MySQL Connector/C++ 8.2 Developer Guide

Abstract

This manual describes how to install and configure MySQL Connector/C++ 8.2, which provides C++ and plain C interfaces for communicating with MySQL servers, and how to use Connector/C++ to develop database applications.

Connector/C++ 8.2 is highly recommended for use with MySQL Server 8.2, 8.1, 8.0, and 5.7. Please upgrade to Connector/C++ 8.2.

For notes detailing the changes in each release of Connector/C++, see MySQL Connector/C++ Release Notes.

For legal information, see the Legal Notices.

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

Licensing information. This product may include third-party software, used under license. If you are using a *Commercial* release of MySQL Connector/C++, see the MySQL Connector/C++ Commercial Release 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 Connector/C++, see the MySQL Connector/C++ Community Release 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: 2023-11-29 (revision: 77346)

Table of Contents

Preface and Legal Notices

This manual describes how to install and configure MySQL Connector/C++ 8.2, and how to use it to develop database applications.

Legal Notices

License Restrictions

Warranty Disclaimer

Restricted Rights Notice

Hazardous Applications Notice

Trademark Notice

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Chapter 1 Introduction to Connector/C++

MySQL Connector/C++ 8.2 is a MySQL database connector for C++ applications that connect to MySQL servers. Connector/C++ can be used to access MySQL servers that implement a document store, or in a traditional way using SQL statements. The preferred development environment for Connector/C++ 8.2 is to enable development of C++ applications using X DevAPI, or plain C applications using X DevAPI for C, but Connector/C++ 8.2 also enables development of C++ applications that use the legacy JDBC-based API from Connector/C++ 1.1.

Connector/C++ applications that use X DevAPI or X DevAPI for C require a MySQL server that has X Plugin enabled. Connector/C++ applications that use the legacy JDBC-based API neither require nor support X Plugin.

For more detailed requirements about required MySQL versions for Connector/C++ applications, see Platform Support and Prerequisites.

For notes detailing the changes in each release of Connector/C++, see MySQL Connector/C++ Release Notes.

Connector/C++ Benefits

MySQL Connector/C++ offers the following benefits for C++ users compared to the MySQL C API provided by the MySQL client library:

- Convenience of pure C++.
- Support for these application programming interfaces:
  - X DevAPI
  - X DevAPI for C
  - Legacy JDBC 4.0-based API
- Support for the object-oriented programming paradigm.
- Reduced development time.
- Licensed under the GPL with the FLOSS License Exception.
- Available under a commercial license upon request.

X DevAPI and X DevAPI for C

Connector/C++ implements X DevAPI, which enables connecting to MySQL servers that implement a document store with X Plugin. X DevAPI also enables applications to execute SQL statements.

Connector/C++ also implements a similar interface called X DevAPI for C for use by applications written in plain C.

For general information about X DevAPI, see X DevAPI User Guide. For reference information specific to the Connector/C++ implementation of X DevAPI and X DevAPI for C, see *MySQL Connector/C++ X DevAPI Reference* in the *X DevAPI* section of MySQL Documentation.

Legacy JDBC API and JDBC Compatibility

Connector/C++ implements the JDBC 4.0 API, if built to include the legacy JDBC connector:

- Connector/C++ binary distributions include the JDBC connector.
- If you build Connector/C++ from source, the JDBC connector is not built by default, but can be included by enabling the `WITH_JDBC` `CMake` option. See Chapter 4, *Installing Connector/C++ from Source*.

The Connector/C++ JDBC API is compatible with the JDBC 4.0 API. Connector/C++ does not implement the entire JDBC 4.0 API, but does feature these classes: `Connection`, `DatabaseMetaData`, `Driver`, `PreparedStatement`, `ResultSet`, `ResultSetMetaData`, `Savepoint`, `Statement`.

The JDBC 4.0 API defines approximately 450 methods for the classes just mentioned. Connector/C++ implements approximately 80% of these.

> **Note**
>
> The legacy JDBC connector in Connector/C++ 8.2 is based on the connector provided by Connector/C++ 1.1. For more information about using the JDBC API in Connector/C++ 8.2, see MySQL Connector/C++ 1.1 Developer Guide.

Platform Support and Prerequisites

To see which platforms are supported, visit the Connector/C++ downloads page.

On Windows platforms, Commercial and Community Connector/C++ distributions require the Visual C++ Redistributable for Visual Studio. The Redistributable is available at the Visual Studio Download Center; install it before installing Connector/C++. The acceptable Redistributable versions depend on your Connector/C++ version:

- Connector/C++ 8.0.19 and higher: VC++ Redistributable 2017 or higher.
- Connector/C++ 8.0.14 to 8.0.18: VC++ Redistributable 2015 or higher.

The following requirements apply to building and running Connector/C++ applications, and to building Connector/C++ itself if you build it from source:

- To run Connector/C++ applications, the MySQL server requirements depend on the API the application uses:
  - Connector/C++ applications that use X DevAPI or X DevAPI for C require a server from MySQL 8.2 (8.2.0) 8.1 (8.1.0 or higher), MySQL 8.0 (8.0.11 or higher), or MySQL 5.7 (5.7.12 or higher), with X Plugin enabled. For MySQL 8.2, 8.1, and MySQL 8.0, X Plugin is enabled by default. For MySQL 5.7, X Plugin must be enabled explicitly. (Some X Protocol features may not work with MySQL 5.7.)
  - Applications that use the JDBC API can use a server from MySQL 5.6 or higher. X Plugin is neither required nor supported.
- To build Connector/C++ applications:
  - The MySQL version does not apply.
  - On Windows, Microsoft Visual Studio is required. The acceptable MSVC versions depend on your Connector/C++ version and the type of linking you use:
    - Connector/C++ 8.0.20 and higher: Same as Connector/C++ 8.0.19, with the addition that binary distributions are also compatible with MSVC 2017 using the static X DevAPI connector library.

This means that binary distributions are fully compatible with MSVC 2019, and fully compatible with MSVC 2017 with the exception of the static legacy (JDBC) connector library.

- Connector/C++ 8.0.19: Connector/C++ binary distributions are compatible with projects built using MSVC 2019 (using either dynamic or static connector libraries) or MSVC 2017 (using dynamic connector libraries).
- Connector/C++ 8.0.14 to 8.0.18: MSVC 2017 or 2015.
- Connector/C++ prior to 8.0.14: MSVC 2015.

- To build Connector/C++ from source:
  - The MySQL C API client library may be required:
    - For Connector/C++ built without the JDBC connector (which is the default), the client library is not needed.
    - To build Connector/C++ with the JDBC connector, configure Connector/C++ with the `WITH_JDBC` `CMake` option enabled. In this case, the JDBC connector requires a client library from MySQL 8.2 (8.2.0 or higher), 8.1 (8.1.0 or higher), MySQL 8.0 (8.0.11 or higher) or MySQL 5.7 (5.7.9 or higher).
  - On Windows, Microsoft Visual Studio is required. The acceptable MSVC versions depend on your Connector/C++ version:
    - Connector/C++ 8.0.19 and higher: MSVC 2019 or 2017.
    - Connector/C++ 8.0.14 to 8.0.18: MSVC 2017 or 2015.
    - Connector/C++ prior to 8.0.14: MSVC 2015.

Chapter 2 Obtaining Connector/C++

Connector/C++ binary and source distributions are available, in platform-specific packaging formats. To obtain a distribution, visit the Connector/C++ downloads page. It is also possible to clone the Connector/C++ Git source repository.

- Connector/C++ binary distributions are available for Microsoft Windows, and for Unix and Unix-like platforms. See Chapter 3, *Installing Connector/C++ from a Binary Distribution*.
- Connector/C++ source distributions are available as compressed `tar` files or Zip archives and can be used on any supported platform. See Chapter 4, *Installing Connector/C++ from Source*.
- The Connector/C++ source code repository uses Git and is available at GitHub. See Chapter 4, *Installing Connector/C++ from Source*.

Chapter 3 Installing Connector/C++ from a Binary Distribution

To obtain a Connector/C++ binary distribution, visit the Connector/C++ downloads page.

For some platforms, Connector/C++ binary distributions are available in platform-specific packaging formats. Binary distributions are also available in more generic format, in the form of compressed `tar` files or Zip archives.

> **Note**
>
> Generic Linux packages do not contain Connector/C++ static libraries. If you intend to link your application to a static library, consider installing a package that is specific to the platform on which you build your final application.

For descriptions here that refer to documentation files, those files have names such as `CONTRIBUTING.md`, `README.md`, `README.txt`, `README`, `LICENSE.txt`, `LICENSE`, `INFO_BIN`, and `INFO_SRC`. (Prior to Connector/C++ 8.0.14, the information file is `BUILDINFO.txt` rather than `INFO_BIN` and `INFO_SRC`.)

- Installation on Windows
- Installation on Linux
- Installation on macOS
- Installation on Solaris
- Installation Using a tar or Zip Package

Installation on Windows

> **Important**
>
> On Windows platforms, Commercial and Community Connector/C++ distributions require the Visual C++ Redistributable for Visual Studio. The Redistributable is available at the Visual Studio Download Center; install it before installing Connector/C++. For information about which VC ++ Redistributable versions are acceptable, see Platform Support and Prerequisites.

These methods of installing binary distributions are available on Windows:

- **Windows MSI Installer.** As of Connector/C++ 8.0.12, an MSI Installer is available for Windows. To use the MSI Installer (`.msi` file), launch it and follow the prompts in the screens it presents. The MSI Installer can install components for these connectors:

  - The connector for X DevAPI (including X DevAPI for C).
  - The connector for the legacy JDBC API.

  For each connector, there are two components:

  - The DLL component includes the connector DLLs and libraries to satisfy runtime dependencies. The DLL component is required to run Connector/C++ application binaries that use the connector.
  - The Developer component includes header files, static libraries, and import libraries for DLLs. The Developer component is required to build from source Connector/C++ applications that use the connector.

  The MSI Installer requires administrative privileges. It begins by presenting a welcome screen that enables you to continue the installation or cancel it. If you continue the installation, the MSI Installer overview screen enables you to select the type of installation to perform:

- The **Complete** installation installs the DLL and Developer components for both connectors.
- The **Typical** installation installs the DLL component for both connectors.
- The **Custom** installation enables you to specify the installation location and select which components to install. The DLL and Developer components for the X DevAPI connector are preselected, but you can override the selection. The Developer component for a connector cannot be selected without also selecting the connector DLL component.

The MSI Installer performs these actions:

- It checks whether the required Visual C++ Redistributable for Visual Studio is present. If not, the installer asks you to install it and exits with an error. For information about which VC++ Redistributable versions are acceptable, see Platform Support and Prerequisites.
- It installs documentation files.

To install Connector/C++ from the command line in batch mode, use a command similar to:

```
msiexec.exe /i packages\mysql-connector-cpp-commercial-8.X.X-winx64.msi /qn /lvx*
msi_install.log ALLUSERS=1 INSTALLDIR=C:\tmp\c-cpp-unpacked INSTALLLEVEL=4
```

To uninstall Connector/C++ from the command line in batch mode, use a command similar to:

```
msiexec.exe /x packages\mysql-connector-cpp-commercial-8.X.X-winx64.msi /qn /lvx*
msi_uninstall.log
```

- **Zip archive package without installer.** To install from a Zip archive package (`.zip` file), see Installation Using a tar or Zip Package.

