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 either the MySQL Forums or MySQL Mailing Lists, where you can discuss your issues with other MySQL users.

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

Document generated on: 2019-05-03 (revision: 61947)
# Table of Contents

1 MySQL Shell Features ............................................................................................................. 1

2 Installing MySQL Shell ........................................................................................................ 3
   2.1 Installing MySQL Shell on Microsoft Windows ......................................................... 3
   2.2 Installing MySQL Shell on Linux .............................................................................. 3
   2.3 Installing MySQL Shell on macOS ........................................................................... 5
   2.4 Starting MySQL Shell ................................................................................................. 5

3 Using MySQL Shell Commands ........................................................................................... 7
   3.1 MySQL Shell Commands ......................................................................................... 7

4 Getting Started with MySQL Shell ....................................................................................... 11
   4.1 MySQL Shell Connections ...................................................................................... 11
      4.1.1 Connecting using Individual Parameters ......................................................... 13
      4.1.2 Using Encrypted Connections ........................................................................ 14
      4.1.3 Connections in JavaScript and Python ............................................................. 15
   4.2 Pluggable Password Store .......................................................................................... 16
      4.2.1 Pluggable Password Configuration Options .................................................... 17
      4.2.2 Working with Credentials ............................................................................... 18
   4.3 MySQL Shell Sessions ............................................................................................... 18
      4.3.1 MySQL Shell Sessions Explained ..................................................................... 19
   4.4 MySQL Shell Global Variables .................................................................................. 20
   4.5 Using a Pager ............................................................................................................. 20

5 MySQL Shell Code Execution ............................................................................................... 23
   5.1 Active Language ........................................................................................................ 23
   5.2 Interactive Code Execution ....................................................................................... 24
   5.3 Code Autocompletion ............................................................................................... 25
   5.4 Code History ............................................................................................................. 27
   5.5 Batch Code Execution .............................................................................................. 28
   5.6 Output Formats .......................................................................................................... 29
      5.6.1 Table Format ...................................................................................................... 30
      5.6.2 Tab Separated Format ....................................................................................... 30
      5.6.3 Vertical Format ................................................................................................. 30
      5.6.4 JSON Format Output ........................................................................................ 30
      5.6.5 JSON Wrapping ............................................................................................... 31
      5.6.6 Result Metadata .............................................................................................. 32
   5.7 API Command Line Interface ...................................................................................... 33

6 Reporting with MySQL Shell ............................................................................................... 35
   6.1 Creating and Registering User-Defined Reports ....................................................... 35
   6.2 Running MySQL Shell Reports .................................................................................. 38

7 MySQL Shell Utilities .......................................................................................................... 41
   7.1 Upgrade Checker Utility ........................................................................................... 41
   7.2 JSON Import Utility .................................................................................................. 47
      7.2.1 Importing JSON documents with the mysqlsh command interface ................. 49
      7.2.2 Importing JSON documents with the --import command ................................ 50
      7.2.3 Conversions for representations of BSON data types ........................................ 51

8 MySQL Shell Application Log ............................................................................................. 53
   8.1 Application Log ........................................................................................................... 53

9 Customizing MySQL Shell .................................................................................................. 55
   9.1 Working With Start-Up Scripts ................................................................................. 55
   9.2 Adding Module Search Paths ...................................................................................... 56
      9.2.1 Environment Variables .................................................................................... 56
      9.2.2 Startup Scripts ................................................................................................. 56
   9.3 Customizing the Prompt ............................................................................................ 56
   9.4 Configuring MySQL Shell Options .......................................................................... 57

A MySQL Shell Command Reference ....................................................................................... 61
   A.1 mysqlsh — The MySQL Shell ..................................................................................... 61
Chapter 1 MySQL Shell Features

The following features are available in MySQL Shell.

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.

Supported Languages

MySQL Shell processes code in the following languages: JavaScript, Python and SQL. Any entered 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”.

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 5.5, “Batch Code Execution”.

Output Formats

MySQL Shell can return results in table, tabbed, or vertical format, or as pretty or raw 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 5.6, “Output Formats”.

Application Log

MySQL Shell can be configured to log information about the execution process. For more information see Chapter 8, MySQL Shell Application Log.

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 legacy MySQL Protocol. A minimal set of features from the X DevAPI are available for sessions created using the legacy MySQL protocol.

Global Session

Interaction with a MySQL Server is done through a Session object. For Python and JavaScript, a Session can be created through the `getSession` function of the `mysqlx` module. If a session is created in JavaScript mode using any of these methods, it is available only in JavaScript mode. The same happens if the session is created in Python mode. These sessions cannot be used in SQL mode.

For SQL Mode, the concept of Global Session is supported by the MySQL Shell. A Global Session is created when the connection information is passed to MySQL Shell using command options, or by using the `\connect` command.

The Global Session is used to execute statements in SQL mode and the same session is available in both Python or JavaScript modes. When a Global Session is created, a variable called `session` is set in the scripting languages, so you can execute code in the different languages by switching the active mode.

For more information, see Section 4.3, “MySQL Shell Sessions”.
Chapter 2 Installing MySQL Shell

Table of Contents

2.1 Installing MySQL Shell on Microsoft Windows ................................................................. 3
2.2 Installing MySQL Shell on Linux ..................................................................................... 3
2.3 Installing MySQL Shell on macOS ................................................................................... 5
2.4 Starting MySQL Shell ........................................................................................................ 5

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:
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`.  

Installing MySQL Shell with the MySQL Yum Repository

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

- 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
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
```

## Install 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:

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.)

### 2.4 Starting MySQL Shell

You need an account name and password to establish a session using MySQL Shell. Replace `user` with your account name.

Open a terminal window (command prompt on Windows) and start MySQL Shell with the following command:

