Creating a relocatable `mariadb-plugin` CMake package

This guide describes a small CMake package installed by MariaDB's `Development` component. Its purpose is to let third-party developers build MariaDB plugins outside the MariaDB source tree.

The package itself can be created with MariaDB's existing CMake 3.12 minimum. External pure and client plugins can also use CMake 3.12. A dirty plugin built with MSVC should use CMake 3.15 or newer so that its runtime library can be selected reliably with CMake's `MSVC_RUNTIME_LIBRARY` target property. The optional Windows runtime-dependency installation described in Chapter 11 uses CMake 3.21 or newer. This does not raise the minimum on Linux.

After MariaDB is installed, a plugin project should be able to write:

```
find_package(mariadb-plugin CONFIG REQUIRED)
```

and use these targets:

Target	What it provides	Binary library added to the link
`MariaDB::mysqlservices`	Installed plugin API headers and `mysqlservices`	`mysqlservices` on every platform
`MariaDB::server`	Plugin API plus internal server headers and the server build's ABI settings	`server.lib` on Windows; nothing on Linux
`MariaDB::client_plugin`	MariaDB Connector/C headers needed by a client plugin	Nothing

Terminology used in this guide

"Pure" and "dirty" are not established MariaDB or CMake terminology. This guide uses them as short names for an important dependency boundary. "Dirty" is not a criticism of the plugin; it only means that the plugin deliberately depends on server internals.

Term	Definition	CMake dependency
Pure server plugin	A module loaded by `mariadbd` that includes only the public server-plugin and service headers. It calls server functionality only through `mysqlservices` and has no unresolved server symbols.	`MariaDB::mysqlservices`
Dirty server plugin	A module loaded by `mariadbd` that includes internal server headers or directly references a server symbol not provided through `mysqlservices`. It must match the installed server's private ABI.	`MariaDB::server`
Storage engine	A server plugin that uses internal types such as `handler` and `THD`. For this package it is a specialized dirty server plugin, with the conventional `ha_` output name.	`MariaDB::server`
Client plugin	A module loaded by MariaDB Connector/C, not by `mariadbd`. The example here is a client authentication plugin. It needs Connector/C headers but no MariaDB link library.	`MariaDB::client_plugin`

On Windows a dirty plugin links `server.lib`. On Linux it does not link a server library and intentionally leaves direct server references unresolved; `mariadbd` resolves them when loading the module. A pure plugin must instead resolve all its symbols at link time on every platform.

The existing in-tree `RECOMPILE_FOR_EMBEDDED` option is not a good public name for this distinction. It describes a historical build mechanism, while an external plugin needs to state which API boundary it uses. In particular, a normal dynamically loaded storage engine may require server-private headers without being "recompiled for embedded." The external convenience function therefore uses `DIRTY` and `STORAGE_ENGINE`. For source compatibility it may also accept `RECOMPILE_FOR_EMBEDDED` as a legacy alias for `DIRTY`, but new external projects should not use that historical name.

The important idea is that a CMake target describes both a library and the requirements for using it. A developer who links to `MariaDB::mysqlservices` must automatically get the correct include directory and compile definition. They should not have to find MariaDB headers separately.

`MariaDB::server` is deliberately platform-dependent. On Windows it is an imported library backed only by `server.lib`. It does not refer to or require `server.dll`. On Linux it is a header-only CMake target. A Linux server plugin is allowed to keep references to server symbols unresolved; `mariadbd` supplies those symbols when it loads the plugin.

1. Account for the current header layout

MariaDB already installs the relevant headers in the `Development` component. Keep doing that. In a normal installation the important directories are conceptually:

```
<prefix>/<INSTALL_INCLUDEDIR>/                      Connector/C headers
<prefix>/<INSTALL_INCLUDEDIR>/server/mysql/         plugin and service headers
<prefix>/<INSTALL_INCLUDEDIR>/server/private/       internal server headers
```

The names are misleading. A well-behaved server plugin does not need MariaDB server internals, but the currently installed public plugin API is nevertheless located below a directory named `server`. In particular, `mysql/plugin.h` includes `mysql/services.h`, and `mysql/services.h` includes the individual `mysql/service_*.h` files. The compiler include root needed for these headers is therefore:

```
<prefix>/<INSTALL_INCLUDEDIR>/server
```

For example, Debian installs `mysql/services.h` as `/usr/include/mariadb/server/mysql/services.h`, while the RPM layout uses `/usr/include/mysql/server/mysql/services.h`. This public plugin API directory must not be confused with `server/private`, which contains server guts used by storage engines and other dirty plugins.

