MySQL Shell 8.0 (part of MySQL 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, see Chapter 6, *Using MySQL AdminAPI*.

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.

Document generated on: 2021-01-29 (revision: 68617)

Table of Contents

Chapter 1 MySQL Shell Features

The following features are available in MySQL Shell.

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, MySQL Shell bundles Python 3.7.7. 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.

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.

- The AdminAPI enables you to administer MySQL instances, using them to create InnoDB Clusters, InnoDB ReplicaSets, 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. Similarly, InnoDB ReplicaSet enables you to administer a set of MySQL instances running asynchronous GTID-based replication. AdminAPI also provides operations to configure users for MySQL Router, to make integration with InnoDB Cluster and InnoDB ReplicaSet as simple as possible. See Chapter 6, *Using MySQL AdminAPI*.

- The 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 7, *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 8, *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 Interface”.

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 9, *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

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 2015 (available at the Microsoft Download Center) installed on your Windows system.

Requirements

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

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

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.
`\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.

Command	Alias/Shortcut	Description
`\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 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 *`Cluster`* or *`dba.Cluster`* to search for help on the AdminAPI `dba.Cluster()` 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.

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.

Note that 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. 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 Shellconfiguration 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 10.4, “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 7.1.5, “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

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.

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 secon
```

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:

```
shell> 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:

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

```
shell> 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 password-less 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.

Instead of a TCP connection, you can connect using a Unix socket file or a Windows named pipe. For instructions, see Section 4.3.2, “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 4.3.3, “Using Encrypted Connections”.

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 4.3.4, “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 password-less 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.

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:

```
shell> 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 4.3.3, “Using Encrypted Connections”. For instructions to establish a compressed connection, see Section 4.3.4, “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:

```
shell> 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:

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

4.3.2 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.

4.3.3 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-version=version`: The TLS protocols permitted for encrypted connections, specified as a comma separated list. For example `--tls-version=TLSv1.1,TLSv1.2`.
- `--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.

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.

4.3.4 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.

4.3.4.1 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 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:

  ```
  shell> 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:

  ```
  shell> 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:

```
shell> 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:

```
shell> 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
```

```
shell> 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&compres
```

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

- 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.

4.3.4.2 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:

```
shell> 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
```

```
shell> 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, available on all platforms supported by the MySQL server (as long as MySQL client package is installed), and offers persistent storage. See `mysql_config_editor` — MySQL Configuration Utility.
- 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 10.4, “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 and InnoDB ReplicaSet administration functions using the AdminAPI. See Chapter 6, *Using 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 10.4, “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 7.1, “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 8, *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 7.2, “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 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.

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 10.3, “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.

Chapter 5 MySQL Shell Code Execution

Table of Contents

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.

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

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

```
shell> mysqlsh < code.js
```

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

```
shell> 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

When MySQL Shell is in SQL mode, autocompletion tries to complete any word with all possible completions that match. In SQL mode the following can be autocompleted:

- SQL keywords - List of known SQL keywords. Matching is case-insensitive.
- SQL snippets - Certain common snippets, such as `SHOW CREATE TABLE`, `ALTER TABLE`, `CREATE TABLE`, and so on.
- Table names - If there is an active schema and database name caching is not disabled, all the tables of the active schema are used as possible completions.

As a special exception, if a backtick is found, only table names are considered for completion. In SQL mode, autocompletion is not context aware, meaning there is no filtering of completions based on the SQL grammar. In other words, autocompleting **`SEL`** returns `SELECT`, but it could also include a table called selfies.

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,lastw
```

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*"`.

Note that regardless of the filters set in the history ignore list, the last executed statement is always available to be recalled by pressing the Up arrow, 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 Interface”.

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:

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

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

```
shell> mysqlsh < code.js
```

This example shows how to redirect SQL code to standard input for execution:

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

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 10.4, "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.sql("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.sql("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.sql("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.sql("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.sql("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.sql("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`)

```
shell> echo "select * from world_x.city where countrycode='AUT'" | mysqlsh --json --sql --uri user@localhos
or
shell> echo "select * from world_x.city where countrycode='AUT'" | mysqlsh --json=pretty --sql --uri user@
{
    "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`)

```
shell> echo "select * from world_x.city where countrycode='AUT'" | mysqlsh --json=raw --sql --uri user@localho
{"hasData":true,"rows":[{"ID":1523,"Name":"Wien","CountryCode":"AUT","District":"Wien","Info":{"Population":16
```

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 Interface

MySQL Shell exposes much of its functionality using an API command syntax that enables you to easily integrate `mysqlsh` with other tools. This functionality is similar to using the `--execute` option, but the command interface uses a simplified argument syntax which reduces the quoting and escaping that can be required by terminals. For example if you want to create an InnoDB Cluster using a `bash` script, you could use this functionality.

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.
- `cluster` - represents an InnoDB Cluster.
- `dba` - provides access to InnoDB cluster administration functions using the AdminAPI. See Chapter 6, *Using MySQL AdminAPI*.
- `shell` - global provides access to MySQL Shell functions, such as `shell.options` for configuring MySQL Shell options (see Section 10.4, “Configuring MySQL Shell Options”), and `shell.reports` for running MySQL Shell reports (see Section 7.1, “Reporting with MySQL Shell”).
- `util` - provides access to MySQL Shell utilities. See Chapter 8, *MySQL Shell Utilities*.

API Command Line Integration Syntax

When you start MySQL Shell on the command-line using the following special syntax, the `--` indicates the end of the list of options and everything after it is treated as a command and its arguments.

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

where the following applies:

- *`shell_object`* is a string which maps to a MySQL Shell global object.
- *`object_method`* is the name of the method provided by the *`shell_object`*. The method names can be provided following either the JavaScript, Python or an alternative command line typing friendly format, where all known methods 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, `getCluster` becomes `get-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 *`object_method`* must match one of the global object's methods in one of the valid formats (JavaScript, Python or command line friendly). If they do not correspond to a valid global object and its methods, MySQL Shell exits with status 10.

API 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. For example, special characters that are handled by the system shell (`bash`, `cmd`, and so on) should be avoided and if quoting is needed, only the quoting of the parent shell should be considered. In other words, if “foo bar” is used as a parameter in `bash`, the quotes are stripped and escapes are handled.

There are two types of arguments that can be used in the list of arguments: positional arguments and named arguments. Positional arguments are for example simple types such as strings, numbers, boolean, null. Named arguments are key value pairs, where the values are simple types. Their usage must adhere to the following pattern:

```
[ positional_argument ]* [ { named_argument* } ]* [ named_argument ]*
```

The rules for using this syntax are:

- all parts of the syntax are optional and can be given in any order
- nesting of curly brackets is forbidden
- all the key values supplied as named arguments must have unique names inside their scope. The scope is either ungrouped or in a group (inside the curly brackets).

These arguments are then converted into the arguments passed to the method call in the following way:

- all ungrouped named arguments independent to where they appear are combined into a single dictionary and passed as the last parameter to the method
- named arguments grouped inside curly brackets are combined into a single dictionary
- positional arguments and dictionaries resulting from grouped named arguments are inserted into the *`arguments`* list in the order they appear on the command line

API Interface Examples

Using the API integration, calling MySQL Shell commands 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='password
```

  This maps to the equivalent command in MySQL Shell:

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

- 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
```

  This maps to the equivalent command in MySQL Shell:

```
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
```

  This maps to the equivalent command in MySQL Shell:

```
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
```

  This maps to the equivalent command in MySQL Shell:

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

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

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

  This maps to the equivalent command in MySQL Shell:

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

Chapter 6 Using MySQL AdminAPI

Table of Contents

This chapter covers MySQL AdminAPI, provided with MySQL Shell, which enables you to administer MySQL instances, using them to create InnoDB Clusters, InnoDB ReplicaSets, and integrating MySQL Router.

6.1 MySQL AdminAPI

This section provides an overview of AdminAPI and what you need to know to get started.

MySQL Shell includes the AdminAPI, which is accessed through the `dba` global variable and its associated methods. The `dba` variable's methods provide operations which enable you to deploy, configure, and administer InnoDB Cluster 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 and updating users that enable you to integrate your InnoDB Cluster and InnoDB ReplicaSet.

MySQL Shell provides two scripting 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 (part of MySQL 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:

```
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.

>trimmed for brevity<
```

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.

This section applies to using InnoDB Cluster or InnoDB ReplicaSet and consists of:

- Deployment Scenarios
- Installing the Components
- Configuring Host Name
- Specifying Instances
- Persisting Settings
- Retrieving a Handler Object
- Creating User Accounts for Administration
- Verbose Logging
- Finding the Primary
- Scripting AdminAPI

Deployment Scenarios

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 want to test a deployment before committing to a full production deployment, the provided sandbox feature enables you to quickly set up a test environment on your local machine. Sandbox server instances are created with the required configuration and you can experiment to become familiar with the technologies employed.

Important

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

Installing the 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 install needs to only be done once on the local machine. The following methods of installing are available:

Downloading and installing the components using the following documentation:

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

Important

Always use the matching version of components, for example run MySQL Shell 8.0.25 to administer instances running MySQL 8.0.25 with MySQL Router 8.0.25.

Once you have installed the software required, choose to follow either Section 6.2, “MySQL InnoDB Cluster” or Section 6.3, “MySQL InnoDB ReplicaSet”.

Configuring Host Name

In a production deployment, the instances which you use run on separate machines, therefore each machine must have a unique host name and be able to resolve the host names of the other machines which run server instances. If this is not the case, you can:

- configure each machine to map the IP of each other machine to a host name. See your operating system documentation for details. This is the recommended solution.
- set up a DNS service
- 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 or hostname that resolves to an IPv6 address for instance connection strings and with options such as `localAddress`, `groupSeeds` 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 is correctly configured, execute the following query to see how the instance reports its own address to other servers and try to connect to that MySQL server from other hosts using the returned address:

```
SELECT coalesce(@@report_host, @@hostname);
```

Specifying Instances

One of the core concepts of using AdminAPI is understanding connections to the MySQL instances which make up your InnoDB Cluster or InnoDB ReplicaSet. The requirements for connections to the instances when administering, and for the connections between the instances themselves, are:

- only TCP/IP connections are supported, using Unix sockets or named pipes is not supported. InnoDB Cluster and InnoDB ReplicaSet are intended to be used in a local area network, running over a wide area network is not recommended.
- only classic MySQL protocol connections are supported, X Protocol is not supported.

Tip

Your applications can use X Protocol, this requirement is for administration operations using AdminAPI.