```
mysqlsh --uri user@localhost
```

You are prompted to input your password. Note that the maximum password length that is accepted for connecting to MySQL Shell is 128 characters.
Chapter 3 Using MySQL Shell Commands

Table of Contents

3.1 MySQL Shell Commands ................................................................. 7

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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Alias/Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\help</td>
<td>h or ?</td>
<td>Print help about MySQL Shell, or search the online help.</td>
</tr>
<tr>
<td>\quit</td>
<td>q or exit</td>
<td>Exit MySQL Shell.</td>
</tr>
<tr>
<td>\</td>
<td></td>
<td>In SQL mode, begin multiple-line mode. Code is cached and executed when an empty line is entered.</td>
</tr>
<tr>
<td>\status</td>
<td>s</td>
<td>Show the current MySQL Shell status.</td>
</tr>
<tr>
<td>\js</td>
<td></td>
<td>Switch execution mode to JavaScript.</td>
</tr>
<tr>
<td>\py</td>
<td></td>
<td>Switch execution mode to Python.</td>
</tr>
<tr>
<td>\sql</td>
<td></td>
<td>Switch execution mode to SQL.</td>
</tr>
<tr>
<td>\connect</td>
<td>c</td>
<td>Connect to a MySQL Server.</td>
</tr>
<tr>
<td>\reconnect</td>
<td></td>
<td>Reconnect to the same MySQL Server.</td>
</tr>
<tr>
<td>\use</td>
<td>u</td>
<td>Specify the schema to use.</td>
</tr>
<tr>
<td>\source</td>
<td>.</td>
<td>Execute a script file using the active language.</td>
</tr>
<tr>
<td>\warnings</td>
<td>W</td>
<td>Show any warnings generated by a statement.</td>
</tr>
<tr>
<td>\nowarnings</td>
<td>w</td>
<td>Do not show any warnings generated by a statement.</td>
</tr>
<tr>
<td>\history</td>
<td></td>
<td>View and edit command line history.</td>
</tr>
<tr>
<td>\rehash</td>
<td></td>
<td>Manually update the autocomplete name cache.</td>
</tr>
<tr>
<td>\option</td>
<td></td>
<td>Query and change MySQL Shell configuration options.</td>
</tr>
<tr>
<td>\show</td>
<td></td>
<td>Run the specified report using the provided options and arguments.</td>
</tr>
<tr>
<td>\watch</td>
<td></td>
<td>Run the specified report using the provided options and arguments, and refresh the results at regular intervals.</td>
</tr>
</tbody>
</table>

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.
Connect and Reconnect Commands

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** - introduces the `dba` global object and the InnoDB cluster AdminAPI.
- **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.
- **X DevAPI** - details the `mysqlx` module as well as the capabilities of the X DevAPI which enable working with MySQL as a Document Store.

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 `?` and `*` to match one or 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 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 4.1, “MySQL Shell Connections”.

Status Command

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:3306

Use the --mysql (--mc) option to create a ClassicSession, enabling you to use the 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 is used to execute code from a script 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.

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.

For more information, see Section 5.4, “Code History”.

**Rehash Command**

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

**Option Command**

The `\option` command enables you to query and change MySQL Shell configuration options in all modes. You can use the `\option` command to list the configuration options that have been set and show how their value was last changed. You can also use it to set and unset options, either for the session, or persistently in the MySQL Shell configuration file. For instructions and a list of the configuration options, see Section 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 4.5, “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 6.2, “Running MySQL Shell Reports”.
4.1 MySQL Shell Connections

MySQL Shell can connect to MySQL Server using both the X Protocol and the classic MySQL protocol. You can configure the MySQL server instance that MySQL Shell is connected to in the following ways:

- When you start MySQL Shell using the command parameters. See Section 4.1.1, “Connecting using Individual Parameters”.
- When MySQL Shell is running using the `\connect` command. See Section 3.1, “MySQL Shell Commands”.
- When running in Python or JavaScript mode using the `shell.connect('instance')` method.

These different ways of connecting to a MySQL server instance all support specifying the connection as:

- A URI type string, such as `myuser@example.com:3306/main-schema`. See Connecting using a URI String for the full syntax.
- As key-value pairs, 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. This means you can supply connection parameters using key-value pairs as a JSON object in JavaScript, or using key-value pairs in a dictionary in Python. See Connecting using Key-value Pairs for the full syntax.

See Connecting Using a URI 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 4.2, “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:
MySQL Shell Connections

- If you are connecting using a URI type string, place a : after the user in the URI type 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

If the connection to the server is lost, MySQL Shell does not attempt to reconnect automatically. Use the \reconnect command to make MySQL Shell try several reconnection attempts for the current global session with the previously supplied parameters.

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:

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

To enable compression for the connection, use the compression connection parameter, for example:

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

When set to true (or 1), this option enables compression of all information sent between the client and the server if both support compression. The default is no compression (false or 0). If you are connecting using command parameters, the equivalent parameter is --compress (-C). Compression is available for MySQL Shell connections using classic MySQL protocol only. You can set the defaultCompress MySQL Shell configuration option to enable compression for every global session. The MySQL Shell \status command shows whether or not compression is enabled for the session.

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 equals 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. In this case, if a TCP port is not specified the default value of 3306 is used.
Connecting using Individual Parameters

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 type string, specify `user@.`
- If you are connecting using a 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 type string.

In URI type strings, the path to a Unix socket file or Windows named pipe must either be encoded using percent encoding, or surrounded with parentheses, which removes the need to percent encode characters such as the common directory separator (`/`). If the path to a Unix socket file is included in the URI type 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%2Fmysqld.sock
mysql-js> \connect user@localhost?socket=(/tmp/mysqld.sock)
mysql-js> \connect user@/tmp%2Fmysqld.sock
mysql-js> \connect user@(/tmp/mysqld.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\pipe
```

**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 and Connecting Using a URI or Key-Value Pairs.

### 4.1.1 Connecting using Individual Parameters

In addition to specifying connection parameters using a path, 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.
Use the following parameters 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 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 equals 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.

To enable compression for the session, specify the `--compress (-C)` parameter. This parameter enables compression of all information sent between the client and the server if both support compression. Compression is available for MySQL Shell connections using classic MySQL protocol only. In a URI type string, the equivalent parameter is `compression`. The MySQL Shell `\status` command shows whether or not compression is enabled for the session.

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.

For example to override `user` from the URI:

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

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, with compression enabled:

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

### 4.1.2 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 security state of the connection to the server.

• `--ssl-ca=filename`: The path to a file in PEM format that contains a list of trusted SSL Certificate Authorities.

• `--ssl-capath=directory`: The path to a directory that contains trusted SSL Certificate Authority certificates in PEM format.

• `--ssl-cert=filename`: 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=filename`: The name of the SSL key file in PEM format to use for establishing an encrypted connection.

• `--ssl-crl=filename`: The path to a file containing certificate revocation lists in PEM format.

• `--ssl-crlpath=directory`: 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.

Alternatively, the SSL options can be encoded as part of a URI type 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 type string must be percent encoded, for example:

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

See [Connecting Using a URI or Key-Value Pairs](#) for more information.

### 4.1.3 Connections in JavaScript and Python

When a connection is made using the command options or by using any of the MySQL Shell commands, a global session object is created. This session is global because once created, it can be used in any of the MySQL Shell execution modes.

Any global session object is available in JavaScript or Python modes because a variable called `session` holds a reference to it.

In addition to the global session object, sessions can be established and assigned to a different variable by using the functions available in the `mysql` and `mysqlx` JavaScript and Python modules.

For example, the following functions are provided by these modules:

- `mysqlx.getSession(connectionData[, password])`
  - The returned object can be `Session` if the object was created or retrieved using a `Session` instance, and `ClassicSession` if the object was created or retrieved using a `ClassicSession` instance.

- `mysql.getClassicSession(connectionData[, password])`
  - The returned object is a `ClassicSession` which uses the traditional MySQL protocol and has a limited development API.

`connectionData` can be either a URI type string or key-value pairs containing the connection parameters. See [Connecting Using a URI or Key-Value Pairs](#).
Sessions created using either `mysql.get ClassicSession(connection_data)` or `mysqlx.getSession(connection_data)` 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`.

The following example shows how to create a Session using the X Protocol:

```javascript
mysql-js> var mysession1=mysqlx.getSession('root@localhost:33060', 'password');
<Session:root@localhost>:33060
```

The following example shows how to create a ClassicSession, with compression enabled for the connection:

```javascript
mysql-js> var mysession2=mysqlx.getSession('root@localhost:33060?compression=true', 'password');
<ClassicSession:root@localhost:3306>
```

### 4.1.3.1 Using Encrypted Connections in Code

To establish an encrypted connection, set the SSL information in the `connectionData` dictionary. For example:

```javascript
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"});
```

See [Connecting using Key-value Pairs](#) for more information.

