MySQL Shell 8.0 (part of MySQL 8.0)

Abstract

MySQL Shell is an advanced client and code editor for MySQL Server. 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 InnoDB Cluster.

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: 2020-07-15 (revision: 66681)

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 3.4.3. 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.

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 X DevAPI enables you 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.
- The AdminAPI enables you to work with InnoDB cluster, which provides an integrated solution for high availability and scalability using InnoDB based MySQL databases, without requiring advanced MySQL expertise. See InnoDB Cluster.

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 6, *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 7, *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 8, *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 Server.
`\reconnect`		Reconnect to the same MySQL Server.
`\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.
`\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.

Command	Alias/Shortcut	Description
`\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 and Reconnect 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.

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 a 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 9.4, “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 6.1.5, “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 se
```

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

4.2.1 Creating the Session Global Object While Starting MySQL Shell

When you start MySQL Shell from the command line, you can specify connection parameters using separate command options for each value, such as the user name, host, and port. For instructions

and examples to start MySQL Shell and connect to a MySQL Server instance in this way, see Section 4.3.1, “Connecting using Individual Parameters”. When you use this connection method, you can add one of these options to choose the type of session object to create at startup to be the `session` global object:

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

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

```
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&compress
  ```

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

- If you are using key-value pairs to specify connection parameters, either with MySQL Shell's `\connect` command or with the `shell.connect()`, `shell.openSession()`,

`mysqlx.getSession()`, `mysql.getSession()`, or `mysql.getClassicSession()` function, use the `compression` parameter in the dictionary of options to specify whether compression is required, preferred, or disabled. For example:

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

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

4.3.4.2 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 9.4, “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 InnoDB Cluster.
- `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 9.4, “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 6.1, “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 7, *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 6.2, “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 9.3, “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,la
```

5.5 Code History

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

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

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

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

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

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

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

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

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

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
```

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 a 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 a 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 9.4, “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@l
{
    "hasData": true,
    "rows": [
        {
            "ID": 1523,
            "Name": "Wien",
            "CountryCode": "AUT",
            "District": "Wien",
            "Info": {
                "Population": 1608144
            }
        },
        {
            "ID": 1524,
            "Name": "Graz",
            "CountryCode": "AUT",
            "District": "Steiermark",
            "Info": {
                "Population": 240967
            }
        },
        {
            "ID": 1525,
            "Name": "Linz",
            "CountryCode": "AUT",
            "District": "North Austria",
            "Info": {
                "Population": 188022
            }
        },
        {
            "ID": 1526,
            "Name": "Salzburg",
```

```
            "CountryCode": "AUT",
            "District": "Salzburg",
            "Info": {
                "Population": 144247
            }
        },
        {
            "ID": 1527,
            "Name": "Innsbruck",
            "CountryCode": "AUT",
            "District": "Tiroli",
            "Info": {
                "Population": 111752
            }
        },
        {
            "ID": 1528,
            "Name": "Klagenfurt",
            "CountryCode": "AUT",
            "District": "Kärnten",
            "Info": {
                "Population": 91141
            }
        }
    ],
    "executionTime": "0.0067 sec",
    "affectedRowCount": 0,
    "affectedItemsCount": 0,
    "warningCount": 0,
    "warningsCount": 0,
    "warnings": [],
    "info": "",
    "autoIncrementValue": 0
}
```


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

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

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 InnoDB Cluster.
- `shell` - global provides access to MySQL Shell functions, such as `shell.options` for configuring MySQL Shell options (see Section 9.4, “Configuring MySQL Shell Options”), and `shell.reports` for running MySQL Shell reports (see Section 6.1, “Reporting with MySQL Shell”).
- `util` - provides access to MySQL Shell utilities. See Chapter 7, *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='p
```

This maps to the equivalent command in MySQL Shell:

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

- 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 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 6.1, “Reporting with MySQL Shell”.
- Extension objects are available from MySQL Shell 8.0.17. See Section 6.2, “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 6.3, “MySQL Shell Plugins”.

6.1 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 6.1.1, “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 6.3, “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 6.1.5, “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 6.1.6, “Built-in MySQL Shell Reports”.

6.1.1 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.

6.1.2 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 case insensitive 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-*”).

6.1.3 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 6.3, “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`.

6.1.4 Example MySQL Shell Report

This example user-defined report `sessions` shows which sessions currently exist.