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 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')
```

Assuming you are running MySQL Shell in the default interactive mode, you are prompted for your password. 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 are no longer prompted for it.

Persisting Settings

The AdminAPI commands you use to work with an InnoDB Cluster, InnoDB ReplicaSet, and their server instances modify the configuration of the MySQL on the instance. Depending on the way MySQL Shell is connected to an instance and the version of MySQL installed on the instance, these configuration changes can be persisted to the instance automatically. Persisting settings to the instance ensures that configuration changes are retained after the instance restarts, for background information see `SET PERSIST`. This is essential for reliable usage, for example if settings are not persisted then an instance which has been 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. On local instances which do not support persisting configuration changes automatically, you need to make the changes locally, 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. This means that 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, see Configuring Instances with `dba.configureLocalInstance()`.

Retrieving a Handler Object

When you are working with AdminAPI, you use a handler object which represents the InnoDB Cluster or InnoDB ReplicaSet. You assign this object to a variable, and then use the operations available to monitor and administer the InnoDB Cluster or InnoDB ReplicaSet. To be able to retrieve the handler object, you establish a connection to one of the instances which belong to the InnoDB Cluster 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 object to work with the cluster, for example to add instances or check the cluster's status. If you want to retrieve a cluster again at a later date, for example after restarting MySQL Shell, use the `dba.getCluster([name],[options])` function. For example:

```
mysql-js> var cluster1 = dba.getCluster()
```

Similarly, use the `dba.getReplicaSet()` operation to retrieve an InnoDB ReplicaSet. For example:

```
mysql-js> var replicaset1 = dba.getReplicaSet()
```

If you do not specify a *`name`* then the default object is returned. 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 active session, in other words the same instance as the 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:

```
mysql-js> shell.connect(secondary_member)
mysql-js> var cluster1 = dba.getCluster(testCluster, {connectToPrimary:false})
```

Tip

Remember that secondary instances have `super_read_only=ON`, so you cannot write changes to them.

Creating User Accounts for Administration

The user account used to administer an instance does not have to be the root account, however the user needs to be assigned full read and write privileges on the metadata tables in addition to full MySQL administrator privileges (`SUPER`, `GRANT OPTION`, `CREATE`, `DROP` and so on). In this procedure the user `icadmin` is shown in InnoDB Cluster examples, and `rsadmin` in InnoDB ReplicaSet examples.

Important

The user name and password of an administrator must be the same on all instances.

In version 8.0.20 and later, use the `setupAdminAccount(user)` operation to create or upgrade a MySQL user account with the necessary privileges to administer an InnoDB Cluster or InnoDB ReplicaSet. 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()` operation.

The mandatory *`user`* argument is the name of the MySQL account you want to create or upgrade to be used to administrator the account. The format of the user names accepted by the `setupAdminAccount()` operation follows the standard MySQL account name format, 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`*, 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.
```

If you already have an administration user, for example created with a version prior to 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 administration user is compatible. For example, to upgrade the user named `icadmin` issue:

```
mysql-js> myCluster.setupAdminAccount('icadmin', {'update':1})
Updating user icadmin@%.
Account icadmin@% was successfully updated.
```

In versions prior to 8.0.20, the preferred method to create users for administration is using the `clusterAdmin` option with the `dba.configureInstance()` operation. 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 example the root user is used. For example:

```
mysql-js> dba.configureInstance('root@ic-1:3306', {clusterAdmin: "'icadmin'@'ic-1%'"});
```

The format of the user names accepted by the `setupAdminAccount()` operation and the `clusterAdmin` option follows the standard MySQL account name format, see Specifying Account Names.

If only read operations are needed (such as for monitoring purposes), an account with more restricted privileges can be used. See Configuring Users for AdminAPI.

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:

```
shell> mysqlsh --log-level=DEBUG3
```

The `DEBUG3` level is recommended, see `--log-level` for more information. 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`. See Chapter 9, *MySQL Shell Logging and Debug* for more information.

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. To configure MySQL Shell to do this, see Section 9.3, “Logging AdminAPI Operations”.

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 can be 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.

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:

```
shell> 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 file is shown here:

```
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');
}
```

AdminAPI is also supported by MySQL Shell's Section 5.8, “API Command Line Interface”. This 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()
```

6.2 MySQL InnoDB Cluster

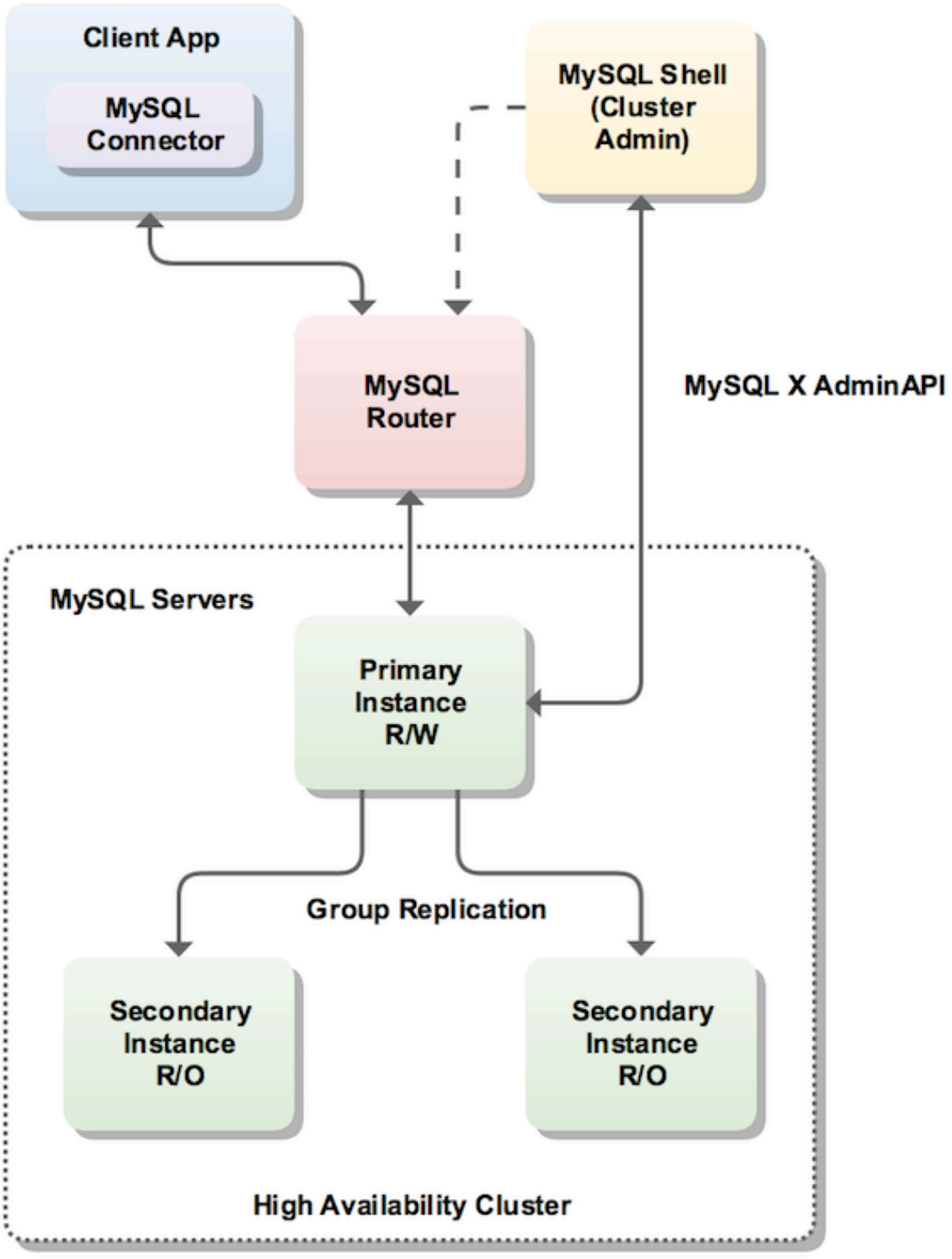
MySQL InnoDB Cluster provides a complete high availability solution for MySQL. By using the AdminAPI included with MySQL Shell you can easily configure and administer a group of at least three MySQL server instances to function as an InnoDB Cluster. In this procedure the host name `ic-number` is used in examples. Each MySQL server instance runs MySQL Group Replication, which provides the mechanism to replicate data within InnoDB Clusters, with built-in failover. AdminAPI removes the need to work directly with Group Replication in InnoDB Clusters, but for more information see Group Replication which explains the details. 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.

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.

The following diagram shows an overview of how these technologies work together:

Figure 6.1 InnoDB Cluster overview



6.2.1 MySQL 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.

- Group Replication members can contain tables using a storage engine other than `InnoDB`, for example `MyISAM`. Such tables cannot be written to by Group Replication, and therefore when using InnoDB Cluster. To be able to write to such tables with InnoDB Cluster, convert all such tables to `InnoDB` before using the instance in an InnoDB Cluster.

- 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.

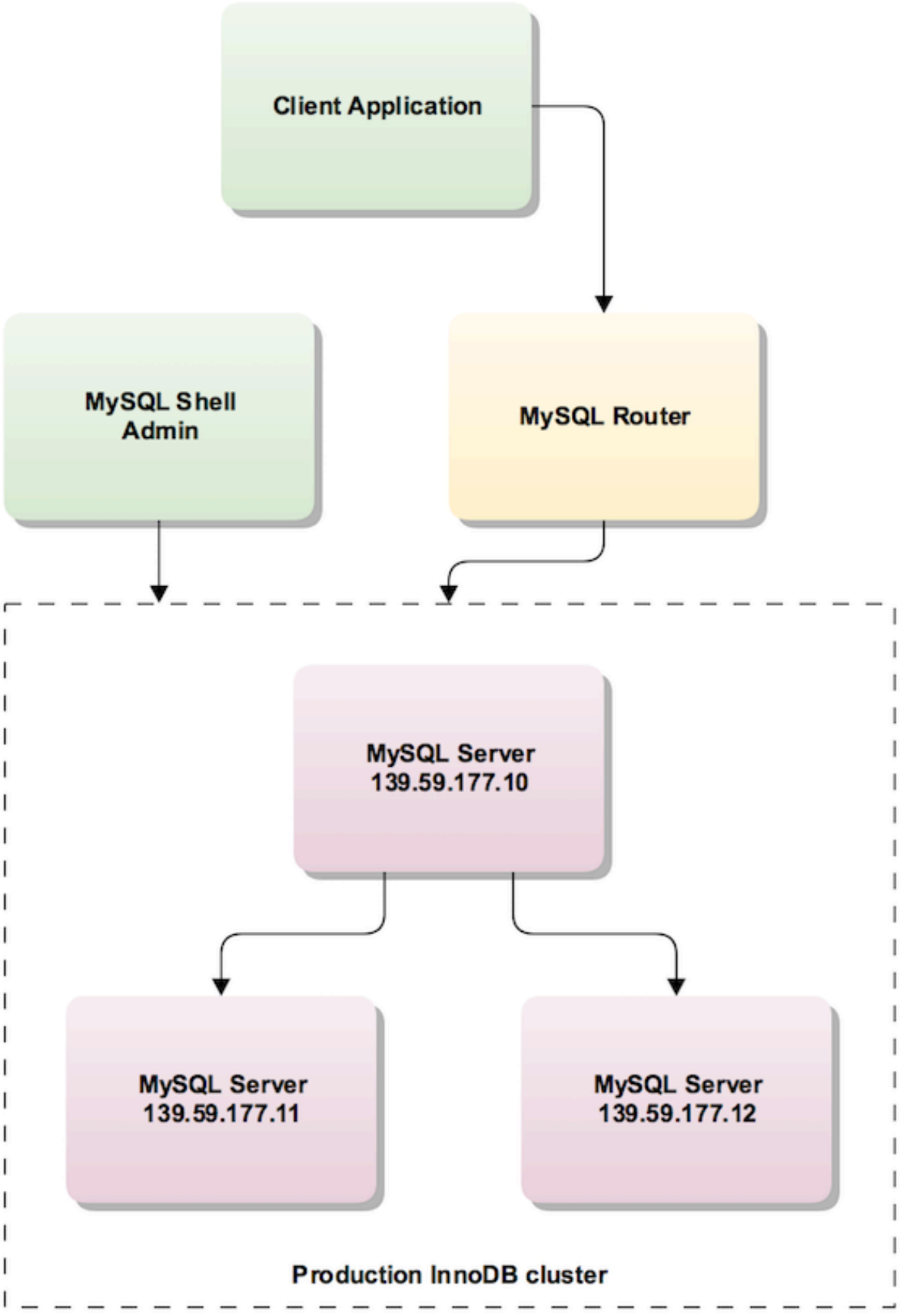
- From version 8.0.23, instances should be configured to use the parallel replication applier. See Configuring the Parallel Replication Applier.