  In addition to the standard Zip archive packages, packages are available that were built in debug mode. However, applications should use the same build mode as Connector/C++. If you install Connector/C++ packages built in debug mode, build applications in debug mode. If you install Connector/C++ packages built in release mode, build applications in release mode.

Installation on Linux

These methods of installing binary distributions are available on Linux:

- **RPM package.** RPM packages are available for Linux (as of Connector/C++ 8.0.12). The packages are distinguished by their base names (the full names include the Connector/C++ version and suffixes):
  - `mysql-connector-c++`: This package provides the shared connector library implementing X DevAPI and X DevAPI for C.
  - `mysql-connector-c++-jdbc`: This package provides the shared legacy connector library implementing the JDBC API.
  - `mysql-connector-c++-devel`: This package installs development files required for building applications that use Connector/C++ libraries provided by the other packages, and static connector libraries. This package depends on the shared libraries provided by the other packages. It cannot be installed by itself without the other two packages.
- **Debian package.** Debian packages are available for Linux (as of Connector/C++ 8.0.14). The packages are distinguished by their base names (the full names include the Connector/C++ version and suffixes):
  - `libmysqlcppconn8-1`: This package provides the shared connector library implementing X DevAPI and X DevAPI for C.

  - `libmysqlcppconn7`: This package provides the shared legacy connector library implementing the JDBC API.
  - `libmysqlcppconn-dev`: This package installs development files required for building applications that use Connector/C++ libraries provided by the other packages, and static connector libraries. This package depends on the shared libraries provided by the other packages. It cannot be installed by itself without the other two packages.

- **Compressed tar file.** To install from a compressed `tar` file (`.tar.gz` file), see Installation Using a tar or Zip Package.

Installation on macOS

These methods of installing binary distributions are available on macOS:

- **DMG package.** DMG (disk image) packages for macOS are available as of Connector/C++ 8.0.12. A DMG package provides shared and static connector libraries implementing X DevAPI and X DevAPI for C, and the legacy connector library implementing the JDBC API. The package also includes OpenSSL libraries, public header files, and documentation files.
- **Compressed tar file.** To install from a compressed `tar` file (`.tar.gz` file), see Installation Using a tar or Zip Package.

Installation on Solaris

These methods of installing binary distributions are available on Solaris:

- **Compressed tar file.** To install from a compressed `tar` file (`.tar.gz` file), see Installation Using a tar or Zip Package.

Installation Using a tar or Zip Package

Connector/C++ binary distributions are available for several platforms, packaged in the form of compressed `tar` files or Zip archives, denoted here as *`PACKAGE`*`.tar.gz` or *`PACKAGE`*`.zip`.

> **Note**
>
> Generic Linux packages do not contain Connector/C++ static libraries.

To unpack a compressed `tar` file, use this command in the intended installation directory:

```
tar zxvf PACKAGE.tar.gz
```

To install from a Zip archive package (`.zip` file), use `WinZip` or another tool that can read `.zip` files to unpack the file into the location of your choosing.

Chapter 4 Installing Connector/C++ from Source

Table of Contents

This chapter describes how to install Connector/C++ using a source distribution or a copy of the Git source repository.

4.1 Source Installation System Prerequisites

To install Connector/C++ from source, the following system requirements must be satisfied:

- Build Tools
- MySQL Client Library
- Boost C++ Libraries
- SSL Support

Build Tools

You must have the cross-platform build tool `CMake` (3.0 or higher).

You must have a C++ compiler that supports C++17 (as of Connector/C++ 8.0.33).

MySQL Client Library

To build Connector/C++ from source, the MySQL C API client library may be required:

- Building the JDBC connector requires a client library from MySQL 8.2 (8.2.0 or higher), 8.1 (8.1.0 or higher), MySQL 8.0 (8.0.11 or higher), or MySQL 5.7 (5.7.9 or higher). This occurs when Connector/C++ is configured with the `WITH_JDBC` `CMake` option enabled to include the JDBC connector.
- For Connector/C++ built without the JDBC connector, the client library is not needed.

Typically, the MySQL client library is installed when MySQL is installed. However, check your operating system documentation for other installation options.

To specify where to find the client library, set the `MYSQL_DIR` `CMake` option appropriately at configuration time as necessary (see Section 4.4, “Connector/C++ Source-Configuration Options”).

Boost C++ Libraries

To compile Connector/C++ the Boost C++ libraries are needed only if you build the legacy JDBC API or if the version of the C++ standard library on your system does not implement the UTF8 converter (`codecvt_utf8`).

If the Boost C++ libraries are needed, Boost 1.59.0 or newer must be installed. To obtain Boost and its installation instructions, visit the official Boost site.

After Boost is installed, use the `WITH_BOOST` `CMake` option to indicate where the Boost files are located (see Section 4.4, “Connector/C++ Source-Configuration Options”):

```
cmake [other_options] -DWITH_BOOST=/usr/local/boost_1_59_0
```

Adjust the path as necessary to match your installation.

SSL Support

Use the `WITH_SSL` `CMake` option to specify which SSL library to use when compiling Connector/C++. OpenSSL 1.0.x or higher is required. Your other options are:

- As of Connector/C++ 8.0.18, it is possible to compile against OpenSSL 1.1.
- As of Connector/C++ 8.0.30, it is possible to compile against OpenSSL 3.0.

For more information about `WITH_SSL` and SSL libraries, see Section 4.4, “Connector/C++ Source-Configuration Options”.

4.2 Obtaining and Unpacking a Connector/C++ Source Distribution

To obtain a Connector/C++ source distribution, visit the Connector/C++ downloads page. Alternatively, clone the Connector/C++ Git source repository.

A Connector/C++ source distribution is packaged as a compressed `tar` file or Zip archive, denoted here as *`PACKAGE`*`.tar.gz` or *`PACKAGE`*`.zip`. A source distribution in `tar` file or Zip archive format can be used on any supported platform.

The distribution when unpacked includes an `INFO_SRC` file that provides information about the product version and the source repository from which the distribution was produced. The distribution also includes other documentation files such as those listed in Chapter 3, *Installing Connector/C++ from a Binary Distribution*.

To unpack a compressed `tar` file, use this command in the intended installation directory:

```
tar zxvf PACKAGE.tar.gz
```

After unpacking the distribution, build it using the appropriate instructions for your platform later in this chapter.

To install from a Zip archive package (`.zip` file), use `WinZip` or another tool that can read `.zip` files to unpack the file into the location of your choosing. After unpacking the distribution, build it using the appropriate instructions for your platform later in this chapter.

To clone the Connector/C++ code from the source code repository located on GitHub at https://github.com/mysql/mysql-connector-cpp, use this command:

```
git clone https://github.com/mysql/mysql-connector-cpp.git
```

That command should create a `mysql-connector-cpp` directory containing a copy of the entire Connector/C++ source tree.

The `git clone` command sets the sources to the `master` branch, which is the branch that contains the latest sources. Released code is in the `8.0` branche (the `8.0` branch contains the same sources as the `master` branch). If necessary, use `git checkout` in the source directory to select the desired branch. For example, to build Connector/C++ 8.0:

```
cd mysql-connector-cpp
git checkout 8.0
```

After cloning the repository, build it using the appropriate instructions for your platform later in this chapter.

After the initial checkout operation to get the source tree, run `git pull` periodically to update your source to the latest version.

4.3 Installing Connector/C++ from Source

To install Connector/C++ from source, verify that your system satisfies the requirements outlined in Section 4.1, “Source Installation System Prerequisites”.

- Configuring Connector/C++
- Specifying External Dependencies
- Building Connector/C++
- Installing Connector/C++
- Verifying Connector/C++ Functionality

Configuring Connector/C++

Use `CMake` to configure and build Connector/C++. Only out-of-source-builds are supported, so create a directory to use for the build and change location into it. Then configure the build using this command, where *`concpp_source`* is the directory containing the Connector/C++ source code:

```
cmake concpp_source
```

It may be necessary to specify other options on the configuration command. Some examples:

- By default, these installation locations are used:
  - `/usr/local/mysql/connector-c++-8.0` (Unix and Unix-like systems)
  - *`User_home`*`/MySQL/"MySQL Connector C++ 8.0"` (Windows)

  To specify the installation location explicitly, use the `CMAKE_INSTALL_PREFIX` option:

  ```
  -DCMAKE_INSTALL_PREFIX=path_name
  ```

- On Windows, you can use the `-G` and `-A` options to select a particular generator:
  - `-G "Visual Studio 16" -A x64` (64-bit builds)
  - `-G "Visual Studio 16" -A Win32` (32-bit builds)

  Consult the `CMake` manual or check `cmake --help` to find out which generators are supported by your `CMake` version. (However, it may be that your version of `CMake` supports more generators than can actually be used to build Connector/C++.)

- If the Boost C++ libraries are needed, use the `WITH_BOOST` option to specify their location:

  ```
  -DWITH_BOOST=path_name
  ```

- By default, the build creates dynamic (shared) libraries. To build static libraries, enable the `BUILD_STATIC` option:

  ```
  -DBUILD_STATIC=ON
  ```

- By default, the legacy JDBC connector is not built. To include the JDBC connector in the build, enable the `WITH_JDBC` option:

  ```
  -DWITH_JDBC=ON
  ```

> **Note**
>
> If you configure and build the test programs later, use the same `CMake` options to configure them as the ones you use to configure Connector/C++

(`-G`, `WITH_BOOST`, `BUILD_STATIC`, and so forth). Exceptions: Path name arguments will differ, and you need not specify `CMAKE_INSTALL_PREFIX`.

For information about `CMake` configuration options, see Section 4.4, “Connector/C++ Source-Configuration Options”.

Specifying External Dependencies

Use `CMake` options to configure and build Connector/C++ with external sources that you can substitute for the required third-party dependencies currently bundled with the connector. If the dependency is an external library, then the library is linked dynamically to the connector. In contrast, bundled third-party libraries used by connector are linked statically to it.

Note

Using an external third-party library that cannot be linked to the connector dynamically causes the build to fail, even when the static library is available.

The supported options are:

- `WITH_BOOST`
- `WITH_LZ4`
- `WITH_MYSQL`
- `WITH_PROTOBUF`
- `WITH_SSL`
- `WITH_ZLIB`
- `WITH_ZSTD`

For example, to use an external installation of Protobuf, instead of building it from bundled sources, specify the `WITH_PROTOBUF` option and provide the path name to the location where `CMake` can find the alternative dependency.

Note

If an external dependency cannot be found (or is unusable), then the build fails. No attempt is made to locate the bundled source.

```
cmake [other_options] -DWITH_PROTOBUF=path_name_to_protobuf_install
```

To configure the standard system-wide location for an external dependency, use the literal value `system` rather than providing a path name. For example:

```
-DWITH_SSL=system
```

For information about `CMake` configuration options, see Section 4.4, “Connector/C++ Source-Configuration Options”.

External dependencies make it possible to use shared third-party libraries that are linked dynamically to the connector. This can be an advantage because, for example, you cannot use the connector static library with an application that also links to a Protobuf library.

When running an application that is linked to the connector dynamic library, the third-party libraries on which the connector depends should be correctly found if they are placed in the file system next to the connector library. The application should also work when the libraries are installed at the standard system-wide locations. This assumes that the external third-party dependency version is expected by Connector/C++.