```
def sessions(session, args, options):
    sys = session.get_schema('sys')
    session_view = sys.get_table('session')
    query = session_view.select(
        'thd_id', 'conn_id', 'user', 'db', 'current_statement',
        'statement_latency AS latency', 'current_memory AS memory')
    if (options.has_key('limit')):
        limit = int(options['limit'])
        query.limit(limit)

    result = query.execute()
    report = [result.get_column_names()]
    for row in result.fetch_all():
        report.append(list(row))
```

```
    return {'report': report}

shell.register_report(
    'sessions',
    'list',
    sessions,
    {
        'brief': 'Shows which sessions exist.',
        'details': ['You need the SELECT privilege on sys.session view and the underlying tables and fu
        'options': [
            {
                'name': 'limit',
                'brief': 'The maximum number of rows to return.',
                'shortcut': 'l',
                'type': 'integer'
            }
        ],
        'argc': '0'
    }
)
```

6.1.5 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});
```

6.1.6 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 6.1.5, “Running MySQL Shell Reports” explains how to run reports in each of these ways.

6.1.6.1 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 6.1.5, “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%'
```

6.1.6.2 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 6.1.5, “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
```

6.1.6.3 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 6.1.5, “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
```

6.2 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.

6.2.1 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.

6.2.2 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 6.1, “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 6.1, “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 6.1 Supported data type pairs for extension objects

JavaScript	Python
Boolean	Boolean
String	String

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

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

6.2.3 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 6.3, “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`.

6.2.4 Example MySQL Shell Extension Objects

Example 6.1 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 6.2 Creating and Registering Extension Objects - JavaScript

This example creates an extension object with the function `listTables()` as a member, and registers it directly as the MySQL Shell global object `tools`:

```
// Define a listTables function that will be exposed by the global object tools

function listTables(session, schemaName, options) {
...
}

// Create an extension object and add the listTables function to it as a member

var object = shell.createExtensionObject()

shell.addExtensionObjectMember(object, "listTables", listTables,

                      {
                        brief:"Retrieves the tables from a given schema.",
                        details: ["Retrieves the tables of the schema named schemaName.",
                                  "If excludeCollections is true, the collection tables will not be ret
                        parameters:
                        [
                          {
                            name: "session",
                            type: "object",
                            class: "Session",
                            brief: "An X Protocol session object."
                          },
                          {
                            name: "schemaName",
                            type: "string",
                            brief: "The name of the schema from which the table list will be pulled."
                          },
                          {
                            name: "options",
                            type: "dictionary",
                            brief: "Additional options that affect the function behavior.",
                            options: [
                              {
                                name: "excludeViews",
                                type: "bool",
                                brief: "If set to true, the views will not be included on the list, def
                              },
                              {
                                name: "excludeCollections",
                                type: "bool",
                                brief: "If set to true, the collections will not be included on the lis
                              }
                            ]
                          },
                        ]
                      });


// Register the extension object as the global object "tools"

shell.registerGlobal("tools", object, {brief:"Global object for ExampleCom administrator tools",
                     details:[
                        "Global object to access homegrown ExampleCom administrator tools.",
                        "Add new tools to this global object as members with shell.addExtensionObjectMem
```

In JavaScript mode, the function is called like this:

```
mysql-js> tools.listTables(session, "world_x", {excludeViews: true})
```

In Python mode, the function is called like this:

```
mysql-py> tools.list_tables(session, "world_x", {"excludeViews": True})
```

6.3 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.

6.3.1 Creating MySQL Shell Plugins

MySQL Shell plugins can be used to contain functions that are registered as MySQL Shell reports (see Section 6.1, “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 6.2, “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`.

6.3.1.1 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
```

6.3.2 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.

6.3.3 Example MySQL Shell Plugins

Example 6.3 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 t
                                                                          "parameters": [
                                                                            {
                                                                              "name":"session",
                                                                              "type":"object",
                                                                              "class":"Session",
                                                                              "brief": "The session to be used on the c
                                                                            },
                                                                            {
                                                                              "name":"id",
                                                                              "type":"integer",
                                                                              "brief": "The ID of the process to be kil
```

```
                                                                    }
                                                                    ]
                                                                })
except Exception as e:
    shell.log("ERROR", "Failed to register ext.process.kill ({0}).".
       format(str(e).rstrip()))
```

Here, the user runs the report `proc` using the MySQL Shell `\show` command, then uses the `ext.process.kill()` function to stop one of the listed processes:

```
mysql-py> \show proc
+----+-----------------+-----------------+---------+-----------------------------------------------
| ID | USER            | HOST            | COMMAND | INFO
+----+-----------------+-----------------+---------+-----------------------------------------------
| 66 | root            | localhost:53998 | Query   | PLUGIN: SELECT ID, USER, HOST, COMMAND, INFO FROM
| 67 | root            | localhost:34022 | Sleep   | NULL
| 4  | event_scheduler | localhost       | Daemon  | NULL
+----+-----------------+-----------------+---------+-----------------------------------------------

mysql-py> ext.process.kill(session, 67)
mysql-py> \show proc
+----+-----------------+-----------------+---------+-----------------------------------------------
| ID | USER            | HOST            | COMMAND | INFO
+----+-----------------+-----------------+---------+-----------------------------------------------
| 66 | root            | localhost:53998 | Query   | PLUGIN: SELECT ID, USER, HOST, COMMAND, INFO FROM
| 4  | event_scheduler | localhost       | Daemon  | NULL
+----+-----------------+-----------------+---------+-----------------------------------------------
```

Chapter 7 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 provides the following functions:

`checkForServerUpgrade()`	An upgrade checker utility that enables you to verify whether MySQL server instances are ready for upgrade. See Section 7.1, “Upgrade Checker Utility”.
`importJSON()`	A JSON import utility that enables you to import JSON documents to a MySQL Server collection or table. See Section 7.2, “JSON Import 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 7.3, “Parallel Table Import Utility”.
`dumpInstance()`, `dumpSchemas()`	An instance dump utility and schema dump utility that can export all schemas or a selected schema from a MySQL instance into an Oracle Cloud Infrastructure Object Storage bucket or a set of local files. See Section 7.4, “Instance Dump Utility and Schema 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 7.5, “Dump Loading Utility”.

7.1 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 8.0.19, 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, or 8.0.19. 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 8.0.19
...
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
```

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":"JS
```

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 8.0.15, and returns JSON output:

```
mysqlsh -- util checkForServerUpgrade user@localhost:3306 --target-version=8.0.15 --output-format=JSON
```

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=
```

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 8.0.19...

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 e
  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 us
  utf8mb4 instead, for improved Unicode support.
  More information: https://dev.mysql.com/doc/refman/8.0/en/charset-unicode-utf8mb3.html

  dbtest.view1.col1 - column's default character set: utf8

4) Table names in the mysql schema conflicting with new tables in 8.0
  No issues found
```

```
5) Partitioned tables using engines with non native partitioning
  Error: In MySQL 8.0 storage engine is responsible for providing its own
  partitioning handler, and the MySQL server no longer provides generic
  partitioning support. InnoDB and NDB are the only storage engines that
  provide a native partitioning handler that is supported in MySQL 8.0. A
  partitioned table using any other storage engine must be altered—either to
  convert it to InnoDB or NDB, or to remove its partitioning—before upgrading
  the server, else it cannot be used afterwards.
  More information:
    https://dev.mysql.com/doc/refman/8.0/en/upgrading-from-previous-series.html#upgrade-configuration-chang

  dbtest.part1_hash - MyISAM engine does not support native partitioning

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://dev.mysql.com/doc/refman/8.0/en/mysql-nutshell.html#mysql-nutshell-removals

  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://dev.mysql.com/doc/refman/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://dev.mysql.com/doc/relnotes/mysql/8.0/en/news-8-0-13.html#mysqld-8-0-13-sql-synt

  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://dev.mysql.com/doc/refman/8.0/en/added-deprecated-removed.html#optvars-removed

  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://dev.mysql.com/doc/refman/8.0/en/upgrading-from-previous-series.html#upgrade-caching-sha2-pa
    https://dev.mysql.com/doc/refman/8.0/en/upgrading-from-previous-series.html#upgrade-caching-sha2-pa

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, Warnin or Notice.
		dbObject	A string identify the databa object to which the messa relates.
		description	If availab a string with a specific descrip of the issue with the databa 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.	

7.2 JSON Import Utility

MySQL Shell's JSON import utility, 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 7.2.3, "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 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 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 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 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.

7.2.1 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 7.2.3, “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
```

7.2.2 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 7.2.3, “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.

7.2.3 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 --ignoreR
```

7.3 Parallel Table Import Utility

MySQL Shell's parallel table import utility, 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, divides 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 invoke the parallel table import utility, you specify the mapping between the fields in the data file and the columns in the MySQL table. You can set field- and line-handling options as for the `LOAD DATA` command to handle data files in arbitrary formats. 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.

