL Shell's dump loading utility `util.loadDump()`. The following errors might be returned:

- Error number: `53000`; Symbol: `SHERR_LOAD_MANIFEST_EXPIRED_PARS`

  Message: The PARs in the manifest file have expired, the expiration time was set to: %s

- Error number: `53001`; Symbol: `SHERR_LOAD_MANIFEST_PAR_MISMATCH`

  Message: The provided PAR must be a file on the dump location: '%s'

- Error number: `53002`; Symbol: `SHERR_LOAD_SPLITTING_DDL_FAILED`

  Message: Error splitting DDL script for table %s: %s

- Error number: `53003`; Symbol: `SHERR_LOAD_SECONDARY_ENGINE_ERROR`

  Message: The table %s has a secondary engine set, but not all indexes have been recreated

- Error number: `53004`; Symbol: `SHERR_LOAD_FAILED_TO_DISABLE_BINLOG`

  Message: 'SET sql_log_bin=0' failed with error: %s

- Error number: `53005`; Symbol: `SHERR_LOAD_WORKER_THREAD_FATAL_ERROR`

  Message: Error loading dump

- Error number: `53006`; Symbol: `SHERR_LOAD_UNSUPPORTED_DUMP_VERSION`

Message: Unsupported dump version

- Error number: 53007; Symbol: SHERR_LOAD_UNSUPPORTED_DUMP_CAPABILITIES

  Message: Unsupported dump capabilities

- Error number: 53008; Symbol: SHERR_LOAD_INCOMPLETE_DUMP

  Message: Incomplete dump

- Error number: 53009; Symbol: SHERR_LOAD_UNSUPPORTED_SERVER_VERSION

  Message: Loading dumps is only supported in MySQL 5.7 or newer

- Error number: 53010; Symbol: SHERR_LOAD_DUMP_NOT_MDS_COMPATIBLE

  Message: Dump is not MDS compatible

- Error number: 53011; Symbol: SHERR_LOAD_SERVER_VERSION_MISMATCH

  Message: MySQL version mismatch

- Error number: 53012; Symbol: SHERR_LOAD_UPDATE_GTID_GR_IS_RUNNING

  Message: The updateGtidSet option cannot be used on server with group replication running.

- Error number: 53013; Symbol: SHERR_LOAD_UPDATE_GTID_APPEND_NOT_SUPPORTED

  Message: Target MySQL server does not support updateGtidSet:'append'.

- Error number: 53014; Symbol: SHERR_LOAD_UPDATE_GTID_REQUIRES_SKIP_BINLOG

  Message: The updateGtidSet option on MySQL 5.7 target server can only be used if the skipBinlog option is enabled.

- Error number: 53015; Symbol: SHERR_LOAD_UPDATE_GTID_REPLACE_REQUIRES_EMPTY_VARIABLES

  Message: The updateGtidSet:'replace' option can be used on target server version only if GTID_PURGED and GTID_EXECUTED are empty, but they are not.

- Error number: 53016; Symbol: SHERR_LOAD_UPDATE_GTID_REPLACE_SETS_INTERSECT

  Message: The updateGtidSet:'replace' option can only be used if gtid_subtract(gtid_executed,gtid_purged) on target server does not intersect with the dumped GTID set.

- Error number: 53017; Symbol: SHERR_LOAD_UPDATE_GTID_REPLACE_REQUIRES_SUPERSET

  Message: The updateGtidSet:'replace' option can only be used if the dumped GTID set is a superset of the current value of gtid_purged on target server.

- Error number: 53018; Symbol: SHERR_LOAD_UPDATE_GTID_APPEND_SETS_INTERSECT

  Message: The updateGtidSet:'append' option can only be used if gtid_executed on target server does not intersect with the dumped GTID set.

- Error number: 53019; Symbol: SHERR_LOAD_INVISIBLE_PKS_UNSUPPORTED_SERVER_VERSION

  Message: The 'createInvisiblePKs' option requires server 8.0.24 or newer.

- Error number: 53020; Symbol: SHERR_LOAD_REQUIRE_PRIMARY_KEY_ENABLED

Message: sql_require_primary_key enabled at destination server

- Error number: `53021`; Symbol: `SHERR_LOAD_DUPLICATE_OBJECTS_FOUND`

  Message: Duplicate objects found in destination database

- Error number: `53022`; Symbol: `SHERR_LOAD_DUMP_WAIT_TIMEOUT`

  Message: Dump timeout

- Error number: `53023`; Symbol: `SHERR_LOAD_INVALID_METADATA_FILE`

  Message: Invalid metadata file %s

- Error number: `53024`; Symbol: `SHERR_LOAD_PARSING_METADATA_FILE_FAILED`

  Message: Could not parse metadata file %s: %s

- Error number: `53025`; Symbol: `SHERR_LOAD_LOCAL_INFILE_DISABLED`

  Message: local_infile disabled in server

- Error number: `53026`; Symbol: `SHERR_LOAD_PROGRESS_FILE_ERROR`

  Message: Error loading load progress file '%s': %s

- Error number: `53027`; Symbol: `SHERR_LOAD_PROGRESS_FILE_UUID_MISMATCH`

  Message: Progress file was created for a server with UUID %s, while the target server has UUID: %s

- Error number: `53028`; Symbol: `SHERR_LOAD_MANIFEST_UNKNOWN_OBJECT`

  Message: Unknown object in manifest: %s

Error numbers in the range 54000-54999 are for connection and network errors experienced by MySQL Shell's dump loading utility `util.loadDump()`, or by MySQL Shell's instance dump utility `util.dumpInstance()`, schema dump utility `util.dumpSchemas()`, and table dump utility `util.dumpTables()`. In most cases, the error code matches the HTTP error involved – for example, error 54404 occurs when the target of a URL is not found (HTTP 404 Not Found). The following errors might be returned:

- Error number: `54000`; Symbol: `SHERR_DL_COMMON_CONNECTION_ERROR`

  Message: %sConnection error: %s.

- Error number: `54100` to `54511`; Symbol: `SHERR_NETWORK_[`*`HTTP error name`*`]`

  Message: Context-specific message

11.7 Diagnostics Utilities

MySQL Shell diagnostic utilities enable you to analyze the performance of your servers and generate diagnostics reports on overall health, performance under load, and individual queries.

11.7.1 collectDiagnostics Utility

The debugging and diagnostics utility `util.debug.collectDiagnostics()` enables you to collect diagnostic data on your MySQL server.

- About the Utility
- Requirements and Restrictions

- Running the Utility
- Options for Collecting Diagnostics

About the Utility

The diagnostic report is generated as a zip file to either the local directory or a specified path.

`util.debug.collectDiagnostics()` enables you to collect raw diagnostic data from standalone servers, members of replication topologies, InnoDB Clusters, and MySQL HeatWave Service DB Systems.

The utility generates files in both TSV and YAML format.

Requirements and Restrictions

The following requirements apply to exports using the diagnostics collection utility:

- MySQL 5.7 or later is required.
- The utility must be run as root.

Running the Utility

The diagnostics utility has the following signature:

```
util.debug.collectDiagnostics("path/",{options})
```

- `"path"`: you can specify a path, filename, or path and filename.

  If a filename is not provided, the file is written to the specified location and the filename `mysql-diagnostics-YYYYMMDD-HHMMSS.zip` is used.

  If a filename is provided without a path, the file is written to the current directory.

- `options`: dictionary of options that can be omitted if empty. See Options for Collecting Diagnostics for the available options.

  If options are not defined, the utility generates a default set of diagnostics. Each option adds one or more reports to the output.

The following example, run on April 6th, 2022, at 10:02:06AM, generates a default set of diagnostics, `mysql-diagnostics-20220406-100206.zip`, in the `C:/Temp/` directory:

```
util.debug.collectDiagnostics("C:/Temp/")
```

The following example generates a default set of diagnostics in a file named `myDiagnostics.zip` in the `C:/Temp/` directory:

```
util.debug.collectDiagnostics("C:/Temp/myDiagnostics.zip")
```

The trailing forward slash is required to define a path. If you omit it, the utility creates a file named `Temp.zip` in the named path. `C:/Temp.zip` for example. It is not possible to overwrite an existing file.

Host information (`host_info`) is collected from the localhost, only. It is not possible to collect host information from a remote host. If your MySQL server is running on a remote host, only the MySQL server information is collected.

On Microsoft Windows platforms, host information is collected using the `MSInfo` utility. This spawns an additional progress dialog while the utility is running.

Options for Collecting Diagnostics

`allMembers: [true|false]`

Default `false`. If set to `true`, generates diagnostics for all members of a managed topology, such as InnoDB Cluster, and pings each member of the topology. Each diagnostic is prefixed with a number, 1 (one) for the server MySQL Shell is connected to, and incremented for each member detected.

Ping results (`ping.txt`) are generated only if Shell is connected to a member of the topology on the localhost. It is not possible to request ping results from a remote host.

`allMembers: true` does not create additional reports. All collected data is included in the default reports.

`innodbMutex: [true|false]`

Default `false`. If set to `true`, collects the output of `SHOW ENGINE INNODB MUTEX`.

This option generates the following additional report:

- `innodb_mutex`: lists the output of `SHOW ENGINE INNODB MUTEX`.

Note

This option can impact performance.

`schemaStats: [true|false]`

Default `false`. If set to `true`, collects schema size statistics.

This option generates the following additional report:

- `schema_object_overview`: lists the contents of the `schema_object_overview` view.
- `top_biggest_tables`: lists the largest tables on the connected server and the slow performance indicators.

`slowQueries: [true|false]`

Default `false`. If set to `true`, collects slow query information from the The Slow Query Log.

This option requires you to enable `slow_log` on the target server and configure its output to `TABLE`.

This option generates the following additional report:

- `slow_log`: lists the contents of the `mysql.slow_log` table.

`ignoreErrors: [true|false]`

Default `false`. If set to `true`, ignores any errors generated by the queries used to generate the diagnostic reports.

`customSql`: *`array`*

One or more SQL statements to run.

For example:

```
{"customSql": ["statement1", "statement2", "statement3"]}
```

`customShell`: *`array`*

One or more shell (DOS, BOURNE, and so on) commands to run.

For example:

```
{"customShell": ["command1", "command2", "command3"]}
```

Important

These commands run with the privileges of the user running MySQL Shell and should be used with caution.

11.7.2 collectHighLoadDiagnostics Utility

- About the Utility
- Requirements and Restrictions
- Running the Utility
- Options for Collecting High Load Diagnostics

About the Utility

`util.debug.collectHighLoadDiagnostics()` runs multiple iterations of diagnostic reporting on your MySQL server, enabling you to analyze multiple aspects of your server while under load.

The High Load diagnostic report is generated as a zip file to either the local directory or a specified path.

The utility generates files in both TSV and YAML format.

Requirements and Restrictions

The following requirements apply to exports using the diagnostics collection utility:

- MySQL 5.7 or later is required.
- The utility must be run as root.

Running the Utility

The diagnostics utility has the following signature:

```
util.debug.collectHighLoadDiagnostics(path, {options})
```

The following example writes the zip file to the user's temp directory, performs 5 iterations of collection, and enables all Performance Schema instruments and consumers:

```
    util.debug.collectHighLoadDiagnostics("/home/userName/temp/highLoad.zip", {iterations: 5, pfsInstrume
```

Note

The data returned by this utility also includes data collected by `util.debug.collectDiagnostics()`.