### 4.2 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:

```javascript
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/[N]ever (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 type 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

### 4.2.1 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.

```plaintext
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:

```javascript
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.

```plaintext
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 URIs 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.

### 4.2.2 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`).

### 4.3 MySQL Shell Sessions

This section explains the different types of sessions in MySQL Shell and how to create and configure
them.
4.3.1 MySQL Shell Sessions Explained

MySQL Shell is a unified interface to operate MySQL Server through scripting languages such as JavaScript or Python. To maintain compatibility with previous versions, SQL can also be executed in certain modes. A connection to a MySQL server is required. In MySQL Shell these connections are handled by a `Session` object.

The following types of Session object are available:

- **Session**: Use this session type for new application development to communicate with MySQL server instances which have the X Protocol enabled. It offers the best integration with MySQL Server, and therefore, it is used by default.

- **ClassicSession**: Use this session type to interact with MySQL Servers that do not have the X Protocol enabled. The development API available for this type of session is very limited. For example, there are no CRUD operations, no collection handling, and binding is not supported.

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

Choosing a MySQL Shell Session Type

MySQL Shell creates a Session object by default. You can either configure the session type using MySQL Shell command options, the `scheme` element of a URI type string, or provide an option to the `\connect` command. To choose which type of session should be created when starting MySQL Shell, use one of these options:

- `--mysqlx` (`--mx`) creates a Session, connected using X Protocol.
- `--mysql` (`--mc`) creates a ClassicSession, connected using MySQL protocol.

To choose which type of session to use when defining a URI type string use one of these options:

- Specify `mysqlx` to create an X Protocol session. The X Plugin must be installed on the server instance, see Using MySQL as a Document Store for more information.
- Specify `mysql` to create a classic MySQL protocol session.

For more information, see Connecting Using a URI or Key-Value Pairs.

Creating a Session Using Shell Commands

If you open MySQL Shell without specifying connection parameters, MySQL Shell opens without an established global session. It is possible to establish a global session once MySQL Shell has been started using the MySQL Shell `\connect` command, where `URI` is a URI type string as defined at Connecting using a URI String. For example:

- `\connect --mysqlx | --mx URI`: Creates a Session using X Protocol.
- `\connect --mysql | --mc URI`: Creates a ClassicSession using MySQL protocol.

For example:

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

If you do not specify a protocol with the `\connect` command, MySQL Shell automatically attempts to use X Protocol for the session's connection, and falls back to MySQL protocol if X Protocol is unavailable. The protocol option `--ma`, which specified that behavior explicitly, is now deprecated. The
use of a single dash with the short form options (that is, \texttt{-mx} and \texttt{-mc}) is also deprecated from version \texttt{8.0.13} of MySQL Shell.

Alternatively, you can use the \texttt{shell.connect('URI')} method. For example this is equivalent to the above \texttt{\connect>} command:

\begin{verbatim}
mysql-je> shell.connect('mysqlx://user@localhost')
\end{verbatim}

\section*{4.4 MySQL Shell Global Variables}

MySQL Shell reserves certain variables as global variables, which are assigned to commonly used objects in scripting. This section describes the available global variables and provides examples of working with them. The global variables are:

- \texttt{session} represents the global session if one has been established.
- \texttt{db} represents a schema if one has been defined, for example by a URI type string.
- \texttt{dba} represents the AdminAPI, a component of InnoDB cluster which enables you to administer clusters of server instances. See InnoDB Cluster.
- \texttt{shell} provides general purpose functions, for example to configure MySQL Shell. \texttt{shell.reports} provides built-in or user-defined MySQL Shell reports as functions, with the name of the report as the function.
- \texttt{util} provides utility functions, for example to check server instances before an upgrade. See Chapter 7, MySQL Shell Utilities.

\begin{center}
\begin{tcolorbox}
\textbf{Important}
These words are reserved and cannot be used, for example as names of variables.
\end{tcolorbox}
\end{center}

\section*{4.5 Using a Pager}

You can configure MySQL Shell to use an external pager tool such as \texttt{less} or \texttt{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 \texttt{shell.options[pager]} = "" MySQL Shell option, a string which specifies the external command that displays the paged output. This string can optionally contain command line arguments which are passed to the external pager command. Correctness of the new value is not checked. An empty string disables the pager.

  Default value: empty string.

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

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

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

- Use the \texttt{\pager | \P command} MySQL Shell command to set the value of \texttt{shell.options["pager"]} option. If called with no arguments, restores the initial value of \texttt{shell.options["pager"]} option.
Using a Pager

(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

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.

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:

```
\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.

### 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:

```javascript
var mySession = mysqlx.getSession('user:pwd@localhost');
var result = mySession.world_x.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()`
Multiple-line Support

- `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:

```javascript
mysql-js> var mySession = mysqlx.getSession('user:pwd@localhost');
```

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

```javascript
mysql-js> mySession.world_x.countryinfo.find();
```

### 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:

```sql
mysql-sql> \n
... 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
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
shell.options['devapi.dbObjectHandles']=true
```

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

### 5.4 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:

```shell
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.

### 5.5 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 5.7, “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`
Executable Scripts

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.

5.6 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.

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.
5.6.1 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`.

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

<table>
<thead>
<tr>
<th>actor_id</th>
<th>first_name</th>
<th>last_name</th>
<th>last_update</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PENELLOPE</td>
<td>GUINESS</td>
<td>2006-02-15 4:34:33</td>
</tr>
<tr>
<td>2</td>
<td>NICK</td>
<td>WAHLBERG</td>
<td>2006-02-15 4:34:33</td>
</tr>
<tr>
<td>3</td>
<td>ED</td>
<td>CHASE</td>
<td>2006-02-15 4:34:33</td>
</tr>
</tbody>
</table>

5.6.2 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`.

```sql
> echo "select * from sakila.actor limit 3;" | mysqlsh --classic --uri user@192.0.2.20:33460
```

```
actor_id  first_name  last_name  last_update  
1          PENELLOPE   GUINESS    2006-02-15 4:34:33  
2          NICK        WAHLBERG  2006-02-15 4:34:33  
3          ED          CHASE     2006-02-15 4:34:33  
```

5.6.3 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`.

5.6.4 JSON Format Output

The JSON format options print result sets as either raw JSON (`json/raw`) or pretty-printed JSON (`json`). To get this output format, start MySQL Shell with the `--result-format=json` command line option (for pretty-printed JSON) or the `--result-format=json/raw` command line option, or set the MySQL Shell configuration option `resultFormat` to `json` (for pretty-printed JSON) or `json/raw`.

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 5.6.5, “JSON Wrapping”.

JSON Format in Interactive Mode (Pretty-Printed JSON)

```sql
mysql-sql> select * from sakila.actor limit 3;
```
5.6.5 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.
5.6.6 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.
5.7 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 MySQL Shell objects are available:

- **dba** - global object for accessing the InnoDB cluster functions
- **cluster** - global object for accessing an InnoDB cluster
- **shell** - MySQL Shell global
- **shell.options** - for configuring MySQL Shell options
- **util** - global object for accessing 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:

```bash
$ mysqlsh -- util check-for-server-upgrade { --user=root --host=localhost --port=3301 } --password='password' --outputFormat=JSON --config-path=/etc/mysql/my.cnf
```

This maps to the equivalent command in MySQL Shell:

```javascript
mysql-js> util.checkForServerUpgrade({user:'root', host:'localhost', port:3301}, {password:'password', outputFormat:'JSON', configPath:'/etc/mysql/my.cnf'})
```

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

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

This maps to the equivalent command in MySQL Shell:

```javascript
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`:

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

This maps to the equivalent command in MySQL Shell:

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

• To check the status of an InnoDB cluster using the sandbox instance listening on port 1234:

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

This maps to the equivalent command in MySQL Shell:

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

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

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

This maps to the equivalent command in MySQL Shell:

```javascript
mysql-js> shell.options.set_persist('history.autoSave', true);
```
Chapter 6 Reporting with MySQL Shell

Table of Contents

6.1 Creating and Registering User-Defined Reports ................................................................. 35
6.2 Running MySQL Shell Reports .............................................................................................. 38

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.

For the report to be automatically loaded when MySQL Shell starts, you must store the function and its registration in a script in the `init.d` folder in the MySQL Shell user configuration path. The file extension is used to determine the language in which MySQL Shell processes the script, so it must be `.js` or `.py` as appropriate. Section 6.1, “Creating and Registering User-Defined Reports” has instructions to create and register your reports.

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

In MySQL Shell 8.0.16, the following built-in report is available:

`query` Executes the single SQL statement that is provided as an argument. This report has no report-specific options, but the standard options for the `\show` and `\watch` commands may be used. For examples, see Section 6.2, “Running MySQL Shell Reports”.

6.1 Creating and Registering User-Defined 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.

MySQL Shell classes reports by the type of output that they return, as follows:

- **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.

- **Report type**: Returns free-form output, which MySQL Shell displays using YAML.

- **Print type**: Prints its own output to screen.

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.
The signature for the Python or JavaScript function to be registered as a MySQL Shell report must be as follows:

```python
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.

Your report can provide its output in any of the following ways:

- Return a list of lists, with the first list consisting of the names of columns, and the remainder of the lists consisting of the content of rows. MySQL Shell can display this output as a table (the default) or in vertical format. 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”.

- Return 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 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.

To register your user-defined report with MySQL Shell, call the `shell.registerReport` method in JavaScript or `shell.registerReport` in Python. The syntax method is as follows:

```python
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.
Registering your report

- 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:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brief</td>
<td>A brief description of the report.</td>
</tr>
<tr>
<td>details</td>
<td>A detailed description of the report, provided as an array of strings. This is provided when you use the <code>\help</code> command or the <code>--help</code> option with the <code>\show</code> command.</td>
</tr>
<tr>
<td>options</td>
<td>Any report-specific options that the report can accept. Each dictionary in the array describes one option, and must contain the following keys:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>name (string, required): The name of the option in the long form, which must be a valid scripting identifier.</td>
</tr>
<tr>
<td></td>
<td>brief (string, optional): A brief description of the option.</td>
</tr>
<tr>
<td></td>
<td>shortcut (string, optional): An alternate name for the option as a single alphanumeric character.</td>
</tr>
<tr>
<td></td>
<td>details (array of strings, optional): A detailed description of the option. This is provided when you use the <code>\help</code> command or the <code>--help</code> option with the <code>\show</code> command.</td>
</tr>
<tr>
<td></td>
<td>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 <code>\show</code> or <code>\watch</code> command, and must have a valid value when you run the report using the <code>shell.reports</code> object.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>argc</td>
<td>A string specifying the number of additional arguments that the report expects, which can be one of the following:</td>
</tr>
<tr>
<td></td>
<td>An exact number of arguments, which is specified as a single number.</td>
</tr>
<tr>
<td></td>
<td>Zero or more arguments, which is specified as an asterisk.</td>
</tr>
<tr>
<td></td>
<td>A range of argument numbers, which is specified as two numbers separated by a dash (for example, “1-5”).</td>
</tr>
</tbody>
</table>
• 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-*”).

**Saving your report**

Your report must be saved as a file in the `init.d` folder in the MySQL Shell user configuration path, so that MySQL Shell can identify and load it as a report. The default path for this folder is `~/.mysqlsh/init.d` in Unix and `%AppData%\MySQL\mysqlsh\init.d` 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.

The file name for your report does not have to match the report name, but it is a best practice for it to do so. The file extension must be `.js` or `.py` to match the scripting language used for the report. The file extension is not case-sensitive.

**Example**

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

```python
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')
    try:
        limit = int(options['limit'])
    except SystemError:
        limit = 0
    if limit > 0:
        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 functions used by it.'],
        'options': [  
            {  
                'name': 'limit',
                'brief': 'The maximum number of rows to return.',
                'shortcut': 'l',
                'type': 'integer'
            },
        ],
        'argc': '0'
    })
```

---

**6.2 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
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:

```bash
\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:

```bash
\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:

```bash
\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:

```python
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.

```python
report = shell.reports.sessions(shell.getSession(), [], {'limit':10});
```
Chapter 7 MySQL Shell Utilities

Table of Contents

7.1 Upgrade Checker Utility ......................................................... 41
7.2 JSON Import Utility ........................................................... 47
  7.2.1 Importing JSON documents with the mysqlsh command interface ................................................. 49
  7.2.2 Importing JSON documents with the --import command ................................................................. 50
  7.2.3 Conversions for representations of BSON data types ................................ 51

MySQL Shell includes utilities for working with MySQL. To access the utilities from within MySQL Shell, use the \texttt{util} global object, which provides the following functions:

\begin{itemize}
  \item \texttt{checkForServerUpgrade()} An upgrade checker utility that enables you to verify whether MySQL server instances are ready for upgrade.
  \item \texttt{importJSON()} A JSON import utility that enables you to import JSON documents to a MySQL Server collection or table.
\end{itemize}

7.1 Upgrade Checker Utility

The \texttt{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 \texttt{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 \texttt{\textbackslash status} command.

\begin{itemize}
  \item \textbf{Note}  
    \begin{enumerate}
    \item The upgrade checker utility does not support checking MySQL Server instances at a version earlier than 5.7.
    \item 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.
    \end{enumerate}
\end{itemize}

From MySQL Shell 8.0.16, the upgrade checker utility can check the configuration file (\texttt{my.cnf} or \texttt{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 \texttt{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.

