

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: 2019-02-19 (revision: 60974)

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 OS X	5
2.4 Starting MySQL Shell	5
3 Getting Started with MySQL Shell	7
3.1 MySQL Shell Connections	7
3.1.1 Connecting using Individual Parameters	9
3.1.2 Using Encrypted Connections	11
3.1.3 Connections in JavaScript and Python	11
3.2 Pluggable Password Store	12
3.2.1 Pluggable Password Configuration Options	13
3.2.2 Working with Credentials	14
3.3 MySQL Shell Sessions	15
3.3.1 MySQL Shell Sessions Explained	15
3.4 MySQL Shell Global Variables	16
3.5 Using a Pager	17
4 MySQL Shell Code Execution	19
4.1 Active Language	19
4.2 Interactive Code Execution	19
4.3 Code Autocompletion	21
4.4 MySQL Shell Code History	23
4.5 Batch Code Execution	24
4.6 Output Formats	25
4.6.1 Table Format	25
4.6.2 Tab Separated Format	26
4.6.3 Vertical Format	26
4.6.4 JSON Format Output	26
4.6.5 JSON Wrapping	27
4.6.6 Result Metadata	29
4.7 API Command Line Interface	29
5 Configuring MySQL Shell	33
5.1 MySQL Shell Commands	33
6 MySQL Shell Utilities	37
6.1 Upgrade Checker Utility	37
6.2 JSON Import Utility	43
6.2.1 Importing JSON documents with the mysqlsh command interface	45
6.2.2 Importing JSON documents with the --import command	46
6.2.3 Conversions for representations of BSON data types	47
7 MySQL Shell Application Log	49
7.1 Application Log	49
8 Customizing MySQL Shell	51
8.1 Working With Start-Up Scripts	51
8.2 Adding Module Search Paths	52
8.2.1 Environment Variables	52
8.2.2 Startup Scripts	52
8.3 Customizing the Prompt	53
8.4 Configuring MySQL Shell	53
A MySQL Shell Command Reference	57
A.1 <code>mysqlsh</code> — The MySQL Shell	57

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.

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 5.1, “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 non-interactive 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 4.5, “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 4.6, “Output Formats”](#).

Multiple-line Support

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](#).

Application Log

MySQL Shell can be configured to log information about the execution process. For more information see [Chapter 7, 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 3.3, "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 OS X	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:
 - Follow the steps given in [Adding the MySQL APT Repository](#), paying special attention to the following:
 - During the installation of the configuration package, when asked in the dialogue box to configure the repository, make sure you choose MySQL 8.0 as the release series you want.
 - Make sure you do not skip the step for updating package information for the MySQL APT repository:

```
sudo apt-get update
```

- Install MySQL Shell with this command:

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

- If you already have the [MySQL APT repository](#) as a software repository on your system, do the following:
 - Update package information for the MySQL APT repository:

```
sudo apt-get update
```

- Update the MySQL APT repository configuration package with the following command:

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

When asked in the dialogue box to configure the repository, make sure you choose MySQL 8.0 as the release series you want.

- Install MySQL Shell with this command:

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

Installing MySQL Shell with the MySQL Yum Repository

For Linux distributions supported by the [MySQL Yum repository](#), follow these steps to install MySQL Shell:

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

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

For dnf-enabled systems, do this instead:

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


Then, follow the steps given in [Adding the MySQL Yum Repository](#) to install the new release package, `mysql80-community-release`.

- If you do not yet have the [MySQL Yum repository](#) as a software repository on your system, follow the steps given in [Adding the MySQL Yum Repository](#).
- Install MySQL Shell with this command:

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

For dnf-enabled systems, do this instead:

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

Installing MySQL Shell from Direct Downloads from the MySQL Developer Zone

RPM, Debian, and source packages for installing MySQL Shell are also available for download at [Download MySQL Shell](#).

2.3 Installing MySQL Shell on OS X

To install MySQL Shell on OS X, do the following:

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

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 Getting Started with MySQL Shell

Table of Contents

3.1 MySQL Shell Connections	7
3.1.1 Connecting using Individual Parameters	9
3.1.2 Using Encrypted Connections	11
3.1.3 Connections in JavaScript and Python	11
3.2 Pluggable Password Store	12
3.2.1 Pluggable Password Configuration Options	13
3.2.2 Working with Credentials	14
3.3 MySQL Shell Sessions	15
3.3.1 MySQL Shell Sessions Explained	15
3.4 MySQL Shell Global Variables	16
3.5 Using a Pager	17

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

3.1 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 3.1.1, “Connecting using Individual Parameters”](#).
- When MySQL Shell is running using the `\connect` command. See [Section 5.1, “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 3.2, “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:

- 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:

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

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

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

```
mysql-js> \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.

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

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

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



Important

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

For more information on connecting with Unix socket files and Windows named pipes, see [Connecting to the MySQL Server](#) and [Connecting Using a URI or Key-Value Pairs](#).

3.1.1 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 (-ppassword)` 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
```

3.1.2 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=name`: 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.

3.1.3 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.getClassicSession(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:

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

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

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

3.1.3.1 Using Encrypted Connections in Code

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

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

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

3.2 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:

```
mysql-js> \connect user@localhost:3310
Creating a session to 'user@localhost:3310'
Please provide the password for 'user@localhost:3310': *****
```



```
Save password for 'user@localhost:3310'? [Y]es/[N]o/[Ne[v]er (default No): y
```

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



Important

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

Passwords are only persisted when they are entered manually. If a password is provided using either a server URI 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 non-interactive 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

3.2.1 Pluggable Password Configuration Options

To configure the pluggable password store, use the `shell.options` interface, see [Section 8.4, “Configuring MySQL Shell”](#). The following options configure the pluggable password store.