Options for Collecting High Load Diagnostics

iterations: number	Default is 2. Number of iterations of high load diagnostic data collection.
delay: number	Default is 300 seconds. Number of seconds between iterations of high load diagnostic data collection.

innodbMutex: [true | false]

Default is false. If set to true, the output of `SHOW ENGINE INNODB MUTEX` is collected.

Note

This parameter can affect performance.

pfsInstrumentation: [current | medium | full]

Default is current. Defines which Performance Schema instruments and consumers are used. Possible values are:

- `current`: The currently enabled Performance Schema instruments and consumers. No changes are made to your server's configuration.
- `medium`: Enables all consumers except `%history` and `%history_long`, and all instruments except `wait/synch/%`.
- `full`: Enables all consumers and all instruments.

customSql: *`array`*

One or more SQL statements to run. You can control when the statements are run with the following prefixes:

- `PRE`: Default. The statement is run once, before the metrics collection iterations begin.
- `POST`: The statement is run once, after the metrics collection iterations complete.
- `ITER`: The statement is run once for each iteration of the metrics collection.

For example:

```
{"customSql": ["statement1", "statement2", "statement3"]}
```

customShell *`array`*

One or more Shell commands to run. You can control when the commands are run with the following prefixes:

- `BEFORE`: Default. The statement is run once, before the metrics collection iterations begin.
- `DURING`: The statement is run once for each iteration of the metrics collection.
- `AFTER`: The statement is run once, after the metrics collection iterations complete.

For example:

```
{"customShell": ["command1", "command2", "command3"]}
```

Important

These commands run with the privileges of the user running MySQL Shell and should be used with caution.

11.7.3 collectSlowQueryDiagnostics Utility

- About the Utility
- Requirements and Restrictions
- Running the Utility
- Options for Collecting Diagnostics

About the Utility

`util.debug.collectSlowQueryDiagnostics()` runs multiple iterations of diagnostic reporting on your MySQL server, enabling you to analyze multiple aspects of your server while a specified query is processed.

The diagnostic report is generated as a zip file to either the local directory or a specified path.

The utility generates files in both TSV and YAML format.

Requirements and Restrictions

The following requirements apply to exports using the diagnostics collection utility:

- MySQL 5.7 or later is required.
- The utility must be run as root.

Running the Utility

The diagnostics utility has the following signature:

```
util.debug.collectSlowQueryDiagnostics("path", "query", {options})
```

Note

The data returned by this utility also includes the default data collected by `util.debug.collectDiagnostics()` and `util.debug.collectHighLoadDiagnostics()`.

- `"path"`: the location the diagnostics archive is written to. If empty, it is written to the current directory.

  `"query"`: the SQL query to analyze.

  `"options"`: dictionary of optional arguments. See Options for Collecting Diagnostics.

In addition to the contents of the `util.debug.collectHighLoadDiagnostics()` diagnostics, `util.debug.collectSlowQueryDiagnostics` collects the following information:

- The `EXPLAIN` output of the query.
- The Optimizer trace of the query.
- DDL of the tables used in the query.
- Warnings generated by the query.

Options for Collecting Diagnostics

delay: *number*	Number of seconds to wait between iterations of data collection. Default is 30. Data is collected only as long as the defined query runs. When the query is complete, the data collection stops.

innodbMutex: true | false

If true, also collects the output of `SHOW ENGINE INNODB MUTEX`. This command is disabled by default, as it can have an impact on production performance.

pfsInstrumentation: [current | medium | full]

Defines which Performance Schema instruments and consumers are used. Possible values are:

- `current`: Default. The currently enabled Performance Schema instruments and consumers. No changes are made to your server's configuration.
- `medium`: Enables all consumers except `%history` and `%history_long`, and all instruments except `wait/synch/%`.
- `full`: Enables all consumers and all instruments.

customSql: *`array`*

One or more SQL statements to run. You can control when the statements are run with the following prefixes:

- `BEFORE`, or nothing: Default. The custom SQL is run once, before the metrics collection iterations begin.
- `AFTER`: The custom SQL is run once, after the metrics collection iterations complete.
- `DURING`: The custom SQL is run once for each iteration of the metrics collection.

For example:

```
{"customSql": ["statement1", "statement2", "statement3"]}
```

customShell: *`array`*

One or more shell (DOS, BOURNE, and so on) commands to run. You can control when the commands are run with the following prefixes:

- `BEFORE`: Default. The command(s) run once, before the metrics collection iterations begin.
- `DURING`: The command(s) run once for each iteration of the metrics collection.
- `AFTER`: The command(s) run once, after the metrics collection iterations complete.

For example:

```
{"customShell": ["command1", "command2", "command3"]}
```

Important

These commands run with the privileges of the user running MySQL Shell and should be used with caution.

Chapter 12 MySQL Shell Logging and Debug

Table of Contents

12.1 Application Log 352
12.2 Verbose Output 353
12.3 System Logging for User SQL Statements 354
12.4 MySQL Shell SQL Logging 355
12.5 Logging AdminAPI Operations 357

You can use MySQL Shell's logging feature to verify the state of MySQL Shell while it is running and to troubleshoot any issues.

By default, MySQL Shell sends logging information at logging level 5 (error, warning, and informational messages) to an application log file. You can also configure MySQL Shell to send the information to an optional additional viewable location, and (from MySQL 8.0.17) to the console as verbose output.

You can control the level of detail to be sent to each destination. For the application log and additional viewable location, you can specify any of the available levels as the maximum level of detail. For verbose output, you can specify a setting that maps to a maximum level of detail. The following levels of detail are available:

Table 12.1 Logging levels in MySQL Shell

Logging Level - Numeric	Logging Level - Text	Meaning	Verbose Setting
1	`none`	No logging	0
2	`internal`	Internal Error	1
3	`error`	Error	1
4	`warning`	Warning	1
5	`info`	Informational	1
6	`debug`	Debug	2
7	`debug2`	Debug2	3
8	`debug3`	Debug3	4

From MySQL Shell 8.0.24, you can choose to send SQL statements that you issue interactively in MySQL Shell's SQL mode to the operating system's system logging facility (`syslog` on Unix, or the Windows Event Log). SQL statements that would be excluded from the MySQL Shell code history are not sent to the system logging facility.

By default, MySQL Shell does not log or output SQL statements that are executed by MySQL Shell itself in the course of AdminAPI operations. From MySQL Shell 8.0.18, you can activate logging for these statements if you want to observe the progress of these operations in terms of SQL execution, in addition to the messages returned during the operations. The statements are written to the MySQL Shell application log file as informational messages provided that the logging level is set to 5 or above. They are also sent to the console as verbose output provided that the verbose setting is 1 or above.

By default, MySQL Shell sends all logging for a program to the same application log file, and all output for a program to the same destination. From MySQL Shell 8.0.26, the function `shell.create_context` can be used in MySQL Shell's Python mode to support multithreading by Python programs. The function is used inside a new Python thread to create a scope which isolates logging, interrupts, and delegates. The context wrapper handles and isolates output printed to `stdout` and `stderr` and diagnostic output, and also user input, with separate handling for passwords. You can also create an individual application log file specific to the thread.

For instructions to configure the application log and the optional additional destination, which is `stderr` on Unix-based systems or the `OutputDebugString()` function on Windows systems, see Section 12.1, “Application Log”.

For instructions to send logging information to the console as verbose output, see Section 12.2, “Verbose Output”.

For instructions to send interactive SQL statements to the system logging facility, see Section 12.3, “System Logging for User SQL Statements”.

For instructions to activate logging for SQL statements that are executed by AdminAPI operations, see Section 12.5, “Logging AdminAPI Operations”.

12.1 Application Log

The location of the MySQL Shell application log file is the user configuration path and the file is named `mysqlsh.log`. By default, MySQL Shell sends logging information at logging level 5 (error, warning, and informational messages) to this file.

Note

Log messages are timestamped in UTC format.

To change the level of logging information that is sent, or to disable logging to the application log file, choose one of these options:

- Use the `--log-level` command-line option when starting MySQL Shell.
- Use the MySQL Shell `\option` command to set the `logLevel` MySQL Shell configuration option. For instructions to use this command, see Section 13.4, “Configuring MySQL Shell Options”.
- Use the `shell.options` object to set the `logLevel` MySQL Shell configuration option. For instructions to use this configuration interface, see Section 13.4, “Configuring MySQL Shell Options”.

The available logging levels are as listed in Table 12.1, “Logging levels in MySQL Shell”. If you specify a logging level of 1 or `none` for the option, logging to the application log file is disabled. All other values leave logging enabled and set the level of detail in the log file. The option requires a value.

With the `--log-level` command-line option, you can specify the logging level using its text name or the numeric equivalent, so the following examples have the same effect:

```
$> mysqlsh --log-level=4
$> mysqlsh --log-level=warning
```

With the `logLevel` MySQL Shell configuration option, you can only specify a numeric logging level.

If you prepend the logging level with @ (at sign), log entries are output to an additional viewable location as well as being written to the MySQL Shell log file. The following examples have the same effect:

```
$> mysqlsh --log-level=@8
$> mysqlsh --log-level=@debug3
```

On Unix-based systems, the log entries are output to `stderr` in the output format that is currently set for MySQL Shell. This is the value of the `resultFormat` MySQL Shell configuration option, unless JSON wrapping has been activated by starting MySQL Shell with the `--json` command line option.

On Windows systems, the log entries are printed using the `OutputDebugString()` function, whose output can be viewed in an application debugger, the system debugger, or a capture tool for debug output.

The MySQL Shell log file format is plain text and entries contain a timestamp and description of the problem, along with the logging level from the above list. For example:

```
2016-04-05 22:23:01: Error: Default Domain: (shell):1:8: MySQLError: You have an error
in your SQL syntax; check the manual that corresponds to your MySQL server version for
the right syntax to use near '' at line 1 (1064) in session.runSql("select * from t
limit").execute().all();
```

Log File Location on Windows

On Windows, the default path to the application log file is `%APPDATA%\MySQL\mysqlsh\mysqlsh.log`. To find the location of `%APPDATA%` on your system, echo it from the command line. For example:

```
C:>echo %APPDATA%

C:\Users\exampleuser\AppData\Roaming
```

On Windows, the path is the `%APPDATA%` folder specific to the user, with `MySQL\mysqlsh` added. Using the above example the path would be `C:\Users\exampleuser\AppData\Roaming\MySQL\mysqlsh\mysqlsh.log` .

If you want the application log file to be stored in a different location, you can override the default user configuration path by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`. The value of this variable replaces `%AppData%\MySQL\mysqlsh\` on Windows.

From MySQL Shell 8.0.27, you can also use the `--log-file` option to override the user configuration path when you run `mysqlsh` from the command line. The `--log-file` option applies to the individual MySQL Shell instance, meaning that different instances can write to different locations.

Log File Location on Unix-based Systems

For a machine running Unix, the default path to the application log file is `~/.mysqlsh/mysqlsh.log` where “~” represents the user's home directory. The environment variable `HOME` also represents the user's home directory. Appending `.mysqlsh` to the user's home directory determines the default path to the log.

If you want the application log file to be stored in a different location, you can override the default user configuration path by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`. The value of this variable replaces `~/.mysqlsh/` on Unix.