- 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.

6.2.2 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 6.5, “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 Installing the Components.

The following diagram illustrates the scenario you work with in this section:


Client Application
MySQL Shell
Admin
MySQL Router
MySQL Server
139.59.177.10
MySQL Server
139.59.177.11
MySQL Server
139.59.177.12
Production InnoDB cluster

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 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

6.2.2.1 Deploying a New Production InnoDB Cluster

The following sections describe how to deploy a new production InnoDB Cluster.

- Configuring Production Instances
- Creating the Cluster
- Adding Instances to a Cluster
- User Accounts Created by InnoDB Cluster
- Configuring InnoDB Cluster Ports

Configuring Production Instances

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 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 command returns a descriptive text message about the operation's result.

The *instance* definition is the connection data for the instance, 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.
- `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 Creating User Accounts for Administration.
- `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.
- deprecated, and scheduled for removal in a future version

  `clearReadOnly` - a boolean value used to confirm that `super_read_only` should be set to off, see Super Read-only and Instances.
- `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 instace 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 Creating the 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 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 Creating User Accounts for Administration.

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 Super Read-only and Instances for more information.

Creating the 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 Users for AdminAPI for more information on cluster administration users.

The `dba.createCluster()` operation supports MySQL Shell's `interactive` option. When `interactive` is on, prompts appear in the following situations:

- when run on an instance that belongs to a cluster and the `adoptFromGr` option is false, you are asked if you want to adopt an existing cluster
- when the `force` option is not used (not set to `true`), you are asked to confirm the creation of a multi-primary cluster

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 timeout 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 6.2.5, “Working with InnoDB Cluster”.

Adding Instances to a Cluster

Use the `Cluster.addInstance(`*`instance`*`)` function to add more instances to the cluster, where *`instance`* is connection information to a configured instance, see Configuring Production Instances. 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

You need a minimum of three instances in the cluster to make it tolerant to the failure of one instance. Adding further instances increases the tolerance to failure of an instance. To add an instance to the cluster issue:

```
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.
```

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 Customizing InnoDB Clusters 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 6.2.2.2, “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 chose 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 i
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 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 Super Read-only and Instances for more information.

Once you have your cluster deployed you can configure MySQL Router to provide high availability, see Section 6.4, “MySQL Router”.

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 the internal users depends on whether the `ipAllowlist` option has been configured. If `ipAllowlist` is not configured, it defaults to `AUTOMATIC` and the internal users are created using both the wildcard `%` character and `localhost` for the hostname value. When `ipAllowlist` has been configured, for each address in the `ipAllowlist` list an internal user is created. 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()`.

In addition, the *`Cluster`*`.rejoinInstance()` operation can also result in a new internal user being generated when the `ipAllowlist` option is used to specify a hostname. For example by issuing:

```
Cluster.rejoinInstance({ipAllowlist: "192.168.1.1/22"});
```

all previously existing internal users are removed and a new internal user is created, taking into account the `ipAllowlist` value used.

For more information on the internal users required by Group Replication, see User Credentials For Distributed Recovery.

Configuring InnoDB Cluster Ports

Instances that belong to a cluster use different ports for different types of communication. 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 Reference.

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'}
```

6.2.2.2 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()`.

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 Adding Instances to a 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.

6.2.2.3 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 Super Read-only and Instances for more information.

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.

6.2.3 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",
                "mode": "R/W",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            },
            "ic-2:3306": {
                "address": "ic-2:3306",
                "mode": "R/O",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            },
            "ic-3:3306": {
                "address": "ic-3:3306",
                "mode": "R/O",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            }
        }
    },
    "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 Securing your Cluster.
- `status`: The status of this element of the cluster. For the overall cluster this describes the high availability provided by this 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.
- `topology`: The instances which have been added to the cluster.
- `Host name of instance`: The host name of an instance, for example localhost:3310.
- `role`: what function this instance provides in the cluster. Currently only HA, for high availability.
- `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.
- `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.

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, 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` is 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.
  - `groupName` is the group's name, a UUID.
- 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.
  - `memberRole` the Member Role as reported by the Group Replication plugin, see the `MEMBER_ROLE` column of the `replication_group_members` table.
  - `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 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"
    }
]
```

6.2.4 Working with Instances

This section describes the AdminAPI operations which apply to instances. You can configure instances before using them with InnoDB Cluster, check the state of an instance, and so on.

- Using `dba.checkInstanceConfiguration()`
- Configuring Instances with `dba.configureLocalInstance()`
- Checking Instance State

Using `dba.checkInstanceConfiguration()`

Before creating a production deployment from server instances you need to check that MySQL on each instance is correctly configured. In addition to `dba.configureInstance()`, which checks the configuration as part of configuring an instance, you can use the `dba.checkInstanceConfiguration(`*`instance`*`)` function. This ensures that the *`instance`* satisfies the Section 6.2.1, “MySQL 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 Users for AdminAPI. 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 server |
| gtid_mode                | OFF           | ON             | Update read-only variable and restart the server |
| server_id                | 1             |                | Update read-only variable and restart the server |
+--------------------------+---------------+----------------+--------------------------------------------------+

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 required before you can proceed. 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.

Configuring Instances with `dba.configureLocalInstance()`

Instances which do not support persisting configuration changes automatically (see 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 recommended method 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`*:

```
shell> 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.

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.

6.2.5 Working with InnoDB Cluster

This section explains how to work with InnoDB Cluster, and how to handle common administration tasks.

- Removing Instances from the InnoDB Cluster
- Dissolving an InnoDB Cluster
- Changing a Cluster's Topology

Removing Instances from the 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.

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 10.4, “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.

Important

The `force` option should only be used with `Cluster.removeInstance(instance)` when you want to ignore any errors, for example unprocessed transactions or an instance being `UNREACHABLE`, and do not plan to reuse the instance with the cluster. Ignoring errors when removing an instance from the cluster could result in an instance which is not in synchrony with the cluster, preventing it from rejoining the cluster at a later time. Only use the `force` option when you plan to no longer use the instance with the cluster, in all other cases you should always try to recover the instance and only remove it when it is available and healthy, in other words with the status `ONLINE`.

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.

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 UDFs 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 to become the new primary, use the `Cluster.setPrimaryInstance(instance)` function, where `instance` specifies the connection to the instance which should become the new primary. This enables you to configure the underlying Group Replication group to choose a specific instance as the new primary, bypassing the election process.

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).

6.2.6 Configuring InnoDB Cluster

This section describes how to use AdminAPI to configure an InnoDB Cluster.

- Setting Options for InnoDB Cluster
- Customizing InnoDB Clusters
- Configuring the Election Process
- Configuring Failover Consistency
- Configuring Automatic Rejoin of Instances
- Configuring the Parallel Replication Applier
- Securing your Cluster
- Creating an Allowlist of Servers

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 Configuring Automatic Rejoin of Instances.
- `exitStateAction`: string value indicating the Group Replication exit state action. See Configuring Automatic Rejoin of Instances.
- `memberWeight`: integer value with a percentage weight for automatic primary election on failover. See Configuring the Election Process.
- `tag:`*`option`*: built-in and user-defined tags to be associated to the cluster. See Section 6.2.9, “Tagging the 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.
- `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 Creating the Cluster.
- `failoverConsistency`: string value indicating the consistency guarantees that the cluster provides. See Configuring Automatic Rejoin of Instances.

This option is changeable at the per instance level only:

- `label`: a string identifier of the instance

Customizing InnoDB Clusters

When you create a cluster and add instances to it, values such as the group name, the local address, and the seed instances are configured automatically by AdminAPI. These 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()`.

To customize the name of the replication group created by InnoDB Cluster, pass the `groupName` option to the `dba.createCluster()` command. This sets the `group_replication_group_name` system variable. The name must be a valid UUID.

To customize the address which an instance provides for connections from other instances, pass the `localAddress` option to the `dba.createCluster()` and `cluster.addInstance()` commands. 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.

To customize the instances used as seeds when an instance joins the cluster, pass the `groupSeeds` option to the `dba.createCluster()` and *`Cluster`*`.addInstance()` operations. Seed instances are contacted when a new instance joins a cluster and are used to provide data to the new instance. The addresses of the seed instances are specified as a comma separated list such as `host1:port1`,`host2:port2`. This configures the `group_replication_group_seeds` system variable. When a new instance is added to a cluster, the local address of this instance is automatically

appended to the list of group seeds of all online cluster members in order to allow them to use the new instance to rejoin the group if necessary.

Note

the instances in the seed list are used according to the order in which they appear in the list. This means that a user specified seed is used first and preferred over automatically added instances.

For more information see the documentation of the system variables configured by these AdminAPI options.

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 precentage 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})
```

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.

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 0, which means that instances do not try to automatically rejoin. When you are 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`.

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.

Securing your 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

Once you have configured a cluster to use encrypted connections you must add the servers to the `ipAllowlist`.

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 permitted values are `DISABLED`, `REQUIRED`, and `AUTO`. These modes are defined as:

- Setting `createCluster({memberSslMode:'DISABLED'})` ensures SSL encryption is disabled for the seed instance in the cluster.
- Setting `createCluster({memberSslMode:'REQUIRED'})` then SSL encryption is enabled for the seed instance in the cluster. If it cannot be enabled an error is raised.
- Setting `createCluster({memberSslMode:'AUTO'})` (the default) then SSL encryption is automatically enabled if the server instance supports it, or disabled if the server does not support it.

Note

When using the commercial version of MySQL, SSL is enabled by default and you might need to configure the allowlist for all instances. See Creating an Allowlist of Servers.

When you issue the `cluster.addInstance()` and `cluster.rejoinInstance()` commands, SSL encryption on the instance is enabled or disabled based on the setting found for the seed instance.

When using `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. Use the `ignoreSslError` option set to false to ensure that sandbox instances are deployed with SSL support, issuing an error if SSL support cannot be provided. When `ignoreSslError` is true, which is the default, no error is issued during the operation if the SSL support cannot be provided and the server instance is deployed without SSL support.

Creating an Allowlist of Servers

When using a cluster's `createCluster()`, `addInstance()`, and `rejoinInstance()` methods you can optionally specify a list of approved servers that belong to the cluster, 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. 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.

6.2.7 Troubleshooting InnoDB Cluster

This section describes how to troubleshoot an InnoDB Cluster.

- Rejoining an Instance to a Cluster
- Restoring a Cluster from Quorum Loss
- Rebooting a Cluster from a Major Outage
- Rescanning a Cluster

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 Super Read-only and Instances for more information.

In the case where an instance has not had its configuration persisted (see 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 it's 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.

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-1:3306]

  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`.