Except for Windows, it should be possible to run an application linked to the connector dynamic library when the connector library and the third-party libraries are placed in a nonstandard location, provided that these locations were stored as runtime paths when building the application (`gcc -rpath` option).

For Windows, an application that is linked to the connector shared library can be run only if the connector library and the libraries are stored either:

- In the Windows system folder
- In the same folder as the application
- In a folder listed in the `PATH` environment variable

If the application is linked to the connector static library, it remains true that the required libraries must be found in one of the preceding locations.

Building Connector/C++

After configuring the Connector/C++ distribution, build it using this command:

```
cmake --build . --config build_type
```

The `--config` option is optional. It specifies the build configuration to use, such as `Release` or `Debug`. If you omit `--config`, the default is `Debug`.

> **Important**
>
> If you specify the `--config` option on the preceding command, specify the same `--config` option for later steps, such as the steps that install Connector/C++ or that build test programs.

If the build is successful, it creates the connector libraries in the build directory. (For Windows, look for the libraries in a subdirectory with the same name as the *`build_type`* value specified for the `--config` option.)

- If you build dynamic libraries, they have these names:
  - `libmysqlcppconn8.so.1` (Unix)
  - `libmysqlcppconn8.2.dylib` (macOS)
  - `mysqlcppconn8-1-vs14.dll` (Windows)
- If you build static libraries, they have these names:
  - `libmysqlcppconn8-static.a` (Unix, macOS)
  - `mysqlcppconn8-static.lib` (Windows)

If you enabled the `WITH_JDBC` option to include the legacy JDBC connector in the build, the following additional library files are created.

- If you build legacy dynamic libraries, they have these names:
  - `libmysqlcppconn.so.7` (Unix)
  - `libmysqlcppconn.7.dylib` (macOS)
  - `mysqlcppconn-7-vs14.dll` (Windows)
- If you build legacy static libraries, they have these names:
  - `libmysqlcppconn-static.a` (Unix, macOS)

- `mysqlcppconn-static.lib` (Windows)

Installing Connector/C++

To install Connector/C++, use this command:

```
cmake --build . --target install --config build_type
```

Verifying Connector/C++ Functionality

To verify connector functionality, build and run one or more of the test programs included in the `testapp` directory of the source distribution. Create a directory to use and change location into it. Then issue the following commands:

```
cmake [other_options] -DWITH_CONCPP=concpp_install concpp_source/testapp
cmake --build . --config=build_type
```

`WITH_CONCPP` is an option used only to configure the test application. *`other_options`* consists of the options that you used to configure Connector/C++ itself (`-G`, `WITH_BOOST`, `BUILD_STATIC`, and so forth). *`concpp_source`* is the directory containing the Connector/C++ source code, and *`concpp_install`* is the directory where Connector/C++ is installed:

The preceding commands should create the `devapi_test` and `xapi_test` programs in the `run` directory of the build location. If you enable `WITH_JDBC` when configuring the test programs, the build also creates the `jdbc_test` program.

Before running test programs, ensure that a MySQL server instance is running with X Plugin enabled. The easiest way to arrange this is to use the `mysql-test-run.pl` script from the MySQL distribution. For MySQL 8.0, X Plugin is enabled by default, so invoke this command in the `mysql-test` directory of that distribution:

```
perl mysql-test-run.pl --start-and-exit
```

For MySQL 5.7, X Plugin must be enabled explicitly, so add an option to do that:

```
perl mysql-test-run.pl --start-and-exit --mysqld=--plugin-load=mysqlx
```

The command should start a test server instance with X Plugin enabled and listening on port 13009 instead of its standard port (33060).

Now you can run one of the test programs. They accept a connection-string argument, so if the server was started as just described, you can run them like this:

```
run/devapi_test mysqlx://root@127.0.0.1:13009
run/xapi_test mysqlx://root@127.0.0.1:13009
```

The connection string assumes availability of a `root` user account without any password and the programs assume that there is a `test` schema available (assumptions that hold for a server started using `mysql-test-run.pl`).

To test `jdbc_test`, you need a MySQL server, but X Plugin is not required. Also, the connection options must be in the form specified by the JDBC API. Pass the user name as the second argument. For example:

```
run/jdbc_test tcp://127.0.0.1:13009 root
```

4.4 Connector/C++ Source-Configuration Options

Connector/C++ recognizes the `CMake` options described in this section.

Table 4.1 Connector/C++ Source-Configuration Option Reference

Formats	Description	Default
BUILD_STATIC	Whether to build a static librarty	OFF
BUNDLE_DEPENDENCIES	Whether to bundle external dependency libraries with the connector	OFF
CMAKE_BUILD_TYPE	Type of build to produce	Debug
CMAKE_INSTALL_DOCDIR	Documentation installation directory	
CMAKE_INSTALL_INCLUDEDIR	Header file installation directory	
CMAKE_INSTALL_LIBDIR	Library installation directory	
CMAKE_INSTALL_PREFIX	Installation base directory	/usr/local
MAINTAINER_MODE	For internal use only	OFF
MYSQLCLIENT_STATIC_BINDING	Whether to link to the shared MySQL client library	ON
MYSQLCLIENT_STATIC_LINKING	Whether to statically link to the MySQL client library	OFF
MYSQL_CONFIG_EXECUTABLE	Path to the mysql_config program	${MYSQL_DIR}/bin/mysql_config
MYSQL_DIR	MySQL Server installation directory	
STATIC_MSVCRT	Use the static runtime library	
WITH_BOOST	The Boost source directory	system
WITH_DOC	Whether to generate Doxygen documentation	OFF
WITH_JDBC	Whether to build legacy JDBC library	OFF
WITH_LZ4	The LZ4 source directory	
WITH_MYSQL	The MySQL Server source directory	system
WITH_PROTOBUF	The Protobuf source directory	
WITH_SSL	The SSL source directory	system
WITH_ZLIB	The ZLIB source directory	
WITH_ZSTD	The ZSTD source directory	

- `-DBUILD_STATIC=`*`bool`*

  By default, dynamic (shared) libraries are built. If this option is enabled, static libraries are built instead.

- `-DBUNDLE_DEPENDENCIES=`*`bool`*

  This is an internal option used for creating Connector/C++ distribution packages.

- `-DCMAKE_BUILD_TYPE=`*`type`*

  The type of build to produce:

  - `Debug`: Disable optimizations and generate debugging information. This is the default.
  - `Release`: Enable optimizations.

  - `RelWithDebInfo`: Enable optimizations and generate debugging information.

- `-DCMAKE_INSTALL_DOCDIR=dir_name`

  The documentation installation directory, relative to `CMAKE_INSTALL_PREFIX`. If not specified, the default is to install in `CMAKE_INSTALL_PREFIX`.

  This option requires that `WITH_DOC` be enabled.

  This option was added in Connector/C++ 8.0.14.

- `-DCMAKE_INSTALL_INCLUDEDIR=dir_name`

  The header file installation directory, relative to `CMAKE_INSTALL_PREFIX`. If not specified, the default is `include`.

  This option was added in Connector/C++ 8.0.14.

- `-DCMAKE_INSTALL_LIBDIR=dir_name`

  The library installation directory, relative to `CMAKE_INSTALL_PREFIX`. If not specified, the default is `lib64` or `lib`.

  This option was added in Connector/C++ 8.0.14.

- `-DCMAKE_INSTALL_PREFIX=dir_name`

  The installation base directory (where to install Connector/C++).

- `-DMAINTAINER_MODE=bool`

  This is an internal option used for creating Connector/C++ distribution packages. It was added in Connector/C++ 8.0.12.

- `-DMYSQLCLIENT_STATIC_BINDING=bool`

  Whether to link to the shared MySQL client library. This option is used only if `MYSQLCLIENT_STATIC_LINKING` is disabled to enable dynamic linking of the MySQL client library. In that case, if `MYSQLCLIENT_STATIC_BINDING` is enabled (the default), Connector/C++ is linked to the shared MySQL client library. Otherwise, the shared MySQL client library is loaded and mapped at runtime.

  This option applies only if you are building the legacy JDBC connector (that is, only if `WITH_JDBC` is enabled). It was added in Connector/C++ 8.0.16.

- `-DMYSQLCLIENT_STATIC_LINKING=bool`

  Whether to link statically to the MySQL client library. The default depends on the legacy JDBC connector that you are building:

  - From Connector/C++ 8.0.33, the default is `OFF` (use dynamic linking to the client library). Enabling this option disables dynamic linking to the client library.
  - For Connector/C++ 8.0.16 to 8.0.32, the default is `ON` (use static linking to the client library). Disabling this option enables dynamic linking to the client library. `CMake` verifies that the current compiler and standard libraries can build without errors at configuration time.

  This option applies only if you are building the legacy JDBC connector (that is, only if `WITH_JDBC` is enabled). It was added in Connector/C++ 8.0.16.

- `-DMYSQL_CONFIG_EXECUTABLE=file_name`

  The path to the `mysql_config` program.

  On non-Windows systems, `CMake` checks to see whether `MYSQL_CONFIG_EXECUTABLE` is set. If not, `CMake` tries to locate `mysql_config` in the default locations.

  This option applies only if you are building the legacy JDBC connector (that is, only if `WITH_JDBC` is enabled).

- `-DMYSQL_DIR=dir_name`

  The directory where MySQL is installed.

  This option applies only if you are building the legacy JDBC connector (that is, only if `WITH_JDBC` is enabled).

- `-DSTATIC_MSVCRT=bool`

  (Windows only) Use the static runtime library (the `/MT*` compiler option). This option might be necessary if code that uses Connector/C++ also uses the static runtime library.

- `-DWITH_BOOST={system|path_name}`

  This option specifies which BOOST header file to use when compiling Connector/C++ with an external dependency. The option value to use:

  - `system`: Use the system BOOST header file.
  - *`path_name`* is the path name to the file to use.

  For consistency with `CMake` conventions, `BOOST_DIR` or `BOOST_ROOT_DIR` can be used instead of `WITH_BOOST` to indicate the base location of the dependency. As an alternative that implies the `WITH_BOOST` option (without specifying it), use `BOOST_INCLUDE_DIR` to provide the header file location instead of deriving it from the `BOOST_ROOT_DIR` value.

  This option applies only if you are building the legacy JDBC connector (that is, only if `WITH_JDBC` is enabled).

- `-DWITH_DOC=bool`

  Whether to enable generating the Doxygen documentation. As of Connector/C++ 8.0.16, enabling this option also causes the Doxygen documentation to be built by the `all` target.

- `-DWITH_JDBC=bool`

  Whether to build the legacy JDBC connector. This option is disabled by default. If it is enabled, Connector/C++ 8.0 applications can use the legacy JDBC API, just like Connector/C++ 1.1 applications.

- `-DWITH_LZ4={system|path_name}`

  This option specifies which LZ4 installation to use when compiling Connector/C++ with an external dependency. The option value to use:

  - `system`: Use the system LZ4 location.
  - *`path_name`* is the path name to the installation location to use.

  For consistency with `CMake` conventions, `LZ4_DIR` or `LZ4_ROOT_DIR` can be used instead of `WITH_LZ4` to indicate the base location of the dependency.

  To imply the `WITH_LZ4` option but with more fine-grained specification of installation directories, use `LZ4_INCLUDE_DIR` or `LZ4_LIB_DIR` to indicate the header file (or library) location instead of deriving it from the `LZ4_ROOT_DIR` value. To specify a list of external libraries to link to, use `LZ4_LIBRARY` instead of the `WITH_LZ4` option.