From MySQL Shell 8.0.27, you can also use the `--log-file` option to override the user configuration path when you run `mysqlsh` from the command line. The `--log-file` option applies to the individual MySQL Shell instance, meaning that different instances can write to different locations.

12.2 Verbose Output

From MySQL 8.0.17, you can send MySQL Shell logging information to the console to help with debugging. Logging messages sent to the console are given the `verbose:` prefix. When you send logging information to the console, it is still sent to the application log file.

To send logging information to the console as verbose output, choose one of these options:

- Use the `--verbose` command-line option when starting MySQL Shell.
- Use the MySQL Shell `\option` command to set the `verbose` MySQL Shell configuration option. For instructions to use this command, see Section 13.4, “Configuring MySQL Shell Options”.
- Use the `shell.options` object to set the `verbose` MySQL Shell configuration option. For instructions to use this configuration interface, see Section 13.4, “Configuring MySQL Shell Options”.

The available settings are as listed in Table 12.1, “Logging levels in MySQL Shell”. The settings for the `verbose` option display messages at the following levels of detail:

0	No messages. Equivalent to a logging level of 1 for the application log.
1	Internal error, error, warning, and informational messages. Equivalent to a logging level of 5 for the application log.
2	Adds `debug` messages. Equivalent to a logging level of 6 for the application log.
3	Adds `debug2` messages. Equivalent to a logging level of 7 for the application log.
4	Adds `debug3` messages, the highest level of detail. Equivalent to a logging level of 8 for the application log.

If the `verbose` option is not set on the command line or in the configuration file, or if you specify a setting of `0` for the option, verbose output to the console is disabled. All other values enable verbose output and set the level of detail for the messages sent to the console. If you specify the option without a value, which is permitted as a command-line option when starting MySQL Shell (`--verbose`) but not with other methods of setting the option, setting 1 (internal error, error, warning, and informational messages) is used.

12.3 System Logging for User SQL Statements

From MySQL 8.0.24, SQL statements that you issue in MySQL Shell's SQL mode can be sent to the operating system's system logging facility. On Unix, this is `syslog`; on Windows, it is the Windows Event Log. The destination where logged messages appear is system dependent. On Linux, the destination is often the `/var/log/messages` file.

When you activate system logging for SQL statements, the following items are written to the system logging facility:

- SQL statements that you issue interactively in MySQL Shell's SQL mode.
- Single SQL statements that you execute by entering them immediately after the `\sql` command while in MySQL Shell's JavaScript or Python mode.
- Instances of the `\source` command that you issue interactively in MySQL Shell's SQL mode.

The following items are excluded and are not written to the system logging facility:

- The contents of a script file that you execute using the `\source` command. Only the `\source` command itself is written to the system logging facility.
- SQL statements that MySQL Shell executes itself in the course of AdminAPI operations. You can log these to the MySQL Shell application log file, as explained in Section 12.5, "Logging AdminAPI Operations".
- SQL statements that would be excluded from the MySQL Shell code history, as specified by the `history.sql.ignorePattern` MySQL Shell configuration option, or the `--histignore` command-line option (which sets the value of `history.sql.ignorePattern` for the current session only).

To send SQL statements that you issue in MySQL Shell's SQL mode to the operating system's system logging facility, choose one of these options:

- Use the `--syslog` command-line option when starting MySQL Shell.
- Use the MySQL Shell `\option` command to set the `history.sql.syslog` MySQL Shell configuration option. For instructions to use this command, see Section 13.4, "Configuring MySQL Shell Options".

- Use the `shell.options` object to set the `history.sql.syslog` MySQL Shell configuration option. For instructions to use this configuration interface, see Section 13.4, “Configuring MySQL Shell Options”.

System logging for SQL statements only takes place when MySQL Shell is started in interactive mode, so either a normal start or a start with the `--interactive` option. It does not take place if the `--execute` or `--file` options are used at startup to run `mysqlsh` in batch mode to process a command or file.

Log message format

The log message for an SQL statement is formatted as a series of key-value pairs separated by a space character. The key-value pairs are as follows:

SYSTEM_USER =	The login name of the operating system user, or `--` if this user name is unknown.
MYSQL_USER =	The name of the MySQL user, or `--` if this user name is unknown.
CONNECTION_ID =	The identifier for the MySQL Shell connection.
DB_SERVER =	The server’s host name, or `--` if the host name is unknown.
DB =	The default database, or `--` if no database has been selected.
QUERY =	The text of the logged SQL statement.

The log message is truncated to 1024 bytes if it exceeds that length.

Here is a sample of output generated on Linux by using `--syslog`. This output is formatted for readability; each logged message actually takes a single line.

```
Mar  1 17:35:33 myhost mysqlsh[33060]:
SYSTEM_USER=hanna_j MYSQL_USER=hanna
CONNECTION_ID=14 DB_SERVER=localhost DB='--'
QUERY='create table test.test (c int, my_row_id BIGINT AUTO_INCREMENT INVISIBLE PRIMARY KEY);'
```

12.4 MySQL Shell SQL Logging

From MySQL Shell 8.0.30, you can log all SQL statements executed by MySQL Shell commands or utilities to the MySQL Shell log file.

Note

`logSql` replaces `dba.logSql` if `dba.logSql` is disabled, `logSql` takes precedence. However, if `dba.logSql` is enabled, it takes precedence over `logSql`, but only for log messages in the `dba.*` context.

The application log level must be set to at least `INFO (5)`, for these messages to be written to the MySQL Shell log.

SQL Logging Options

MySQL Shell SQL logging can be enabled, disabled, or altered in one of the following ways:

- Command line: `--log-sql=`*`logOption`* in your MySQL Shell startup command.
- MySQL Shell configuration options: `shell.options['logSql']='`*`logOption`*`'` during your MySQL Shell session.

The following options are available:

off	No MySQL Shell SQL statements are logged.
error	(Default) only MySQL Shell failed SQL statements with are logged.
on	All MySQL Shell SQL statements are logged, except those which match the ignore pattern defined in `logSql.ignorePattern` and `logSql.ignorePatternUnsafe`. See Filtering SQL Logging for more information.
all	All MySQL Shell SQL statements are logged, except those which match the ignore pattern defined in the `logSql.ignorePatternUnsafe]`. See Filtering SQL Logging for more information.
unfiltered	All MySQL Shell SQL statements are logged, no filtering is performed.

Filtering SQL Logging

The log is filtered using a colon-separated list of glob patterns. The following options are available:

- `logSql.ignorePattern`: This option defines a colon-separated list of statement patterns to filter out. Default value is `*SELECT*:*SHOW*`.
- `logSql.ignorePatternUnsafe`: This option defines a colon-separated list of statement patterns to filter out. Default value is `*IDENTIFIED*:*PASSWORD*`.

For information on working with `shell.options`, see Section 13.4, “Configuring MySQL Shell Options”.

Log Format

The log messages use the following format:

```
Date&Time: LogLevel: LogContext: tid= : SQL:
```

Date&Time	Date and time of the log message.
LogLevel	The log level. For more information on log levels, seeTable 12.1, “Logging levels in MySQL Shell”. All successful SQL statements and error messages for unsuccessful statements are logged with `INFO` log level.
LogContext	The origin of the log message. Can be one of the following values: • main: base MySQL Shell context. • sql: SQL mode context. • js: JavaScript mode context. • py: Python mode context. • *`object.method`*: Global object method context. For example, `Dba.createCluster` or `Cluster.status`.
tid	The MySQL thread ID.
SQL	The logged SQL statement or error message.

The following example shows an SQL INFO message with a `Dba.deploySandboxInstance` context:

```
2022-06-17 15:06:00: Info: Dba.deploySandboxInstance: tid=9: SQL: SET SESSION `autocommit` = 1
```

12.5 Logging AdminAPI Operations

Note

Deprecated in MySQL Shell 8.0.30. For SQL logging in MySQL Shell 8.0.30 onwards, see Section 12.4, “MySQL Shell SQL Logging”.

From MySQL Shell 8.0.18, you can include SQL statements that are executed in the course of AdminAPI operations as part of the MySQL Shell logging information. By default, MySQL Shell does not log these statements, and just logs the messages returned during the operations. Activating logging for these statements lets you observe the progress of the operations in terms of SQL execution, which can help with problem diagnosis for any errors.

When you activate logging for SQL statements from AdminAPI operations, the statements are written to the MySQL Shell application log file as informational messages, provided that the logging level is set to 5 (which is the default for MySQL Shell's logging level) or above. If an additional viewable location was specified with the logging level, the statements are sent there too. The statements are also sent to the console as verbose output if the verbose option is set to 1 or above. Any passwords included in the SQL statements are masked for logging and display and are not recorded or shown.

SQL statements executed by AdminAPI sandbox operations (`dba.deploySandboxInstance()`, `dba.startSandboxInstance()`, `dba.stopSandboxInstance()`, `dba.killSandboxInstance()`, and `dba.deleteSandboxInstance()`) are always excluded from logging and verbose output, even if you have activated logging for regular AdminAPI operations.

To log SQL statements executed by AdminAPI operations, choose one of these options:

- Use the `--dba-log-sql` command-line option when starting MySQL Shell.
- Use the MySQL Shell `\option` command to set the `dba.logSql` MySQL Shell configuration option. For instructions to use this command, see Section 13.4, “Configuring MySQL Shell Options”.
- Use the `shell.options` object to set the `dba.logSql` MySQL Shell configuration option. For instructions to use this configuration interface, see Section 13.4, “Configuring MySQL Shell Options”.

The available settings for the option are follows:

0	Do not log SQL statements executed by AdminAPI operations. This setting is the default behavior if the option is not set on the command line or in the configuration file, and can be set to deactivate this type of logging after use if you only needed it temporarily.
1	Log SQL statements that are executed by AdminAPI operations, with the exceptions of `SELECT` statements, `SHOW` statements, and statements executed by sandbox operations.
2	Log SQL statements that are executed by regular AdminAPI operations in full, including `SELECT` and `SHOW` statements, but do not log statements executed by sandbox operations.

If you specify the option without a value, which is permitted for a command-line option when starting MySQL Shell (`--dba-log-sql`) but not with other methods of setting the option, setting 1 is used.

Chapter 13 Customizing MySQL Shell

Table of Contents

13.1 Working With Startup Scripts 359
13.2 Adding Module Search Paths 360
13.2.1 Module Search Path Environment Variables 361
13.2.2 Module Search Path Variable in Startup Scripts 361
13.3 Customizing the Prompt 362
13.4 Configuring MySQL Shell Options 363

MySQL Shell offers these customization options for you to change its behavior and code execution environment to suit your preferences:

- Create startup scripts that are executed when MySQL Shell is started in JavaScript or Python mode. See Section 13.1, “Working With Startup Scripts”.
- Add non-standard module search paths for JavaScript or Python mode. See Section 13.2, “Adding Module Search Paths”.
- Customize the MySQL Shell prompt. See Section 13.3, “Customizing the Prompt”.
- Set configuration options to change MySQL Shell's behavior for the current session or permanently. See Section 13.4, “Configuring MySQL Shell Options”.

13.1 Working With Startup Scripts

When MySQL Shell is started in JavaScript or Python mode, and also when you switch to JavaScript or Python mode for the first time, MySQL Shell searches for startup scripts to be executed. The startup scripts are JavaScript or Python specific scripts containing the instructions to be executed when MySQL Shell first enters the corresponding language mode. Startup scripts let you customize the JavaScript or Python code execution environment in any of these ways:

- Adding additional search paths for Python or JavaScript modules.
- Defining global functions or variables.
- Carrying out any other possible initialization through JavaScript or Python.