Rebooting a Cluster from a Major Outage

If your cluster suffers from a complete outage, you can ensure it is reconfigured correctly using `dba.rebootClusterFromCompleteOutage()`. This operation takes the instance which MySQL Shell is currently connected to and uses its metadata to recover the cluster. In the event that a cluster's instances have completely stopped, the instances must be started and only then can the cluster be started. For example if the machine a sandbox cluster was running on has been restarted, and the instances were at ports 3310, 3320 and 3330, issue:

```
mysql-js> dba.startSandboxInstance(3310)
mysql-js> dba.startSandboxInstance(3320)
mysql-js> dba.startSandboxInstance(3330)
```

This ensures the sandbox instances are running. In the case of a production deployment you would have to start the instances outside of MySQL Shell. Once the instances have started, you need to connect to an instance with the GTID superset, which means the instance which had applied the most transaction before the outage. If you are unsure which instance contains the GTID superset, connect to any instance and follow the interactive messages from the `dba.rebootClusterFromCompleteOutage()` operation, which detects if the instance you are connected to contains the GTID superset. Reboot the cluster by issuing:

```
mysql-js> var cluster = dba.rebootClusterFromCompleteOutage();
```

The `dba.rebootClusterFromCompleteOutage()` operation then follows these steps to ensure the cluster is correctly reconfigured:

- 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, in other words the transactions applied by the cluster. If the currently connected instance does not contain the GTID superset, the operation aborts with that information. See the subsequent paragraphs for more information.
- If the instance contains the GTID superset, the cluster is recovered based on the metadata of the instance.
- Assuming you are running MySQL Shell in interactive mode, a wizard is run that checks which instances of the cluster are currently reachable and asks if you want to rejoin any discovered instances to the rebooted cluster.
- Similarly, in interactive mode the wizard also detects instances which are currently not reachable and asks if you would like to remove such instances from the rebooted cluster.

If you are not using MySQL Shell's interactive mode, you can use the `rejoinInstances` and `removeInstances` options to manually configure instances which should be joined or removed during the reboot of the cluster.

If you encounter an error such as `The active session instance isn't the most updated in comparison with the ONLINE instances of the Cluster's metadata.` then the instance you are connected to does not have the GTID superset of transactions applied by the cluster. In this situation, connect MySQL Shell to the instance suggested in the error message and issue `dba.rebootClusterFromCompleteOutage()` from that instance.

Tip

To manually detect which instance has the GTID superset rather than using the interactive wizard, check the `gtid_executed` variable on each instance. For example issue:

```
mysql-sql> SHOW VARIABLES LIKE 'gtid_executed';
```

The instance which has applied the largest GTID set of transactions contains the GTID superset.

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.

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.

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 InnoDB Cluster and Auto-increment.

6.2.8 Upgrading an 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 8.1, “Upgrade Checker Utility” to verify instances are ready for the upgrade.

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), if it discovers the metadata version is 0.0.0 (upgrade in progress) it does not proceed with the metadata refresh 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 both version 1.x.x and 2.x.x. metadata, and the version can change between TTL refreshes. This ensures routing continues while the cluster is upgraded.

Although it is possible to have instances in a cluster which 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 a mix of versions, if 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 should not be relied on for long term use.

6.2.8.1 Rolling Upgrades

When upgrading the metadata schema of clusters deployed by MySQL Shell versions before 8.0.19, a rolling upgrade of existing MySQL Router instances is required. This process minimizes disruption to applications during the upgrade. The rolling upgrade process must be performed in the following order:

1. Run the latest MySQL Shell version, connect the global session to the cluster and issue `dba.upgradeMetadata()`. This step ensures that any MySQL Router accounts configured for the cluster have their privileges modified to be compatible with the new version. The upgrade function stops if an outdated MySQL Router instance is detected, at which point you can stop the upgrade process in MySQL Shell to resume later.

2. Upgrade any detected out of date MySQL Router instances to the latest version. It is recommended to use the same MySQL Router version as MySQL Shell version.

3. Continue or restart the `dba.upgradeMetadata()` operation to complete the metadata upgrade.

6.2.8.2 Upgrading InnoDB Cluster Metadata

As AdminAPI evolves, some releases might require you to upgrade the metadata of existing 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 8.0.19 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()`, and `Cluster.options()`.

This `dba.upgradeMetadata()` operation compares the version of the metadata schema found on the cluster MySQL Shell is currently connected to with the version of the metadata schema supported by this MySQL Shell version. If the installed metadata version is lower, an upgrade process is started. The `dba.upgradeMetadata()` operation then upgrades 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 cluster, only then can the MySQL Router application be upgraded. To get information on which of the MySQL Router instances registered with a cluster require the metadata upgrade, 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"
        }
    }
}
```

Warning

A cluster which is using the new metadata cannot be administered by earlier MySQL Shell versions, for example once you upgrade to version 8.0.19 you can no longer use version 8.0.18 or earlier to administer the cluster.

To upgrade a 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. For example:

```
mysql-js> \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 you encounter an error related to the cluster administration user missing privileges, use the *Cluster*.setupAdminAccount() operation with the update option to grant the user the correct privileges. See Configuring Users for AdminAPI.

6.2.8.3 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.

6.2.9 Tagging the Metadata

From version 8.0.21, a configurable tag framework is available, to allow the metadata of a Cluster or 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 MySQL Router usage, which enable a compatible MySQL Router to support hiding instances from applications. The following tags are reserved for this purpose:

- `_hidden` instructs MySQL Router to exclude 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 of key-value pairs that can be associated with Clusters, ReplicaSets or their specific instances. Tag values can be any JSON value. This 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 set the tag *`test`* to `true` on the *`myCluster`* instance `ic-1`, issue:

```
mysql-js> myCluster.setInstanceOption(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 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 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 Routers bootstrapped against the InnoDB Cluster 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 the 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()` operation to enable the `_hidden` and `_disconnect_existing_sessions_when_hidden` tags:

```
mysql-js> myCluster.setInstanceOption(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 disconne
2020-07-03 16:32:16 routing INFO [7fa9d164c700] Routing routing:testCluster_x_ro listening on 64470 got re
2020-07-03 16:32:16 routing INFO [7fa9d164c700] Routing routing:testCluster_x_rw listening on 64460 got re
2020-07-03 16:32:16 routing INFO [7fa9d164c700] Routing routing:testCluster_rw listening on 6446 got reque
2020-07-03 16:32:16 routing INFO [7fa9d164c700] Routing routing:testCluster_ro listening on 6447 got reque
```

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": []
        }
    }
}
```

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:

```
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"
                }
            ]
        }
    }
}
```

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 region which it is operating in. For example, you could set a custom tag named location, with a value of EMEA on the cluster.

6.2.10 InnoDB Cluster Tips

This section describes some information which is good to know when using InnoDB Cluster.

- Super Read-only and Instances
- Configuring Users for AdminAPI

- InnoDB Cluster and Auto-increment
- InnoDB Cluster and Binary Log Purging
- Resetting Recovery Account Passwords

Super Read-only and Instances

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`.

Configuring Users for AdminAPI

Working with instances that belong to an InnoDB Cluster or InnoDB ReplicaSet requires that you connect to the instances with a user that has the correct privileges to administer the instances. AdminAPI provides the following ways to administer suitable users:

- In version 8.0.20 and later, use the `setupAdminAccount(user)` operation, which creates or upgrades a MySQL user account with the necessary privileges to administer an InnoDB Cluster or InnoDB ReplicaSet.
- In versions prior to 8.0.20, the preferred method to create users for administration is using the `clusterAdmin` option with the `dba.configureInstance()` operation.

For more information, see Creating User Accounts for Administration. If you want to manually configure an administration user, that user requires the following privileges, all with `GRANT OPTION`:

- 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`.

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 administer accounts is using the operations described at Creating User Accounts for Administration

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 give the user *`your_user`* the privileges needed to monitor InnoDB Cluster issue:

```
GRANT SELECT ON mysql_innodb_cluster_metadata.* TO your_user@'%';
GRANT SELECT ON performance_schema.global_status TO your_user@'%';
GRANT SELECT ON performance_schema.global_variables TO your_user@'%';
GRANT SELECT ON performance_schema.replication_applier_configuration TO your_user@'%';
GRANT SELECT ON performance_schema.replication_applier_status TO your_user@'%';
GRANT SELECT ON performance_schema.replication_applier_status_by_coordinator TO your_user@'%';
GRANT SELECT ON performance_schema.replication_applier_status_by_worker TO your_user@'%';
GRANT SELECT ON performance_schema.replication_connection_configuration TO your_user@'%';
GRANT SELECT ON performance_schema.replication_connection_status TO your_user@'%';
GRANT SELECT ON performance_schema.replication_group_member_stats TO your_user@'%';
GRANT SELECT ON performance_schema.replication_group_members TO your_user@'%';
GRANT SELECT ON performance_schema.threads TO your_user@'%' WITH GRANT OPTION;
```

For more information, see Account Management Statements.

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.

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 6.2.2.2, “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.

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.

6.2.11 Known 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.

- If a session type is not specified when creating the global session, MySQL Shell provides automatic protocol detection which attempts to first create a NodeSession and if that fails it tries to create a ClassicSession. With an InnoDB cluster that consists of three server instances, where there is one read-write port and two read-only ports, this can cause MySQL Shell to only connect to one of the read-only

instances. Therefore it is recommended to always specify the session type when creating the global session.

- When adding non-sandbox server instances (instances which you have configured manually rather than using `dba.deploySandboxInstance()`) to a cluster, MySQL Shell is not able to persist any 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 persisted 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 only `a` happens, the instance does not rejoin the cluster after a restart.

  If `b` also happens, and you observe that the instance did not rejoin the cluster after a restart, you cannot use the recommended `dba.rebootClusterFromCompleteOutage()` in this situation to get the cluster back online. This is because 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.

  To avoid this problem it is strongly recommended to use `dba.configureInstance()` before adding instances to a cluster in order to persist the configuration changes.

- The use of the `--defaults-extra-file` option to specify an option file is not supported by InnoDB Cluster server instances. InnoDB Cluster only supports a single option file on instances and no extra option files are supported. Therefore for any operation working with the instance's option file the main one should be specified. If you want to use multiple option files you have to configure the files manually and make sure they are updated correctly considering the precedence rules of the use of multiple option files and ensuring that the desired settings are not incorrectly overwritten by options in an extra unrecognized option file.

- Attempting to use instances with a host name that resolves to an IP address which does not match a real network interface fails with an error that `This instance reports its own address as` *`the hostname`*. This is not supported by the Group Replication communication layer. On Debian based instances this means instances cannot use addresses such as `user@localhost` because localhost resolves to a non-existent IP (such as 127.0.1.1). This impacts on using a sandbox deployment, which usually uses local instances on a single machine.

  A workaround is to configure the `report_host` system variable on each instance to use the actual IP address of your machine. Retrieve the IP of your machine and add `report_host=`*`IP of your machine`* to the `my.cnf` file of each instance. You need to ensure the instances are then restarted to make the change.

- When executing `dba.createCluster()` or adding an instance to an existing InnoDB Cluster by running *`Cluster`*`.addInstance()`, the following errors are logged to MySQL error log:

```
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'
```

  These messages are harmless and relate to the way AdminAPI starts Group Replication.

- When using 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`. Sandbox instances are not designed for production, therefore they are considered transient and are not supported for upgrade.

  A workaround for this issue 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.

- 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. Therefore, replication between one InnoDB Cluster and another is also not supported.

6.3 MySQL InnoDB ReplicaSet

This section documents InnoDB ReplicaSet, added in version 8.0.19.