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

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

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

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

shell.options.credentialStore.savePasswords = "value"

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

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

The default value for this option is `prompt`.

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

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

The default value for this option is an empty list.

3.2.2 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`).

3.3 MySQL Shell Sessions

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

3.3.1 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 URI` 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, `-mx` and `-mc`) is also deprecated from version 8.0.13 of MySQL Shell.

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

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

3.4 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:

- `session` represents the global session if one has been established.
- `db` represents a schema if one has been defined, for example by a URI type string.
- `dba` represents the AdminAPI, a component of InnoDB cluster which enables you to administer clusters of server instances. See [InnoDB Cluster](#).
- `shell` provides general purpose functions, for example to configure MySQL Shell.
- `util` provides utility functions, for example to check server instances before an upgrade.



Important

These words are reserved and cannot be used, for example as names of variables.

3.5 Using a Pager

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

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

Default value: empty string.

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

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

- Configure the `--pager` MySQL Shell option, which overrides the initial value of `shell.options["pager"]` option even if it was persisted and `PAGER` environment variable is configured.
- Use the `\pager | \P command` MySQL Shell command to set the value of `shell.options["pager"]` option. If called with no arguments, restores the initial value of `shell.options["pager"]` option (the one MySQL Shell had at startup. Strings can be marked with `"` characters or not. For example, to configure the pager:
 - pass in no `command` or an empty string to restore the initial pager
 - pass in `more` to configure MySQL Shell to use the `more` command as the pager
 - pass in `more -10` to configure MySQL Shell to use the `more` command as the pager with the option `-10`

The MySQL Shell output that is passed to the external pager tool is forwarded with no filtering. If MySQL Shell is using a prompt with color (see [Section 8.3, “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 4 MySQL Shell Code Execution

Table of Contents

4.1 Active Language	19
4.2 Interactive Code Execution	19
4.3 Code Autocompletion	21
4.4 MySQL Shell Code History	23
4.5 Batch Code Execution	24
4.6 Output Formats	25
4.6.1 Table Format	25
4.6.2 Tab Separated Format	26
4.6.3 Vertical Format	26
4.6.4 JSON Format Output	26
4.6.5 JSON Wrapping	27
4.6.6 Result Metadata	29
4.7 API Command Line Interface	29

This section explains how code execution works in MySQL Shell.

4.1 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
```

4.2 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 4.1, "Active Language"](#). When executed, the results of the operation are displayed on-screen.

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

```
var mySession = mysqlx.getSession('user:pwd@localhost');
var result = mySession.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()`
 - `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:

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

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

```
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:

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

4.3 Code Autocompletion

MySQL Shell supports autocompletion of text preceding the cursor by pressing the **Tab** key. The [Section 5.1, “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 4.1, “Active Language”](#).

Autocompletion supports the following text objects:

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

By default autocompletion is enabled, to change this behavior see [Configuring Autocompletion](#).

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

Autocompleting SQL

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

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

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

Autocompleting JavaScript and Python

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

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

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

Configuring Autocompletion

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

To disable this behavior you can:

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

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

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

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

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

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

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

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

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

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

4.4 MySQL Shell 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 `shell.options["history.maxSize"]=number` 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.

History entries are saved to the `~/mysqlsh/history` file on Linux and Mac, or the `%AppData%\MySQL\mysqlsh\history` file on 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 history file 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.

Issuing the MySQL Shell command `\history` 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 `shell.options["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. 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.

4.5 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 4.7, “API Command Line Interface”](#).

In batch mode, all the command logic described at [Section 4.2, “Interactive Code Execution”](#) is not available, only valid code for the active language can be executed. When processing SQL code, it is executed statement by statement using the following logic: read/process/print result. When processing non-SQL code, it is loaded entirely from the input source and executed as a unit. Use the `--interactive` (or `-i`) command-line option to configure MySQL Shell to process the input source as if it were being issued in interactive mode; this enables all the features provided by the Interactive mode to be used in batch processing.



Note

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

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

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

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

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

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

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

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

Executable Scripts

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

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

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

4.6 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 8.4, “Configuring MySQL Shell”](#). 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.

4.6.1 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`.

```
mysql-sql> select * from sakila.actor limit 3;
+-----+-----+-----+-----+
| actor_id | first_name | last_name | last_update |
+-----+-----+-----+-----+
| 1 | PENELOPE | GUINNESS | 2006-02-15 4:34:33 |
| 2 | NICK | WAHLBERG | 2006-02-15 4:34:33 |
| 3 | ED | CHASE | 2006-02-15 4:34:33 |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql-sql>
```

4.6.2 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`.

```
>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             PENELOPE     GUINNESS     2006-02-15 4:34:33
2             NICK         WAHLBERG     2006-02-15 4:34:33
3             ED           CHASE        2006-02-15 4:34:33
```

4.6.3 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`.

4.6.4 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 4.6.5, “JSON Wrapping”](#).

JSON Format in Interactive Mode (Pretty-Printed JSON)

```
mysql-sql> select * from sakila.actor limit 3;
{
  "duration": "0.00 sec",
```

```

"info": "",
"row_count": 3,
"rows": [
  [
    1,
    "PENELOPE",
    "GUINNESS",
    {
      "year": 2006,
      "month": 1,
      "day": 15,
      "hour": 4,
      "minute": 34,
      "second": 33.0
    }
  ],
  [
    2,
    "NICK",
    "WAHLBERG",
    {
      "year": 2006,
      "month": 1,
      "day": 15,
      "hour": 4,
      "minute": 34,
      "second": 33.0
    }
  ],
  [
    3,
    "ED",
    "CHASE",
    {
      "year": 2006,
      "month": 1,
      "day": 15,
      "hour": 4,
      "minute": 34,
      "second": 33.0
    }
  ]
],
"warning_count": 0
}
mysql-sql>

```

JSON Format in Interactive Mode (Raw JSON)