The relevant startup script is loaded when you start or restart MySQL Shell in either JavaScript or Python mode, and also the first time you change to the other one of those modes while MySQL Shell is running. After this, MySQL Shell does not search for startup scripts again, so implementing updates to a startup script requires a restart of MySQL Shell if you have already entered the relevant mode. When MySQL Shell is started in SQL mode or you switch to that mode, no startup script is loaded.

The startup scripts are optional, and you can create them if you want to use them for customization. The startup scripts must be named as follows:

- For JavaScript mode: `mysqlshrc.js`
- For Python mode: `mysqlshrc.py`

You can place your startup scripts in any of the locations listed below. MySQL Shell searches all of the stated paths, in the order stated, for startup scripts with the file name `mysqlshrc` and the file extension that matches the scripting mode that is being initialized (`.js` by default if MySQL Shell is started with no language mode specified). Note that MySQL Shell executes all appropriate startup scripts found for the scripting mode, in the order they are found. If something is defined in two different startup scripts, the script executed later takes precedence.

1. In the platform's standard global configuration path.
   - On Windows: `%PROGRAMDATA%\MySQL\mysqlsh\mysqlshrc.[js|py]`
   - On Unix: `/etc/mysql/mysqlsh/mysqlshrc.[js|py]`
2. In the `share/mysqlsh` subdirectory of the MySQL Shell home folder, which can be defined by the environment variable `MYSQLSH_HOME`, or identified by MySQL Shell. If `MYSQLSH_HOME` is not defined, MySQL Shell identifies its own home folder as the parent folder of the folder named `bin` that contains the `mysqlsh` binary, if such a folder exists. (For many standard installations it is therefore not necessary to define `MYSQLSH_HOME`.)
   - On Windows: `%MYSQLSH_HOME%\share\mysqlsh\mysqlshrc.[js|py]`
   - On Unix: `$MYSQLSH_HOME/share/mysqlsh/mysqlshrc.[js|py]`
3. In the folder containing the mysqlsh binary, but only if the MySQL Shell home folder described in option 2 is neither specified nor identified by MySQL Shell in the expected standard location.
   - On Windows: `<mysqlsh binary path>\mysqlshrc.[js|py]`
   - On Unix: `<mysqlsh binary path>/mysqlshrc.[js|py]`
4. In the MySQL Shell user configuration path, as defined by the environment variable `MYSQLSH_USER_CONFIG_HOME`.
   - On Windows: `%MYSQLSH_USER_CONFIG_HOME%\mysqlshrc.[js|py]`
   - On Unix: `$MYSQLSH_USER_CONFIG_HOME/mysqlshrc.[js|py]`
5. In the platform's standard user configuration path, but only if the MySQL Shell user configuration path described in option 4 is not specified.
   - On Windows: `%APPDATA%\MySQL\mysqlsh\mysqlshrc.[js|py]`
   - On Unix: `$HOME/.mysqlsh/mysqlshrc.[js|py]`

13.2 Adding Module Search Paths

When you use the `require()` function in JavaScript or the `import` function in Python, the module search paths listed for the `sys.path` variable are used to search for the specified module. MySQL Shell initializes the `sys.path` variable to contain the following module search paths:

- The folders specified by the module search path environment variable (`MYSQLSH_JS_MODULE_PATH` in JavaScript mode, or `PYTHONPATH` in Python mode).
- For JavaScript, the subfolder `share/mysqlsh/modules/js` of the MySQL Shell home folder, or the subfolder `/modules/js` of the folder containing the `mysqlsh` binary, if the home folder is not present.
- For Python, installation-dependent default paths, as for Python's standard import machinery.

MySQL Shell can also load the built-in modules `mysql` and `mysqlx` using the `require()` or `import` function, and these modules do not need to be specified using the `sys.path` variable.

For JavaScript mode, MySQL Shell loads the first module found in the specified location that is (in order of preference) a file with the specified name, or a file with the specified name plus the file extension `.js`, or an `init.js` file contained in a folder with the specified name. For Python mode, Python's standard import machinery is used to load all modules for MySQL Shell.

For JavaScript mode, from MySQL Shell 8.0.19, MySQL Shell also provides support for loading of local modules by the `require()` function. If you specify the module name or path prefixed with `./`

or `../`, in batch mode, MySQL Shell searches for the specified module in the folder that contains the JavaScript file or module currently being executed. In interactive mode, given one of those prefixes, MySQL Shell searches in the current working directory. If the module is not found in that folder, MySQL Shell proceeds to check the module search paths specified by the `sys.path` variable.

You can add further module search paths to the `sys.path` variable either by appending them to the module search path environment variable for JavaScript mode or Python mode (see Section 13.2.1, “Module Search Path Environment Variables”), or by appending them directly to the `sys.path` variable using the MySQL Shell startup script for JavaScript mode or Python mode (see Section 13.2.2, “Module Search Path Variable in Startup Scripts”). You can also modify the `sys.path` variable at runtime, which changes the behavior of the `require()` or `import` function immediately.

13.2.1 Module Search Path Environment Variables

You can add folders to the module search path by adding them to the appropriate language-specific module search path environment variable. MySQL Shell includes these folders in the module search paths when you start or restart MySQL Shell. If you want to add to the search path immediately, modify the `sys.path` variable directly.

For JavaScript, add folders to the `MYSQLSH_JS_MODULE_PATH` environment variable. The value of this variable is a list of paths separated by a semicolon character.

For Python, add folders to the `PYTHONPATH` environment variable. The value of this variable is a list of paths separated by a semicolon character on Windows platforms, or by a colon character on Unix platforms.

For JavaScript, folders added to the environment variable are placed at the end of the `sys.path` variable value, and for Python, they are placed at the start.

Note that Python's behavior for loading modules is not controlled by MySQL Shell; the normal import behaviors for Python apply.

13.2.2 Module Search Path Variable in Startup Scripts

The `sys.path` variable can be customized using the MySQL Shell startup script `mysqlshrc.js` for JavaScript mode or `mysqlshrc.py` for Python mode. For more information on the startup scripts and their locations, see Section 13.1, “Working With Startup Scripts”. Using the startup script, you can append module paths directly to the `sys.path` variable.

Note that each startup script is only used in the relevant language mode, so the module search paths specified in `mysqlshrc.js` for JavaScript mode are only available in Python mode if they are also listed in `mysqlshrc.py`.

For Python modify the `mysqlshrc.py` file to append the required paths into the `sys.path` array:

```
# Import the sys module
import sys

# Append the additional module paths
sys.path.append('~/custom/python')
sys.path.append('~/other/custom/modules')
```

For JavaScript modify the `mysqlshrc.js` file to append the required paths into the `sys.path` array:

```
// Append the additional module paths
  sys.path = [...sys.path, '~/custom/js'];
  sys.path = [...sys.path, '~/other/custom/modules'];
```

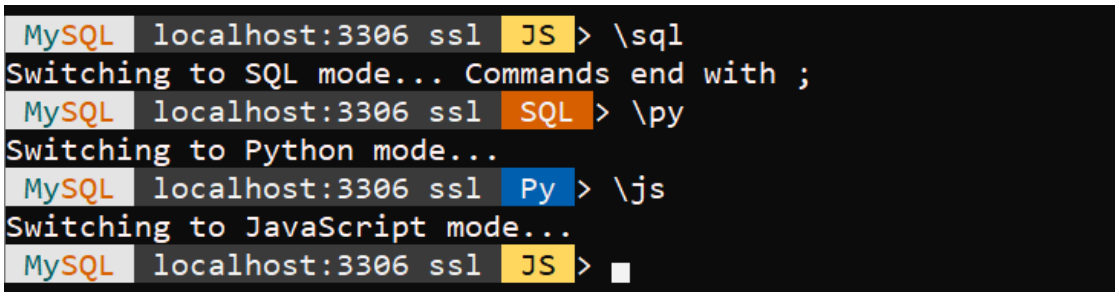
A relative path that you append to the `sys.path` array is resolved relative to the current working directory.

The startup scripts are loaded when you start or restart MySQL Shell in either JavaScript or Python mode, and also the first time you change to the other one of those modes while MySQL Shell is running. After this, MySQL Shell does not search for startup scripts again, so implementing updates to a startup script requires a restart of MySQL Shell if you have already entered the relevant mode. Alternatively, you can modify the `sys.path` variable at runtime, in which case the `require()` or `import` function uses the new search paths immediately.

13.3 Customizing the Prompt

The prompt of MySQL Shell can be customized using prompt theme files. To customize the prompt theme file, either set the `MYSQLSH_PROMPT_THEME` environment variable to a prompt theme file name, or copy a prompt theme file to the `~/.mysqlsh/` directory on Linux and macOS, or the `%AppData%\Roaming\MySQL\mysqlsh\` directory on Windows. The file must be named `prompt.json`, and MySQL Shell must be restarted before changes take effect.

Figure 13.1 mysql shell prompt



There are six parts that can make up the prompt:

- **Status**: Whether it is a production system and whether the connection has been lost.
- **MySQL**: A reminder that you are working with a MySQL database.
- **Connection**: Which host you are connected to, and on which port that SSL is being used.
- **Schema**: The current default schema.
- **Mode**: The mode you are using: `JS` = JavaScript, `PY` = Python, and `SQL` = SQL.
- **End**: The prompt ends with `>`.

The user configuration path for the `mysqlsh` directory where the `prompt.json` is located can be overridden on all platforms by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`. The value of this variable replaces `%AppData%\Roaming\MySQL\mysqlsh\` on Microsoft Windows or `~/.mysqlsh/` on Unix.

On Microsoft Windows, find the prompt theme files in the following directory: `%programfiles%\MySQL\MySQL Shell 8.0\share\mysqlsh\prompt\`.

On macOS, find the prompt theme files in the following directory: `/usr/local/mysql-shell/share/mysqlsh/prompt`.

On Linux, find the prompt theme files in the following directory: `/usr/share/mysqlsh/prompt/`.

The format of the prompt theme file is described in the `README.prompt` file. Some sample prompt theme files are also included, for example, `prompt_256.json`:

In the sample prompt theme `prompt_256.json`, there is an object with the classes `disconnected %host%` and `%is_production%`. The variables are defined in this file or come from MySQL Shell

itself, for example, `%host` and `%port`. In this example, the host is included in the environment variable `PRODUCTION_SERVERS`.

```
"variables" : {
      "is_production": {
      "match" : {
      "pattern": "*;%host%;*",
      "value": ";%env:PRODUCTION_SERVERS%;"
      },
```

The background and foreground colors are defined using the `bg` and `fg` elements. These elements allow you to customize the colors used in the prompt. Specify the colors in one of the following ways:

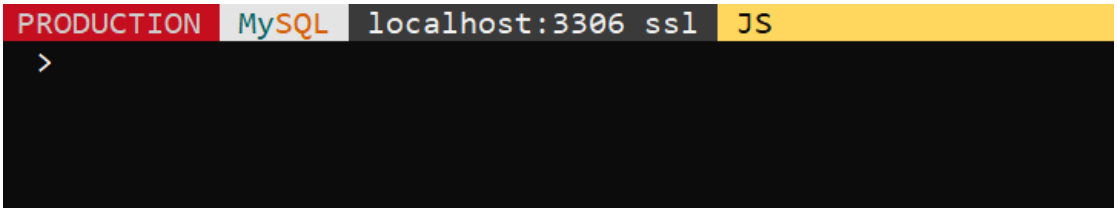
- **By Name**: Use a color defined by name.
- **By Index**: Use a value between 0 and 255 (inclusive) where 0 is black, 63 light blue, 127 magenta, 193 yellow, and 255 is white.
- **By RGB**: Use a value in the `#rrggbb` format. The terminal must support `TrueColor`.