This should not be treated as a permanent public path. Sergei Golubchik clarified in the MDEV-40608 review that the additional nesting was unintended: the expected RPM path was `/usr/include/mysql/plugin.h`, not `/usr/include/mysql/server/mysql/plugin.h`, and the installation should eventually be fixed.

The examples in this guide use the layout that MariaDB 11.8 packages actually contain today. When the installation rule is corrected, update `MariaDB::plugin_api` to publish the new include root. Do not make third-party projects test layouts or spell either physical path themselves. Their source continues to use `#include <mysql/plugin.h>` and their CMake code continues to link `MariaDB::mysqlservices`; only the exported target changes. This is one of the main benefits of expressing the header dependency through a CMake target.

If the corrected destination differs between installation layouts, introduce a layout-specific variable for the plugin API include root. Do not compute it in the installed config with `..`, and do not make the config guess whether a particular header layout is present.

The CMake targets below point to those installed directories. CMake does not install headers merely because an include directory is mentioned on a target; the existing `install(FILES ...)` and `install(DIRECTORY ...)` rules remain necessary.

2. Create targets for the three sets of headers

Put this in an appropriate MariaDB CMake file after the installation-directory variables have been defined. The names before `EXPORT_NAME` are private names used inside the MariaDB build. The installed names are shown in the comments.

```
# Public server-plugin API: MariaDB::plugin_api
add_library(mariadb_plugin_api INTERFACE)
set_target_properties(mariadb_plugin_api PROPERTIES
  EXPORT_NAME plugin_api
)
target_compile_definitions(mariadb_plugin_api INTERFACE
  MYSQL_DYNAMIC_PLUGIN
)
target_include_directories(mariadb_plugin_api INTERFACE
  $<BUILD_INTERFACE:${PROJECT_SOURCE_DIR}/include>
  $<BUILD_INTERFACE:${PROJECT_BINARY_DIR}/include>
  $<INSTALL_INTERFACE:${INSTALL_INCLUDEDIR}/server>
)

# Internal server API: MariaDB::server_headers
add_library(mariadb_server_headers INTERFACE)
set_target_properties(mariadb_server_headers PROPERTIES
  EXPORT_NAME server_headers
)
target_link_libraries(mariadb_server_headers INTERFACE
  mariadb_plugin_api
)
target_include_directories(mariadb_server_headers INTERFACE
  $<BUILD_INTERFACE:${PROJECT_SOURCE_DIR}/sql>
  $<BUILD_INTERFACE:${PROJECT_BINARY_DIR}/sql>
  $<INSTALL_INTERFACE:${INSTALL_INCLUDEDIR}/server/private>
)

# The installed package config adds ABI definitions such as DBUG_OFF,
# WITH_WSREP, and _FILE_OFFSET_BITS from the server build profile described
# in Chapter 7. Do not hard-code a Release profile here: Debug builders differ.

if(WIN32)
  target_compile_definitions(mariadb_server_headers INTERFACE
    _WIN32_WINNT=0x0A00
    NOMINMAX
    NOSERVICE
    WIN32_LEAN_AND_MEAN
  )
endif()

# Connector/C client-plugin API: MariaDB::client_plugin
add_library(mariadb_client_plugin INTERFACE)
set_target_properties(mariadb_client_plugin PROPERTIES
  EXPORT_NAME client_plugin
)
target_include_directories(mariadb_client_plugin INTERFACE
  $<INSTALL_INTERFACE:${INSTALL_INCLUDEDIR}>
)
```

If MariaDB also uses `mariadb_client_plugin` from its build tree, add the Connector/C source and binary include directories as `BUILD_INTERFACE` entries. They are not needed merely to create the installed package.

Why use `$<BUILD_INTERFACE:...>` and `$<INSTALL_INTERFACE:...>`? The source-tree paths are valid while MariaDB itself is being built. They must not appear in an installed package. The install entries are relative to the installation prefix, so the complete installation can later be moved.

Do not write this:

```
# Wrong: this stores the installation machine's absolute path in the package.
target_include_directories(mariadb_plugin_api INTERFACE
  ${CMAKE_INSTALL_PREFIX}/${INSTALL_INCLUDEDIR}/server
)
```