  If you specify both `LZ4_LIBRARY` and `LZ4_LIB_DIR`, then `LZ4_LIB_DIR` is used as an additional prefix when finding the library file and `LZ4_LIBRARY` should be relative to that prefix. On Windows, `LZ4_LIBRARY` should point at the import library of the DLL.

- `-DWITH_MYSQL={system|path_name}`

  The location where the MySQL sources are installed. The client library is linked statically when you specify this option unless you also request `MYSQLCLIENT_STATIC_LINKING=OFF`. The option value to use:

  - `system`: Use the system MYSQL location.
  - *`path_name`* is the path name to the installation location to use.

  This option applies only if you are building the legacy JDBC connector (that is, only if `WITH_JDBC` is enabled).

  For consistency with `CMake` conventions, `MYSQL_DIR` or `MYSQL_ROOT_DIR` can be used instead of `WITH_MYSQL` to indicate the base location of the dependency.

  To imply the `WITH_MYSQL` option but with more fine-grained specification of installation directories, use `MYSQL_INCLUDE_DIR` or `MYSQL_LIB_DIR` to indicate the header file (or library) location instead of deriving it from the `MYSQL_ROOT_DIR` value. To specify a list of external libraries to link to, use `MYSQL_LIBRARY` instead of the `WITH_MYSQL` option.

  If you specify both `MYSQL_LIBRARY` and `MYSQL_LIB_DIR`, then `MYSQL_LIB_DIR` is used as an additional prefix when finding the library file and `MYSQL_LIBRARY` should be relative to that prefix. On Windows, `MYSQL_LIBRARY` should point at the import library of the DLL.

- `-DWITH_PROTOBUF={system|path_name}`

  This option specifies which Protobuf installation to use when compiling Connector/C++ with an external dependency. Although the library in Connector/C++ binary packages still links in Protobuf

statically, using this option makes it possible to build from external sources a variant that links in Protobuf dynamically.

The option value to use:

- `system`: Use the system Protobuf location.
- *`path_name`* is the path name to the installation location to use.

For consistency with `CMake` conventions, `PROTOBUF_DIR` or `PROTOBUF_ROOT_DIR` can be used instead of `WITH_PROTOBUF` to indicate the base location of the dependency.

To imply the `WITH_PROTOBUF` option but with more fine-grained specification of installation directories, use `PROTOBUF_INCLUDE_DIR` or `PROTOBUF_LIB_DIR` to indicate the header file (or library) location instead of deriving it from the `PROTOBUF_ROOT_DIR` value. To specify a list of external libraries to link to, use `PROTOBUF_LIBRARY` instead of the `WITH_PROTOBUF` option.

If you specify both `PROTOBUF_LIBRARY` and `PROTOBUF_LIB_DIR`, then `PROTOBUF_LIB_DIR` is used as an additional prefix when finding the library file and `PROTOBUF_LIBRARY` should be relative to that prefix. On Windows, `PROTOBUF_LIBRARY` should point at the import library of the DLL.

Similarly, specifying `PROTOBUF_BIN_DIR` makes it possible to locate the binaries required to use the dependency and find the compiler.

- `-DWITH_SSL={system|`*`path_name`*`}`

  This option specifies which SSL library to use when compiling Connector/C++. The option value to use:

  - `system`: Use the system OpenSSL library.
  - *`path_name`* is the path name to the SSL installation to use. It should be the path to the installed OpenSSL library, and must point to a directory containing a `lib` subdirectory with OpenSSL libraries that are already built. Specifying a path name for the OpenSSL installation can be preferable to using `system` because it can prevent `CMake` from detecting and using an older or incorrect OpenSSL version installed on the system.

  For consistency with `CMake` conventions, `SSL_DIR` or `SSL_ROOT_DIR` (`OPENSSL_ROOT_DIR`) can be used instead of `WITH_SSL` to indicate the base location of the dependency.

  To imply the `WITH_SSL` option but with more fine-grained specification of installation directories, use `OPENSSL_INCLUDE_DIR` or `OPENSSL_LIB_DIR` to indicate the header file (or library) location instead of deriving it from the `SSL_ROOT_DIR` value. To specify a list of external libraries to link to, use `SSL_LIBRARY` instead of the `WITH_SSL` option.

  If you specify both `SSL_LIBRARY` and `OPENSSL_LIB_DIR`, then `OPENSSL_LIB_DIR` is used as an additional prefix when finding the library file and `SSL_LIBRARY` should be relative to that prefix. On Windows, `SSL_LIBRARY` should point at the import library of the DLL.

- `-DWITH_ZLIB={system|path_name}`

  This option specifies which ZLIB installation to use when compiling Connector/C++ with an external dependency. The option value to use:

  - `system`: Use the system ZLIB location.
  - *`path_name`* is the path name to the installation location to use.

  For consistency with `CMake` conventions, `ZLIB_DIR` or `ZLIB_ROOT_DIR` can be used instead of `WITH_ZLIB` to indicate the base location of the dependency.

  To imply the `WITH_ZLIB` option but with more fine-grained specification of installation directories, use `ZLIB_INCLUDE_DIR` or `ZLIB_LIB_DIR` to indicate the header file (or library) location instead of deriving it from the `ZLIB_ROOT_DIR` value. To specify a list of external libraries to link to, use `ZLIB_LIBRARY` instead of the `WITH_ZLIB` option.

  If you specify both `ZLIB_LIBRARY` and `ZLIB_LIB_DIR`, then `ZLIB_LIB_DIR` is used as an additional prefix when finding the library file and `ZLIB_LIBRARY` should be relative to that prefix. On Windows, `ZLIB_LIBRARY` should point at the import library of the DLL,

- `-DWITH_ZSTD={system|path_name}`

  This option specifies which ZSTD installation to use when compiling Connector/C++ with an external dependency. The option value to use:

  - `system`: Use the system ZSTD location.
  - *`path_name`* is the path name to the installation location to use.

  For consistency with `CMake` conventions, `ZSTD_DIR` or `ZSTD_ROOT_DIR` can be used instead of `WITH_ZSTD` to indicate the base location of the dependency.

  To imply the `WITH_ZSTD` option but with more fine-grained specification of installation directories, use `ZSTD_INCLUDE_DIR` or `ZSTD_LIB_DIR` to indicate the header file (or library) location instead of deriving it from the `ZSTD_ROOT_DIR` value. To specify a list of external libraries to link to, use `ZSTD_LIBRARY` instead of the `WITH_ZSTD` option.

  If you specify both `ZSTD_LIBRARY` and `ZSTD_LIB_DIR`, then `ZSTD_LIB_DIR` is used as an additional prefix when finding the library file and `ZSTD_LIBRARY` should be relative to that prefix. On Windows, `ZSTD_LIBRARY` should point at the import library of the DLL.

Chapter 5 Building Connector/C++ Applications

Table of Contents

This chapter provides guidance on building Connector/C++ applications:

- General considerations for building Connector/C++ applications successfully. See Section 5.1, “Building Connector/C++ Applications: General Considerations”.
- Information about building Connector/C++ applications that applies to specific platforms such as Windows, macOS, generic Linux, and Solaris. See Section 5.2, “Building Connector/C++ Applications: Platform-Specific Considerations”.

For discussion of the programming interfaces available to Connector/C++ applications, see Chapter 1, *Introduction to Connector/C++*.

5.1 Building Connector/C++ Applications: General Considerations

This section discusses general considerations to keep in mind when building Connector/C++ applications. For information that applies to particular platforms, see the section that applies to your platform in Section 5.2, “Building Connector/C++ Applications: Platform-Specific Considerations”.

Commands shown here are as given from the command line (for example, as invoked from a `Makefile`). The commands apply to any platform that supports `make` and command-line build tools such as `g++`, `cc`, or `clang`, but may need adjustment for your build environment.

- Build Tools and Configuration Settings
- C++17 Support
- Connector/C++ Header Files
- Connector/C++ Version Macros
- Boost Header Files
- Link Libraries
- Runtime Libraries
- Using the Connector/C++ Dynamic Library
- Using the Connector/C++ Static Library

Build Tools and Configuration Settings

It is important that the tools you use to build your Connector/C++ applications are compatible with the tools used to build Connector/C++ itself. Ideally, build your applications with the same tools that were used to build the Connector/C++ binaries.

To avoid issues, ensure that these factors are the same for your applications and Connector/C++ itself:

- Compiler version.
- Runtime library.
- Runtime linker configuration settings.

To avoid potential crashes, the build configuration of Connector/C++ should match the build configuration of the application using it. For example, do not use a release build of Connector/C++ with a debug build of the client application.

To use a different compiler version, release configuration, or runtime library, first build Connector/C++ from source using your desired settings (see Chapter 4, *Installing Connector/C++ from Source*), then build your applications using those same settings.

Connector/C++ binary distributions include an `INFO_BIN` file that describes the environment and configuration options used to build the distribution. If you installed Connector/C++ from a binary distribution and experience build-related issues on a platform, it may help to check the settings that were used to build the distribution on that platform. Binary distributions also include an `INFO_SRC` file that provides information about the product version and the source repository from which the distribution was produced. (Prior to Connector/C++ 8.0.14, look for `BUILDINFO.txt` rather than `INFO_BIN` and `INFO_SRC`.)

C++17 Support

X DevAPI uses C++17 language features (as of Connector/C++ 8.0.33). To compile Connector/C++ applications that use X DevAPI, enable C++17 support in the compiler using the `-std=c++17` option. This option is not needed for applications that use X DevAPI for C (which is a plain C API) or the legacy JDBC API (which is based on plain C++), unless the application code uses C++17.

Connector/C++ Header Files

The API an application uses determines which Connector/C++ header files it should include. The following include directives work under the assumption that the include path contains `$MYSQL_CPPCONN_DIR/include`, where `$MYSQL_CPPCONN_DIR` is the Connector/C++ installation location. Pass an `-I $MYSQL_CPPCONN_DIR/include` option on the compiler invocation command to ensure this.

- For applications that use X DevAPI:

```
#include <mysqlx/xdevapi.h>
```

- For applications that use X DevAPI for C:

```
#include <mysqlx/xapi.h>
```

- For applications that use the legacy JDBC API, the header files are version dependent:
  - As of Connector/C++ 8.0.16, a single `#include` directive suffices:

    ```
    #include <mysql/jdbc.h>
    ```

  - Prior to Connector/C++ 8.0.16, use this set of `#include` directives:

    ```
    #include <jdbc/mysql_driver.h>
    #include <jdbc/mysql_connection.h>
    #include <jdbc/cppconn/*.h>
    ```

    The notation `<jdbc/cppconn/*.h>` means that you should include all header files from the `jdbc/cppconn` directory that are needed by your application. The particular files needed depend on the application.

  - Legacy code that uses Connector/C++ 1.1 has `#include` directives of this form:

```
#include <mysql_driver.h>
#include <mysql_connection.h>
#include <cppconn/*.h>
```

To build such code with Connector/C++ 8.0 without modifying it, add `$MYSQL_CPPCONN_DIR/include/jdbc` to the include path.

To compile code that you intend to link statically against Connector/C++, define a macro that adjusts API declarations in the header files for usage with the static library. For details, see Using the Connector/C++ Static Library.