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:

```java
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. A user account with `ALL` privileges is required. 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 8.0.16, 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, or 8.0.16. 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:

```sql
mysqlsh> \status
MySQL Shell version 8.0.16
...
Server version:     5.7.24-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:

```sql
mysqlsh> util.checkForServerUpgrade('user@example.com:3306', {"password":"password", "targetVersion":"8.0.11", "configPath":"C:\ProgramData\MySQL\MySQL Server 8.0\my.ini"})
```

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:

```sql
mysqlsh> util.checkForServerUpgrade('user@example.com:3306', {"password":"password", "outputFormat":"JSON", "configPath":"C:\ProgramData\MySQL\MySQL Server 8.0\my.ini"})
```

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 5.7, “API Command Line Interface”. The following example checks a MySQL server for upgrade to release 8.0.15, and returns JSON output:

```bash
mysqlsh -- util.checkForServerUpgrade user@localhost:3306 --target-version=8.0.15 --output-format=JSON --config-path=/etc/mysql/my.cnf
```
The connection data can also be specified as named options grouped together by using curly brackets, as in the following example, which also shows that lower case and hyphens can be used for the method name rather than camelCase:

```
mysqlsh -- util check-for-server-upgrade { --user=user --host=localhost --port=3306 } --target-version=8.0.15 --output-format=JSON --config-path=/etc/mysql/my.cnf
```

The following example uses a Unix socket connection and shows the older format for invoking the utility from the command line, which is still valid:

```
./bin/mysqlsh --socket=/tmp/mysql.sock --user=user -e "util.checkForServerUpgrade()"
```

To get help for the upgrade checker utility, issue:

```
mysqlsh> util.help("checkForServerUpgrade")
```

`util.checkForServerUpgrade()` 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 8.0.16...

1) Usage of old temporal type
   No issues found

2) Usage of db objects with names conflicting with reserved keywords in 8.0
   Warning: The following objects have names that conflict with reserved keywords that are new to 8.0. Ensure queries sent by your applications use 'quotes' when referring to them or they will result in errors.
   
   dbtest.System - Table name
   dbtest.System.JSON_TABLE - Column name
   dbtest.System.cube - Column name

3) Usage of utf8mb3 charset
   Warning: The following objects use the utf8mb3 character set. It is recommended to convert them to use utf8mb4 instead, for improved Unicode support.
   
   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) Foreign key constraint names longer than 64 characters
   No issues found

6) Usage of obsolete MAXDB sql_mode flag
   No issues found

7) Usage of obsolete sql_mode flags
   No issues found

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

9) 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);`
   
   dbtest.table1 - Partition p0 is in shared tablespace tbsp4
Upgrade Checker Utility

dbtest.table1 - Partition pl is in shared tablespace tbsp4

10) Usage of removed functions
   No issues found

11) 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.
   dbtest.view1 - VIEW uses removed GROUP BY DESC syntax
   dbtest.func1 - FUNCTION uses removed GROUP BY ASC syntax

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

13) 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.
    log_builtin_as_identified_by_password - is set and will be removed
    show_compatibility_56 - is set and will be removed

14) 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
    [...]

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

16) 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:

Errors:  6
Warnings: 32
Notices:  0

6 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 checks.
check. Each JSON object has the following key-value pairs:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
<td>The message level, one of Error, Warning, or Notice.</td>
</tr>
<tr>
<td>dbObject</td>
<td>A string identifying the database object to which the message relates.</td>
</tr>
<tr>
<td>description</td>
<td>If available, a string with a further description of the issue.</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The ID of the manual check, which is a unique string.</td>
</tr>
<tr>
<td>title</td>
<td>A short description of the manual check.</td>
</tr>
<tr>
<td>level</td>
<td>The message level, either Warning or Notice.</td>
</tr>
<tr>
<td>description</td>
<td>A long description of the manual check, with information and advice.</td>
</tr>
<tr>
<td>documentationLink</td>
<td>If available, a link to documentation with further information or advice.</td>
</tr>
</tbody>
</table>
### 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:

```sql
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:

```javascript
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 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.

- `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.
Importing JSON documents with the mysqlsh command interface

```sql
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:

```sql
mysql-js> util.importJson('/europe/regions.json', {schema: 'mydb', table: 'regions', tableColumn: 'jsondata', convertBsonTypes: true});
```

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:

```sql
```

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:

```bash
mysqlsh user@host:port/mydb -- util importJson <path> [options]
```

For information on this syntax, see Section 5.7, “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 connection URI, 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 connection URI, 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.

```shell
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 connection URI:

```shell
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.

```shell
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:

<table>
<thead>
<tr>
<th>BSON Type</th>
<th>JSON Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Simple value containing the value of the field.</td>
</tr>
<tr>
<td>Timestamp</td>
<td>MySQL timestamp created using the <code>time_t</code> value.</td>
</tr>
<tr>
<td>Decimal</td>
<td>Simple value containing a string representation of the decimal value.</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer value.</td>
</tr>
<tr>
<td>Regular expression</td>
<td>String containing the regular expression only, and ignoring the options. A warning is printed if options are present.</td>
</tr>
<tr>
<td>Binary data</td>
<td>Base64 string.</td>
</tr>
<tr>
<td>ObjectID</td>
<td>Simple value containing the value of the field.</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimalAsDouble: true</td>
<td>Convert the value of the BSON “decimal” type to the MySQL DOUBLE type, rather than a string.</td>
</tr>
<tr>
<td>ignoreRegex: true</td>
<td>Disable conversion of regular expressions (the BSON “regex” type). The data is imported as an embedded JSON document exactly as in the input file.</td>
</tr>
<tr>
<td>ignoreRegexOptions: false</td>
<td>Include the options associated with a regular expression in the string, as well as the regular expression itself (in the format /&lt;regular expression&gt;/&lt;options&gt;). 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.</td>
</tr>
<tr>
<td>ignoreBinary: true</td>
<td>Disable conversion of the BSON “binData” type. The data is imported as an embedded JSON document exactly as in the input file.</td>
</tr>
</tbody>
</table>

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:

```bash
gsql user@localhost/mydb --import /europe/regions.json regions jsondata --convertBsonTypes --ignoreRegex
```
Chapter 8 MySQL Shell Application Log

Table of Contents

8.1 Application Log ................................................................. 53

This section explains the application log.

8.1 Application Log

MySQL Shell can be configured to generate an application log file with information about issues of varying severity. You can use this information to verify the state of MySQL Shell while it is running.

The location of the log file is the user configuration path and the file is named mysqlsh.log.

By default, logging is disabled in MySQL Shell. To enable logging use the --log-level command-line option when starting MySQL Shell, or use either the \option command or the shell.options object to set the logLevel MySQL Shell configuration option. If the option is not set on the command line or in the configuration file, or if you specify a logging level of 1 or none, logging is disabled. All other values enable logging and set the level of detail in the log as follows:

<table>
<thead>
<tr>
<th>Logging Level - Numeric</th>
<th>Logging Level - Text</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>none</td>
<td>None, the default</td>
</tr>
<tr>
<td>2</td>
<td>internal</td>
<td>Internal Error</td>
</tr>
<tr>
<td>3</td>
<td>error</td>
<td>Error</td>
</tr>
<tr>
<td>4</td>
<td>warning</td>
<td>Warning</td>
</tr>
<tr>
<td>5</td>
<td>info</td>
<td>Informational</td>
</tr>
<tr>
<td>6</td>
<td>debug</td>
<td>Debug</td>
</tr>
<tr>
<td>7</td>
<td>debug2</td>
<td>Debug2</td>
</tr>
<tr>
<td>8</td>
<td>debug3</td>
<td>Debug3</td>
</tr>
</tbody>
</table>