3. Export `mysqlservices`

Add the public plugin API as a usage requirement of the existing `mysqlservices` static library:

```
set_target_properties(mysqlservices PROPERTIES
  EXPORT_NAME mysqlservices
)

target_link_libraries(mysqlservices PUBLIC
  mariadb_plugin_api
)

# mysqlservices only defines service-pointer variables and uses no CRT code.
# Do not let its object files select a CRT for the consuming plugin DLL.
if(MSVC)
  target_compile_options(mysqlservices PRIVATE /Zl)
endif()

# A "pure" server plugin must resolve every symbol at link time.
# MSVC already enforces this. GNU/Linux needs --no-undefined.
# Use target_link_libraries rather than target_link_options because MariaDB
# still supports CMake 3.12; target_link_options was added in CMake 3.13.
if(CMAKE_SYSTEM_NAME STREQUAL "Linux")
  target_link_libraries(mysqlservices INTERFACE
    "${LINK_FLAG_NO_UNDEFINED}"
  )
endif()
```

`PUBLIC` is intentional. It means that a target using `mysqlservices` also inherits the plugin API include root and `MYSQL_DYNAMIC_PLUGIN`. It does not inherit `server/private`, `DBUG_OFF`, or other server-internal settings.

On MSVC, `/Zl` omits the default CRT library name from the object files in `mysqlservices.lib`. This is appropriate because those objects contain only service-pointer definitions. It prevents a pure plugin's CRT choice from being dictated by the static library. Check the result in Windows CI with:

```
dumpbin /directives mysqlservices.lib
```

The output should contain no `/DEFAULTLIB` reference to `MSVCRT`, `MSVCRTD`, `LIBCMT`, or `LIBCMTD`, and no conflicting `RuntimeLibrary` directive.

Making `--no-undefined` an interface option also makes the rule automatic: a pure plugin that uses `MariaDB::mysqlservices` cannot accidentally acquire a dependency on an unexported server symbol.

4. Split `server.lib` from `server.dll` on Windows

Do not put the real `server` shared-library target into the export set. An export of that target describes both `server.lib` and `server.dll`, which would make the development package require the server runtime package.

Instead, split the two artifacts between the existing components:

```
if(WIN32 AND TARGET server)
  install(TARGETS server
    # The DLL is needed only to run the server.
    RUNTIME DESTINATION ${INSTALL_BINDIR}
            COMPONENT Server

    # The import library is enough to link an external plugin.
    ARCHIVE DESTINATION ${INSTALL_LIBDIR}
            COMPONENT Development
  )
endif()
```

On Windows, CMake classifies a DLL as `RUNTIME` and its import library as `ARCHIVE`. Because this `install(TARGETS)` call has no `EXPORT` option, it creates no dependency from `Development` to `Server`.

Replace the existing installation of the `server` target with the split rule; do not add a second rule that installs the same DLL or import library again.

5. Install one export set

Use one export set for `mysqlservices` and its header-only dependencies. The real Windows `server` target is intentionally absent.

```
set(MARIADB_PLUGIN_CMAKEDIR
  "${INSTALL_LIBDIR}/cmake/mariadb-plugin"
)

install(TARGETS
    mysqlservices
    mariadb_plugin_api
    mariadb_server_headers
    mariadb_client_plugin
  EXPORT mariadb-plugin-targets

  # mysqlservices is a development file.
  ARCHIVE DESTINATION ${INSTALL_LIBDIR}
          COMPONENT Development
)

install(EXPORT mariadb-plugin-targets
  FILE mariadb-plugin-targets.cmake
  NAMESPACE MariaDB::
  DESTINATION "${MARIADB_PLUGIN_CMAKEDIR}"
  COMPONENT Development
)
```

If MariaDB's existing helper around `install(TARGETS)` performs additional packaging work, extend that helper to accept `EXPORT mariadb-plugin-targets` instead of bypassing it. The essential result is that `mysqlservices` and the three header targets above belong to the export set, while `server` does not.