Connector/C++ Version Macros

Starting with Connector/C++ 8.0.30, version-related macros are defined in public header files. The intent of the macros is to provide a way to systematically and predictably maintain version numbering of the Connector/C++ product. The following table describes the version-related macros.

Macro Name	Description
`MYSQL_CONCPP_VERSION_MAJOR`	Major number of the product version; currently **8**.
`MYSQL_CONCPP_VERSION_MINOR`	Minor number of the product version; currently **00**.
`MYSQL_CONCPP_VERSION_MICRO`	Micro number of the product version; initially **30**.
`MYSQL_CONCPP_VERSION_NUMBER`	Full Connector/C++ version number, which combines the major, minor, and micro numbers. For example, the combined version number **8000030** represents Connector/C++ 8.0.30.

> **Note**
>
> The version numbers maintained by these macros apply to the Connector/C++ product only and are unrelated to API or ABI versions, which are handled separately.

Connector/C++ applications that use X DevAPI, X DevAPI for C, or the legacy JDBC API can specify the `MYSQL_CONCPP_VERSION_NUMBER` macro to add conditional tests that determine the inclusion or exclusion of feature dependencies, based on which Connector/C++ version introduced the dependency. For example, it is possible to use the `MYSQL_CONCPP_VERSION_NUMBER` macro in the following cases:

- When a Connector/C++ application needs a guard that checks for features introduced after the specified version. The following example specifies version 8.0.32, which has the macro defined in public header files. The same conditional-compilation directive also works when the macro is not defined (with pre-8.0.30 header files), because the value is treated as 0.

```
#if MYSQL_CONCPP_VERSION_NUMBER > 8000032
  // use some 8.0.32+ feature
#endif
```

- When a Connector/C++ application requires all features introduced before the specified version.

```
#if MYSQL_CONCPP_VERSION_NUMBER < 8000032
  // this usage is OK; it compiles with 8.0.31 and all previous versions
#endif
```

- When a Connector/C++ application that uses X DevAPI also uses the `CharacterSet::utf8mb3` enumeration constant or any of the new `utf8mb4` collation members. If the application compiles with the pre-8.0.30 connector, then it is possible to guard the use of these new API elements.

```
#if MYSQL_CONCPP_VERSION_NUMBER >= 8000030
  if (CharacterSet::utf8mb3 == cs)
#else
```

```
  if (CharacterSet::utf8 == cs)
#endif
  {
    // cs is the id of the utf8 character set
  }
```

- When a Connector/C++ application that uses X DevAPI needs to check the name of the `utf8mb3` character set or any of its collations, and it must also be compiled with the pre-8.0.30 connector.

```
#if MYSQL_CONCPP_VERSION_NUMBER >= 8000030
  if ("utf8mb3" == characterSetName(cs))
#else
  if ("utf8" == characterSetName(cs))
#endif
  {
    // cs is the id of the utf8 character set
  }
```

> **Note**
>
> Alternatively, you can compare against numeric enumeration constant value, which should work regardless of the connector version.

- When a Connector/C++ application that uses the legacy JDBC API needs to check the name of the `utf8mb3` character set or any of its collations, and it must also be compiled with the pre-8.0.30 connector.

```
#if MYSQL_CONCPP_VERSION_NUMBER >= 8000030
  if ("utf8mb3" == metadata->getColumnCharset(column))
#else
  if ("utf8" == metadata->getColumnCharset(column))
#endif
  {
    // column is the column index using the utf8 character set
  }
```

Do not use the `MYSQL_CONCPP_VERSION_NUMBER` macro to check against versions earlier than Connector/C++ 8.0.30, which can produce unreliable results. For example:

```
#if MYSQL_CONCPP_VERSION_NUMBER > 8000028
  // this does not compile the with 8.0.29 connector!
#endif

#if MYSQL_CONCPP_VERSION_NUMBER < 8000028
  // this compiles with the 8.0.29 connector!
#endif
```

Boost Header Files

The Boost header files are needed under these circumstances:

- Prior to Connector/C++ 8.0.16, on Unix and Unix-like platforms for applications that use X DevAPI or X DevAPI for C, if you build using `gcc` and the version of the C++ standard library on your system does not implement the UTF8 converter (`codecvt_utf8`).
- Prior to Connector/C++ 8.0.23, to compile Connector/C++ applications that use the legacy JDBC API.

If the Boost header files are needed, Boost 1.59.0 or newer must be installed, and the location of the headers must be added to the include path. To obtain Boost and its installation instructions, visit the official Boost site.

Link Libraries

When running an application that uses the shared Connector/C++ library, the library and its runtime dependencies must be found by the dynamic linker. The dynamic linker must be properly configured to

find Connector/C++ libraries and their dependencies. This includes adding `-lresolv` explicitly to the compile/link command.

Building Connector/C++ using OpenSSL makes the connector library dependent on OpenSSL dynamic libraries. In that case:

- When linking an application to Connector/C++ dynamically, this dependency is relevant only at runtime.
- When linking an application to Connector/C++ statically, link to the OpenSSL libraries as well. On Linux, this means adding `-lssl -lcrypto` explicitly to the compile/link command. On Windows, this is handled automatically.

On Windows, link to the dynamic version of the C++ Runtime Library.

Runtime Libraries

X DevAPI for C applications need `libstdc++` at runtime. Depending on your platform or build tools, a different library may apply. For example, the library is `libc++` on macOS; see Section 5.2.2, “macOS Notes”.

If an application is built using dynamic link libraries, those libraries must be present not just on the build host, but on target hosts where the application runs. The dynamic linker must be properly configured to find those libraries and their runtime dependencies, as well as to find Connector/C++ libraries and their runtime dependencies.

Connector/C++ libraries built by Oracle depend on the OpenSSL libraries. The latter must be installed on the system in order to run code that links against Connector/C++ libraries. Another option is to put the OpenSSL libraries in the same location as Connector/C++, in which case, the dynamic linker should find them next to the connector library. See also Section 5.2.1, “Windows Notes”, and Section 5.2.2, “macOS Notes”.

> **Note**
>
> The TLSv1 and TLSv1.1 connection protocols are no longer supported as of Connector/C++ 8.0.28, making TLSv1.2 the earliest supported connection protocol.

Using the Connector/C++ Dynamic Library

The Connector/C++ dynamic library name depends on the platform. These libraries implement X DevAPI and X DevAPI for C, where *A* in the library name represents the ABI version:

- `libmysqlcppconn8.so.`*A* (Unix)
- `libmysqlcppconn8.`*A*`.dylib` (macOS)
- `mysqlcppconn8-`*A*`-vs`*NN*`.dll`, with import library `vs`*NN*`/mysqlcppconn8.lib` (Windows)

For the legacy JDBC API, the dynamic libraries are named as follows, where *B* in the library name represents the ABI version:

- `libmysqlcppconn.so.`*B* (Unix)
- `libmysqlcppconn.`*B*`.dylib` (macOS)
- `mysqlcppconn-`*B*`-vs`*NN*`.dll`, with import library `vs`*NN*`/mysqlcppconn-static.lib` (Windows)

On Windows, the `vs`*NN* value in library names depends on the MSVC toolchain version used to build the libraries. (Connector/C++ libraries provided by Oracle use `vs14`, and they are compatible with MSVC 2019 and 2017.) This convention enables using libraries built with different versions of MSVC on the same system. See also Section 5.2.1, “Windows Notes”.

To build code that uses X DevAPI or X DevAPI for C, add `-lmysqlcppconn8` to the linker options. To build code that uses the legacy JDBC API, add `-lmysqlcppconn`.

You must also indicate whether to use the 64-bit or 32-bit libraries by specifying the appropriate library directory. Use an `-L` linker option to specify `$MYSQL_CONCPP_DIR/lib64` (64-bit libraries) or `$MYSQL_CONCPP_DIR/lib` (32-bit libraries), where `$MYSQL_CPPCONN_DIR` is the Connector/C++ installation location. On FreeBSD, `/lib64` is not used. The library name always ends with `/lib`.

To build a Connector/C++ application that uses X DevAPI, has sources in `app.cc`, and links dynamically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR = Connector/C++ installation location
CPPFLAGS = -I $(MYSQL_CONCPP_DIR)/include -L $(MYSQL_CONCPP_DIR)/lib64
LDLIBS = -lmysqlcppconn8
CXXFLAGS = -std=c++17
app : app.cc
```

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
g++ -std=c++17 -I .../include -L .../lib64 app.cc -lmysqlcppconn8 -o app
```

To build a plain C application that uses X DevAPI for C, has sources in `app.c`, and links dynamically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR = Connector/C++ installation location
CPPFLAGS = -I $(MYSQL_CONCPP_DIR)/include -L $(MYSQL_CONCPP_DIR)/lib64
LDLIBS = -lmysqlcppconn8
app : app.c
```

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
cc -I .../include -L .../lib64 app.c -lmysqlcppconn8 -o app
```

> **Note**
>
> The resulting code, even though it is compiled as plain C, depends on the C++ runtime (typically `libstdc++`, though this may differ depending on platform or build tools; see Runtime Libraries).

To build a plain C++ application that uses the legacy JDBC API, has sources in `app.c`, and links dynamically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR = Connector/C++ installation location
CPPFLAGS = -I $(MYSQL_CONCPP_DIR)/include -L $(MYSQL_CONCPP_DIR)/lib64
LDLIBS = -lmysqlcppconn
app : app.c
```

The library option in this case is `-lmysqlcppcon`, rather than `-lmysqlcppcon8` as for an X DevAPI or X DevAPI for C application.

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
cc -I .../include -L .../lib64 app.c -lmysqlcppconn -o app
```

> **Note**
>
> When running an application that uses the Connector/C++ dynamic library, the library and its runtime dependencies must be found by the dynamic linker. See Runtime Libraries.

Using the Connector/C++ Static Library

It is possible to link your application with the Connector/C++ static library. This way there is no runtime dependency on the connector, and the resulting binary can run on systems where Connector/C++ is not installed.

> **Note**
>
> Even when linking statically, the resulting code still depends on all runtime dependencies of the Connector/C++ library. For example, if Connector/C++ is built using OpenSSL, the code has a runtime dependency on the OpenSSL libraries. See Runtime Libraries.

The Connector/C++ static library name depends on the platform. These libraries implement X DevAPI and X DevAPI for C:

- `libmysqlcppconn8-static.a` (Unix, macOS)
- `vs*NN*/mysqlcppconn8-static.lib` (Windows)

For the legacy JDBC API, the static libraries are named as follows:

- `libmysqlcppconn-static.a` (Unix, macOS)
- `vs*NN*/mysqlcppconn-static.lib` (Windows)

> **Note**
>
> Generic Linux packages do not contain any Connector/C++ static libraries. If you intend to link your application to a static library, consider installing a package that is specific to the platform on which you build your final application.

On Windows, the `vs*NN*` value in library names depends on the MSVC toolchain version used to build the libraries. (Connector/C++ libraries provided by Oracle use `vs14`, and they are compatible with MSVC 2019 and 2017.) This convention enables using libraries built with different versions of MSVC on the same system. See also Section 5.2.1, “Windows Notes”.