6.3.1 Introducing InnoDB ReplicaSet

The AdminAPI includes support for InnoDB ReplicaSet, that enables you to administer a set of MySQL instances running asynchronous GTID-based replication in a similar way to InnoDB Cluster. An InnoDB ReplicaSet consists of a 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 having to manually configure it. This makes InnoDB ReplicaSet a quick and easy way to get MySQL replication and MySQL Router up and running, making it well suited to scaling out reads, and provides 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 the replication setup has been adopted, you administer it in the same way as an InnoDB ReplicaSet deployed from scratch. This enables you to take advantage of AdminAPI and MySQL Router without the need to create a new ReplicaSet. For more information see Section 6.3.4, “Adopting an Existing Replication Set Up”.

InnoDB ReplicaSet Limitations

An InnoDB ReplicaSet has several limitations compared to an InnoDB Cluster and thus, 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 have not yet been applied by the time of the halt could become lost.
- No protection against inconsistencies after an unexpected exit or unavailability. If a failover promotes a secondary while the former primary is still available (for example due to a network partition), inconsistencies could be introduced because of the split-brain.
- InnoDB ReplicaSet does not support a multi-primary mode. Data consistency cannot be guaranteed with classic replication topologies that allow writes in all members.
- Read scale-out is limited because 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 scenarios or use-cases, this might have an impact on the source. For example, lots of very small updates going on.

6.3.2 Deploying InnoDB ReplicaSet

You deploy InnoDB ReplicaSet in a similar way to InnoDB Cluster. First you configure some MySQL server instances, the minimum is two instances, see Section 6.1, “MySQL AdminAPI”. One functions as the primary, in this tutorial `rs-1`; the other instance functions as the secondary, in this tutorial `rs-2`; which replicates the transactions applied by the primary. This is the equivalent of the source and replica known from asynchronous MySQL replication. Then you connect to one of the instances using MySQL Shell, and create a ReplicaSet. Once the ReplicaSet has been created, you can add instances to it.

InnoDB ReplicaSet is compatible with sandbox instances, which you can use to deploy locally, for example for testing purposes. See 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.

InnoDB ReplicaSet Prerequisites

To use InnoDB ReplicaSet you should be aware of the following prerequisites:

- Only instances running MySQL version 8.0 and later are supported
- Only GTID-based replication is supported, binary log file position replication is not compatible 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 maximum one primary instance, and 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 that are added to a ReplicaSet, the more monitoring has to be done.
- The ReplicaSet must be entirely managed by MySQL Shell. For example, the replication account is created and managed by MySQL Shell. Making configuration changes to the instance outside of MySQL Shell, for example using SQL statements directly to change the primary, is not supported. Always use MySQL Shell to work with InnoDB ReplicaSet.

AdminAPI and InnoDB ReplicaSet enable you to work with MySQL replication without a deep understanding of the underlying concepts. However, for background information see Replication.

Configuring InnoDB ReplicaSet Instances

Use `dba.configureReplicaSetInstance(instance)` to configure each instance you want to use in your replica set. MySQL Shell can either connect to an instance and then configure it, or you can pass in `instance` to configure a specific remote instance. To use an instance in a ReplicaSet, it must support persisting settings. See 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 Creating User Accounts for Administration. Alternatively, the `dba.configureReplicaSetInstance()` operation can optionally create an administrator account, if the `clusterAdmin` option is provided. The account is created with the correct set of privileges required to manage InnoDB ReplicaSet.

Tip

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 which are not compatible with InnoDB ReplicaSet are configured if possible. The cluster administrator account is created with the privileges required for InnoDB ReplicaSet.

Creating an InnoDB ReplicaSet

Once you have configured your instances, connect to an instance and use `dba.createReplicaSet()` to create a managed ReplicaSet that uses MySQL asynchronous replication, as opposed to MySQL Group Replication used by InnoDB Cluster. The MySQL instance which MySQL Shell is currently 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 you want to check the operation but not actually make any changes to the instances, use the `dryRun` option. This checks and shows what actions the MySQL Shell would take to create the ReplicaSet. 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.add_instance() to add more asynchronously replicated instances to this replicaset
and rs.status() to check its status.
```

To verify that the operation was successful, you work with the returned `ReplicaSet` object. For example this provides the *`ReplicaSet`*`.status()` operation, which displays information about the ReplicaSet. We already assigned the returned `ReplicaSet` 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.

6.3.3 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 synchrony 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. A custom administrator account with the required grants can be created while an instance is configured with `dba.configureReplicaSetInstance()`. See Configuring InnoDB ReplicaSet Instances.

Recovery Methods for InnoDB ReplicaSet

When new instances are added to an InnoDB ReplicaSet they need to be provisioned with the existing data which it contains. This can be done automatically using one of the following methods:

- MySQL Clone, which 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. It does not rely on there being a complete binary log of all transactions applied by the InnoDB ReplicaSet.

Warning

All previous data on the instance being added is destroyed during a clone operation. All MySQL settings not stored in tables are however maintained.

- incremental recovery, which relies on MySQL Replication to apply all missing transactions on the new instance. If the amount of transactions missing on the new instance is small, this can be the fastest method. However, this method is only usable if at least one online instance in the InnoDB ReplicaSet has a complete binary log, containing the entire transaction history of the InnoDB ReplicaSet. This method cannot be used if the binary logs have been purged from all members or if the binary log was only enabled after databases already existed in the instance. If there is a very large amount of transactions to apply, there could be a long 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 6.2.2.2, “Using MySQL Clone with InnoDB Cluster”. This section covers the differences when adding instances to a ReplicaSet.

Adding Instances to a ReplicaSet

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, see Configuring InnoDB ReplicaSet Instances.

For example, to add the instance at `rs-2` with user `rsadmin`, issue:

```
mysql-js> rs.addInstance('rsadmin@rs-2')

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 executed 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):
```

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.

```
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.
```

Assuming the instance is valid for InnoDB ReplicaSet usage, recovery proceeds. In this case the newly joining instance uses MySQL Clone to copy all of the transactions it has not yet applied from the primary, then it joins the ReplicaSet as an online instance. To verify, use the *`ReplicaSet`*`.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.

If you want to override the interactive MySQL Shell mode trying to 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 6.2.2.2, “Using MySQL Clone with InnoDB Cluster”.

6.3.4 Adopting an Existing Replication Set Up

As an alternative to creating a new InnoDB ReplicaSet, you can also 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 Prerequisites, 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, starting from the instance MySQL Shell's global session is connected to. The configuration of all instances is checked during 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. All instances that are part of the topology are automatically added to the ReplicaSet. 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 they could be changed 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.add_instance() 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 from scratch. From this point you must administer the InnoDB ReplicaSet using only AdminAPI.

6.3.5 Working with InnoDB ReplicaSet

You work with an InnoDB ReplicaSet in much the same way as you would work with an InnoDB Cluster. For example as seen in Adding Instances to a ReplicaSet, you assign a `ReplicaSet` object to a variable and call operations that administer the ReplicaSet, such as *`ReplicaSet`*`.addInstance()` to add instances, which is the equivalent of *`Cluster`*`.addInstance()` in InnoDB Cluster. Thus, much of the documentation at Section 6.2.5, “Working with InnoDB Cluster” also applies to InnoDB ReplicaSet. The following operations are supported by `ReplicaSet` objects:

- You get online help for `ReplicaSet` objects, and the AdminAPI, using `\help ReplicaSet` or *`ReplicaSet`*`.help()` and `\help dba` or `dba.help()`. See Section 6.1, “MySQL AdminAPI”.
- You can quickly check the name of a `ReplicaSet` object using either `name` or *`ReplicaSet`*`.getName()`. For example the following are equivalent:

```
mysql-js> rs.name
example
mysql-js> rs.getName()
example
```

- You check information about a ReplicaSet using the *`ReplicaSet`*`.status()` operation, which supports the `extended` option to get different levels of detail. For example:
  - the default for `extended` is 0, 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.
  - 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.
  - 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 very 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 of 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()`.
- You change the instances being used for a ReplicaSet using the *`ReplicaSet`*`.addInstance()` and *`ReplicaSet`*`.removeInstance()` operations. See Adding Instances to a ReplicaSet, and Removing Instances from the InnoDB Cluster.
- Use *`ReplicaSet`*`.rejoinInstance()` to add an instance that was removed back to a ReplicaSet, for example after a failover.
- Use the *`ReplicaSet`*`.setPrimaryInstance()` operation to safely perform a change of the primary of a ReplicaSet to another instance. See Planned Changes of the ReplicaSet Primary.
- Use the *`ReplicaSet`*`.forcePrimaryInstance()` operation to perform a forced failover of the primary. See Forcing the Primary Instance in a ReplicaSet.
- You work with the MySQL Router instances which have been bootstrapped against a ReplicaSet in exactly the same way as with InnoDB Cluster. See Working with a Cluster's Routers for information on *`ReplicaSet`*`.listRouters()` and *`ReplicaSet`*`.removeRouterMetadata()`. For specific information on using MySQL Router with InnoDB ReplicaSet see Using ReplicaSets with MySQL Router.

- From version 8.0.23 InnoDB ReplicaSet supports and enables the parallel replication applier, sometimes referred to as a multi-threaded replica. Using the parallel replication applier with InnoDB ReplicaSet requires that your instances have the correct settings configured. If you are upgrading from an earlier version, instances require an updated configuration. For each instance that belongs to the InnoDB ReplicaSet, update the configuration by issuing `dba.configureReplicaSetInstance(instance)`. Note that usually `dba.configureReplicaSetInstance()` is used before adding the instance to a replica set, but in this special case there is no need to remove the instance and the configuration change is made while it is online. For more information, see Configuring the Parallel Replication Applier.

  InnoDB ReplicaSet instances report information about the parallel replication applier in the output of the *`ReplicaSet`*`.status(extended=1)` operation under the `replication` field.

For more information, see the linked InnoDB Cluster sections.

The following operations are specific to InnoDB ReplicaSet and can only be called against a `ReplicaSet` object:

Planned Changes of the ReplicaSet Primary

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 replica set 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 Forcing the Primary Instance in a ReplicaSet.

During a change of primary, 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, these problematic secondary instances must be either repaired or removed from the ReplicaSet for the fail over to be possible.

Forcing the Primary Instance in a ReplicaSet

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. In the event that you cannot connect to the primary, and you cannot use *`ReplicaSet`*`.setPrimaryInstance()` to safely perform a switchover to a new primary as described at Planned Changes of the ReplicaSet Primary, 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 given (or is null), the most up-to-date instance is automatically selected and promoted to be the new primary. If a target instance is provided, it is promoted to a primary, while other

reachable secondary instances are switched to replicate from the 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 actually 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 it 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 replica set. If at a later date you find a way to recover the instance, it must be removed from the ReplicaSet and re-added as a new instance. If there were any secondary instances that could not be switched to the new primary during the failover, they are also considered invalid.

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 is still able to 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. In many cases, 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.

InnoDB ReplicaSet Locking

From version 8.0.20, AdminAPI uses a locking mechanism to avoid different operations from performing changes on an InnoDB ReplicaSet simultaneously. Previously, different instances of MySQL Shell could connect to an InnoDB ReplicaSet at the same time and execute AdminAPI operations simultaneously. This could lead to inconsistent instance states and errors, for example if *`ReplicaSet`*`.addInstance()` and *`ReplicaSet`*`.setPrimaryInstance()` were executed in parallel.

The InnoDB ReplicaSet operations have the following locking:

- `dba.upgradeMetadata()` and `dba.createReplicaSet()` are globally exclusive operations. This means that if MySQL Shell executes these operations on an InnoDB ReplicaSet, no other operations can be executed against the InnoDB ReplicaSet or any of its instances.
- *`ReplicaSet`*`.forcePrimaryInstance()` and *`ReplicaSet`*`.setPrimaryInstance()` are operations that change the primary. This means that if MySQL Shell executes these operations against an InnoDB ReplicaSet, no other operations which change the primary, or instance change operations can be executed until the first operation completes.
- *`ReplicaSet`*`.addInstance()`, *`ReplicaSet`*`.rejoinInstance()`, and *`ReplicaSet`*`.removeInstance()` are operations that change an instance. This means that if MySQL Shell executes these operations on an instance, the instance is locked for any further instance change operations. However, this lock is only at the instance level and multiple instances in an InnoDB ReplicaSet can each execute one of this type of operation simultaneously. In other words, at most one instance change operation can be executed at a time, per instance in the InnoDB ReplicaSet.
- `dba.getReplicaSet()` and *`ReplicaSet`*`.status()` are InnoDB ReplicaSet read operations and do not require any locking.

In practice, if you try to execute an InnoDB ReplicaSet related operation while another operation that cannot be executed concurrently is still running, you get an error indicating that a lock on a needed resource failed to be acquired. In this case, you should wait for the running operation which holds the lock to complete, and only then try to execute 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, the *`ReplicaSet`*`.addInstance()` operation failed because the lock on the primary instance (`rs1:3306`) could not be acquired, for example because a *`ReplicaSet`*`.setPrimaryInstance()` operation (or other similar operation) was still running.

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 6.2.9, “Tagging the 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 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.

6.4 MySQL Router

This section describes how to integrate MySQL Router with InnoDB Cluster and InnoDB ReplicaSet. For background information on MySQL Router, see MySQL Router 8.0.

6.4.1 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 boostrap 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, 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 due to an unexpected failure of an instance, MySQL Router detects the change and adjusts the routing to the remaining instances automatically. This 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.

Configuring the MySQL Router User

When MySQL Router connects to an InnoDB Cluster or InnoDB ReplicaSet, it requires a user account which 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 a number of 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, issue:

```
mysqlsh> testCluster.setupRouterAccount(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 were 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:

```
mysqlsh> testCluster.setupRouterAccount(myOldRouter, {'update':1})
```

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:

```
shell> mysqlrouter --bootstrap icadmin@ic-1:3306 --user=mysqlrouter
```

You are prompted for the instance password and encryption key for MySQL Router to use. This encryption 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 being 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, this means that the first connection to port 6446 would be redirected to the ic-1 instance, the second connection to port 6446 would be 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:

```
shell> 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.

```
shell> mysqlsh --uri root@localhost:6442
```

To verify which instance you are actually connected to, simply issue an SQL query against the `port` status variable.

```
mysql-js> \sql
Switching to SQL mode... Commands end with ;
mysql-sql> select @@port;
+--------+
| @@port |
+--------+
|   3310 |
+--------+
```

Using ReplicaSets with MySQL Router

You can use MySQL Router 8.0.19 and later to bootstrap against an InnoDB ReplicaSet, see Section 6.4, “MySQL Router”. 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 exactly the same way as with InnoDB Cluster. See Working with a Cluster's Routers for information on `ReplicaSet.listRouters()` and `ReplicaSet.removeRouterMetadata()`.

6.4.2 Using AdminAPI and MySQL Router

This section explains how to use MySQL Router and AdminAPI.

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. Exactly 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 assume there is a single-primary sandbox cluster deployment with three server instances and the instance listening at port 3310 is the current primary. Simulate the instance leaving the cluster unexpectedly:

```
mysql-js> dba.killSandboxInstance(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. Notice that 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 shows that the InnoDB Cluster provided us with automatic failover, that MySQL Router has automatically reconnected us to the new primary instance, and that we have high availability.

Working with a Cluster's Routers

You can bootstrap multiple instances of MySQL Router against a cluster or 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:

```
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"
        }
    }
}
```

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. See Section 6.2.8.2, “Upgrading InnoDB Cluster Metadata”.

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 your 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”, we decide to remove this old instance from the metadata by issuing:

```
mysql-js> cluster.removeRouterMetadata('myRouter1')
```

The MySQL Router instance specified is unregistered from the cluster by removing it from the InnoDB Cluster metadata.

6.5 AdminAPI MySQL Sandboxes

This section explains how to set up a sandbox deployment with AdminAPI. Initially deploying and using local sandbox instances of MySQL is a good way to start your exploration of AdminAPI. You can fully test out the functionality locally, prior to deployment on your production servers. AdminAPI has built-in functionality for creating sandbox instances that are correctly configured to work with InnoDB Cluster 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. See Section 6.2.2, “Deploying a Production InnoDB Cluster” for more information.

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 which you are running MySQL Shell. Therefore, to specify a sandbox instance you supply the port number which the MySQL sandbox instance is listening on.

- Deploying Sandbox Instances
- Managing Sandbox Instances

Deploying Sandbox Instances

The MySQL AdminAPI adds the `dba` global variable to MySQL Shell, which provides functions for administration of sandbox instances. In this example setup, you create three sandbox instances using `dba.deploySandboxInstance(port_number)`. To deploy a new sandbox instance which is bound to port 3310, issue:

```
mysql-js> dba.deploySandboxInstance(3310)
```

The argument passed to `deploySandboxInstance()` is the TCP port number where the MySQL Server instance listens for connections. 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`.

The root user's password for the instance is prompted for.

> **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. For each additional sandbox instance issue:

```
mysql-js> dba.deploySandboxInstance(port_number)
```

To follow this tutorial, use port numbers 3310, 3320 and 3330 for the three sandbox server instances. Issue:

```
mysql-js> dba.deploySandboxInstance(3320)
mysql-js> dba.deploySandboxInstance(3330)
```

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/mysqld` issue:

```
PATH=/home/user/mysql-latest/bin/mysqld:$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.

Managing Sandbox Instances

Once a sandbox instance is running, it is possible to change its status at any time using the following:

- To stop a sandbox instance use `dba.stopSandboxInstance(instance)`. This stops the instance gracefully, unlike `dba.killSandboxInstance(instance)`.
- To start a sandbox instance use `dba.startSandboxInstance(instance)`.
- To kill a sandbox instance use `dba.killSandboxInstance(instance)`. This stops the instance without gracefully stopping it and is useful in simulating unexpected halts.
- To delete a sandbox instance use `dba.deleteSandboxInstance(instance)`. This completely removes the sandbox instance from your file system.

Chapter 7 Extending MySQL Shell

Table of Contents

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 7.1, “Reporting with MySQL Shell”.
- Extension objects are available from MySQL Shell 8.0.17. See Section 7.2, “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 7.3, “MySQL Shell Plugins”.

7.1 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 7.1.1, “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 7.3, “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 7.1.5, “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 7.1.6, “Built-in MySQL Shell Reports”.

7.1.1 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.

7.1.2 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-*”).

7.1.3 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 7.3, “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`.

7.1.4 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 functi
        'options': [
            {
                'name': 'limit',
                'brief': 'The maximum number of rows to return.',
                'shortcut': 'l',
                'type': 'integer'
            }
        ],
        'argc': '0'
    }
)
```

7.1.5 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%'
```

Note that 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});
```

7.1.6 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 7.1.5, “Running MySQL Shell Reports” explains how to run reports in each of these ways.

7.1.6.1 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 7.1.5, “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%'
```

7.1.6.2 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 7.1.5, “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
```

7.1.6.3 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 7.1.5, “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
```

7.2 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.

7.2.1 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.

7.2.2 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 7.1, “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 7.1, “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 7.1 Supported data type pairs for extension objects

JavaScript	Python
Boolean	Boolean
String	String
Integer	Long
Number	Float
Null	None
Array	List
Map	Dictionary

An extension object is literally the same object in both languages.

7.2.3 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 7.3, “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`.

7.2.4 Example MySQL Shell Extension Objects

Example 7.1 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 7.2 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 returne
                         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, default is
                              },
                              {
                                name: "excludeCollections",
                                type: "bool",
                                brief: "If set to true, the collections will not be included on the list, defa
                              }
                            ]
                          },
                        ]
                      });


// 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.addExtensionObjectMember()."
```

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})
```

7.3 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.

7.3.1 Creating MySQL Shell Plugins

MySQL Shell plugins can be used to contain functions that are registered as MySQL Shell reports (see Section 7.1, “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 7.2, “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`.

7.3.1.1 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
```

7.3.2 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.

7.3.3 Example MySQL Shell Plugins

Example 7.3 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.sql(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.sql("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
                                                                          "parameters": [
                                                                            {
                                                                              "name":"session",
                                                                              "type":"object",
                                                                              "class":"Session",
                                                                              "brief": "The session to be used on the
                                                                            },
                                                                            {
                                                                              "name":"id",
                                                                              "type":"integer",
                                                                              "brief": "The ID of the process to be ki
                                                                            }
                                                                            ]
                                                                          })
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 INF
| 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 INFORMA
| 4  | event_scheduler | localhost       | Daemon  | NULL
+----+-----------------+-----------------+---------+-----------------------------------------------------------
```

Chapter 8 MySQL Shell Utilities

Table of Contents

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 8.1, “Upgrade Checker Utility”.
`importJSON()`	A JSON import utility that enables you to import JSON documents to a MySQL Server collection or table. See Section 8.2, “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 8.3, “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 8.4, “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 8.5, “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 8.6, “Dump Loading Utility”.

8.1 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. From MySQL Shell 8.0.13, 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.

You can use the upgrade checker utility to check MySQL 5.7 server instances for compatibility errors and issues for upgrading. From MySQL Shell 8.0.13, you can also use it to check MySQL 8.0 server instances at another GA status release within the MySQL 8.0 release series. 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.

From MySQL Shell 8.0.16, 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 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 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.

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. In MySQL Shell 8.0.22, you can specify release 8.0.11 (the first MySQL Server 8.0 GA release), 8.0.12, 8.0.13, 8.0.14, 8.0.15, 8.0.16, 8.0.17, 8.0.18, 8.0.19, 8.0.20, 8.0.21, or 8.0.22. 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 current MySQL Shell release number.
`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 for 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 8.0.22
...
Server version:               5.7.25-log MySQL Community Server (GPL)
...
mysqlsh> util.checkForServerUpgrade()
```

The following command checks the MySQL server at URI `user@example.com:3306` for upgrade to the first MySQL Server 8.0 GA status release (8.0.11). 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.
```

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"
```

From MySQL 8.0.13, 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 Interface”. The following example checks a MySQL server for upgrade to release 8.0.21, and returns JSON output:

```
mysqlsh -- util checkForServerUpgrade user@localhost:3306 --target-version=8.0.21 --output-format=JSON --c
```