Named colors are used in this example, with a text `PRODUCTION` output if the Boolean `is_production` elements returns `TRUE`.

```
  "production" : {
  "text": " PRODUCTION ",
  "bg": "red",
  "fg": "white"
  }
```

These elements output the prompt in the following format:

Figure 13.2 mysql shell prompt



Color display depends on the support available from the terminal. Most terminals support 256 colors in Linux and Mac. In Windows, color support requires either a 3rd party terminal program with support for ANSI/VT100 escapes, or Windows 10. By default, MySQL Shell attempts to detect the terminal type and handle colors appropriately. If auto-detection does not work for your terminal type, or if you want to modify the color mode due to accessibility requirements or for other purposes, you can define the environment variable `MYSQLSH_TERM_COLOR_MODE` to force MySQL Shell to use a specific color mode. The possible values for this environment variable are `rgb`, `256`, `16`, and `nocolor`.

On startup, if an error is found in the prompt theme file, an error message is printed and a default prompt theme is used. Some of the sample prompt theme files require a special font (for example `SourceCodePro+Powerline+Awesome+Regular.ttf`). If you set the `MYSQLSH_PROMPT_THEME` environment variable to an empty value, MySQL Shell uses a minimal prompt with no color.

13.4 Configuring MySQL Shell Options

You can configure MySQL Shell to match your preferences, for example to start up to a certain programming language or to provide output in a particular format. Configuration options can be set for the current session only, or options can be set permanently by persisting changes to the MySQL Shell configuration file. Online help for all options is provided. You can configure options using the MySQL Shell `\option` command, which is available in all MySQL Shell modes for querying and changing configuration options. Alternatively in JavaScript and Python modes, use the `shell.options` object.

Valid Configuration Options

The following configuration options can be set using either the `\option` command or `shell.options` scripting interface:

optionName	DefaultValue	Type	Effect
`autocomplete.nameCa`	true	boolean	Enable database name caching for autocompletion.
`batchContinueOnErro`	false	boolean (READ ONLY)	In SQL batch mode, force processing to continue if an error is found. Set to `true` by adding `--force` on the command line. See Appendix A, *MySQL Shell Command Reference*.
`connectTimeout`	10	float greater than 0	The time in seconds to wait before the connection of any session not using AdminAPI times out.
`credentialStore.exclu`	`emptyFilters`	array	An array of URLs for which automatic password storage is disabled, supports glob characters `*` and `?`.
`credentialStore.helpe`	Depends on platform	string	Name of the credential helper used to fetch or store passwords. A special value `default` is supported to use the platform's default helper. The special value `>disabled<` disables the credential store.
`credentialStore.savePass`	`Prompt`	string	Controls automatic password storage, supported values: `always`, `prompt` or `never`.
`dba.connectTimeout`	5	float greater than 0	The time in seconds to wait before the connection of any session using AdminAPI times out.
`dba.connectivityChe`	true	boolean	Defines if connectivity checks are performed for *`cluster`*`.addInstance()`, *`clusterSet`*`.createReplicaClus` and *`replicaSet`*`.addInstance()`,

optionName	DefaultValue	Type	Effect
			using the defined SSL configuration. If an SSL error occurs, the command stops and an error is returned.
`dba.gtidWaitTimeout`	60	integer greater than 0	The time in seconds to wait for GTID transactions to be applied, when required by AdminAPI operations (see Section 7.9, “Modifying or Dissolving an InnoDB Cluster”).
`logFile`	Path to the MySQL Shell log file.	string	Displays the path to the MySQL Shell log file. (Read-only) This value can only be changed from the command line, using the `--log-file=path/to/logfile.log` option. See Appendix A, *MySQL Shell Command Reference*.
`dba.logSql`	0	integer ranging from 0 to 2	(Deprecated in MySQL Shell 8.0.30. Use `logSql` instead.) Log SQL statements that are executed by AdminAPI operations (see Chapter 12, *MySQL Shell Logging and Debug*).
`dba.restartWaitTimeout`	60	integer greater than 0	The time in seconds to wait for transactions to be applied during a recovery operation. Use to configure a longer timeout when a joining instance has to recover a large amount of data. See Section 7.4.6, “Using MySQL Clone with InnoDB Cluster”).
`defaultCompress`	false	boolean	Request compression for information sent between the client and the server in every global session. Affects classic MySQL protocol connections only (see Section 4.3.7,

optionName	DefaultValue	Type	Effect
			"Using Compressed Connections").
defaultMode	None	string (sql, js or py)	The mode to use when MySQL Shell is started (SQL, JavaScript or Python).
devapi.dbObjectHandles	true	boolean	Enable table and collection name handles for the X DevAPI db object.
history.autoSave	false	boolean	Save (true) or clear (false) entries in the MySQL Shell code history when you exit the application (see Section 5.5, "Code History").
history.maxSize	1000	integer	The maximum number of entries to store in the MySQL Shell code history.
history.sql.ignorePattern	*SELECT*:SHOW*	string	Strings that match these patterns are not added to the MySQL Shell code history.
history.sql.syslog	false	boolean	Send interactive SQL statements to the operating system's system logging facility (see Section 12.3, "System Logging for User SQL Statements").
interactive	true	boolean (READ ONLY)	Enable interactive mode.
logLevel	info	integer ranging from 1 to 8 or any of none, internal, error, warning, info, debug, debug2, debug3, respectively	Set a logging level for the application log (see Chapter 12, *MySQL Shell Logging and Debug*).
logSql	error	string (off, error, on, all)	Log SQL statements that are executed by MySQL Shell operations (see Section 12.4, "MySQL Shell SQL Logging"). This log option does not log user SQL statements executed interactively from the SQL mode, only SQL statements executed by MySQL Shell operations. Logs all

optionName	DefaultValue	Type	Effect
			SQL statements except those defined in the `logSql.ignorePatternUnsaf` and `logSql.ignorePattern` options. • off: no SQL statements are logged. • error: only SQL statements with error messages are logged when an error occurs. • `on`: logs all SQL statements except those defined in the `logSql.ignorePatternUns` and `logSql.ignorePattern` options. • `all`: logs all SQL statements except those defined in `logSql.ignorePatternUns`
`logSql.ignorePattern`	`*SELECT*:*SHOW*`	string	Colon-separated list of glob pattern to filter out of SQL statements logged by `logSql` (see Section 12.4, "MySQL Shell SQL Logging").
`logSql.ignorePattern`	`*IDENTIFIED*:*PASSWORD*`	string	Colon-separated list of statement patterns to filter out of SQL statements logged by `logSql` (see Section 12.4, "MySQL Shell SQL Logging").
`mysqlPluginDir`	MySQL Shell's MySQL plugin directory. That is, the `lib/mysql/plugins` directory of your MySQL Shell installation on Linux platforms, and `lib\mysql\plugins` on Windows platforms.	string	Set a persistent path to a plugin directory.
`oci.configFile`	The default location for your platform.	string	Set a persistent path to an OCI CLI config file.
`oci.profile`	DEFAULT	string	Specify which profile to use in the OCI CLI config file.

optionName	DefaultValue	Type	Effect
`pager`	None	string	Use the specified external pager tool to display text and results. Command-line arguments for the tool can be added (see Section 4.6, “Using a Pager”).
`passwordsFromStdin`	false	boolean	Read passwords from `stdin` instead of terminal.
`resultFormat`	table	string (table, tabbed, vertical, json | json/pretty, ndjson | json/raw, json/array)	The default output format for printing result sets (see Section 5.7, “Output Formats”).
`sandboxDir`	Depends on platform	string	The sandbox directory. On Windows, the default is `C:\Users\MyUser\MySQL\mysql-sandboxes`, and on Unix systems, the default is `$HOME/mysql-sandboxes`.
`showColumnTypeInfo`	false	boolean	In SQL mode, display column metadata for result sets.
`showWarnings`	true	boolean	In SQL mode, automatically display SQL warnings if any.
`ssh.bufferSize`	10240	integer greater than 0	The buffer size in bytes for data transfer through an SSH tunnel (see Section 4.3.6, “Using an SSH Tunnel”).
`ssh.configFile`	empty	string	The path to a custom SSH configuration file that replaces the standard SSH configuration file `~/.ssh/config` as the default for SSH tunneling (see Section 4.3.6, “Using an SSH Tunnel”).
`useWizards`	true	boolean	(Read-only) Indicates if interactive prompting and wizards are enabled by default
`verbose`	0	integer ranging from 0 to 4	Enable verbose output to the console and set a level of detail (see Chapter 12, *MySQL*

optionName	DefaultValue	Type	Effect
			Shell Logging and Debug).

Note

String values are case-sensitive.

Options listed as “READ ONLY” cannot be modified.

The `outputFormat` option is now deprecated. Use `resultFormat` instead.

Using the `\option` Command

The MySQL Shell `\option` command enables you to query and change configuration options in all modes, enabling configuration from SQL mode in addition to JavaScript and Python modes.

The command is used as follows:

- `\option -h, --help [filter]` - print help for options matching *`filter`*.
- `\option -l, --list [--show-origin]` - list all the options. `--show-origin` augments the list with information about how the value was last changed, possible values are:
  - `Command line`
  - `Compiled default`
  - `Configuration file`
  - `Environment variable`
  - `User defined`
- `\option` *`option_name`* - print the current value of the option.
- `\option [--persist]` *`option_name value or name=value`* - set the value of the option and if `--persist` is specified save it to the configuration file.
- `\option --unset [--persist] <option_name>` - reset option's value to default and if `--persist` is specified, removes the option from the MySQL Shell configuration file.

Note

The value of *`option_name`* and *`filter`* are case-sensitive.

See Valid Configuration Options for a list of possible values for *`option_name`*.

Using the `shell.options` Configuration Interface

The `shell.options` object is available in JavaScript and Python mode to change MySQL Shell option values. You can use specific methods to configure the options, or key-value pairs as follows:

```
MySQL JS > shell.options['history.autoSave']=1
```

In addition to the key-value pair interface, the following methods are available:

- `shell.options[optionName]`: lists the current value of the option.
- `shell.options.set(`*`optionName, value`*`)`: sets the *`optionName`* to *`value`* for this session, the change is not saved to the configuration file.

- `shell.options.setPersist(optionName, value)`: sets the *`optionName`* to *`value`* for this session, and saves the change to the configuration file. In Python mode, the method is `shell.options.set_persist`.
- `shell.options.unset(optionName)`: resets the *`optionName`* to the default value for this session, the change is not saved to the configuration file.
- `shell.options.unsetPersist(optionName)`: resets the *`optionName`* to the default value for this session, and saves the change to the configuration file. In Python mode, the method is `shell.options.unset_persist`.

Option names are treated as strings, and as such should be surrounded by `'` characters. See Valid Configuration Options for a list of possible values for *`optionName`*.