```

mysql-sql> select * from sakila.actor limit 3;
{"duration":"0.00 sec","info":"","row_count":3,"rows":[[1,"PENELOPE","GUINNESS",{"year":2006,"month":1,"day":15,"hour":4,"minute":34,"second":33.0}],["NICK","WAHLBERG",{"year":2006,"month":1,"day":15,"hour":4,"minute":34,"second":33.0}],["ED","CHASE",{"year":2006,"month":1,"day":15,"hour":4,"minute":34,"second":33.0}]]]
mysql-sql>

```

4.6.5 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.

MySQL Shell Output with JSON Wrapping

```
shell>echo "select * from sakila.actor limit 3;" | mysqlsh --json=raw --sqlc --uri user@192.0.2.20:3306
{"duration":"0.00 sec","info":"","row_count":3,"rows":[[1,"PENELOPE","GUINNESS",{"year":2006,"month":1,"day":15}]]}

shell>echo "select * from sakila.actor limit 3;" | mysqlsh --json --sqlc --uri user@192.0.2.20:3306
or
shell>echo "select * from sakila.actor limit 3;" | mysqlsh --json=pretty --sqlc --uri user@192.0.2.20:3306
{
  "duration": "0.00 sec",
  "info": "",
  "row_count": 3,
  "rows": [
    [
      1,
      "PENELOPE",
      "GUINNESS",
      {
        "year": 2006,
        "month": 1,
        "day": 15,
        "hour": 4,
        "minute": 34,
        "second": 33.0
      }
    ],
    [
      2,
      "NICK",
      "WAHLBERG",
      {
        "year": 2006,
        "month": 1,
        "day": 15,
        "hour": 4,
        "minute": 34,
        "second": 33.0
      }
    ],
    [
      3,
      "ED",
      "CHASE",
      {
        "year": 2006,
        "month": 1,
        "day": 15,
        "hour": 4,
        "minute": 34,
        "second": 33.0
      }
    ]
  ],
  "warning_count": 0
}
shell>
```


4.6.6 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.

4.7 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:

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

This maps to the equivalent command in MySQL Shell:

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

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

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

This maps to the equivalent command in MySQL Shell:

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

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

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

This maps to the equivalent command in MySQL Shell:

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

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

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

This maps to the equivalent command in MySQL Shell:

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

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

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

This maps to the equivalent command in MySQL Shell:

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

Chapter 5 Configuring MySQL Shell

Table of Contents

5.1 MySQL Shell Commands 33

This section describes the commands which configure MySQL Shell from the interactive code editor. For information on the `mysqlsh` command options see [Appendix A, MySQL Shell Command Reference](#).

5.1 MySQL Shell Commands

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

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

Help Command

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

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

- [AdminAPI](#) - 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 `?` to match any single character and `*` to match multiple characters in a search. The wildcard characters can be used one or more times in the pattern. The following namespaces can also be used when searching for help:

- `dba` for AdminAPI
- `mysqlx` for X DevAPI
- `mysql` for ShellAPI for classic 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 a single topic matches the pattern searched for, the help is displayed. If multiple topics match the pattern searched for a list of possible help topics is displayed, and you can view the topic using the search pattern displayed.

Connect and Reconnect Commands

The `\connect` command is used to connect to a MySQL Server using an URI type string. See [Connecting using a URI String](#).

For example:

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

If a password is required you are prompted for it.

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

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

Use the `--mysql` (`--mc`) option to create a ClassicSession, enabling you to use 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 4.4, “MySQL Shell 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 4.3, “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 8.4, “Configuring MySQL Shell”](#).

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.

Chapter 6 MySQL Shell Utilities

Table of Contents

6.1 Upgrade Checker Utility	37
6.2 JSON Import Utility	43
6.2.1 Importing JSON documents with the mysqlsh command interface	45
6.2.2 Importing JSON documents with the --import command	46
6.2.3 Conversions for representations of BSON data types	47

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

<code>checkForServerUpgrade()</code>	An upgrade checker utility that enables you to verify whether MySQL server instances are ready for upgrade.
<code>importJSON()</code>	A JSON import utility that enables you to import JSON documents to a MySQL Server collection or table.

6.1 Upgrade Checker Utility

The `util.checkForServerUpgrade()` function is an upgrade checker utility that enables you to verify whether MySQL server instances are ready for upgrade. From MySQL Shell 8.0.13, you can select a target MySQL Server release to which you plan to upgrade, ranging from the first MySQL Server 8.0 General Availability (GA) release (8.0.11), up to the MySQL Server release number that matches the current MySQL Shell release number. The upgrade checker utility carries out the automated checks that are relevant for the specified target release, and advises you of further relevant checks that you should make manually.

You can use the upgrade checker utility to check MySQL 5.7 server instances for compatibility errors and issues for upgrading. From MySQL Shell 8.0.13, you can also use it to check MySQL 8.0 server instances at another GA status release within the MySQL 8.0 release series. If you invoke `checkForServerUpgrade()` without specifying a MySQL Server instance, the instance currently connected to the global session is checked. To see the currently connected instance, issue the `\status` command.



Note

1. The upgrade checker utility does not support checking MySQL Server instances at a version earlier than 5.7.
2. MySQL Server only supports upgrade between GA releases. Upgrades from non-GA releases of MySQL 5.7 or 8.0 are not supported. For more information on supported upgrade paths, see [Upgrade Paths](#).

The upgrade checker utility can 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:

```
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:

<code>password</code>	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.
<code>targetVersion</code>	The target MySQL Server version to which you plan to upgrade. In MySQL Shell 8.0.14, you can specify release 8.0.11 (the first MySQL Server 8.0 GA release), 8.0.12, 8.0.13, or 8.0.14. If you specify the short form version number 8.0, or omit the <code>targetVersion</code> option, the utility checks for upgrade to the MySQL Server release number that matches the current MySQL Shell release number.
<code>outputFormat</code>	The format in which the output from the upgrade checker utility is returned. The default if you omit the option is text format (<code>TEXT</code>). If you specify <code>JSON</code> , well-formatted JSON output is returned instead, in the format listed in JSON output for the upgrade checker utility . Do not specify the command line option <code>--json</code> when you start MySQL Shell to use the upgrade checker utility, as that option does not produce the output in the correct format.