6. Create the package config and version files

Create `cmake/mariadb-plugin-config.cmake.in`. The exported targets file creates the ordinary targets. The small block below creates `MariaDB::server` by hand:

```
CMAKE
```

```
@PACKAGE_INIT@

# Public installation-layout variables. These name the MariaDB installation
# selected by find_package(mariadb-plugin), not the consumer project's prefix.
set(MARIADB_SERVER_VERSION "@SERVER_VERSION@")
set(MARIADB_INSTALL_PREFIX "${PACKAGE_PREFIX_DIR}")
set(MARIADB_INSTALL_BINDIR "@PACKAGE_INSTALL_BINDIR@")
set(MARIADB_INSTALL_SCRIPTDIR "@PACKAGE_INSTALL_SCRIPTDIR@")
set(MARIADB_INSTALL_LIBDIR "@PACKAGE_INSTALL_LIBDIR@")
set(MARIADB_INSTALL_PLUGINDIR "@PACKAGE_INSTALL_PLUGINDIR@")
set(MARIADB_INSTALL_DOCDIR "@PACKAGE_INSTALL_DOCDIR@")
set(MARIADB_INSTALL_MANDIR "@PACKAGE_INSTALL_MANDIR@")
set(MARIADB_INSTALL_SHAREDIR "@PACKAGE_INSTALL_SHAREDIR@")
if("@INSTALL_SYSCONF2DIR@" STREQUAL "")
  set(MARIADB_INSTALL_SYSCONF2DIR "")
else()
  set(MARIADB_INSTALL_SYSCONF2DIR "@PACKAGE_INSTALL_SYSCONF2DIR@")
endif()

include("${CMAKE_CURRENT_LIST_DIR}/mariadb-plugin-targets.cmake")
include("${CMAKE_CURRENT_LIST_DIR}/mariadb-plugin-build-profile.cmake")

# Apply the fixed ABI profile of the installed server. These tests do not
# depend on the configuration chosen by the third-party project.
if(MARIADB_PLUGIN_SERVER_DBUG_OFF)
  set_property(TARGET MariaDB::server_headers APPEND PROPERTY
    INTERFACE_COMPILE_DEFINITIONS DBUG_OFF
  )
endif()

if(MARIADB_PLUGIN_SERVER_WITH_WSREP)
  set_property(TARGET MariaDB::server_headers APPEND PROPERTY
    INTERFACE_COMPILE_DEFINITIONS WITH_WSREP
  )
endif()

if(MARIADB_PLUGIN_SERVER_NDEBUG_WORKAROUND)
  set_property(TARGET MariaDB::server_headers APPEND PROPERTY
    INTERFACE_COMPILE_DEFINITIONS NDEBUG
  )
endif()

if(NOT "${MARIADB_PLUGIN_SERVER_FILE_OFFSET_BITS}" STREQUAL "")
  set_property(TARGET MariaDB::server_headers APPEND PROPERTY
    INTERFACE_COMPILE_DEFINITIONS
      "_FILE_OFFSET_BITS=${MARIADB_PLUGIN_SERVER_FILE_OFFSET_BITS}"
  )
endif()

if(WIN32)
  # This target represents the import library only. It deliberately has no
  # IMPORTED_LOCATION for server.dll and therefore does not require the DLL.
  set_and_check(_mariadb_server_import_library
    "@PACKAGE_INSTALL_LIBDIR@/server@CMAKE_IMPORT_LIBRARY_SUFFIX@"
  )
  add_library(MariaDB::server UNKNOWN IMPORTED)
  set_target_properties(MariaDB::server PROPERTIES
    IMPORTED_LOCATION "${_mariadb_server_import_library}"
    INTERFACE_LINK_LIBRARIES MariaDB::server_headers
  )
  unset(_mariadb_server_import_library)
else()
  # No linkable server library on Linux: expose only the internal headers.
  add_library(MariaDB::server INTERFACE IMPORTED)
  set_target_properties(MariaDB::server PROPERTIES
    INTERFACE_LINK_LIBRARIES MariaDB::server_headers
  )
endif()

include("${CMAKE_CURRENT_LIST_DIR}/mariadb-plugin-functions.cmake")
```

Create `cmake/mariadb-plugin-build-profile.cmake.in`. It is deliberately small and contains only facts captured from the server configuration that produced this Development package:

```
set(MARIADB_PLUGIN_SERVER_DBUG_OFF
  @MARIADB_PLUGIN_SERVER_DBUG_OFF@)
```

```
set(MARIADB_PLUGIN_SERVER_WITH_WSREP
  @MARIADB_PLUGIN_SERVER_WITH_WSREP@)
set(MARIADB_PLUGIN_SERVER_NDEBUG_WORKAROUND
  @MARIADB_PLUGIN_SERVER_NDEBUG_WORKAROUND@)
set(MARIADB_PLUGIN_SERVER_FILE_OFFSET_BITS
  "@MARIADB_PLUGIN_SERVER_FILE_OFFSET_BITS@")

# Empty on non-MSVC platforms. Use CMake's property spelling, not /MD or /MT.
set(MARIADB_PLUGIN_SERVER_MSVC_RUNTIME_LIBRARY
  "@MARIADB_PLUGIN_SERVER_MSVC_RUNTIME_LIBRARY@")
```

These are ordinary package variables as well as inputs used to construct the imported targets. They let a developer inspect the server profile, but normal plugin code should consume `MariaDB::server` and `mariadb_add_plugin()` rather than reimplementing the tests.

`UNKNOWN IMPORTED` is appropriate for `server.lib`: CMake passes that file to the linker without pretending that it is the server DLL. `set_and_check()` also gives a clear error if the `Development` component is incomplete and the import library is missing.