Use the commands to configure MySQL Shell options as follows:

```
MySQL JS > shell.options.set('history.maxSize', 5000)
MySQL JS > shell.options.setPersist('useWizards', 'true')
MySQL JS > shell.options.setPersist('history.autoSave', 1)
```

Return options to their default values as follows:

```
MySQL JS > shell.options.unset('history.maxSize')
MySQL JS > shell.options.unsetPersist('useWizards')
```

Configuration File

The MySQL Shell configuration file stores the values of the option to ensure they are persisted across sessions. Values are read at startup and when you use the persist feature, settings are saved to the configuration file.

The location of the configuration file is the user configuration path and the file is named `options.json`. Assuming that the default user configuration path has not been overridden by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`, the path to the configuration file is:

- on Windows `%APPDATA%\MySQL\mysqlsh`
- on Unix `~/.mysqlsh` where `~` represents the user's home directory.

The configuration file is created the first time you customize a configuration option. This file is internally maintained by MySQL Shell and should not be edited manually. If an unrecognized option or an option with an incorrect value is found in the configuration file on startup, MySQL Shell exits with an error.

Appendix A MySQL Shell Command Reference

Table of Contents

A.1 mysqlsh — The MySQL Shell 371

This appendix describes the `mysqlsh` command.

A.1 mysqlsh — The MySQL Shell

MySQL Shell is an advanced command-line client and code editor for MySQL. In addition to SQL, MySQL Shell also offers scripting capabilities for JavaScript and Python. For information about using MySQL Shell, see MySQL Shell 8.0. When MySQL Shell is connected to the MySQL Server through the X Protocol, the X DevAPI can be used to work with both relational and document data, see Using MySQL as a Document Store. MySQL Shell includes the AdminAPI that enables you to work with InnoDB Cluster, InnoDB ClusterSet, and InnoDB ReplicaSet deployments; see Chapter 6, *MySQL AdminAPI*.

Many of the options described here are related to connections between MySQL Shell and a MySQL Server instance. See Section 4.3, “MySQL Shell Connections” for more information.

`mysqlsh` supports the following command-line options.

Table A.1 mysqlsh Options

Option Name	Description	Introduced	Deprecated
--	Start of API command line integration		
--auth-method	Authentication method to use		
--cluster	Connect to an InnoDB cluster	8.0.4	
--column-type-info	Print metadata for columns in result sets	8.0.14	
--compress	Compress all information sent between client and server	8.0.14	
--compression-algorithms	Permitted compression algorithms for connections to server	8.0.18	
--zstd-compression-level	Compression level for connections to server that use any compression algorithm over X Protocol or for zstd only on classic connections.	8.0.18	
--connect-timeout	Connection timeout for global session	8.0.13	
--credential-store-helper	The Secret Store helper for passwords	8.0.12	
--database	The schema to use (alias for --schema)		

Option Name	Description	Introduced	Deprecated
--dba	Enable X Protocol on connection with MySQL 5.7 server		
--dba-log-sql	Log SQL statements that are executed by AdminAPI operations	8.0.18	8.0.30
--dbpassword	Password to use when connecting to server		8.0.13
--dbuser	MySQL user name to use when connecting to server		8.0.13
--execute	Execute the command and quit		
--fido-register-factor	Register a FIDO device for authentication to the server	8.0.29	
--file	File to process in batch mode		
--force	Continue in SQL and batch modes even if errors occur		
--get-server-public-key	Request RSA public key from server		
--help	Display help message and exit		
--histignore	Strings that are not added to the history	8.0.3	
--host	Host on which MySQL server instance is located		
--import	Import JSON documents from a file or standard input	8.0.13	
--interactive	Emulate Interactive mode in batch mode		
--js, --javascript	Start in JavaScript mode		
--json	Print output in JSON format		
--log-file	Log file location for this instance	8.0.27	
--log-level	Specify logging level		
--log-sql	Log all MySQL Shell-generated SQL statements to the MySQL Shell log file.	8.0.30	
-ma	Detect transport protocol for session automatically	8.0.3	8.0.13

Option Name	Description	Introduced	Deprecated
--mysql, --mc	Create a session using classic MySQL protocol	8.0.3	
--mysql-plugin-dir	Directory where the client-side plugins are installed	8.0.27	
--mysqlx, --mx	Create a session using X Protocol	8.0.3	
--name-cache	Enable automatic loading of table names based on the active default schema	8.0.4	
--no-name-cache	Disable autocompletion	8.0.4	
--no-password	No password is provided for this connection		
--no-wizard, --nw	Disable the interactive wizards		
--pager	The external pager tool used to display output	8.0.13	
--password	Password to use when connecting to server		
--password1	Password 1 for multifactor authentication (equivalent to --password)	8.0.28	
--password2	Password 2 for multifactor authentication	8.0.28	
--password3	Password 3 for multifactor authentication	8.0.28	
--passwords-from-stdin	Read the password from stdin		
--port	TCP/IP port number for connection		
--py, --python	Start in Python mode		
--pyc	Execute a Python command and quit. Any options specified after this are treated as arguments of the processed command.	8.0.31	
--pym	Password 3 for multifactor authentication	8.0.22	
--quiet-start	Start without printing introductory information	8.0.14	
--recreate-schema	Drop and recreate schema		

Option Name	Description	Introduced	Deprecated
--redirect-primary	Ensure connection to an InnoDB cluster's primary	8.0.4	
--redirect-secondary	Ensure connection to an InnoDB cluster's secondary	8.0.4	
--replicaset	Ensures that the target server belongs to an InnoDB ReplicaSet, and populates the rs global variable with the InnoDB ReplicaSet the target instance belongs to	8.0.20	
--result-format	Set the output format for this session	8.0.14	
--save-passwords	How passwords are stored in the Secret Store	8.0.12	
--schema	The schema to use		
--server-public-key-path	Path name to file containing RSA public key		
--show-warnings	Show warnings after each statement if there are any (in SQL mode)		
--socket	Unix socket file or Windows named pipe to use (classic MySQL protocol only)		
--sql	Start in SQL mode, auto-detecting protocol to use for connection		
--sqlc	Start in SQL mode using a classic MySQL protocol connection		
--sqlx	Start in SQL mode using an X Protocol connection	8.0.3	
--ssh	URI for connection to SSH server	8.0.28	
--ssh-config-file	Configuration file for connection to SSH server	8.0.28	
--ssh-identity-file	Identity file for connection to SSH server	8.0.28	
--ssl	Enable an SSL connection. Deprecated in version 1.0.10; use --ssl-mode instead		8.0.3

Option Name	Description	Introduced	Deprecated
--ssl-ca	File that contains list of trusted SSL Certificate Authorities		
--ssl-capath	Directory that contains trusted SSL Certificate Authority certificate files		
--ssl-cert	File that contains X.509 certificate		
--ssl-cipher	List of permissible encryption ciphers for connections that use TLS protocols up through TLSv1.2		
--ssl-crl	File that contains certificate revocation lists		
--ssl-crlpath	Directory that contains certificate revocation list files		
--ssl-key	File that contains X.509 key		
--ssl-mode	Desired security state of connection to server		
--syslog	Log interactive SQL statements to the system logging facility	8.0.24	
--tabbed	Display output in tab separated format		
--table	Display output in table format		
--tls-ciphersuites	TLS v1.3 cipher to use	8.0.18	
--tls-version	Permissible TLS protocol for encrypted connections		
--uri	Session information in URI format		
--user	MySQL user name to use when connecting to server		
--verbose	Activate verbose output to the console	8.0.17	
--version	Display version information and exit		
--vertical	Display all SQL results vertically		
--zstd-compression-level	Compression level for connections to server that use zstd compression	8.0.18	

- `--help`, `-?`

  Display a help message and exit.

- `--`

  Marks the end of the list of mysqlsh options and the start of a command and its arguments for MySQL Shell's API command line integration. You can execute methods of the MySQL Shell global objects from the command line using this syntax:

  ```
  mysqlsh [options]  -- object method [arguments]
  ```

  See Section 5.8, “API Command Line Integration” for more information.

- `--auth-method=method`

  Authentication method to use for the account. Depends on the authentication plugin used for the account's password. For MySQL Shell connections using classic MySQL protocol, specify the name of the authentication plugin, for example `caching_sha2_password`. For MySQL Shell connections using X Protocol, specify one of the following options:

AUTO	Let the library select the authentication method.
FALLBACK	Let the library select the authentication method, but do not use any authentication method that is not compatible with MySQL 5.7.
FROM_CAPABILITIES	Let the library select the authentication method, using the capabilities announced by the server instance.
MYSQL41	Use the challenge-response authentication protocol supported by MySQL 4.1 and later, which does not send a plaintext password. This option is compatible with accounts that use the `mysql_native_password` authentication plugin.
PLAIN	Send a plaintext password for authentication. Use this option only with encrypted connections. This option can be used to authenticate with cached credentials for an account that uses the `caching_sha2_password` authentication plugin, provided there is an SSL connection. See Using X Plugin with the Caching SHA-2 Authentication Plugin.
SHA256_MEMORY	Authenticate using a hashed password stored in memory. This option can be used to authenticate with cached credentials for an account that uses the `caching_sha2_password` authentication plugin, where there is a non-SSL connection. See Using X Plugin with the Caching SHA-2 Authentication Plugin.

  For MySQL Shell connections using classic MySQL protocol, specify the name of the authentication plugin used by the user account, for example `caching_sha2_password` (which is the default for user accounts created in MySQL 8.0). MySQL Shell uses the MySQL client library for client-side authentication for these connections. The following authentication methods require additional configuration:

clear_text_password	The `mysql_clear_password` client-side plugin is required for simple LDAP authentication. It is built in to the MySQL client library, but for security it is not enabled by default. From MySQL Shell 8.0.27, MySQL Shell enables and uses the plugin when you specify it with the `--auth-method=clear_text_password` connection option. This authentication type is only suitable for a secure connection that uses SSL or sockets, so you must configure the secure connection before using it. Note that with

the option `ssl-mode=preferred`, the SSL connection is not guaranteed, so a connection with this option set is not considered to be an SSL connection. For more information, see Section 4.3.4, "Using Encrypted Connections".

authentication_ldap_sasl_client	The `authentication_ldap_sasl_client` client-side plugin is for SASL-based LDAP authentication, including GSSAPI/Kerberos authentication. It is not built in to the MySQL client library, but it is shipped in the MySQL Server packages. To load it, you must use the `--mysql-plugin-dir` option (available from MySQL Shell 8.0.27) to specify a path to the plugin in the MySQL Server packages.
authentication_kerberos_client	The `authentication_kerberos_client` client-side plugin is for Kerberos authentication. It is not built in to the MySQL client library, but it is shipped in the MySQL Server packages. To load it, you must use the `--mysql-plugin-dir` option (available from MySQL Shell 8.0.27) to specify a path to the plugin in the MySQL Server packages.

Cached ticket-granting tickets (TGTs) for Kerberos authentication are supported from MySQL 8.0.27 when the `--auth-method` option is used to specify the `authentication_ldap_sasl_client` or `authentication_kerberos_client` plugin, and the `--mysql-plugin-dir` option is used to provide a path to the plugin. To use cached TGTs, do not specify a user and password in the connection options. When you specify one of these plugins and do not specify a user and password, MySQL Shell does not supply the system user name, does not prompt for a password, and does not attempt to use the Secret Store helper to retrieve or store credentials.