For example, the following commands verify then check the MySQL server instance currently connected to the global session, with output in text format:

```
mysqlsh> \status
MySQL Shell version 8.0.14
...
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 is supplied as part of the options dictionary, and the output is returned in the default text format:

```
mysqlsh> util.checkForServerUpgrade('user@example.com:3306', {"password":"password", "targetVersion":"8.0.11"})
```

The following command checks the same MySQL server for upgrade to the MySQL Server release number that matches the current MySQL Shell release number (the default), and returns JSON output for further processing:

```
mysqlsh> util.checkForServerUpgrade('user@example.com:3306', {"password":"password", "outputFormat":"JSON"})
```

From MySQL 8.0.13, you can start the upgrade checker utility from the command line using the `mysqlsh` command interface. For information on this syntax, see [Section 4.7, “API Command Line Interface”](#). The following example checks a MySQL server for upgrade to release 8.0.13, and returns JSON output:

```
mysqlsh -- util checkForServerUpgrade user@localhost:3306 --target-version=8.0.13 --output-format=JSON
```

The connection data can also be specified as named options grouped together by using curly brackets, as in the following example, which also shows that lower case and hyphens can be used for the method name rather than camelCase:

```
mysqlsh -- util check-for-server-upgrade { --user=user --host=localhost --port=3306 } --target-version=8.0
```

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

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

To get help for the upgrade checker utility, issue:

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

`util.checkForServerUpgrade()` no longer returns a value (before MySQL Shell 8.0.13, the value 0, 1, or 2 was returned).

When you invoke the upgrade checker utility, MySQL Shell connects to the server instance and tests the settings described at [Preparing Your Installation for Upgrade](#). For example:

```
The MySQL server at example.com:3306, version
5.7.25-enterprise-commercial-advanced - MySQL Enterprise Server - Advanced Edition (Commercial),
will now be checked for compatibility issues for upgrade to MySQL 8.0.14...

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 error.
   More information: https://dev.mysql.com/doc/refman/en/keywords.html

   dbtest.System - Table name
   dbtest.System.JSON_TABLE - Column name
   dbtest.System.cube - Column name

3) Usage of utf8mb3 charset
   Warning: The following objects use the utf8mb3 character set. It is recommended to convert them to use
   utf8mb4 instead, for improved Unicode support.
   More information: https://dev.mysql.com/doc/refman/8.0/en/charset-unicode-utf8mb3.html

   dbtest.view1.coll - 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);'
More information: https://dev.mysql.com/doc/refman/8.0/en/mysql-nutshell.html#mysql-nutshell-removals

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

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.
More information: https://dev.mysql.com/doc/relnotes/mysql/8.0/en/news-8-0-13.html#mysqld-8-0-13-sql-syntax

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
ERROR: This check requires server to be at minimum at 8.0.11 version

More information:
https://dev.mysql.com/doc/relnotes/mysql/8.0/en/news-8-0-13.html#mysqld-8-0-13-logging

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

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

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

[mysqld]
default_authentication_plugin=mysql_native_password

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

Errors: 4
Warnings: 5
Notices: 0

4 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.
- This example also shows an error message marked with **ERROR** in capitals which was returned because a check did not run successfully. Such error messages are not included in the count of errors at

the end of the report. If you see any messages of this type, investigate why the check failed to run, and if necessary, manually verify that there are no issues for upgrade relating to the checked item.

- 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:

<code>serverAddress</code>	Host name and port number for MySQL Shell's connection to the MySQL server instance that was checked.												
<code>serverVersion</code>	Detected MySQL version of the server instance that was checked.												
<code>targetVersion</code>	Target MySQL version for the upgrade checks.												
<code>errorCount</code>	Number of errors found by the utility.												
<code>warningCount</code>	Number of warnings found by the utility.												
<code>noticeCount</code>	Number of notices found by the utility.												
<code>summary</code>	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.").												
<code>checksPerformed</code>	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: <table> <tr> <td><code>id</code></td> <td>The ID of the check, which is a unique string.</td> </tr> <tr> <td><code>title</code></td> <td>A short description of the check.</td> </tr> <tr> <td><code>status</code></td> <td>"OK" if the check ran successfully, "ERROR" otherwise.</td> </tr> <tr> <td><code>description</code></td> <td>A long description of the check (if available) incorporating advice, or an error message if the check failed to run.</td> </tr> <tr> <td><code>documentationLink</code></td> <td>If available, a link to documentation with further information or advice.</td> </tr> <tr> <td><code>detectedProblems</code></td> <td>An array (which might be empty) of JSON objects representing the errors, warnings, or notices that were found as a result of the check. Each</td> </tr> </table>	<code>id</code>	The ID of the check, which is a unique string.	<code>title</code>	A short description of the check.	<code>status</code>	"OK" if the check ran successfully, "ERROR" otherwise.	<code>description</code>	A long description of the check (if available) incorporating advice, or an error message if the check failed to run.	<code>documentationLink</code>	If available, a link to documentation with further information or advice.	<code>detectedProblems</code>	An array (which might be empty) of JSON objects representing the errors, warnings, or notices that were found as a result of the check. Each
<code>id</code>	The ID of the check, which is a unique string.												
<code>title</code>	A short description of the check.												
<code>status</code>	"OK" if the check ran successfully, "ERROR" otherwise.												
<code>description</code>	A long description of the check (if available) incorporating advice, or an error message if the check failed to run.												
<code>documentationLink</code>	If available, a link to documentation with further information or advice.												
<code>detectedProblems</code>	An array (which might be empty) of JSON objects representing the errors, warnings, or notices that were found as a result of the check. Each												