To compile code that you intend to link statically against Connector/C++, define a macro that adjusts API declarations in the header files for usage with the static library. One way to define the macro is by passing a `-D` option on the compiler invocation command:

- For applications that use X DevAPI, X DevAPI for C, or (as of Connector/C++ 8.0.16) the legacy JDBC API, define the `STATIC_CONCPP` macro. All that matters is that you define it; the value does not matter. For example: `-DSTATIC_CONCPP`
- Prior to Connector/C++ 8.0.16, for applications that use the legacy JDBC API, define the `CPPCONN_PUBLIC_FUNC` macro as an empty string. To ensure this, define the macro as `CPPCONN_PUBLIC_FUNC=`, not as `CPPCONN_PUBLIC_FUNC`. For example: `-DCPPCONN_PUBLIC_FUNC=`

To build a Connector/C++ application that uses X DevAPI, has sources in `app.cc`, and links statically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR = Connector/C++ installation location
CPPFLAGS = -DSTATIC_CONCPP -I $(MYSQL_CONCPP_DIR)/include
LDLIBS = $(MYSQL_CONCPP_DIR)/lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread
CXXFLAGS = -std=c++17
app : app.cc
```

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
g++ -std=c++17 -DSTATIC_CONCPP -I .../include app.cc
  .../lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread -o app
```

> **Note**
>
> To avoid having the linker report unresolved symbols, the compile line must include the OpenSSL libraries and the `pthread` library on which Connector/C++ code depends.

> OpenSSL libraries are not needed if Connector/C++ is built without them, but Connector/C++ distributions built by Oracle do depend on OpenSSL.
>
> The exact list of libraries required by Connector/C++ library depends on the platform. For example, on Solaris, the `socket`, `rt`, and `nsl` libraries might be needed.

To build a plain C application that uses X DevAPI for C, has sources in `app.c`, and links statically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR = Connector/C++ installation location
CPPFLAGS = -DSTATIC_CONCPP -I $(MYSQL_CONCPP_DIR)/include
LDLIBS = $(MYSQL_CONCPP_DIR)/lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread
app : app.c
```

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
cc -DSTATIC_CONCPP -I .../include app.c
  .../lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread -o app
```

To build a plain C application that uses the legacy JDBC API, has sources in `app.c`, and links statically to the connector library, the `Makefile` might look like this:

```
MYSQL_CONCPP_DIR = Connector/C++ installation location
CPPFLAGS = -DCPPCONN_PUBLIC_FUNC= -I $(MYSQL_CONCPP_DIR)/include
LDLIBS = $(MYSQL_CONCPP_DIR)/lib64/libmysqlcppconn-static.a -lssl -lcrypto -lpthread
app : app.c
```

The library option in this case names `libmysqlcppcon-static.a`, rather than `libmysqlcppcon8-static.a` as for an X DevAPI or X DevAPI for C application.

With that `Makefile`, the command `make app` generates the following compiler invocation:

```
cc -std=c++17 --DCPPCONN_PUBLIC_FUNC= -I .../include app.c
  .../lib64/libmysqlcppconn-static.a -lssl -lcrypto -lpthread -o app
```

When building plain C code, it is important to take care of connector's dependency on the C++ runtime, which is introduced by the connector library even though the code that uses it is plain C:

- One approach is to ensure that a C++ linker is used to build the final code. This approach is taken by the `Makefile` shown here:

  ```
  MYSQL_CONCPP_DIR = Connector/C++ installation location
  CPPFLAGS = -DSTATIC_CONCPP -I $(MYSQL_CONCPP_DIR)/include
  LDLIBS = $(MYSQL_CONCPP_DIR)/lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread
  LINK.o = $(LINK.cc) # use C++ linker
  app : app.o
  ```

  With that `Makefile`, the build process has two steps: first compile the application source in `app.c` using a plain C compiler to produce `app.o`, then link the final executable (`app`) using the C++ linker, which takes care of the dependency on the C++ runtime. The commands look something like this:

  ```
  cc -DSTATIC_CONCPP -I .../include -c -o app.o app.c
  g++ -DSTATIC_CONCPP -I .../include app.o
    .../libmysqlcppconn8-static.a -lssl -lcrypto -lpthread -o app
  ```

- Another approach is to use a plain C compiler and linker, but add the `libstdc++` C++ runtime library as an explicit option to the linker. This approach is taken by the `Makefile` shown here:

  ```
  MYSQL_CONCPP_DIR = Connector/C++ installation location
  CPPFLAGS = -DSTATIC_CONCPP -I $(MYSQL_CONCPP_DIR)/include
  LDLIBS = $(MYSQL_CONCPP_DIR)/lib64/libmysqlcppconn8-static.a -lssl -lcrypto -lpthread -lstdc++
  app : app.c
  ```

  With that `Makefile`, the compiler is invoked as follows:

  ```
  cc -DSTATIC_CONCPP -I .../include app.c
  ```

```
.../libmysqlcppconn8-static.a -lssl -lcrypto -lpthread -lstdc++ -o app
```

> **Note**
>
> Even if the application that uses Connector/C++ is written in plain C, the final executable depends on the C++ runtime which must be installed on the target computer on which the application is to run.

5.2 Building Connector/C++ Applications: Platform-Specific Considerations

This section discusses platform-specific considerations to keep in mind when building Connector/C++ applications. For general considerations that apply on a platform-independent basis, see Section 5.1, “Building Connector/C++ Applications: General Considerations”.

5.2.1 Windows Notes

This section describes aspects of building Connector/C++ applications that are specific to Microsoft Windows. For general application-building information, see Section 5.1, “Building Connector/C++ Applications: General Considerations”.

On Windows, applications can be built in different build configurations, which determine the type of the C++ runtime library that is used by the final executable:

- An application can be built in 32-bit or 64-bit mode.
- An application can be built in release or debug mode.
- You can choose between the dynamic runtime library (`/MD` linker option) or static runtime library (`/MT` linker option). Different versions of the MSVC compiler also use different versions of the runtime library.

To build Connector/C++ applications, developers using Windows must satisfy these conditions:

- An acceptable version of Microsoft Visual Studio is required.
- Applications should use the same build configuration as that used to build Connector/C++. Build configuration includes the build mode (release mode or debug mode) and the linker option (for example, `/MD` or `/MDd`).
- Target hosts running client applications must have an acceptable version of the Visual C++ Redistributable for Visual Studio installed.

For information about acceptable versions of Visual Studio and VC++ Redistributable, see Platform Support and Prerequisites.

The following sections provide additional detail about several aspects of building Connector/C++ applications:

- Application Build Configuration Must Match Connector/C++
- Linking Connector/C++ to Applications
- Building Connector/C++ Applications with Microsoft Visual Studio

Application Build Configuration Must Match Connector/C++

It is important to use a compatible compiler version to build applications and Connector/C++. It is also important to build applications using the same build configuration as that used to build Connector/C++. That is, applications should use the same build mode and linker option, to ensure that the connector and the application use the same runtime library.

The following table shows the linker option appropriate for each combination of build mode and runtime library. It also shows for each combination whether a Connector/C++ binary package is available from Oracle. (If not, you must build Connector/C++ from source yourself.)

Table 5.1 Connector/C++ Linker Option Per Build Mode and Runtime Library

Build Mode	Runtime Library	Linker Option	Binary Package Available
Release	Dynamic	`/MD`	Yes
Debug	Dynamic	`/MDd`	Yes
Release	Static	`/MT`	No (build from source)
Debug	Static	`/MTd`	No (build from source)

Standard Connector/C++ binary packages available from Oracle are built in release mode. If you install such a package, build applications in release mode to match. Oracle packages built in debug mode are available as well. To build applications in debug mode, you must either install an Oracle-built Connector/C++ package that was built in debug mode, or build Connector/C++ from source yourself using debug mode.

Linking Connector/C++ to Applications

Connector/C++ binary distributions are available as 64-bit or 32-bit packages, which store libraries under a directory named `lib64` or `lib`, respectively. Package names and certain library file and directory names also include `vs`*`NN`*. The `vs`*`NN`* value in these names depends on the MSVC toolchain version used to build the libraries. This convention enables using libraries built with different versions of MSVC on the same system.

> **Note**
>
> The `vs`*`NN`* value represents the major version of the MSVC toolchain used to build the libraries. Currently it is `vs14`, which is the toolchain used by MSVC 2015 through 2019.

Connector/C++ binary packages include libraries built using the dynamic runtime library in either release mode (`/MD`) or debug mode (`/MDd`). The Connector/C++ libraries are compatible with MSVC 2019 and 2017, and code that uses these libraries can be built with either MSVC 2019 or 2017 using the appropriate linker option (that is, `/MD` for release mode or `/MDd` for debug mode). To build code with a different linker option (`/MT` or `/MTd`), first build Connector/C++ from source with that option (see Section 4.3, “Installing Connector/C++ from Source”), then build applications using the same option.

> **Note**
>
> One exception for compiler version compatibility is that to build applications using the static JDBC legacy connector, MSVC 2019 is required; 2017 does not work.

Connector/C++ is available as a dynamic or static library to use with your application. Which library you choose determines the library files needed, and the location of those files within a Connector/C++ package depends on whether the package was built in release or debug mode. Library files are located under the library directory, which, as previously mentioned, is `lib64` for 64-bit packages or `lib` for 32-bit packages. Denote this directory as *`LIB`*. The following table shows the directory in which to find library files for each type of library (including import libraries, which are used in conjunction with dynamic libraries).

Table 5.2 Connector/C++ Library File Directories

Library Type	Library File Directory (Release Build)	Library File Directory (Debug Build)
Dynamic Library	*`LIB`*	*`LIB`*`/debug`

Library Type	Library File Directory (Release Build)	Library File Directory (Debug Build)
Import Library	*LIB*/vs14	*LIB*/vs14/debug
Static Library	*LIB*/vs14	*LIB*/vs14/debug

For dynamic linking, the following table indicates which dynamic and import library files to use.

Table 5.3 Connector/C++ Dynamic and Import Library Files Per Connector

Connector	Dynamic Library File	Import Library File
X DevAPI, X DevAPI for C	mysqlcppconn8-2-vs14.dll	mysqlcppconn8.lib
JDBC	mysqlcppconn-7-vs14.dll	mysqlcppconn.lib

For the X DevAPI or X DevAPI for C connector, use the dynamic library file named mysqlcppconn8-2-vs14.dll, together with with the import library file named mysqlcppconn8.lib from the import library directory. The 2 in the dynamic library name is the major ABI version number. (This helps when using compatibility libraries with an old ABI together with new libraries having a different ABI.) The libraries installed on your system may have a different ABI version in their file names.

For the legacy JDBC connector, use the dynamic library file named mysqlcppconn-7-vs14.dll, together with the import library file named mysqlcppconn.lib from the import library directory.

For static linking, the following table indicates which static library file to use.

Table 5.4 Connector/C++ Static Library File Per Connector

Connector	Static Library File
X DevAPI, X DevAPI for C	mysqlcppconn8-static.lib
JDBC	mysqlcppconn-static.lib

For the X DevAPI or X DevAPI for C connector, use the static library file named mysqlcppconn8-static.lib from the static library directory.

For the legacy JDBC connector, use the static library file named mysqlcppconn-static.lib from the static library directory.

When building code that uses Connector/C++ libraries, use these guidelines for setting build options in the project configuration:

- As an additional include directory, specify $MYSQL_CPPCONN_DIR/include.
- As an additional library directory, specify the directory containing the libraries the application must link to, as indicated in Table 5.2, “Connector/C++ Library File Directories”. For example, to specify the import or static library directory for building in release mode, use $MYSQL_CONCPP_DIR/lib64/vs14 (for 64-bit libraries) or $MYSQL_CONCPP_DIR/lib/vs14 (for 32-bit libraries). For building in debug mode, change vs14 to vs14/debug.
- To use a dynamic library file (.dll extension), link your application with a .lib import library: mysqlcppconn8.lib to the linker options, or mysqlcppconn.lib for legacy code.
- To use a static library file (.lib extension), link your application with the library: mysqlcppconn8-static.lib, or mysqlcppconn-static.lib for legacy code.