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
```

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()` no longer returns 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
5.7.25-enterprise-commercial-advanced - MySQL Enterprise Server - Advanced Edition (Commercial),
will now be checked for compatibility issues for upgrade to MySQL 8.0.22...

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 errors.
  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://docs.oracle.com/cd/E17952_01/mysql-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://docs.oracle.com/cd/E17952_01/mysql-8.0-en/upgrading-from-previous-series.html#upgrade-configuratio

  dbtest.part1_hash - MyISAM engine does not support native partitioning

6) Foreign key constraint names longer than 64 characters
  No issues found

7) Usage of obsolete MAXDB sql_mode flag
  No issues found

8) Usage of obsolete sql_mode flags
  No issues found

9) ENUM/SET column definitions containing elements longer than 255 characters
  No issues found

10) Usage of partitioned tables in shared tablespaces
  Error: The following tables have partitions in shared tablespaces. Before upgrading to 8.0 they need
  to be moved to file-per-table tablespace. You can do this by running query like
  'ALTER TABLE table_name REORGANIZE PARTITION X INTO
    (PARTITION X VALUES LESS THAN (30) TABLESPACE=innodb_file_per_table);'
  More information: https://docs.oracle.com/cd/E17952_01/mysql-8.0-en/mysql-nutshell.html#mysql-nutshell-remov

  dbtest.table1 - Partition p0 is in shared tablespace tbsp4
  dbtest.table1 - Partition p1 is in shared tablespace tbsp4

11) Circular directory references in tablespace data file paths
  No issues found
```

```
12) Usage of removed functions
  Error: Following DB objects make use of functions that have been removed in
    version 8.0. Please make sure to update them to use supported alternatives
    before upgrade.
  More information:
    https://docs.oracle.com/cd/E17952_01/mysql-8.0-en/mysql-nutshell.html#mysql-nutshell-removals

  dbtest.view1 - VIEW uses removed function PASSWORD

13) Usage of removed GROUP BY ASC/DESC syntax
  Error: The following DB objects use removed GROUP BY ASC/DESC syntax. They need to be altered so that
  ASC/DESC keyword is removed from GROUP BY clause and placed in appropriate ORDER BY clause.
  More information: https://docs.oracle.com/cd/E17952_01/mysql-8.0-relnotes-en/news-8-0-13.html#mysqld-8-0

  dbtest.view1 - VIEW uses removed GROUP BY DESC syntax
  dbtest.func1 - FUNCTION uses removed GROUP BY ASC syntax

14) Removed system variables for error logging to the system log configuration
  No issues found

15) Removed system variables
  Error: Following system variables that were detected as being used will be
    removed. Please update your system to not rely on them before the upgrade.
  More information: https://docs.oracle.com/cd/E17952_01/mysql-8.0-en/added-deprecated-removed.html#optvars

  log_builtin_as_identified_by_password - is set and will be removed
  show_compatibility_56 - is set and will be removed

16) System variables with new default values
  Warning: Following system variables that are not defined in your
    configuration file will have new default values. Please review if you rely on
    their current values and if so define them before performing upgrade.
  More information: https://mysqlserverteam.com/new-defaults-in-mysql-8-0/

  back_log - default value will change
  character_set_server - default value will change from latin1 to utf8mb4
  collation_server - default value will change from latin1_swedish_ci to
    utf8mb4_0900_ai_ci
  event_scheduler - default value will change from OFF to ON
[...]

17) Zero Date, Datetime, and Timestamp values
  Warning: By default zero date/datetime/timestamp values are no longer allowed
    in MySQL, as of 5.7.8 NO_ZERO_IN_DATE and NO_ZERO_DATE are included in
    SQL_MODE by default. These modes should be used with strict mode as they will
    be merged with strict mode in a future release. If you do not include these
    modes in your SQL_MODE setting, you are able to insert
    date/datetime/timestamp values that contain zeros. It is strongly advised to
    replace zero values with valid ones, as they may not work correctly in the
    future.
  More information:
    https://lefred.be/content/mysql-8-0-and-wrong-dates/

  global.sql_mode - does not contain either NO_ZERO_DATE or NO_ZERO_IN_DATE
    which allows insertion of zero dates
  session.sql_mode -  of 2 session(s) does not contain either NO_ZERO_DATE or
    NO_ZERO_IN_DATE which allows insertion of zero dates
  dbtest.date1.d - column has zero default value: 0000-00-00

18) Schema inconsistencies resulting from file removal or corruption
  No issues found

19) Tables recognized by InnoDB that belong to a different engine
  No issues found

20) Issues reported by 'check table x for upgrade' command
  No issues found
```

```
21) New default authentication plugin considerations
  Warning: The new default authentication plugin 'caching_sha2_password' offers
    more secure password hashing than previously used 'mysql_native_password'
    (and consequent improved client connection authentication). However, it also
    has compatibility implications that may affect existing MySQL installations.
    If your MySQL installation must serve pre-8.0 clients and you encounter
    compatibility issues after upgrading, the simplest way to address those
    issues is to reconfigure the server to revert to the previous default
    authentication plugin (mysql_native_password). For example, use these lines
    in the server option file:

    [mysqld]
    default_authentication_plugin=mysql_native_password

    However, the setting should be viewed as temporary, not as a long term or
    permanent solution, because it causes new accounts created with the setting
    in effect to forego the improved authentication security.
    If you are using replication please take time to understand how the
    authentication plugin changes may impact you.
  More information:
    https://docs.oracle.com/cd/E17952_01/mysql-8.0-en/upgrading-from-previous-series.html#upgrade-caching-sha2
    https://docs.oracle.com/cd/E17952_01/mysql-8.0-en/upgrading-from-previous-series.html#upgrade-caching-sha2

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.
- As shown in this example, the upgrade checker utility might also provide 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 for 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.

8.2 JSON Import Utility

MySQL Shell's JSON import utility `util.importJSON()`, introduced in MySQL Shell 8.0.13, 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.

From MySQL Shell 8.0.14, the 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.

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`.

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.

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 8.2.3, “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: "jsondata",
```

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: "jsondata",
```

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.

8.2.1 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 Interface”. 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 8.2.3, “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
```

8.2.2 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 8.2.3, “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.

8.2.3 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
```

8.3 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 8.4, “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.

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.

When choosing a destination for the table export file, note that for import into a MySQL 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 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 using the copy utility of your choice, depending on the operating system you chose for your Compute instance.

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.

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.

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, `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` is a dictionary of options that can be omitted if it is empty. The following options are available for the table export utility:

`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 8.1 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	\

dialect	linesTerminatedBy	fieldsTerminatedBy	fieldsEnclosedBy	fieldsOptionallyEnclosed	fieldsEscapedBy
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.
`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.
`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.

8.4 Parallel Table Import Utility

MySQL Shell's parallel table import utility `util.importTable()`, introduced in MySQL Shell 8.0.17, 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.

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 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.

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.

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.

The data file or files to be imported must be in one 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.

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.

Function

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)
```

`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.
- 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.

`options` is a dictionary of import options that can be omitted if it is empty. The options are listed after the examples.

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.

Examples

The following examples, the first in 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-unix", s
```

```
mysql-py> util.import_table("/tmp/productrange.csv", {"schema": "mydb", "table": "products", "dialect": "csv-u
```

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 interface to invoke the parallel table import utility, 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. 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 this interface, see Section 5.8, “API Command Line Interface”.

Options

The following import options are available for the parallel table import utility to specify where and 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 of data 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 8.2 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.
`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.
`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.

8.5 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 Database Service DB System (a MySQL DB System, for short) or a MySQL Server instance using MySQL Shell's Section 8.6, “Dump Loading Utility” `util.loadDump()`.

MySQL Shell's instance dump utility, schema dump utility, and table dump utility provide Oracle Cloud Infrastructure Object Storage streaming, MySQL Database 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 Database 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 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 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 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.

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 `information_schema`, `mysql`, `ndbinfo`, `performance_schema`, and `sys` schemas are always excluded from an instance dump. 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. You can also choose to include or exclude users and their roles and grants, events, routines, and triggers.

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.

From MySQL Shell 8.0.22, when you export instances or schemas to an Oracle Cloud Infrastructure Object Storage bucket, during the dump you can generate a pre-authenticated request URL for every item. The user account that runs MySQL Shell's dump loading utility `util.loadDump()` uses these to load the dump files without additional access permissions. By default, if the `ocimds` option is set to `true` and an Object Storage bucket name is supplied using the `osBucketName` option, MySQL Shell's instance dump utility and schema dump utility generate pre-authenticated request URLs for the dump files and list them in a single manifest file. The dump loading utility references the manifest file to obtain the URLs and load the dump files. For instructions to generate or deactivate pre-authenticated request URLs, see the description for the `ociParManifest` option.

The following requirements apply to dumps using the instance dump utility, schema dump utility, and table dump utility:

- MySQL 5.7 or later is required for both the source MySQL instance and the destination MySQL instance.
- 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: `BACKUP_ADMIN`, `EVENT`, `RELOAD`, `SELECT`, `SHOW VIEW`, and `TRIGGER`. If the `consistent` option is set to `false`, the `BACKUP_ADMIN` and `RELOAD` privileges are not required. 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.
- 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.
- For import into a MySQL DB System, set the `ocimds` option to `true`, to ensure compatibility with MySQL Database Service.
- For compatibility with MySQL Database 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 Database 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.
- A number of other security related restrictions and requirements apply to items such as tablespaces and privileges for compatibility with MySQL Database 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.

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])
```

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.

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 Database 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 Database Service 8.0.21
...
Compatibility issues with MySQL Database 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.