JSON object has the following key-value pairs:

level	The message level, one of Error, Warning, or Notice.
dbObject	A string identifying the database object to which the message relates.
description	If available, a string with a further description of the issue.

manualChecks

An array of JSON objects, one for each individual upgrade issue that is relevant to your upgrade path and needs to be checked manually (for example, the change of default authentication plugin in MySQL 8.0). Each JSON object has the following key-value pairs:

id	The ID of the manual check, which is a unique string.
title	A short description of the manual check.
level	The message level, either Warning or Notice.
description	A long description of the manual check, with information and advice.

documentationLink

If available, a link to documentation with further information or advice.

6.2 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 standard input to a MySQL Server collection or relational table. The utility checks that the supplied JSON documents are well-formed and inserts them into the target database, removing the need to use multiple `INSERT` statements or write scripts to achieve this task.

From MySQL Shell 8.0.14, the import utility can process BSON (binary JSON) data types that are represented in JSON documents. The data types used in BSON documents are not all natively supported by JSON, but can be represented using extensions to the JSON format. The import utility can process documents that use JSON extensions to represent BSON data types, convert them to an identical or compatible MySQL representation, and import the data value using that representation. The resulting converted data values can be used in expressions and indexes, and manipulated by SQL statements and X DevAPI functions.

You can import the JSON documents to an existing table or collection or to a new one created for the import. If the target table or collection does not exist in the specified database, it is automatically created by the utility, using a default collection or table structure. The default collection is created by calling the `createCollection()` function from a `schema` object. The default table is created as follows:

```
CREATE TABLE `dbname`.`tablename` (
  target_column JSON,
  id INTEGER AUTO_INCREMENT PRIMARY KEY
) CHARSET utf8mb4 ENGINE=InnoDB;
```

The default collection name or table name is the name of the supplied import file (without the file extension), and the default `target_column` name is `doc`.

To convert JSON extensions for BSON types into MySQL types, you must specify the `convertBsonTypes` option when you run the import utility. Additional options are available to control the mapping and conversion for specific BSON data types. If you import documents with JSON extensions for BSON types and do not use this option, the documents are imported in the same way as they are represented in the input file.

The JSON import utility requires an existing X Protocol connection to the server. The utility cannot operate over a classic MySQL protocol connection.

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

```
importJSON (path, options)
```

`path` is a string specifying the file path for the file containing the JSON documents to be imported. (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.

<code>collection:</code> <code>"collection_name"</code>	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 <code>collection</code> , <code>table</code> , or <code>tableColumn</code> options, the utility defaults to using or creating a target collection with the name of the supplied import file (without the file extension).
<code>table: "table_name"</code>	The name of the target table. This is an alternative to specifying a collection. If the table does not exist, the utility creates it.
<code>tableColumn:</code> <code>"column_name"</code>	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 <code>table</code> option but omit the <code>tableColumn</code> option, the default column name <code>doc</code> is used. If you specify the <code>tableColumn</code> option but omit the <code>table</code> option, the name of the supplied import file (without the file extension) is used as the table name.
<code>convertBsonTypes: true</code>	Recognizes and converts BSON data types that are represented using extensions to the JSON format. The default for this option is <code>false</code> . When you specify <code>convertBsonTypes: true</code> , 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 6.2.3, “Conversions for representations of BSON data types” . The <code>convertBsonOid</code> option must also be set to <code>true</code> , which is that option's default setting when you specify <code>convertBsonTypes: true</code> . If you import documents with JSON extensions for BSON types and do not use <code>convertBsonTypes: true</code> , the documents are imported in the same way as they are represented in the input file, as embedded JSON documents.
<code>convertBsonOid: true</code>	Recognizes and converts MongoDB ObjectIDs, which are a 12-byte BSON type used as an <code>_id</code> value for documents, represented in MongoDB Extended JSON strict mode. The default for this option is the value of the <code>convertBsonTypes</code> option, so if that option is set to <code>true</code> , MongoDB ObjectIDs are automatically also converted. When importing data from MongoDB, <code>convertBsonOid</code> must always be set to <code>true</code> if you do not convert the BSON types, because MySQL Server requires the <code>_id</code> value to be converted to the <code>varbinary(32)</code> type.
<code>extractOidTime:</code> <code>"field_name"</code>	Recognizes and extracts the timestamp value that is contained in a MongoDB ObjectID in the <code>_id</code> field for a document, and places it into a separate field in the imported data. <code>extractOidTime</code> names the field in the document that contains the timestamp. The timestamp is the first 4 bytes of the ObjectID, which remains unchanged. <code>convertBsonOid: true</code> must be set to use this option, which is the default when <code>convertBsonTypes</code> is set to <code>true</code> .