For static linking, the application must also be linked with import libraries for the required OpenSSL libraries. If the connector was installed from a binary package provided by Oracle, these are present in the vs14 subdirectory under the main library directory ($MYSQL_CONCPP_DIR/lib64 or

`$MYSQL_CONCPP_DIR/lib`), and the corresponding OpenSSL `.dll` libraries are present in the main library directory.

> **Note**
>
> A Windows application that uses the connector dynamic library must be able to locate it at runtime, as well as its dependencies such as OpenSSL. The common way of arranging this is to copy all the required DLLs to the same location as the application executable.

Building Connector/C++ Applications with Microsoft Visual Studio

To build a Connector/C++ application with Microsoft Visual Studio, follow this procedure:

1. Start a new Visual C++ project in Visual Studio.
2. Set the required include paths.

   From the main menu, select **Project**, **Properties**. This can also be accessed using the hot key **ALT** + **F7**. Under **Configuration Properties**, open the tree view. Select **C/C++**, **General** in the tree view.

   In the **Additional Include Directories** text field:

   - Add the `include/` directory of Connector/C++. This directory should be located within the Connector/C++ installation directory.
   - If Boost is required to build the application, also add the Boost library root directory. (See Section 5.1, “Building Connector/C++ Applications: General Considerations”.)
3. Set the library locations.

   In the tree view, open **Linker**, **General**, **Additional Library Directories**.

   In the **Additional Library Directories** text field, add the Connector/C++ import or static library directory as specified in Table 5.2, “Connector/C++ Library File Directories”. Set appropriate paths for release and debug builds.

   > **Note**
   >
   > For building in debug mode, the Connector/C++ debug package must be installed.
4. Set the connector library to use.

   Open **Linker**, **Input** in the **Property Pages** dialog.

   For building with the Connector/C++ dynamic library, enter the import library name: `mysqlcppconn8.lib`, or `mysqlcppconn.lib` for legacy applications.

   For building with the Connector/C++ static library, enter the static library name: `mysqlcppconn8-static.lib`, or `mysqlcppconn-static.lib` for legacy applications.

   > **Note**
   >
   > Generic Linux packages do not contain Connector/C++ static libraries.
5. Define macros for static linking.

   To compile code that is linked statically with the connector library, you must define a macro that adjusts API declarations in the header files for usage with the static library. By default, the macro is undefined to declare functions to be compatible with an application that calls a DLL.

In the **Project**, **Properties** tree view, under **C++**, **Preprocessor**, enter the appropriate macro into the **Preprocessor Definitions** text field:

- For applications that use X DevAPI, X DevAPI for C, or (as of Connector/C++ 8.0.16) the legacy JDBC API, define the `STATIC_CONCPP` macro. All that matters is that you define it; the value does not matter. For example: `-DSTATIC_CONCPP`
- Prior to Connector/C++ 8.0.16, for applications that use the legacy JDBC API, define the `CPPCONN_PUBLIC_FUNC` macro as an empty string. To ensure this, define the macro as `CPPCONN_PUBLIC_FUNC=`, not as `CPPCONN_PUBLIC_FUNC`.

Notes

- Target hosts running the client application must have the Visual C++ Redistributable for Visual Studio installed. For information about which VC++ Redistributable versions are acceptable, see Platform Support and Prerequisites.
- If your code uses the Connector/C++ dynamic library, it must be present on the target host where the application is run. Copy the appropriate Connector/C++ dynamic library to the same directory as the application executable (see Linking Connector/C++ to Applications). Alternatively, extend the `PATH` environment variable using `SET PATH=%PATH%;C:\path\to\cpp`, or copy the dynamic library to the Windows installation directory, typically `C:\windows`.
- If your code uses the Connector/C++ static library, the required OpenSSL libraries must be found on the target host where the application is run. For Connector/C++ binary distributions, the OpenSSL `.dll` libraries are present in the main library directory (`$MYSQL_CONCPP_DIR/lib64` or `$MYSQL_CONCPP_DIR/lib`). Copy them to the same location as the application executable or to some directory listed in the system `PATH`.

5.2.2 macOS Notes

This section describes aspects of building Connector/C++ applications that are specific to macOS. For general application-building information, see Section 5.1, “Building Connector/C++ Applications: General Considerations”.

The binary distribution of Connector/C++ for macOS is compiled using the macOS native `clang` compiler. For that reason, an application that uses Connector/C++ should be built with the same `clang` compiler.

The `clang` compiler can use two different implementations of the C++ runtime library: either the native `libc++` or the GNU `libstdc++` library. It is important that an application uses the same runtime implementation as Connector/C++ that is, the native `libc++`. To ensure that, the `-stdlib=libc++` option should be passed to the compiler and the linker invocations.

To build a Connector/C++ application that uses X DevAPI, has sources in `app.cc`, and links dynamically to the connector library, the `Makefile` for building on macOS might look like this:

```
MYSQL_CONCPP_DIR = Connector/C++ installation location
CPPFLAGS = -I $(MYSQL_CONCPP_DIR)/include -L $(MYSQL_CONCPP_DIR)/lib64
LDLIBS = -lmysqlcppconn8
CXX = clang++ -stdlib=libc++
CXXFLAGS = -std=c++17
app : app.cc
```

Binary packages for macOS include OpenSSL libraries that are required by code linked with the connector. These libraries are installed in the same location as the connector libraries and should be found there by the dynamic linker.

5.2.3 Generic Linux Notes

This section describes aspects of building Connector/C++ applications that are specific to Linux. Generic Linux packages do not contain Connector/C++ static libraries. For general application-building information, see Section 5.1, “Building Connector/C++ Applications: General Considerations”.

> **Note**
>
> Connector/C++ 8.0.32 provides generic Linux packages for ARM architecture (64 bit). All Connector/C++ versions provide generic Linux packages for Intel architecture (both 32 and 64 bits).

Previously, generic Linux packages were built on the EL7 platform and on that platform GCC is configured to use an older ABI of `libstdc++`. Some of the symbols exported by the library include standard library types in their names, and consequently, are not compatible with the new `CXX11` ABI, which is the default for modern GCC on most platforms (EL7 being an exception). So, unless you build your code on EL7, and use GCC6 or later compiler, it defaults to new `CXX11` ABI and looks for Connector/C++ symbols that have new ABI names in them.

As of Connector/C++ 8.0.30, Connector/C++ uses the new CXX11 ABI. With this change, you might encounter following problems when using Connector/C++ installed from a generic Linux package:

- An upgrade from Connector/C++ 8.0.29 (or earlier) to 8.0.30 (or later) could produce runtime errors after the upgrade, even if the previous version of Connector/C++ ran successfully.
- It will not work with GCC5 or earlier, because the old compiler uses the old ABI and cannot link to code that uses new the ABI.
- It will not work on EL6, EL7, or any other platform that modifies GCC settings to use the old ABI by default. However, in this situation a workaround is to build code under `-D_GLIBCXX_USE_CXX11_ABI=1`.

For a majority of platforms, including EL8, the GCC default was changed to the new ABI.

5.3 Authentication Support

For connections to the server made using the legacy JDBC API (that is, not made using X DevAPI or X DevAPI for C), Connector/C++ supports different client-side authentication plugins and authentication methods for:

- LDAP Authentication
- Kerberos Authentication
- OCI Authentication
- Multifactor Authentication
- FIDO Authentication
- WebAuthn Authentication

LDAP Authentication

LDAP authentication enables Connector/C++ (8.0.22 and later) application programs to connect to MySQL servers using simple LDAP authentication, or SASL LDAP authentication using the SCRAM-SHA-1 authentication method. LDAP authentication requires use of a server from a MySQL Enterprise Edition distribution. For more information about the LDAP authentication plugins, see LDAP Pluggable Authentication.

Connector/C++ binary distributions include the libraries that provide the client-side LDAP authentication plugins, as well as any dependent libraries required by the plugins.

> **Note**
>
> In Connector/C++ 8.0.23, a dependency on the `mysql-client-plugins` package was removed. This package now is required only on hosts where Connector/C++ applications make connections using commercial MySQL server accounts with LDAP authentication. In that case, additional libraries must also be installed: `cyrus-sasl-scram` for installations that use RPM packages and `libsasl2-modules-gssapi-mit` for installations that use Debian packages. These SASL packages provide the support required to use the SCRAM-SHA-256 and GSSAPI/Kerberos authentication methods for LDAP.

If Connector/C++ was installed from a compressed `tar` file or Zip archive, the application program will need to set the `OPT_PLUGIN_DIR` connection option to the appropriate directory so that the bundled plugin library can be found. (Alternatively, copy the required plugin library to the default directory expected by the client library.)

For example:

```
sql::ConnectOptionsMap connection_properties;

// To use simple LDAP authentication ...

connection_properties["userName"] = "simple_ldap_user_name";
connection_properties["password"] = "simple_ldap_password";
connection_properties[OPT_ENABLE_CLEARTEXT_PLUGIN]=true;

// To use SASL LDAP authentication using SCRAM-SHA-1 ...

connection_properties["userName"] = "sasl_ldap_user_name";
connection_properties["password"] = "sasl_ldap_scram_password";

// Needed if Connector/C++ was installed from tar file or Zip archive ...

connection_properties[OPT_PLUGIN_DIR] = "${INSTALL_DIR}/lib{64}/plugin";

auto *driver = get_driver_instance();
auto *con = driver->connect(connection_properties);

// Execute statements ...

con->close();
```

Kerberos Authentication

Kerberos authentication enables Connector/C++ application programs to establish connections for accounts that use the `authentication_kerberos` server-side authentication plugin, provided that the correct Kerberos tickets are available or can be obtained from Kerberos. This capability is available on client hosts running Linux (starting with 8.0.26).

On Windows (starting with 8.0.32), the `OPT_AUTHENTICATION_KERBEROS_CLIENT_MODE` connection option can be set to either `SSPI` (default) or `GSSAPI`. The option permits choosing between SSPI and GSSAPI at runtime for the `authentication_kerberos_client` authentication plugin on Windows. Connector/C++ implements `GSSAPI` mode through the MIT kerberos library and this mode is compatible with the Java SE security tools (for example, `klist` and `kinit` commands) on Windows. In this mode, the ticket search on Windows hosts is restricted to the MIT Kerberos cache only. If the cache has no ticket, the connection fails even if the Windows ticket is valid

Previously, Connector/C++ supported Kerberos authentication through the Windows SSPI Kerberos library only (starting with 8.0.27). SSPI is not capable of acquiring cached credentials that were generated using the `kinit` command. In `SSPI` mode, the Windows single sign-on ticket is used for authentication if the client user provides no password and the authentication method considers the

Windows ticket exclusively. If the ticket is missing or invalid, the connection fails even if the Kerberos cache contains a valid ticket. For more information, see Commands for Windows Clients in SSPI Mode.