For more information, see Section 4.3.5, "Using LDAP and Kerberos Authentication".

- `--cluster`

  Ensures that the target server is part of an InnoDB Cluster and if so, sets the `cluster` global variable to the cluster object.

- `--column-type-info`

  In SQL mode, before printing the returned result set for a query, print metadata for each column in the result set, such as the column type and collation.

  The column type is returned as both the type used by MySQL Shell (`Type`), and the type used by the original database (`DBType`). For MySQL Shell connections using classic MySQL protocol, `DBType` is as returned by the protocol, and for X Protocol connections, `DBType` is inferred from the available information. The column length (`Length`) is returned in bytes.

- `--compress[={required|preferred|disabled}]`, `-C [{required|preferred|disabled}]`

  Controls compression of information sent between the client and the server using this connection. In MySQL Shell 8.0.14 through 8.0.19 this option is available for classic MySQL protocol connections only, and does not use the options `required`, `preferred`, and `disabled`. In those releases, when you specify `--compress`, compression is activated if possible. From MySQL Shell 8.0.20 it is also available for X Protocol connections, and you can optionally specify `required`, `preferred`, or `disabled`. When just `--compress` is specified from MySQL Shell 8.0.20, the meaning is `--compress=required`. See Section 4.3.7, "Using Compressed Connections" for information on using MySQL Shell's compression control in all releases.

- `--compression-algorithms=`*`value`*

Command-Line Format	`--compression-algorithms=value`

Introduced	8.0.18
Type	Set
Default Value	`uncompressed`

Valid Values	`zlib`
	`zstd`
	`uncompressed`

The permitted compression algorithms for connections to the server.

See Section 4.3.7.1, “Compression Control For MySQL Shell 8.0.20 And Later”.

- `--compression-level=`*`level`*

Command-Line Format	`--zstd-compression-level=#`
Introduced	8.0.18
Type	Integer

Compression level for connections to server that use any compression algorithm over X Protocol or for zstd only on classic connections.

See Section 4.3.7.1, “Compression Control For MySQL Shell 8.0.20 And Later”.

- `--zstd-compression-level=`*`level`*

Command-Line Format	`--zstd-compression-level=#`
Introduced	8.0.18
Type	Integer

The compression level to use for connections to the server that use the `zstd` compression algorithm. The permitted levels are from 1 to 22, with larger values indicating increasing levels of compression. The default `zstd` compression level is 3. The compression level setting has no effect on connections that do not use `zstd` compression.

See Section 4.3.7.1, “Compression Control For MySQL Shell 8.0.20 And Later”.

- `--connect-timeout=`*`ms`*

Configures how long MySQL Shell waits (in milliseconds) to establish a global session specified through command-line arguments.

- `--credential-store-helper=`*`helper`*

The Secret Store Helper that is to be used to store and retrieve passwords. See Section 4.4, “Pluggable Password Store”.

- `--database=`*`name`*, `-D` *`name`*

The default schema to use. This is an alias for `--schema`.

- `--dba=enableXProtocol`

Enable X Plugin on connection with a MySQL 5.7 server, so that you can use X Protocol connections for subsequent connections. Requires a connection using classic MySQL protocol. Not relevant for MySQL 8.0 servers, which have X Plugin enabled by default.

- `--dba-log-sql[=0|1|2]`

  Log SQL statements that are executed by AdminAPI operations (excluding sandbox operations). By default, this category of statement is not written to the MySQL Shell application log file or sent to the console as verbose output, even when the `--log-level` and `--verbose` options are set. The value of the option is an integer in the range from 0 to 2. 0 does not log or display this category of statement, which is the default behavior if you do not specify the option. 1 logs SQL statements that are executed by AdminAPI operations, with the exceptions of `SELECT` statements and `SHOW` statements (this is the default setting if you specify the option on the command line without a value). 2 logs SQL statements that are executed by regular AdminAPI operations in full, including `SELECT` and `SHOW` statements. See Chapter 12, *MySQL Shell Logging and Debug* for more information.

- `--log-sql[=off|error|on|all|unfiltered]`

  Log all SQL statements executed by MySQL Shell to the MySQL Shell log file, `mysqlsh.log`

  The following options are available:

off	No MySQL Shell SQL statements are logged.
error	(Default value) only MySQL Shell failed SQL statements with are logged.
on	All MySQL Shell SQL statements are logged, except those which match the ignore pattern defined in `logSql.ignorePattern` and `logSql.ignorePatternUnsafe`. See Filtering SQL Logging for more information.
all	All MySQL Shell SQL statements are logged, except those which match the ignore pattern defined in `logSql.ignorePatternUnsafe`. See Filtering SQL Logging for more information.
unfiltered	All MySQL Shell SQL statements are logged, no filtering is performed.

- `--dbpassword[=`*`password`*`]`

  Deprecated in version 8.0.13 of MySQL Shell. Use `--password[=`*`password`*`]` instead.

- `--dbuser=`*`user_name`*

  Deprecated in version 8.0.13 of MySQL Shell. Use `--user=`*`user_name`* instead.

- `--execute=`*`command`*, `-e` *`command`*

  Execute the command using the currently active language and quit. This option is mutually exclusive with the `--file=`*`file_name`* option.

- `--fido-register-factor`

  The factor or factors for which FIDO device registration must be performed. This option value must be a single value, or two values separated by commas. Each value must be 2 or 3, so the permitted option values are '2', '3', '2,3' and '3,2'. For example:

```
mysqlsh --user=user_name --password1 --fido-register-factor=2
```

```
Enter password: (enter factor 1 password)
```

To register an account for passwordless authentication, with the FIDO device being the only authentication, you use `--fido-register-factor=2` to provide the temporary password. The server moves FIDO authentication to the first factor after registration succeeds.

Note

MySQL Shell prompts for a password if you do not specify one when connecting to the server. After you set up passwordless authentication with a FIDO device, use one of the following methods to bypass the password prompt when you make a connection:

- Specify the connection option `--no-password`, or `--password=` with an empty value.
- Place a colon (:) after the user name in the connection string, for example: `mysqlsh user_name:@localhost`
- Press Enter when the password prompt appears.

MySQL Shell's `--fido-register-factor` option works in the same way as the `mysql` client option. For more details and instructions, see FIDO Pluggable Authentication.

- `--file=file_name`, `-f file_name`

  Specify a file to process in Batch mode. Any options specified after this are used as arguments of the processed file.

- `--force`

  Continue processing in SQL and Batch modes even if errors occur.

- `--histignore=strings`

  Specify strings that are not added to the MySQL Shell history. Strings are separated by a colon. Matching is case insensitive, and the wildcards * and ? can be used. The default ignored strings are specified as "`*IDENTIFIED*:*PASSWORD*`". See Section 5.5, "Code History".

- `--host=host_name`, `-h host_name`

  Connect to the MySQL server on the given host. On Windows, if you specify `--host=.` or `-h .` (giving the host name as a period), MySQL Shell connects using the default named pipe (which has the name `MySQL`), or an alternative named pipe that you specify using the `--socket` option.

- `--get-server-public-key`

  MySQL Shell equivalent of `--get-server-public-key`.

  If `--server-public-key-path=file_name` is given and specifies a valid public key file, it takes precedence over `--get-server-public-key`.

  Important

  Only supported with classic MySQL protocol connections.

  See Caching SHA-2 Pluggable Authentication.

- `--import`

  Import JSON documents from a file or standard input to a MySQL Server collection or relational table, using the JSON import utility. For instructions, see Section 11.2, "JSON Import Utility".

- `--interactive[=full]`, `-i`

  Emulate Interactive mode in Batch mode.

- `--js`, `--javascript`

  Start in JavaScript mode.

- `--json[={off|pretty|raw}]`

  Controls JSON wrapping for MySQL Shell output from this session. This option is intended for interfacing MySQL Shell with other programs, for example as part of testing. For changing query results output to use the JSON format, see `--result-format`.

  When the `--json` option has no value or a value of `pretty`, the output is generated as pretty-printed JSON. With a value of `raw`, the output is generated in raw JSON format. In any of these cases, the `--result-format` option and its aliases and the value of the `resultFormat` MySQL Shell configuration option are ignored. With a value of `off`, JSON wrapping does not take place, and result sets are output as normal in the format specified by the `--result-format` option or the `resultFormat` configuration option.

- `--log-file=`*`path`*

  Change the location of the MySQL Shell application log file `mysqlsh.log` for this MySQL Shell instance. The default location for the application log file is the user configuration path, which defaults to `%APPDATA%\MySQL\mysqlsh\` on Windows or `~/.mysqlsh/` on Unix. You can override the user configuration path for all MySQL Shell instances by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`. The `--log-file` option applies to the individual MySQL Shell instance, meaning that different instances can write to different locations.

- `--log-level=`*`N`*

  Change the logging level for the MySQL Shell application log file `mysqlsh.log`, or disable logging to the file. The option requires a value, which can be either an integer in the range from 1 to 8, or one of `none`, `internal`, `error`, `warning`, `info`, `debug`, `debug2`, or `debug3`. Specifying 1 or `none` disables logging to the application log file. Level 5 (`info`) is the default if you do not specify this option. See Chapter 12, *MySQL Shell Logging and Debug*.

- `-ma`

  Deprecated in version 8.0.13 of MySQL Shell. Automatically attempts to use X Protocol to create the session's connection, and falls back to classic MySQL protocol if X Protocol is unavailable.

- `--mysql`, `--mc`

  Sets the global session created at start up to use a classic MySQL protocol connection. The `--mc` option with two hyphens replaces the previous single hyphen `-mc` option from MySQL Shell 8.0.13.

- `--mysql-plugin-dir=`*`path`*

  Sets a non-persistent path to the client-side authentication plugins by overriding the value of the `shell.options.mysqlPluginDir` setting. Client-side plugins are shipped in the MySQL Server packages and can be located relative to the MySQL base directory (the value of the `basedir` system variable). For example:

  - `C:\program files\mysql\mysql Server 8.0\lib\plugin` on Windows host types
  - `/usr/local/mysql/lib/plugin` on Linux host types

  For a list of the client authentication plugins that ship with the server, see Available Authentication Plugins.

- `--mysqlx`, `--mx`

  Sets the global session created at start up to use an X Protocol connection. The `--mx` option with two hyphens replaces the previous single hyphen `-mx` option from MySQL Shell 8.0.13.

- `--name-cache`

  Enable automatic loading of table names based on the active default schema.

- `--no-name-cache`, `-A`

  Disable loading of table names for autocompletion based on the active default schema and the DevAPI `db` object. Use `\rehash` to reload the name information manually.

- `--no-password`

  When connecting to the server, if the user has a passwordless account, which is insecure and not recommended, or if socket peer-credential authentication is in use (for Unix socket connections), you must use `--no-password` to explicitly specify that no password is provided and the password prompt is not required.

- `--no-wizard`, `-nw`

  Disables the interactive wizards provided by operations such as creating connections, `dba.configureInstance()`, *`Cluster`*`.rebootClusterFromCompleteOutage()` and so on. Use this option when you want to script MySQL Shell and not have the interactive prompts displayed. For more information see Section 5.6, “Batch Code Execution” and Section 5.8, “API Command Line Integration”.