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. 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 parallel table import utility uses `LOAD DATA LOCAL INFILE` statements to upload data chunks from the input file, so the data file to be imported must be in a location that is accessible to the client host as a local disk. 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.

In the MySQL Shell API, the parallel table import utility is a function of the `util` global object, and has the following signature:

```
importTable (filename, options)
```

`filename` is a string specifying the name and path for the file containing the data to be imported. On Windows, backslashes must be escaped in the file path, or you can use forward slashes instead. The data file to be imported must be in a location that is accessible to the client host as a local disk. The data is imported to the MySQL server to which the active MySQL session is connected.

`options` is a dictionary of import options that can be omitted if it is empty. The following options are available 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 data file, given in the order that they map to columns in the target relational table. Use this option if the import file does not contain all the columns of the target table, or if the order of the fields in the import file 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.

`skipRows: number`
: Skip this number of rows of data at the beginning of the file. 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. 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 7.1 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. 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. 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. 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 (`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. If this is not provided, escape sequence interpretation does not occur. The default is as for the specified dialect, or a backslash (\) if the dialect option is omitted. This option is equivalent to the `FIELDS ESCAPED BY` option for the `LOAD DATA` statement.

`characterSet: "`*`charset`*`"`
: Added in MySQL Shell 8.0.21. This option specifies a character set encoding with which the input data file 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 file.

`bytesPerChunk: "size"`	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 divides 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 to the target server. If you do not specify a number of threads, the default maximum is 8. 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.
`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.

The following examples import the data in the 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"
```

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

The following example 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 function returns void, or an exception in case of an error. If the import is stopped partway by the user with **Ctrl+C** or by an error, the utility stops sending data. When the server finishes processing the data it received, messages are returned showing the chunk that was being imported by each thread at the time, the percentage complete, and the number of records that were updated in the target table.

The 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 example:

```
mysqlsh mysql://root:@127.0.0.1:3366 --ssl-mode=DISABLED -- util import-table /r/mytable.dump --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”.

7.4 Instance Dump Utility and Schema Dump Utility

MySQL Shell's instance dump utility and schema dump utility, 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 schemas 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 7.5, “Dump Loading Utility”.

MySQL Shelll's instance dump utility and schema 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 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 or schema 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 and schema dump utility comprise DDL files specifying the schema structure, and tab-separated `.tsv` files containing the data. You can also choose to produce the DDL files only or the data files only, if you want to set up the exported schema as a separate exercise from populating it with the exported data. You can choose whether or not to lock the instance for backup during the dump for data consistency. By default, the dump utilities chunk table data into multiple data files and compress the files.

You can specify schemas or individual tables to exclude from a dump. 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.

The following requirements apply to dumps using the instance dump utility and schema 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 upload method used to transfer files to an Oracle Cloud Infrastructure Object Storage bucket has a file size limit of 1.2 TiB.

- 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 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.
- 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`.
- 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 and schema 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 and schema dump utility are functions of the `util` global object, and have the following signatures:

```
util.dumpInstance(outputUrl[, options])
util.dumpSchemas(schemas, outputUrl[, options])
```

For the schema dump utility, `schemas` specifies a list of one or more schemas to be dumped from the MySQL instance. If most of the schemas are to be dumped, an alternative strategy is to use the instance dump utility and specify the `excludeSchemas` option to list those schemas that are not to be dumped.

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.

If you are dumping to an Oracle Cloud Infrastructure Object Storage bucket, `outputUrl` is a path that will be used to prefix the dump files in the bucket, to simulate a directory structure. Use the `osBucketName` option to provide the name of the Object Storage bucket, and the `osNamespace`