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();
```

The default 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.

**Log File Location on Windows**

On Windows, the default path to the 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`.

**Log File Location on Unix-based Systems**

For a machine running Unix, the default path 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 logs.
Chapter 9 Customizing MySQL Shell

Table of Contents

9.1 Working With Start-Up Scripts .......................................................... 55
9.2 Adding Module Search Paths ............................................................ 56
  9.2.1 Environment Variables ................................................................. 56
  9.2.2 Startup Scripts .............................................................................. 56
9.3 Customizing the Prompt ..................................................................... 56
9.4 Configuring MySQL Shell Options .................................................... 57

MySQL Shell offers the ability to customize the behavior and code execution environment through
startup scripts, which are executed when the application is first run. Using such scripts enables you to:

• Add additional search paths for Python or JavaScript modules.
• Override the default prompt used by the Python and JavaScript modes.
• Define global functions or variables.
• Any other possible initialization through JavaScript or Python.

9.1 Working With Start-Up Scripts

When MySQL Shell enters either into JavaScript or Python mode, it searches for startup scripts to be
executed. The startup scripts are JavaScript or Python specific scripts containing the instructions to be
executed when the corresponding mode is initialized.

Startup scripts must be named as follows:

• For JavaScript mode: mysqlshrc.js
• For Python mode: mysqlshrc.py

MySQL Shell searches the following paths for these files (in order of execution).

On Windows:
1. %PROGRAMDATA%\MySQL\mysqlsh\mysqlshrc.[js|py]
2. %MYSQLSH_HOME%\shared\mysqlsh\mysqlshrc.[js|py]
3. <mysqlsh binary path>\mysqlshrc.[js|py]
4. %APPDATA%\MySQL\mysqlsh\mysqlshrc.[js|py]

On Linux and OSX:
1. /etc/mysql/mysqlsh/mysqlshrc.[js|py]
2. $MYSQLSH_HOME/shared/mysqlsh/mysqlshrc.[js|py]
3. <mysqlsh binary path>/mysqlshrc.[js|py]
4. $HOME/.mysqlsh/mysqlshrc.[js|py]

Warning
The lists above also define the order of searching the paths, so if something is
defined in two different scripts, the script executed later takes precedence.
The environment variable `MYSQLSH_HOME`, used in option 2, defines the root folder of a standard setup of MySQL Shell. If `MYSQLSH_HOME` is not defined it is automatically calculated based on the location of the MySQL Shell binary, therefore on many standard setups it is not required to define `MYSQLSH_HOME`.

If `MYSQLSH_HOME` is not defined and the MySQL Shell binary is not in a standard install folder structure, then the path defined in option 3 in the above lists is used. If using a standard install or if `MYSQLSH_HOME` points to a standard install folder structure, then the path defined in option 3 is not used.

The user configuration path in option 4 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.

## 9.2 Adding Module Search Paths

There are two ways to add additional module search paths:

- Through environment variables
- Through startup scripts

### 9.2.1 Environment Variables

Python uses the `PYTHONPATH` environment variable to allow extending the search paths for python modules. The value of this variable is a list of paths separated by:

- A colon character in Linux and OSX
- A semicolon character in Windows

To achieve this in JavaScript, MySQL Shell supports defining additional JavaScript module paths using the `MYSQLSH_JS_MODULE_PATH` environment variable. The value of this variable is a list of semicolon separated paths.

### 9.2.2 Startup Scripts

The addition of module search paths can be achieved for both languages through the corresponding startup script.

For Python modify the `mysqlshrc.py` file and append the required paths into the `sys.path` array.

```python
# Import the sys module
import sys

# Append the additional module paths
sys.path.append('~/custom/python')
sys.path.append('~/other/custom/modules')
```

For JavaScript the same task is achieved by adding code into the `mysqlshrc.js` file to append the required paths into the predefined `shell.js_module_paths` array.

```javascript
// Append the additional module paths
shell.js.module_paths[shell.js.module_paths.length] = '~/custom/js';
shell.js.module_paths[shell.js.module_paths.length] = '~/other/custom/modules';
```

## 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,
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:

<table>
<thead>
<tr>
<th>optionName</th>
<th>DefaultValue</th>
<th>Type</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>autocomplete.name</td>
<td>true</td>
<td>boolean</td>
<td>Enable database name caching for autocompletion.</td>
</tr>
<tr>
<td>batchContinueOnError</td>
<td>false</td>
<td>boolean</td>
<td>In SQL batch mode, force processing to continue if an error is found.</td>
</tr>
<tr>
<td>dba.gtidWaitTimeout</td>
<td>60</td>
<td>integer</td>
<td>The timeout in seconds to wait for GTID transactions to be applied, when required by AdminAPI operations (see Working with InnoDB Cluster).</td>
</tr>
<tr>
<td>defaultCompress</td>
<td>false</td>
<td>boolean</td>
<td>Enable compression for information sent between the client and the server in every global session (classic MySQL Protocol connections only).</td>
</tr>
<tr>
<td>defaultMode</td>
<td>None</td>
<td>string</td>
<td>The mode to use when MySQL Shell is started (SQL, Javascript or Python).</td>
</tr>
<tr>
<td>devapi.dbObjectHandles</td>
<td>true</td>
<td>boolean</td>
<td>Enable table and collection name handles for the DevAPI db object.</td>
</tr>
<tr>
<td>history.autoSave</td>
<td>false</td>
<td>boolean</td>
<td>Save (true) or clear (false) entries in the MySQL Shell code history when you exit MySQL Shell (see Section 5.4, “Code History”).</td>
</tr>
</tbody>
</table>
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

### Configuration Options

<table>
<thead>
<tr>
<th><code>optionName</code></th>
<th><code>DefaultValue</code></th>
<th><code>Type</code></th>
<th><code>Effect</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>history.maxSize</td>
<td>1000</td>
<td>integer</td>
<td>The maximum number of entries to store in the MySQL Shell code history.</td>
</tr>
<tr>
<td>history.sql.ignorePattern</td>
<td><em>IDENTIFIED</em> : <em>PASSWORD</em></td>
<td>string</td>
<td>Strings that match these patterns are not added to the MySQL Shell code history.</td>
</tr>
<tr>
<td>interactive</td>
<td>true</td>
<td>boolean (READ ONLY)</td>
<td>Enable interactive mode.</td>
</tr>
<tr>
<td>logLevel</td>
<td>5</td>
<td>integer ranging from 1 to 8</td>
<td>Enable the application log and set a logging level (see Section 8.1, &quot;Application Log&quot;).</td>
</tr>
<tr>
<td>pager</td>
<td>None</td>
<td>string</td>
<td>Use the specified external pager tool to display text and results. Command-line arguments for the tool can be added (see Section 4.5, &quot;Using a Pager&quot;).</td>
</tr>
<tr>
<td>passwordsFromStdin</td>
<td>false</td>
<td>boolean</td>
<td>Read passwords from stdin instead of terminal.</td>
</tr>
<tr>
<td>resultFormat</td>
<td>table</td>
<td>string (table, tabbed, vertical, json, json/raw)</td>
<td>The default output format for printing result sets (see Section 5.6, &quot;Output Formats&quot;).</td>
</tr>
<tr>
<td>sandboxDir</td>
<td>Depends on platform string</td>
<td>The sandbox directory. On Windows, the default is <code>C:\Users\MyUser\MySQL\mysql-sandboxes</code>, and on Unix systems, the default is <code>$HOME/mysql-sandboxes</code>.</td>
<td></td>
</tr>
<tr>
<td>showColumnTypeInfo</td>
<td>false</td>
<td>boolean</td>
<td>In SQL mode, display column metadata for result sets.</td>
</tr>
<tr>
<td>showWarnings</td>
<td>true</td>
<td>boolean</td>
<td>In SQL mode, automatically display SQL warnings if any.</td>
</tr>
<tr>
<td>useWizards</td>
<td>true</td>
<td>boolean</td>
<td>Enable wizard mode.</td>
</tr>
</tbody>
</table>

**Note**

String values are case sensitive.

The `outputFormat` option is now deprecated. Use `resultFormat` instead.
Using the `shell.options` Configuration Interface

- 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:

```plaintext
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:

```plaintext
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:

```plaintext
MySQL JS > shell.options.unset('history.maxSize')
MySQL JS > shell.options.unsetPersist('useWizards')
```

**Configuration File**

The MySQL Shell configuration file stores the values of the option to ensure they are persisted across sessions. Values are read at startup and when you use the persist feature, settings are saved to the configuration file.
The location of the configuration file is the user configuration path and the file is named *options.json*. Assuming that the default user configuration path has not been overridden by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`, the path to the configuration file is:

- **on Windows** `%APPDATA%\MySQL\mysqlsh`
- **on Unix** `~/.mysqlsh` where ~ represents the user’s home directory.

The configuration file is created the first time you customize a configuration option. This file is internally maintained by MySQL Shell and should not be edited manually. If an unrecognized option or an option with an incorrect value is found in the configuration file on startup, MySQL Shell exits with an error.
Appendix A MySQL Shell Command Reference

Table of Contents

A.1 mysqlsh — The MySQL Shell ........................................................................................................... 61

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 4.1, “MySQL Shell Connections” for more information.

mysqlsh supports the following command-line options.

Table A.1 mysqlsh Options

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Introduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>--auth-method</td>
<td>Authentication method to use</td>
<td></td>
</tr>
<tr>
<td>--cluster</td>
<td>Connect to an InnoDB cluster</td>
<td>8.0.4</td>
</tr>
<tr>
<td>--connect-timeout</td>
<td>Connection timeout for global session</td>
<td>8.0.13</td>
</tr>
<tr>
<td>--database</td>
<td>The schema to use (alias for --schema)</td>
<td></td>
</tr>
<tr>
<td>--dba</td>
<td>Enable X Protocol on connection with MySQL 5.7 server</td>
<td></td>
</tr>
<tr>
<td>--dbpassword</td>
<td>Password to use when connecting to server</td>
<td></td>
</tr>
<tr>
<td>--dbuser</td>
<td>MySQL user name to use when connecting to server</td>
<td></td>
</tr>
<tr>
<td>--execute</td>
<td>Execute the command and quit</td>
<td></td>
</tr>
<tr>
<td>--file</td>
<td>File to process in Batch mode</td>
<td></td>
</tr>
<tr>
<td>--force</td>
<td>Continue in SQL and Batch modes even if errors occur</td>
<td></td>
</tr>
<tr>
<td>--get-server-public-key</td>
<td>Request RSA public key from server</td>
<td></td>
</tr>
<tr>
<td>--help</td>
<td>Display help message and exit</td>
<td></td>
</tr>
<tr>
<td>--host</td>
<td>Connect to MySQL server on given host</td>
<td></td>
</tr>
<tr>
<td>--interactive</td>
<td>Emulate Interactive mode in Batch mode</td>
<td></td>
</tr>
<tr>
<td>--javascript, --js</td>
<td>Start in JavaScript mode</td>
<td></td>
</tr>
<tr>
<td>--json</td>
<td>Print output in JSON format</td>
<td></td>
</tr>
<tr>
<td>--log-level</td>
<td>Specify logging level</td>
<td></td>
</tr>
<tr>
<td>-ma</td>
<td>Detect session protocol automatically</td>
<td>8.0.3</td>
</tr>
<tr>
<td>--mysql, -mc</td>
<td>Create a session using MySQL protocol</td>
<td>8.0.3</td>
</tr>
<tr>
<td>--mysqlx, -mx</td>
<td>Create a session using X Protocol</td>
<td>8.0.3</td>
</tr>
<tr>
<td>--name-cache</td>
<td>Enable automatic loading of table names based on the active default schema</td>
<td>8.0.4</td>
</tr>
<tr>
<td>--no-name-cache</td>
<td>Disable autocompletion.</td>
<td>8.0.4</td>
</tr>
<tr>
<td>--no-password</td>
<td>No password is provided for this connection</td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
<td>Introduced</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>--no-wizard, --nw</td>
<td>Disable the connection wizard</td>
<td></td>
</tr>
<tr>
<td>--password</td>
<td>Password to use when connecting to server (alias for --dbpassword)</td>
<td></td>
</tr>
<tr>
<td>--passwords-from-stdin</td>
<td>Read the password from stdin</td>
<td></td>
</tr>
<tr>
<td>--port</td>
<td>TCP/IP port number for connection</td>
<td></td>
</tr>
<tr>
<td>--py, --python</td>
<td>Start in Python mode</td>
<td></td>
</tr>
<tr>
<td>--recreate-schema</td>
<td>Drop and recreate schema</td>
<td></td>
</tr>
<tr>
<td>--redirect-primary</td>
<td>Ensure connection to an InnoDB cluster's primary</td>
<td>8.0.4</td>
</tr>
<tr>
<td>--redirect-secondary</td>
<td>Ensure connection to an InnoDB cluster's secondary</td>
<td></td>
</tr>
<tr>
<td>--schema</td>
<td>The schema to use</td>
<td></td>
</tr>
<tr>
<td>--server-public-key-path</td>
<td>Path name to file containing RSA public key</td>
<td></td>
</tr>
<tr>
<td>--show-warnings</td>
<td>Show warnings after each statement if there are any (in SQL mode)</td>
<td></td>
</tr>
<tr>
<td>--socket</td>
<td>For connections to localhost, the Unix socket file to use, or pipe name on Windows (ClassicSessions only)</td>
<td></td>
</tr>
<tr>
<td>--sql</td>
<td>Start in SQL mode, auto-detecting protocol to use for connection</td>
<td></td>
</tr>
<tr>
<td>--sqlc</td>
<td>Start in SQL mode using a ClassicSession</td>
<td></td>
</tr>
<tr>
<td>--sqlx</td>
<td>Start in SQL mode using an X Protocol connection</td>
<td>8.0.3</td>
</tr>
<tr>
<td>--ssl-ca</td>
<td>File that contains list of trusted SSL Certificate Authorities</td>
<td></td>
</tr>
<tr>
<td>--ssl-capath</td>
<td>Directory that contains trusted SSL Certificate Authority certificate files</td>
<td></td>
</tr>
<tr>
<td>--ssl-cert</td>
<td>File that contains X.509 certificate</td>
<td></td>
</tr>
<tr>
<td>--ssl-cipher</td>
<td>Name of the SSL cipher to use</td>
<td></td>
</tr>
<tr>
<td>--ssl-crl</td>
<td>File that contains certificate revocation lists</td>
<td></td>
</tr>
<tr>
<td>--ssl-crlpath</td>
<td>Directory that contains certificate revocation-list files</td>
<td></td>
</tr>
<tr>
<td>--ssl-key</td>
<td>File that contains X.509 key</td>
<td></td>
</tr>
<tr>
<td>--ssl-mode</td>
<td>Security state of connection to server</td>
<td></td>
</tr>
<tr>
<td>--tabbed</td>
<td>Display output in tab separated format</td>
<td></td>
</tr>
<tr>
<td>--table</td>
<td>Display output in table format</td>
<td></td>
</tr>
<tr>
<td>--tls-version</td>
<td>Protocols permitted for encrypted connections</td>
<td></td>
</tr>
<tr>
<td>--uri</td>
<td>Session information in URI format</td>
<td></td>
</tr>
<tr>
<td>--user</td>
<td>MySQL user name to use when connecting to server (alias for --dbuser)</td>
<td></td>
</tr>
<tr>
<td>--version</td>
<td>Display version information and exit</td>
<td></td>
</tr>
<tr>
<td>--vertical</td>
<td>Display all SQL results vertically</td>
<td></td>
</tr>
</tbody>
</table>

- **--help**

Display a help message and exit.

- **--auth-method=method**

Authentication method to use for the account. Depends on the authentication plugin used for the account's password. MySQL Shell currently supports the following methods:
• `mysql_native_password` - see Native Pluggable Authentication
• `mysql_old_password` - see Old Native Pluggable Authentication
• `sha256_password` - see Caching SHA-2 Pluggable Authentication

• `--cluster`
Ensures that the target server is part of an InnoDB cluster and if so, sets the `cluster` global variable to the cluster object.

• `--column-type-info`
In SQL mode, before printing the returned result set for a query, print metadata for each column in the result set, such as the column type and collation.

The column type is returned as both the type used by MySQL Shell (`Type`), and the type used by the original database (`DBType`). For MySQL Shell connections using classic MySQL protocol, `DBType` is as returned by the protocol, and for X Protocol connections, `DBType` is inferred from the available information. The column length (`Length`) is returned in bytes.

• `--compress, -C`
Enable compression for information sent between the client and the server if both support compression. This option is available for classic MySQL Protocol connections only.

• `--connect-timeout`
Configures how long MySQL Shell waits to establish a global session specified through command-line arguments.

• `--database=name`
The default schema to use. This is an alias for `--schema`.

• `--dba=enableXProtocol`
Enable X Plugin on connection with 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.

• `--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.
• **--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**.

**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]**

Emulate Interactive mode in Batch mode.

• **--js**

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**

Specify the logging level. The value can be either an integer in the range from 1 to 8, or one of none, internal, error, warning, info, debug, debug2, or debug3. See Chapter 8, MySQL Shell Application Log.

• **-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 MySQL protocol if X Protocol is unavailable.

• **--mysql(--mc)**

Sets the session created at start up to use a classic MySQL protocol; connection. The **--mc** option with two hyphens replaced the previous **--mc** option in version 8.0.13.

• **--mysqlx(--mx)**

Sets the session created at start up to use an X Protocol connection. The **--mx** option with two hyphens replaced the previous single hyphen **--mx** option in version 8.0.13.
• **--name-cache**

Enable automatic loading of table names based on the active default schema.

• **--no-name-cache**

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**

Disables the connection wizard which provides help when creating connections.

• **--passwords-from-stdin**

Read the password from standard input, rather than from the terminal. This option does not affect any other password behaviors, such as the password prompt.

• **--password[=password], -p password**

The password to use when connecting to the server. The maximum password length that is accepted for connecting to MySQL Shell is 128 characters.

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

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

  • **--password** with no value and no equals 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**

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 schema.

- **--redirect-primary**

  Ensures that the target server is part of an InnoDB cluster and if it is not a primary, finds the cluster's primary and connects to it. MySQL Shell exits with an error if any of the following is true when using this option:

  - Group Replication is not active
  - InnoDB cluster metadata does not exist
  - There is no quorum

- **--redirect-secondary**

  Ensures that the target server is part of an InnoDB cluster 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:

  - 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 in the cluster, for example because there is just one server instance

- **--result-format={table|tabbed|vertical|json|json/raw}**

  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**
    
    Produces pretty-printed JSON.

  - **json/raw**
    
    Produces raw JSON.

  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.
• \texttt{--schema=name, -D name}

The default schema to use.

• \texttt{--server-public-key-path=file_name}

MySQL Shell equivalent of \texttt{--server-public-key-path}.

\begin{itemize}
\item \textbf{Important}
\end{itemize}

Only supported with classic MySQL protocol connections.

See \texttt{caching_sha2_password} plugin \texttt{Caching SHA-2 Pluggable Authentication}.

• \texttt{--show-warnings}

Cause warnings to be shown after each statement if there are any.

• \texttt{--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 \texttt{--socket} with no value and no equals sign, or \texttt{-S} without a value, the default Unix socket file for the 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 \texttt{--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 \texttt{localhost} on Unix or a period (.) on Windows.

• \texttt{--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 \texttt{--sqlx} or \texttt{--sqlc} options. Alternatively specify a protocol to use as part of a URI type string or use the \texttt{--port} option. See Section 4.1, “MySQL Shell Connections” and MySQL Shell Ports Reference for more information.

• \texttt{--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 4.1, “MySQL Shell Connections” and MySQL Shell Ports Reference for more information.

• \texttt{--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 4.1, “MySQL Shell Connections” and MySQL Shell Ports Reference for more information.

• \texttt{--ssl*}
Options that begin with \texttt{--ssl} specify whether to connect to the server using SSL and indicate where to find SSL keys and certificates. The \texttt{mysqlsh} SSL options function in the same way as the SSL options for MySQL Server, see \texttt{Command Options for Encrypted Connections} for more information.

\texttt{mysqlsh} accepts these SSL options: \texttt{--ssl-mode}, \texttt{--ssl-ca}, \texttt{--ssl-capath}, \texttt{--ssl-cert}, \texttt{--ssl-cipher}, \texttt{--ssl-crl}, \texttt{--ssl-crlpath}, \texttt{--ssl-key}, \texttt{--tls-version}.

- \texttt{--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 \texttt{--result-format=tabbed} option.

- \texttt{--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 \texttt{--result-format=table} option.

- \texttt{--uri=\texttt{str}}

Create a connection upon startup, specifying the connection options in a URI string format, see \texttt{Connecting Using a URI or Key-Value Pairs}.

- \texttt{--user=\texttt{user_name}}, \texttt{-u \texttt{user_name}}

The MySQL user name to use when connecting to the server.

- \texttt{--version}, \texttt{-V}

Display version information and exit.

- \texttt{--vertical}, \texttt{-E}

Display results vertically, as when the \texttt{\G} terminator is used for an SQL query. This option is an alias of the \texttt{--result-format=vertical} option.