The following examples import the JSON documents in the file `/tmp/products.json` to the `products` collection in the `mydb` database:

```
mysql-js> util.importJson("/tmp/products.json", {schema: "mydb", collection: "products"})
```

```
mysql-py> util.import_json("/tmp/products.json", {"schema": "mydb", "collection": "products"})
```


The following example has no options specified, so the dictionary is omitted. `mydb` is the active schema for the MySQL Shell session. The utility therefore imports the JSON documents in the file `/tmp/stores.json` to a collection named `stores` in the `mydb` database:

```
mysql-js> \use mydb
mysql-js> util.importJson("/tmp/stores.json")
```

The following example imports the JSON documents in the file `/europe/regions.json` to the column `jsondata` in a relational table named `regions` in the `mydb` database. BSON data types that are represented in the documents by JSON extensions are converted to a MySQL representation:

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

The following example carries out the same import but without converting the JSON representations of the BSON data types to MySQL representations. However, the MongoDB ObjectIDs in the documents are converted as required by MySQL, and their timestamps are also extracted:

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

When the import is complete, or if the import is stopped partway by the user with **Ctrl+C** or by an error, a message is returned to the user showing the number of successfully imported JSON documents, and any applicable error message. The function itself returns void, or an exception in case of an error.

The JSON import utility can also be invoked from the command line. Two alternative formats are available for the command line invocation. You can use the `mysqlsh` command interface, which accepts input only from a file, or the `--import` command, which accepts input from standard input or a file.

6.2.1 Importing JSON documents with the mysqlsh command interface

With the `mysqlsh` command interface, you invoke the JSON import utility as follows:

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

For information on this syntax, see [Section 4.7, “API Command Line Interface”](#). For the JSON import utility, specify the parameters as follows:

<code>user</code>	The user name for the user account that is used to run the JSON import utility.
<code>host</code>	The host name for the MySQL server.
<code>port</code>	The port number for MySQL Shell's connection to the MySQL server. The default port for this connection is 33060.
<code>mydb</code>	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 <code>--schema</code> command line option.
<code>path</code>	The file path for the file containing the JSON documents to be imported.
<code>options</code>	The <code>--collection</code> , <code>--table</code> , and <code>--tableColumn</code> options specify a target collection or a target table and column. The relationships and defaults when the JSON import utility is invoked using the <code>mysqlsh</code> 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 6.2.3, “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
```

6.2.2 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 6.2.3, “Conversions for representations of BSON data types”](#).

The following example reads JSON documents from standard input and imports them to a target named `territories` in the `mydb` database. If no collection or table named `territories` is found, the utility creates a collection named `territories` and imports the documents to it. If you want to create and import the documents to a relational table named `territories`, you must specify a column name as a further parameter.

```
mysqlsh user@localhost/mydb --import - territories
```

The following example with a file path and a target imports the JSON documents in the file `/europe/regions.json` to the column `jsondata` in a relational table named `regions` in the `mydb` database. The schema name is specified using the `--schema` command line option instead of in the connection URI:

```
mysqlsh user@localhost:33062 --import /europe/regions.json regions jsondata --schema=mydb
```

The following example with a file path but no target specified imports the JSON documents in the file `/europe/regions.json`. If no collection or table named `regions` (the name of the supplied import file without the extension) is found in the specified `mydb` database, the utility creates a collection named `regions` and imports the documents to it. If there is already a collection named `regions`, the utility imports the documents to it.

```
mysqlsh user@localhost/mydb --import /europe/regions.json
```

MySQL Shell returns a message confirming the parameters for the import, for example, `Importing from file "/europe/regions.json" to table `mydb`.`regions` in MySQL Server at 127.0.0.1:33062.`

When an import is complete, or if the import is stopped partway by the user with **Ctrl+C** or by an error, a message is returned to the user showing the number of successfully imported JSON documents, and any applicable error message. The process returns zero if the import finished successfully, or a nonzero exit code if there was an error.

6.2.3 Conversions for representations of BSON data types

When you specify the `convertBsonTypes: true` (`--convertBsonTypes`) option to convert BSON data types that are represented by JSON extensions, by default, the BSON types are imported as follows:

Date (“date”)	Simple value containing the value of the field.
Timestamp (“timestamp”)	MySQL timestamp created using the <code>time_t</code> value.
Decimal (“decimal”)	Simple value containing a string representation of the decimal value.
Integer (“int” or “long”)	Integer value.
Regular expression (“regex” plus options)	String containing the regular expression only, and ignoring the options. A warning is printed if options are present.
Binary data (“binData”)	Base64 string.
ObjectID (“objectId”)	Simple value containing the value of the field.

The following control options can be specified to adjust the mapping and conversion of these BSON types. `convertBsonTypes: true` (`--convertBsonTypes`) must be specified to use any of these control options:

<code>ignoreDate: true</code> (<code>--ignoreDate</code>)	Disable conversion of the BSON “date” type. The data is imported as an embedded JSON document exactly as in the input file.
<code>ignoreTimestamp: true</code> (<code>--ignoreTimestamp</code>)	Disable conversion of the BSON “timestamp” type. The data is imported as an embedded JSON document exactly as in the input file.
<code>decimalAsDouble: true</code> (<code>--decimalAsDouble</code>)	Convert the value of the BSON “decimal” type to the MySQL DOUBLE type, rather than a string.
<code>ignoreRegex: true</code> (<code>--ignoreRegex</code>)	Disable conversion of regular expressions (the BSON “regex” type). The data is imported as an embedded JSON document exactly as in the input file.
<code>ignoreRegexOptions: false</code> (<code>--ignoreRegexOptions=false</code>)	Include the options associated with a regular expression in the string, as well as the regular expression itself (in the format <code>/<code><regular expression></code>/<code><options></code></code>). By default, the options are ignored (<code>ignoreRegexOptions: true</code>), but a warning is printed if any options were present. <code>ignoreRegex</code> must be set to the default of <code>false</code> to specify <code>ignoreRegexOptions</code> .
<code>ignoreBinary: true</code> (<code>--ignoreBinary</code>)	Disable conversion of the BSON “binData” type. The data is imported as an embedded JSON document exactly as in the input file.

The following example imports documents from the file `/europe/regions.json` to the column `jsondata` in a relational table named `regions` in the `mydb` database. BSON data types that are represented by JSON extensions are converted to MySQL representations, with the exception of regular expressions, which are imported as embedded JSON documents:

```
mysqlsh user@localhost/mydb --import /europe/regions.json regions jsondata --convertBsonTypes --ignoreRegex
```

Chapter 7 MySQL Shell Application Log

Table of Contents

7.1 Application Log 49

This section explains the application log.

7.1 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:

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

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 8 Customizing MySQL Shell

Table of Contents

8.1 Working With Start-Up Scripts	51
8.2 Adding Module Search Paths	52
8.2.1 Environment Variables	52
8.2.2 Startup Scripts	52
8.3 Customizing the Prompt	53
8.4 Configuring MySQL Shell	53

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.

8.1 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.

8.2 Adding Module Search Paths

There are two ways to add additional module search paths:

- Through environment variables
- Through startup scripts

8.2.1 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.

8.2.2 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.

```
# 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.

```
// 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';
```

8.3 Customizing the Prompt

The prompt of MySQL Shell can be customized using prompt theme files. To customize the prompt theme file, either set the `MYSQLSH_PROMPT_THEME` environment variable to a prompt theme file name, or copy a theme file to the `~/ .mysqlsh/prompt.json` directory on Linux and Mac, or the `%AppData%\MySQL\mysqlsh\prompt.json` directory on Windows.

The user configuration path for the directory can be overridden on all platforms by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`. The value of this variable replaces `%AppData%\MySQL\mysqlsh\` on Windows or `~/ .mysqlsh/` on Unix.

The format of the prompt theme file is described in the `README.prompt` file, and some sample prompt theme files are included. On startup, if an error is found in the prompt theme file, an error message is printed and a default prompt theme is used. Some of the sample prompt theme files require a special font (for example `SourceCodePro+Powerline+Awesome+Regular.ttf`). If you set the `MYSQLSH_PROMPT_THEME` environment variable to an empty value, MySQL Shell uses a minimal prompt with no color.

Color display depends on the support available from the terminal. Most terminals support 256 colors in Linux and Mac. In Windows, color support requires either a 3rd party terminal program with support for ANSI/VT100 escapes, or Windows 10. By default, MySQL Shell attempts to detect the terminal type and handle colors appropriately. If auto-detection does not work for your terminal type, or if you want to modify the color mode due to accessibility requirements or for other purposes, you can define the environment variable `MYSQLSH_TERM_COLOR_MODE` to force MySQL Shell to use a specific color mode. The possible values for this environment variable are `rgb`, `256`, `16`, and `nocolor`.

8.4 Configuring MySQL Shell

You can configure MySQL Shell to match your preferences, for example to start up to a certain programming language or to provide output in a particular format. Configuration options can be set for the current session only, or options can be set permanently by persisting changes to the MySQL Shell configuration file. Online help for all options is provided. You can configure options using the MySQL Shell `\option` command, which is available in all MySQL Shell modes for querying and changing configuration options. Alternatively in JavaScript and Python modes, use the `shell.options` object.

Valid Configuration Options

The following configuration options can be set using either the `\option` command or `shell.options` scripting interface:

optionName	DefaultValue	Type	Effect
<code>autocomplete.nameCache</code>	<code>true</code>	boolean	Enable database name caching for autocomplete.
<code>batchContinueOnError</code>	<code>false</code>	boolean (READ ONLY)	In SQL batch mode, force processing to continue if an error is found.
<code>dba.gtidWaitTimeout</code>	60	integer greater than 0	The timeout in seconds to wait for GTID transactions to be applied, when required by AdminAPI operations (see Working with InnoDB Cluster).
<code>defaultCompress</code>	<code>false</code>	boolean	Enable compression for information sent between the client and the server in every global session (classic MySQL Protocol connections only).

optionName	DefaultValue	Type	Effect
defaultMode	None	string (sql, js or py)	The mode to use when MySQL Shell is started (SQL, Javascript or Python).
devapi.dbObjectHandles	true	boolean	Enable table and collection name handles for the DevAPI db object.
history.autoSave	false	boolean	Save (true) or clear (false) entries in the MySQL Shell code history when you exit MySQL Shell (see Section 4.4, “MySQL Shell Code History”).
history.maxSize	1000	integer	The maximum number of entries to store in the MySQL Shell code history.
history.sql.ignorePatterns	IDENTIFIED*: *PASSWORD*	string	Strings that match these patterns are not added to the MySQL Shell code history.
interactive	true	boolean (READ ONLY)	Enable interactive mode.
logLevel	5	integer ranging from 1 to 8	Enable the application log and set a logging level (see Section 7.1, “Application Log”).
pager	None	string	Use the specified external pager tool to display text and results. Command-line arguments for the tool can be added (see Section 3.5, “Using a Pager”).
passwordsFromStdin	false	boolean	Read passwords from <code>stdin</code> instead of terminal.
resultFormat	table	string (table, tabbed, vertical, json, json/raw)	The default output format for printing result sets (see Section 4.6, “Output Formats”).
sandboxDir	Depends on platform	string	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> .
showColumnTypeInfo	false	boolean	In SQL mode, display column metadata for result sets.
showWarnings	true	boolean	In SQL mode, automatically display SQL warnings if any.
useWizards	true	boolean	Enable wizard mode.



Note

String values are case sensitive.

The `outputFormat` option is now deprecated. Use `resultFormat` instead.