- `--pager=`*`name`*

  The external pager tool used by MySQL Shell to display text output for statements executed in SQL mode and other selected commands such as online help. If you do not set a pager, the pager specified by the `PAGER` environment variable is used. See Section 4.6, “Using a Pager”.

- `--passwords-from-stdin`

  Read the password from standard input, rather than from the terminal. This option does not affect any other password behaviors, such as the password prompt.

- `--password[=`*`password`*`]`, `-p`*`password`*

  The password to use when connecting to the server. The maximum password length that is accepted for connecting to MySQL Shell is 128 characters.

  - `--password=`*`password`* (`-p`*`password`*) with a value supplies a password to be used for the connection. With the long form `--password=`, you must use an equals sign and not a space between the option and its value. With the short form `-p`, there must be no space between the option and its value. If a space is used in either case, the value is not interpreted as a password and might be interpreted as another connection parameter.

    Specifying a password on the command line should be considered insecure. See End-User Guidelines for Password Security. You can use an option file to avoid giving the password on the command line.

  - `--password` with no value and no equal sign, or `-p` without a value, requests the password prompt.

  - `--password=` with an empty value has the same effect as `--no-password`, which specifies that the user is connecting without a password. When connecting to the server, if the user has a passwordless account, which is insecure and not recommended, or if socket peer-credential

authentication is in use (for Unix socket connections), you must use one of these methods to explicitly specify that no password is provided and the password prompt is not required.

- `--password1[=password]`

  `--password1`, `--password2` and `--password3` are the passwords to use for accounts that require multifactor authentication. You can supply up to three passwords. The options work in the same way as the `--password` option, and `--password1` is treated as equivalent to that option. You can specify a password value following the option on the command line (which is insecure), or if the options are given without a password value, MySQL Shell prompts the user for each password in turn. These options are available from MySQL Shell 8.0.28, where they are only supported for classic MySQL protocol connections made using command-line arguments.

- `--password2[=password]`

  The password for the second authentication method for accounts that require multifactor authentication. See the description for the `--password1` option.

- `--password3[=password]`

  The password for the third authentication method for accounts that require multifactor authentication. See the description for the `--password1` option.

- `--port=port_num`, `-P port_num`

  The TCP/IP port number to use for the connection. The default is port 33060.

- `--py`, `--python`

  Start in Python mode.

- `--pyc=pythonCommand`, `-c`

  Execute a Python command and quit. Any options specified after this are treated as arguments of the processed command.

- `--pym`

  Execute the specified Python module as a script in MySQL Shell's Python mode. `--pym` works in the same way as Python's `-m` command line option. This option is available from MySQL Shell 8.0.22.

- `--quiet-start[=1|2]`

  Start without printing introductory information. MySQL Shell normally prints information about the product, information about the session (such as the default schema and connection ID), warning messages, and any errors that are returned during startup and connection. When you specify `--quiet-start` with no value or a value of 1, information about the MySQL Shell product is not printed, but session information, warnings, and errors are printed. With a value of 2, only errors are printed.

- `--recreate-schema`

  Drop and recreate the schema that was specified in the connection options, either as part of a URI-like connection string or using the `--schema`, `--database`, or `-D` option. The schema is deleted if it exists.

- `--redirect-primary`

  Ensures that the target server is part of an InnoDB Cluster or InnoDB ReplicaSet and if it is not the primary, finds the primary and connects to it. MySQL Shell exits with an error if any of the following is true when using this option:

  - No instance is specified

  - On an InnoDB Cluster, Group Replication is not active
  - InnoDB Cluster metadata does not exist
  - There is no quorum

- `--replicaset`

  Ensures that the target server belongs to an InnoDB ReplicaSet, and if so, populates the `rs` global variable with the InnoDB ReplicaSet. You can then administer the InnoDB ReplicaSet using the `rs` global variable, for example by issuing `rs.status()`.

- `--redirect-secondary`

  Ensures that the target server is part of a single-primary InnoDB Cluster or InnoDB ReplicaSet and if it is not a secondary, finds a secondary and connects to it. MySQL Shell exits with an error if any of the following is true when using this option:

  - On an InnoDB Cluster, Group Replication is not active
  - InnoDB Cluster metadata does not exist
  - There is no quorum
  - The cluster is not single-primary and is running in multi-primary mode
  - There is no secondary available, for example because there is just one server instance

- `--result-format={table|tabbed|vertical|json|json/pretty|ndjson|json/raw|json/array}`

  Set the value of the `resultFormat` MySQL Shell configuration option for this session. Formats are as follows:

table	The default for interactive mode, unless another value has been set persistently for the `resultFormat` configuration option in the configuration file, in which case that default applies. The `--table` alias can also be used.
tabbed	The default for batch mode, unless another value has been set persistently for the `resultFormat` configuration option in the configuration file, in which case that default applies. The `--tabbed` alias can also be used.
vertical	Produces output equivalent to the `\G` terminator for an SQL query. The `--vertical` or `-E` aliases can also be used.
json or json/pretty	Produces pretty-printed JSON.
ndjson or json/raw	Produces raw JSON delimited by newlines.
json/array	Produces raw JSON wrapped in a JSON array.

  If the `--json` command line option is used to activate JSON wrapping for output for the session, the `--result-format` option and its aliases and the value of the `resultFormat` configuration option are ignored.

- `--save-passwords={always|prompt|never}`

  Controls whether passwords are automatically stored in the secret store. `always` means passwords are always stored unless they are already in the store or the server URL is excluded by a filter.

`never` means passwords are never stored. `prompt`, which is the default, means users are asked whether to store the password or not. See Section 4.4, “Pluggable Password Store”.

- `--schema=name`, `-D name`

  The default schema to use.

- `--server-public-key-path=file_name`

  MySQL Shell equivalent of `--server-public-key-path`.

  If `--server-public-key-path=file_name` is given and specifies a valid public key file, it takes precedence over `--get-server-public-key`.

  Important

  Only supported with classic MySQL protocol connections.

  See `caching_sha2_password` plugin Caching SHA-2 Pluggable Authentication.

- `--show-warnings={true|false}`

  When true is specified, which is the default, in SQL mode, MySQL Shell displays warnings after each SQL statement if there are any. If false is specified, warning are not displayed.

- `--socket[=path]`, `-S [path]`

  On Unix, when a path is specified, the path is the name of the Unix socket file to use for the connection. If you specify `--socket` with no value and no equal sign, or `-S` without a value, the default Unix socket file for the appropriate protocol is used.

  On Windows, the path is the name of the named pipe to use for the connection. The pipe name is not case-sensitive. On Windows, you must specify a path, and the `--socket` option is available for classic MySQL protocol sessions only.

  You cannot specify a socket if you specify a port or a host name other than `localhost` on Unix or a period (.) on Windows.

- `--sql`

  Start in SQL mode, auto-detecting the protocol to use if it is not specified as part of the connection information. When the protocol to use is not specified, defaults to an X Protocol connection, falling back to a classic MySQL protocol connection. To force a connection to use a specific protocol see the `--sqlx` or `--sqlc` options. Alternatively, specify a protocol to use as part of a URI-like connection string or use the `--port` option. See Section 4.3, “MySQL Shell Connections” and MySQL Shell Ports. for more information.

- `--sqlc`

  Start in SQL mode forcing the connection to use classic MySQL protocol, for example to use MySQL Shell with a server that does not support X Protocol. If you do not specify the port as part of the connection, when you provide this option MySQL Shell uses the default classic MySQL protocol port which is usually 3306. The port you are connecting to must support classic MySQL protocol, so for example if the connection you specify uses the X Protocol default port 33060, the connection fails with an error. See Section 4.3, “MySQL Shell Connections” and MySQL Shell Ports. for more information.

- `--sqlx`

  Start in SQL mode forcing the connection to use X Protocol. If you do not specify the port as part of the connection, when you provide this option MySQL Shell uses the default X Protocol port which

is usually 33060. The port you are connecting to must support X Protocol, so for example if the connection you specify uses the classic MySQL protocol default port 3306, the connection fails with an error. See Section 4.3, “MySQL Shell Connections” and MySQL Shell Ports. for more information.

- `--ssh=str`

  Create an SSH tunnel that provides an encrypted connection to the MySQL server instance. Supply the URI for connection to the SSH server in the format `[user@]host[:port]`, for example:

  ```
  --ssh root@198.51.100.4:2222
  ```

  When you use this option, you must also specify the `--user`, `--host`, and `--port` options, or a URI, for connection to the MySQL server instance. For information on SSH tunnel connections from MySQL Shell, see Section 4.3.6, “Using an SSH Tunnel”.

- `--ssh-config-file=path`

  Specify the path to an SSH configuration file for the connection to the SSH server. You can use the MySQL Shell configuration option `ssh.configFile` to set a custom file as the default if this option is not specified. If `ssh.configFile` has not been set, the default is the standard SSH configuration file `~/.ssh/config`. If you specify `--ssh-config-file` with an empty value, the default file specified by `ssh.configFile` is ignored, and the `~/.ssh/config` file is used instead.

- `--ssh-identity-file=path`

  Specify the path to an identity file for the connection to the SSH server. The default if this option is not specified is the standard private key file in the SSH configuration folder (`~/.ssh/id_rsa`).

- `--ssl*`

  Options that begin with `--ssl` specify whether to connect to the server using SSL and indicate where to find SSL keys and certificates. The `mysqlsh` SSL options function in the same way as the SSL options for MySQL Server, see Command Options for Encrypted Connections for more information.

  `mysqlsh` accepts these SSL options: `--ssl-mode`, `--ssl-ca`, `--ssl-capath`, `--ssl-cert`, `--ssl-cipher`, `--ssl-crl`, `--ssl-crlpath`, `--ssl-key`, `--tls-version`, `--tls-ciphersuites`.

- `--syslog`

  Send SQL statements that you issue in MySQL Shell’s SQL mode to the operating system’s system logging facility (`syslog` on Unix, or the Windows Event Log). System logging for SQL statements only takes place when MySQL Shell is started in interactive mode, so either a normal start or a start with the `--interactive` option. It does not take place if the `--execute` or `--file` options are used at startup to run `mysqlsh` in batch mode. See Section 12.3, “System Logging for User SQL Statements” for more information.

- `--tabbed`

  Display results in tab separated format in interactive mode. The default for that mode is table format. This option is an alias of the `--result-format=tabbed` option.

- `--table`

  Display results in table format in batch mode. The default for that mode is tab separated format. This option is an alias of the `--result-format=table` option.

- `--uri=str`

  Create a connection upon startup, specifying the connection options in a URI-like string as described at Connecting to the Server Using URI-Like Strings or Key-Value Pairs.

- `--user=user_name`, `-u user_name`

  The MySQL user name to use when connecting to the server.

- `--verbose[=0|1|2|3|4]`

  Activate verbose output to the console and specify the level of detail. The value is an integer in the range from 0 to 4. 0 displays no messages, which is the default verbosity setting when you do not specify the option. 1 displays error, warning and informational messages (this is the default setting if you specify the option on the command line without a value). 2, 3, and 4 add higher levels of debug messages. See Chapter 12, *MySQL Shell Logging and Debug* for more information.

- `--version`, `-V`

  Display the version of MySQL Shell and exit.

- `--vertical`, `-E`

  Display results vertically, as when the `\G` terminator is used for an SQL query. This option is an alias of the `--result-format=vertical` option.