option to identify the namespace for the bucket. In this example, the user dumps the `world` schema from the connected MySQL instance to an Object Storage bucket, with the same compatibility modifications as in the previous example:

```
shell-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 both the instance dump utility and the schema 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 (tenancy name) 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`	Exclude the named schemas from the dump. This option is for the instance dump utility only. 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`	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.
`users: [ true | false ]`	Include (`true`) or exclude (`false`) users and their roles and grants in the dump. The default is `true`. This option is for the instance dump utility only. The schema dump utility does not include users, roles, and grants in a dump.
`events: [ true | false ]`	Include (`true`) or exclude (`false`) events for each schema in the dump. The default is `true`.
`routines: [ true | false ]`	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 32 MB (`32M`), 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`. 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.

`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. 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.
- `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).
- `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.

7.5 Dump Loading Utility

MySQL Shell's dump loading utility, 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 dumped using MySQL Shell's Section 7.4, “Instance Dump Utility and Schema 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.

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 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 or schema dump utility, as the `gtidExecuted` field in the `@.json` dump file. To retain these GTIDs on the target MySQL instance for use with replication, set the dump loading utility's `skipBinlog` option to `true` for the import to prevent the generation of new GTIDs as the dump is imported. After the import, update the `gtid_purged` GTID set on the target MySQL instance using the `gtid_executed` GTID set from the source MySQL instance. (This is currently not supported on MySQL DB System.) For details, see the description for the `skipBinlog` option.

MySQL Shell's dump loading utility uses the DDL files and tab-separated `.tsv` data files output by the instance dump utility or schema dump utility 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. 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.

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 `loadprogress.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.

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 instance dump utility and schema dump utility, 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, from MySQL 8.0.21, 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 `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.

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, 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, 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 (tenancy name) 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 `loadprogress.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 available tables 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. Tables are loaded as they become 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 available tables have been loaded 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. 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.

`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. 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 dump loading utility does not apply the `gtid_executed` GTID set from the source MySQL instance on the target MySQL instance. This is currently not supported on MySQL DB System. For a MySQL Server instance, 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, which 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. For MySQL 5.7, the plus sign (`+`) must be omitted, and the `gtid_executed` GTID set 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.

Note

The `gtid_executed` GTID set in the dump metadata includes the GTIDs of all transactions in the `gtid_executed` set on the source MySQL instance, including those relating to schemas that were not included in the dump. If you set the value of `gtid_purged` on the target MySQL instance, and later import a dump of another schema from the same source MySQL instance, do not issue the `SET`

> `@@GLOBAL.gtid_purged` statement again, as the GTIDs will already be present. Do continue to use `skipBinlog: true` for the import so that new GTIDs are not generated during the import.

`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. The default setting `fulltext` defers full-text indexes only. `all` defers all secondary indexes and only creates primary indexes during the table load. `off` means all indexes are created during the table load.
`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 or schema 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.
`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.
`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.
`includeSchemas:` *`array of strings`*	Load only the named schemas from the dump files.
`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 cannot combine this option with the `includeSchemas` option.
`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. Statements for the current user are skipped. The default is `false`, so users are not imported by default. Note that MySQL Shell's schema dump utility does not include users, roles, and grants in a dump, but the instance dump utility can, and does by default.

Chapter 8 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 8.1 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 8.1, “Application Log”.

For instructions to send logging information to the console as verbose output, see Section 8.2, “Verbose Output”.

For instructions to activate logging for SQL statements that are executed by AdminAPI operations, see Section 8.3, “Logging AdminAPI Operations”.

8.1 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 9.4, “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 9.4, “Configuring MySQL Shell Options”.

The available logging levels are as listed in Table 8.1, “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.

8.2 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 9.4, "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 9.4, "Configuring MySQL Shell Options".

The available settings are as listed in Table 8.1, "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.

8.3 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 9.4, “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 9.4, “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 9 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 9.1, “Working With Startup Scripts”.
- Add non-standard module search paths for JavaScript or Python mode. See Section 9.2, “Adding Module Search Paths”.
- Customize the MySQL Shell prompt. See Section 9.3, “Customizing the Prompt”.
- Set configuration options to change MySQL Shell's behavior for the current session or permanently. See Section 9.4, “Configuring MySQL Shell Options”.

9.1 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]`

9.2 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 9.2.1, "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 9.2.2, "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.

9.2.1 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.

9.2.2 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 9.1, "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.

9.3 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`.

9.4 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.
`batchContinueOnError`	false	boolean (READ ONLY)	In SQL batch mode, force processing to continue if an error is found.
`credentialStore.excludeFilter`	empty	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.

optionName	DefaultValue	Type	Effect
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 Working with InnoDB Cluster).
dba.logSql	0	integer ranging from 0 to 2	Log SQL statements that are executed by AdminAPI operations (see Chapter 8, *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 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 8, *MySQL Shell Logging and Debug*).
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-

optionName	DefaultValue	Type	Effect
			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 8, *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 InnoDB Cluster.

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
--interactive	Emulate Interactive mode in batch mode	
--js, --javascript	Start in JavaScript mode	
--json	Print output in JSON format	

Option Name	Description	Introduced
--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	
--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	

Option Name	Description	Introduced
--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 8, *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 7.2, "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 8, *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]`, `-ppassword`

  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` (`-ppassword`) 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.

- `--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 8, *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.