Using the `\option` Command

The MySQL Shell `\option` command enables you to query and change configuration options in all modes, enabling configuration from SQL mode in addition to JavaScript and Python modes.

The command is used as follows:

- `\option -h, --help [filter]` - print help for options matching *filter*.

- `\option -l, --list [--show-origin]` - list all the options. `--show-origin` augments the list with information about how the value was last changed, possible values are:
 - Command line
 - Compiled default
 - Configuration file
 - Environment variable
 - User defined
- `\option option_name` - print the current value of the option.
- `\option [--persist] option_name value` or `name=value` - set the value of the option and if `--persist` is specified save it to the configuration file.
- `\option --unset [--persist] <option_name>` - reset option's value to default and if `--persist` is specified, removes the option from the MySQL Shell configuration file.

**Note**

The value of `option_name` and `filter` are case sensitive.

See [Valid Configuration Options](#) for a list of possible values for `option_name`.

Using the `shell.options` Configuration Interface

The `shell.options` object is available in JavaScript and Python mode to change MySQL Shell option values. You can use specific methods to configure the options, or key-value pairs as follows:

```
MySQL JS > shell.options['history.autoSave']=1
```

In addition to the key-value pair interface, the following methods are available:

- `shell.options.set(optionName, value)` - sets the `optionName` to `value` for this session, the change is not saved to the configuration file.
- `shell.options.setPersist(optionName, value)` - sets the `optionName` to `value` for this session, and saves the change to the configuration file. In Python mode, the method is `shell.options.set_persist`.
- `shell.options.unset(optionName)` - resets the `optionName` to the default value for this session, the change is not saved to the configuration file.
- `shell.options.unsetPersist(optionName)` - resets the `optionName` to the default value for this session, and saves the change to the configuration file. In Python mode, the method is `shell.options.unset_persist`.

Option names are treated as strings, and as such should be surrounded by ' characters. See [Valid Configuration Options](#) for a list of possible values for `optionName`.

Use the commands to configure MySQL Shell options as follows:

```
MySQL JS > shell.options.set('history.maxSize', 5000)
MySQL JS > shell.options.setPersist('useWizards', 'true')
```

```
MySQL JS > shell.options.setPersist('history.autoSave', 1)
```

Return options to their default values as follows:

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

Configuration File

The MySQL Shell configuration file stores the values of the option to ensure they are persisted across sessions. Values are read at startup and when you use the persist feature, settings are saved to the configuration file.

The location of the configuration file is the user configuration path and the file is named `options.json`. Assuming that the default user configuration path has not been overridden by defining the environment variable `MYSQLSH_USER_CONFIG_HOME`, the path to the configuration file is:

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

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

Appendix A MySQL Shell Command Reference

Table of Contents

A.1 `mysqlsh` — The MySQL Shell 57

This appendix describes the `mysqlsh` commands.

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](#).

`mysqlsh` supports the following command-line options.

mysqlsh Options Command-line options available for `mysqlsh`.

- `--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](#)

- `--classic`

Deprecated in version 8.0.3.

Creates a Classic session, to connect using MySQL Protocol.

- `--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 Protocol on connection with server. Requires Classic session.
- `--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.
- `--file=file_name, -f file_name`
Specify file to process in Batch mode.
- `--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 6.2, “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. With no option 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 7, 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.
- `--node`

Deprecated in version 8.0.3.
Creates a Node session connected using X Protocol to a single server.
- `--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], -ppassword`

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

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

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

- `--password` with no value and no 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 <code>resultFormat</code> configuration option in the configuration file, in which case that default applies. The <code>--table</code> alias can also be used.
tabbed	The default for batch mode, unless another value has been set persistently for the <code>resultFormat</code> configuration option in the configuration file, in which case that default applies. The <code>--tabbed</code> alias can also be used.
vertical	Produces output equivalent to the <code>\G</code> terminator for an SQL query. The <code>--vertical</code> or <code>-E</code> 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.

- `--schema=name, -D name`

The default schema to use.

- `--server-public-key-path=file_name`

MySQL Shell equivalent of `--server-public-key-path`.



Important

Only supported with classic MySQL protocol connections.

See `caching_sha2_password` plugin [Caching SHA-2 Pluggable Authentication](#).

- `--show-warnings`

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

- `--socket[=path], -S [path]`

On Unix, when a path is specified, the path is the name of the Unix socket file to use for the connection. If you specify `--socket` with no value and no equals sign, or `-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 `--socket` option is available for Classic MySQL Protocol sessions only.

You cannot specify a socket if you specify a port or a host name other than `localhost` on Unix or a period (`.`) on Windows.

- `--sql`

Start in SQL mode.

- `--sqlc`

Start in SQL mode using a `ClassicSession`.

- `--sqln`

Deprecated in version 8.0.3.

Start in SQL mode using a `NodeSession`.

- `--sqlx`

Start in SQL mode and create connection using X Protocol.

- `--ssl*`

Options that begin with `--ssl` specify whether to connect to the server using SSL and indicate where to find SSL keys and certificates. The `mysqlsh` SSL options function in the same way as the SSL options for MySQL Server, see [Command Options for Encrypted Connections](#) for more information.

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

- `--tabbed`

Display results in tab separated format in interactive mode. The default for that mode is table format. This option is an alias of the `--result-format=tabbed` option.

- `--table`

Display results in table format in batch mode. The default for that mode is tab separated format. This option is an alias of the `--result-format=table` option.

- `--uri=str`

Create a connection upon startup, specifying the connection options in a URI string format, see [Connecting Using a URI or Key-Value Pairs](#).

- `--user=user_name, -u user_name`

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

- `--version, -V`

Display version information and exit.

- `--vertical, -E`

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