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")
```

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 in MySQL Shell's Python mode, 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-py> util.dump_schemas(["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` is a dictionary of options that can be omitted if it is empty. The following options are available for the instance dump utility, the schema dump utility, and the table dump utility, unless otherwise indicated:

`dryRun: [ true | false ]` Display information about what would be dumped with the specified set of options, and about the results of MySQL Database 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`.
`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.
`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.
`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.
`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`).
`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. If a named schema does not exist or is excluded anyway, the utility ignores the item.
`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. 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. Tables named by the `excludeTables` option do not have DDL files or data files in the dump. If a named table does not exist in the schema or the schema is not included in the dump, the utility ignores the item.

`all: [ true | false ]` (Table dump utility only) Setting this option to `true` includes all views and tables from the specified schema in the dump. When you use this option, set the `tables` parameter to an empty array. 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

In MySQL Shell 8.0.21, attempting to import users to a MySQL 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 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 (which is equivalent to `"'user_name'@'%'"`). If a named user account does not exist, the utility ignores the item.

`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, in which case a user account matched by both an `includeUsers` string and an `excludeUsers` string is excluded.

`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`.

`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`.
`triggers: [ true | false ]`	Include (`true`) or exclude (`false`) triggers for each table in the dump. The default is `true`.
`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.
`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.
`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. 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 and the global read lock is released.
`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`.
`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. In order 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. If you set the chunking option to `false`, chunking does not take place and the utility creates one data file for each table.
`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.

`ocimds: [ true | false ]` Setting this option to `true` enables checks and modifications for compatibility with MySQL Database 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 DICTIONARY`, `INDEX DICTIONARY`, 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, 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 pre-authenticated requests are generated for every item in the dump, and the dump files can only be accessed using these request URLs.

`compatibility: array of strings` Apply the specified requirements for compatibility with MySQL Database 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 a comma-separated list:

- `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 Database Service.
- `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 Database

		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 Database 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_role_admin`	Remove the `ROLE_ADMIN` privilege from `GRANT` statements. This privilege can be restricted by MySQL Database Service.
	`strip_tablespaces`	Remove the `TABLESPACE` clause from `GRANT` statements, so all tables are created in their default tablespaces. MySQL Database Service has some restrictions on tablespaces.
`ociParManifest: [ true | false ]`	Setting this option to `true` generates a pre-authenticated request for read access (an Object Read PAR) for every item in the dump, and a manifest file listing all the pre-authenticated request URLs. The pre-authenticated requests expire after a week by default, which you can change using the `ociParExpireTime` option. This option is available from MySQL Shell 8.0.22, 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 utilities, and in MySQL Shell 8.0.22, it is only available for the instance dump utility and schema dump utility. 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. 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 pre-authenticated requests. 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 pre-authenticated request URL for any object, the associated file is deleted and the dump is stopped.

To enable the resulting dump files to be loaded, create a pre-authenticated read request for the manifest file object (`@.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 pre-authenticated read request using any user account that has the required permissions. The pre-authenticated request 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 pre-authenticated request URL gives anyone who has the URL access to the targets identified in the request. Carefully manage the distribution of the pre-authenticated URL you create for the manifest file, and of the pre-authenticated URLs for exported items in the manifest file.

`ociParExpireTime: "`*`string`*`"`

The expiry time for the pre-authenticated request URLs 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. From MySQL Shell 8.0.23, this option is available for all the utilities, and in MySQL Shell 8.0.22, it is only available for the instance dump utility and schema dump utility.

The expiry time must be formatted as an RFC 3339 timestamp, as required by Oracle Cloud Infrastructure when creating a pre-authenticated request. 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 pre-authenticated request creation to fail for the first file in the dump, which stops the dump.

8.6 Dump Loading Utility

MySQL Shell's dump loading utility `util.loadDump()`, introduced in MySQL Shell 8.0.21, supports the import into a MySQL Database Service DB System (a MySQL DB System, for short) or a MySQL Server

instance of schemas or tables dumped using MySQL Shell's Section 8.5, “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.

For import into a MySQL 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 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 Database Service. MySQL Shell's table dump utility does not use this option.

Note

1. 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 DB System configuration.
2. 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 DB System configuration.
3. 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, from MySQL Shell 8.0.22, use the `updateGtidSet` option to either append them to or replace the `gtid_purged` GTID set, depending on the release of the target MySQL instance. This is not currently supported on MySQL DB System due to a permissions restriction. In MySQL Shell 8.0.21, the GTID set can be imported manually, though this is not supported on MySQL DB System.

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 load 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.

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 shell session is used as the target schema, or the `schema` option can be used to name the schema.

You can customize the import with further options in the dump loading utility:

- You can select individual tables or schemas to import or to exclude from the import.

- Users and their roles and grants are excluded by default, but you can choose to import them.
- You can specify a different character set for the data in the target MySQL instance to that used in the dump files.
- You can update the `ANALYZE TABLE` histograms, even after the data has already been loaded.
- You can choose to skip binary logging on the target MySQL instance during the course of the import using a `SET sql_log_bin=0` statement.

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.

To load dump files with pre-authenticated requests from an Object Storage bucket into a MySQL DB System, you need the pre-authenticated read request URL that has been created for the manifest file object (`@.manifest.json`). Also create a pre-authenticated read-write request for a text file in the same prefixed location as the dump files in the Object Storage bucket. This will be the dump loading utility's progress state file, which is required when you are loading dump files with pre-authenticated requests. You can use any user account with the required permissions to create the request. The text file can have any name, and you can create the file or let the utility create it. The content of the file will be in JSON format, so a `.json` file extension is appropriate if you are using one (for example, `progress.json`).

As an alternative to storing the progress state file with the dump files, which is the default, you can use a local file in the location where you run the dump loading utility. If you do not have permissions to create a pre-authenticated read-write request for the progress state file, this method enables you to store progress. When you use a local file, note that the dump cannot be resumed by running the dump loading utility from an alternative location.

With pre-authenticated requests, when you run the dump loading utility, specify the dump's URL as the pre-authenticated request URL for the `@.manifest.json` file. Also specify the progress state file (`progressFile` option) as the pre-authenticated request URL for the file in the Object Storage bucket, or as the file on the local system if you chose that option. The user account that runs the dump loading utility can then load the dump files using the URLs in the manifest file without additional access permissions. If 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.

The DDL files for a dump are loaded by a single thread, but the data is loaded in parallel by the number of threads you select, 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. 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.

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])
```

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 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` is a dictionary of options that can be omitted if it is empty. The following options are available:

`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`.

`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.

`threads:` *`int`* 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.

`progressFile: "`*`string`*`"` A local file location for the dump loading utility's progress state file, which persists progress state for an import. By default, the progress

	state file is named `load-progress.server_uuid.json` and created in the dump directory, but you can change that using this option. Setting `progressFile` to an empty string disables progress state tracking, which means that the dump loading utility cannot resume a partially completed import.
`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.
`waitDumpTimeout: int`	Setting this option activates concurrent loading by specifying 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.
`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.
`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.

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 Database Service instance.

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.

`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 DB System due to a permissions restriction. From MySQL 8.0.23, the option can also be used for a MySQL 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 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 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 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.

`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 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.

`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`.

`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.
`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.
`schema: "`*`string`*`"`	The existing target schema into which a dump produced by MySQL Shell's table dump utility must be loaded. 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.
`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. If a named schema does not exist in the dump files, the utility ignores the item.
`includeSchemas: `*`array of strings`*	Load only the named schemas from the dump files. You can specify both options, in which case a schema name matched by both an `includeSchemas` string and an `excludeSchemas` string is excluded.
`excludeTables: `*`array of strings`*	Exclude the named tables from the import. 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. Tables named by the `excludeTables` option are not uploaded to the

target MySQL instance. If a named table does not exist in the schema or the schema does not exist in the dump files, the dump loading utility ignores the item.

`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, in which case a table name matched by both an `includeTables` string and an `excludeTables` string is excluded.

`loadDdl: [ true | false ]`

Setting this option to `true` imports only the DDL files from the dump, and does not import the data. The default is `false`.

`loadData: [ true | false ]`

Setting this option to `true` imports only the data files from the dump, and does not import the DDL files. The default is `false`.

`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 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.

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 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 (which is equivalent to `"'user_name'@'%'"`). If a named user account does not exist in the dump files, the utility ignores the item.

`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, in which case a user account matched by both an `includeUsers` string and an `excludeUsers` string is excluded.

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` option, to load the DDL file 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"]});
```

Chapter 9 MySQL Shell Logging and Debug

Table of Contents

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 9.1 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

By default, MySQL Shell does not log or output SQL statements that are executed 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.

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 9.1, “Application Log”.

For instructions to send logging information to the console as verbose output, see Section 9.2, “Verbose Output”.

For instructions to activate logging for SQL statements that are executed by AdminAPI operations, see Section 9.3, “Logging AdminAPI Operations”.

9.1 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. 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 10.4, “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 10.4, “Configuring MySQL Shell Options”.

The available logging levels are as listed in Table 9.1, “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:

```
shell> mysqlsh --log-level=4
shell> 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:

```
shell> mysqlsh --log-level=@8
shell> 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.sql("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.

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.

9.2 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 10.4, "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 10.4, "Configuring MySQL Shell Options".

The available settings are as listed in Table 9.1, "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.

9.3 Logging AdminAPI Operations

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 10.4, “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 10.4, “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 10 Customizing MySQL Shell

Table of Contents

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 10.1, “Working With Startup Scripts”.
- Add non-standard module search paths for JavaScript or Python mode. See Section 10.2, “Adding Module Search Paths”.
- Customize the MySQL Shell prompt. See Section 10.3, “Customizing the Prompt”.
- Set configuration options to change MySQL Shell's behavior for the current session or permanently. See Section 10.4, “Configuring MySQL Shell Options”.

10.1 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]`

10.2 Adding Module Search Paths

When you use the `require()` function in JavaScript or the `import` function in Python, the well-known 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 well-known module search paths specified by the `sys.path` variable.

You can add further well-known 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 10.2.1, “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 10.2.2, “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.

10.2.1 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 well-known 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.

10.2.2 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 10.1, “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.

10.3 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 theme file to the `~/.mysqlsh/prompt.json` directory on Linux and Mac, or the `%AppData%\MySQL\mysqlsh\prompt.json` directory on Windows.

The user configuration path for the directory 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.

The format of the prompt theme file is described in the `README.prompt` file, and some sample prompt theme files are included. 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.

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`.

10.4 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.nameCache`	true	boolean	Enable database name caching for autocompletion.

optionName	DefaultValue	Type	Effect
batchContinueOnError	false	boolean (READ ONLY)	In SQL batch mode, force processing to continue if an error is found.
credentialStore.excludeFilters	empty array	array	An array of URLs for which automatic password storage is disabled, supports glob characters * and ?.
credentialStore.helper	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.savePasswords	false	string	Controls automatic password storage, supported values: always, prompt or never.
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 6.2.5, “Working with InnoDB Cluster”).
dba.logSql	0	integer ranging from 0 to 2	Log SQL statements that are executed by AdminAPI operations (see Chapter 9, *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 6.2.2.2, “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 4.3.4, “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	*IDENTIFIED* : *PASSWORD*	string	Strings that match these patterns are not added to the MySQL Shell code history.
interactive	true	boolean (READ ONLY)	Enable interactive mode.
logLevel	Requires a value	integer ranging from 1 to 8	Set a logging level for the application log (see Chapter 9, *MySQL Shell Logging and Debug*).

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.
useWizards	true	boolean	Enable wizard mode.
verbose	1	integer ranging from 0 to 4	Enable verbose output to the console and set a level of detail (see Chapter 9, *MySQL 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.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

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 (part of MySQL 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, see Chapter 6, *Using 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
--	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
--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)	
--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
--dbpassword	Password to use when connecting to server	
--dbuser	MySQL user name to use when connecting to server	
--execute	Execute the command and quit	
--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

Option Name	Description	Introduced
--interactive	Emulate Interactive mode in batch mode	
--js, --javascript	Start in JavaScript mode	
--json	Print output in JSON format	
--log-level	Specify logging level	
-ma	Detect transport protocol for session automatically	8.0.3
--mysql, -mc	Create a session using classic MySQL protocol	8.0.3
--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 (alias for --dbpassword)	
--passwords-from-stdin	Read the password from stdin	
--port	TCP/IP port number for connection	
--py, --python	Start in Python mode	
--quiet-start	Start without printing introductory information	
--recreate-schema	Drop and recreate schema	
--redirect-primary	Ensure connection to an InnoDB cluster's primary	8.0.4
--redirect-secondary	Ensure connection to an InnoDB cluster's secondary	
--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
--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	Name of the SSL cipher to use	

Option Name	Description	Introduced
--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	
--tabbed	Display output in tab separated format	
--table	Display output in table format	
--tls-version	Permissible TLS protocol for encrypted connections	
--uri	Session information in URI format	
--user	MySQL user name to use when connecting to server (alias for --dbuser)	
--verbose	Activate verbose output to the console	8.0.17
--version	Display version information and exit	
--vertical	Display all SQL results vertically	

- `--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 Interface” 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 wih 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.

- `--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 4.3.4, “Using Compressed Connections” for information on using MySQL Shell's compression control in all releases.

- `--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 9, *MySQL Shell Logging and Debug* for more information.

- `--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.

- `--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 8.2, “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-level=`*`N`*

  Change the logging level for the MySQL Shell application log file, 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 9, *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 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.

- `--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 password-less 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 Interface”.

- `--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 password-less 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.

- `--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.

- `--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 Reference 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 Reference 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 Reference for more information.

- `--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`.

- `--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 9, *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.