It is possible to connect to Kerberos-authenticated accounts without giving a user name under these conditions:

- The user has a Kerberos principal name assigned, a MySQL Kerberos account for that principal name exists, and the user has the required tickets.
- The default authentication method must be set to the `authentication_kerberos_client` client-side authentication plugin using the `OPT_DEFAULT_AUTH` connection option.

It is possible to connect without giving a password, provided that the user has the required tickets in the Kerberos cache on Linux or the MIT Kerberos cache on Windows (for example, created by `kinit` or a similar command).

> **Note**
>
> The SSPI Kerberos library is not compatible with Java SE security tools. To use the `kinit` command, the client application must set the `OPT_AUTHENTICATION_KERBEROS_CLIENT_MODE` connection option to `GSSAPI`.

If the required tickets are not present in the Kerberos cache (or the MIT Kerberos cache) and a password was given, Connector/C++ obtains the tickets from Kerberos using that password. If the required tickets are found in the cache, any password given is ignored and the connection might succeed *even if the password is incorrect.*

On client hosts running Windows, you can override the default location of the MIT Kerberos configuration file by setting the `KRB5_CONFIG` environment variable and the default MIT Kerberos credential cache name with the `KRB5CCNAME` environment variable (for example, `KRB5CCNAME=DIR:/mydir/`).

For details about using the MIT Kerberos configuration and cache, see:

- `KRB5_CONFIG`: https://web.mit.edu/kerberos/krb5-devel/doc/admin/conf_files/krb5_conf.html
- `KRB5CCNAME`: https://web.mit.edu/kerberos/krb5-1.12/doc/basic/ccache_def.html

For more information about Kerberos authentication, see Kerberos Pluggable Authentication.

OCI Authentication

OCI authentication enables Connector/C++ application programs to make connections without passwords for accounts that use the `authentication_oci` server-side authentication plugin, provided that the correct configuration entries are available to map to one unique user in a specific Oracle Cloud Infrastructure tenancy. This supported was added in the Connector/C++ 8.0.27 release.

To ensure correct account mapping, the client-side Oracle Cloud Infrastructure configuration must contain a fingerprint of the API key to use for authentication (`fingerprint` entry) and the location of a PEM file with the private part of the API key (`key_file` entry). Both entries should be specified in the `[DEFAULT]` profile of the configuration file. In Connector/C++ 8.0.33, the `OPT_OCI_CLIENT_CONFIG_PROFILE` connection option permits selecting a profile in the configuration file to use for authentication. By default, the value of `OPT_OCI_CLIENT_CONFIG_PROFILE` is the `[DEFAULT]` profile.

Unless an alternative path to the configuration file is specified with the `OPT_OCI_CONFIG_FILE` connection option, the following default locations are used:

- `~/.oci/config` on Linux or Posix host types
- `%HOMEDRIVE%%HOMEPATH%/.oci/config` on Windows host types

If the MySQL user name is not provided as a connection option, then the operating system user name is substituted. Specifically, if the private key and correct Oracle Cloud Infrastructure configuration are present on the client side, then a connection can be made without giving any options.

To support Oracle Cloud Infrastructure ephemeral key-based authentication, Connector/C++ 8.0.33 (and later) obtains the location of the token file from the `security_token_file` entry. For example:

```
[DEFAULT]
fingerprint=59:8a:0b[...]
key_file=~/.oci/sessions/DEFAULT/oci_api_key.pem
tenancy=ocid1.tenancy.oc1.[...]
region=us-ashburn-1
security_token_file=~/.oci/sessions/DEFAULT/token
```

Connector/C++ sends to the server a JSON attribute (named `"token"`) with the value extracted from the `security_token_file` field. If the target file referenced in the profile does not exist, or if the file exceeds a specified maximum value, then Connector/C++ terminates the action and returns an exception with the cause.

Connector/C++ sends an empty token value in the JSON payload if:

- The security-token file is empty.
- The configuration option `security_token_file` is found but the value in the configuration file is empty.

In all other cases, Connector/C++ adds the content of the security-token file intact to the JSON document.

Multifactor Authentication

Starting with Connector/C++ 8.0.28, applications can establish connections using multifactor authentication, such that up to three passwords can be specified at connect time. The `OPT_PASSWORD1`, `OPT_PASSWORD2`, and `OPT_PASSWORD3` connection options are available for specifying the first, second, and third multifactor authentication passwords, respectively.

`OPT_PASSWORD1` is an alias for the existing `OPT_PASSWORD` option; if both are provided, `OPT_PASSWORD` is ignored. For more information about this authentication option, see Multifactor Authentication.

FIDO Authentication

FIDO authentication to MySQL Server supports using devices such as smart cards, security keys, and biometric readers. This authentication method is based on the Fast Identity Online (FIDO) standard. To ensure client applications using the legacy JBDC API are notified when a user is expected to interact with the FIDO device, Connector/C++ 8.0.29 (and later) implements a new `setCallback()` method in the `MySQL_Driver` class that accepts a single callback argument named `Fido_Callback`.

```
class Fido_Callback
{
public:

  Fido_Callback(std::function<void(SQLString)>);

  /**
  * Override this message to receive Fido Action Requests
  */
  virtual void FidoActionRequested(sql::SQLString msg);

};
```

Any connection created by the driver can use the callback, if needed. However, if an application does not set the callback explicitly, `libmysqlclient` determines the behavior by default, which involves printing a message to standard output.

> **Note**
>
> On Windows, the client application must run as administrator. The is a requirement of the `fido2.dll` library, which is used by the `authentication_fido` plugin.

A client application has two options for obtaining a callback from the connector:

- By passing a function or lambda to `Fido_Callback`.

```
driver->setCallBack(Fido_Callback([](SQLString msg) {...}));
```

- By implementing the virtual method `FidoActionRequested`.

```
class MyWindow : public Fido_Callback
{
  void FidoActionRequested(sql::SQLString msg) override;
};

MyWindow window;
driver->setCallBack(window);
```

Setting a new callback always removes the previous callback. To disable the active callback and restore the default behavior, pass `nullptr` as a function callback. Example:

```
driver->setCallBack(Fido_Callback(nullptr));
```

For more information about FIDO authentication, see FIDO Pluggable Authentication.

WebAuthn Authentication

WebAuthn authentication supports both the FIDO and FIDO2 standards. This authentication method overcomes the limitations associated with FIDO authentication that prevented WebAuthn applications like web browsers from authenticating to MySQL Server. To ensure client applications using the legacy JBDC API are notified when a user is expected to interact with the FIDO/FIDO2 device, Connector/C++ 8.2.0 (and later) adds a second callback argument named `WebAuthn_Callback` to the `setCallback()` method in the `MySQL_Driver` class that was introduced for FIDO authentication. The `WebAuthn_Callback` class has a callback method named `ActionRequested()`.

```
class WebAuthn_Callback
{
public:

  WebAuthn_Callback(std::function<void(SQLString)>);

  /**
  * Override this message to receive WebAuthn Action Requests
  */
  virtual void ActionRequested(sql::SQLString msg);

};
```

Set the `WebAuthn_Callback` callback explicitly for authentication to accounts that use WebAuthn authentication. If a `Fido_Callback` callback is registered with a driver instance, then it should be set during authentication for accounts using both FIDO and WebAuthn authentication. It is not permitted to register `Fido_Callback` after first registering `WebAuthn_Callback`.

> **Note**
>
> On Windows, the client application must run as administrator. The is a requirement of the `fido2.dll` library, which is used by the `authentication_webauthn` plugin.

A client application can obtain a callback from the connector, or disable the active callback, as shown in FIDO Authentication. Substitute `WebAuthn_Callback` and `ActionRequested()` as needed.

For more information about WebAuthn authentication, see WebAuthn Pluggable Authentication.

5.4 OpenTelemetry Tracing Support

For applications that use the legacy JDBC API (that is, not X DevAPI or X DevAPI for C) on Linux systems and use OpenTelemetry (OTel) instrumentation, the connector adds query and connection spans to the trace generated by application code and forwards the current OpenTelemetry context to the server. OpenTelemetry tracing was introduced in the Connector/C++ 8.1.0 release.

> **Note**
>
> OTel context forwarding works only with MySQL Enterprise Edition, a commercial product. To learn more about commercial products, see https://www.mysql.com/products/.

Enabling and Disabling Tracing

By default, the connector generates spans only when an instrumented application links with the required OpenTelemetry SDK libraries and configures the trace exporter to send trace data to some destination. If the application code does not use instrumentation, then the legacy connector does not use it either.

Connector/C++ supports a connection property option, `OPT_OPENTELEMETRY`, which has these values:

- `OTEL_DISABLED`: The connector does not create OpenTelemetry spans or forward the OpenTelemetry context to the server.
- `OTEL_PREFERRED`: Default. Use instrumentation in the connection if the required OpenTelemetry instrumentation is available. Otherwise, permit the connection to operate without any OpenTelemetry instrumentation.

The `OPT_OPENTELEMETRY` option also accepts a Boolean value in which `false` corresponds to `OTEL_DISABLED`. `false` is the only accepted Boolean value for this option; setting it to `true` has no meaning and emits an error.

For example, an application can specify `OPT_OPENTELEMETRY` in either form using the `connect()` syntax that takes an option map argument:

```
connection_properties["OPT_OPENTELEMETRY"] = false;
connection_properties["OPT_OPENTELEMETRY"] = OTEL_DISABLED;
```

When you build code that links to Connector/C++ and uses OTel instrumentation, the additional spans generated by the connector appear in the traces generated by your code. Spans generated by the connector are sent to the same destination (trace exporter) where other spans generated by the user code are sent as configured by user code. It is not possible to send spans generated by the connector to any other destination.

This implementation is distinct from the implementation provided through the MySQL client library (or the related `telemetry_client` client-side plugin).

Limitations

- No spans for preparing statements or executing prepared statements are created.

Chapter 6 Connector/C++ Known Issues

To report bugs, use the MySQL Bug System. See How to Report Bugs or Problems.

For notes detailing the changes in each release of Connector/C++, see MySQL Connector/C++ Release Notes.

- Generally speaking, C++ library binaries are less portable than C library binaries. Issues can be caused by name mangling, different Standard Template Library (STL) versions, and using different compilers and linkers for linking against the libraries than were used for building the library itself.

  Even a small change in the compiler version can cause problems. If you obtain error messages that you suspect are related to binary incompatibilities, build Connector/C++ from source, using the same compiler and linker that you use to build and link your application.

  Due to variations between Linux distributions, compiler versions, linker versions, and STL versions, it is not possible to provide binaries for every possible configuration. However, Connector/C++ binary distributions include an `INFO_BIN` file that describes the environment and configuration options used to build the binary versions of the connector libraries. Binary distributions also include an `INFO_SRC` file that provides information about the product version and the source repository from which the distribution was produced. (Prior to Connector/C++ 8.0.14, look for `BUILDINFO.txt` rather than `INFO_BIN` and `INFO_SRC`.)

- To avoid potential crashes, the build configuration of Connector/C++ should match the build configuration of the application using it. For example, do not use a release build of Connector/C++ with a debug build of the client application.

Chapter 7 Connector/C++ Support

For general discussion of Connector/C++, please use the C/C++ community forum.

To report bugs, use the MySQL Bug System. See How to Report Bugs or Problems.

For notes detailing the changes in each release of Connector/C++, see MySQL Connector/C++ Release Notes.

For Licensing questions, and to purchase MySQL Products and Services, please see http://www.mysql.com/buy-mysql/.

Index