Do not add `--no-undefined` to `MariaDB::server`. A dirty Linux plugin is supposed to contain references that are resolved only when the server loads it. The normal Linux linker behaviour for a `MODULE` library permits this.

Then generate and install it:

```
include(CMakePackageConfigHelpers)

# MARIADB_PLUGIN_SERVER_DBUG_OFF must already have been recorded by the same
# configuration-specific logic that controls -DDBUG_OFF. Set the NDEBUG
# workaround according to whether the affected Json_writer header is shipped.
if(NOT DEFINED MARIADB_PLUGIN_SERVER_DBUG_OFF OR
   NOT DEFINED MARIADB_PLUGIN_SERVER_NDEBUG_WORKAROUND)
  message(FATAL_ERROR "MariaDB server ABI profile was not initialized")
endif()
set(MARIADB_PLUGIN_SERVER_WITH_WSREP "${WITH_WSREP}")
set(MARIADB_PLUGIN_SERVER_FILE_OFFSET_BITS "${_FILE_OFFSET_BITS}")
if(MSVC)
  set(MARIADB_PLUGIN_SERVER_MSVC_RUNTIME_LIBRARY
    "${CMAKE_MSVC_RUNTIME_LIBRARY}"
  )
else()
  set(MARIADB_PLUGIN_SERVER_MSVC_RUNTIME_LIBRARY "")
endif()

# This direct configure_file form is suitable for a single-config package.
# For a multi-config generator, generate one profile per configuration as
# explained below and install the selected profile under this filename.
configure_file(
  "${PROJECT_SOURCE_DIR}/cmake/mariadb-plugin-build-profile.cmake.in"
  "${PROJECT_BINARY_DIR}/mariadb-plugin-build-profile.cmake"
  @ONLY
)

configure_package_config_file(
  "${PROJECT_SOURCE_DIR}/cmake/mariadb-plugin-config.cmake.in"
  "${PROJECT_BINARY_DIR}/mariadb-plugin-config.cmake"
  INSTALL_DESTINATION "${MARIADB_PLUGIN_CMAKEDIR}"
  PATH_VARS
    INSTALL_BINDIR
    INSTALL_SCRIPTDIR
    INSTALL_LIBDIR
    INSTALL_PLUGINDIR
    INSTALL_SYSCONF2DIR
    INSTALL_DOCDIR
    INSTALL_MANDIR
    INSTALL_SHAREDIR
)

write_basic_package_version_file(
  "${PROJECT_BINARY_DIR}/mariadb-plugin-config-version.cmake"
  VERSION "${SERVER_VERSION}"
```

```
  COMPATIBILITY SameMinorVersion
)

install(FILES
    "${PROJECT_BINARY_DIR}/mariadb-plugin-config.cmake"
    "${PROJECT_BINARY_DIR}/mariadb-plugin-config-version.cmake"
    "${PROJECT_BINARY_DIR}/mariadb-plugin-build-profile.cmake"
    "${PROJECT_SOURCE_DIR}/cmake/mariadb-plugin-functions.cmake"
  DESTINATION "${MARIADB_PLUGIN_CMAKEDIR}"
  COMPONENT Development
)
```

The config file contains no absolute installation path. The generated targets file computes its prefix from its own location, and `@PACKAGE_INSTALL_LIBDIR@` is generated relative to the same prefix. This is what makes both `mysqlservices` and `server.lib` relocatable.

Use a separate export filename such as `mariadb-plugin-targets.cmake`, rather than naming the export file like the package config. CMake may generate per-configuration companion files and loads them by filename pattern.

The build-profile file must describe the configuration that is actually installed. For a normal MariaDB build the `DBUG_OFF` values are:

Installed server configuration	[illegible]
`Debug`	`FALSE`
`Release`	`TRUE`
`RelWithDebInfo`	`TRUE`
`MinSizeRel`	`TRUE`

Record this value in the same MariaDB CMake logic that adds or omits `-DDBUG_OFF`; do not rediscover it later from the consumer's build type. If a custom builder forces `DBUG_ON`, the recorded value must be `FALSE` even when the configuration is named `Release`.

With a single-config generator, configure the profile from `CMAKE_BUILD_TYPE`. With Visual Studio or another multi-config generator, generate one profile for each configuration and install the one selected by `cmake --install --config <configuration>`. A package containing several server configurations needs corresponding imported configurations and ABI profiles; otherwise publish separate Development packages for Debug and Release servers.

7. Preserve the server ABI for dirty plugins

A pure plugin sees only the plugin and service API. A dirty plugin includes internal server headers and may exchange internal objects with `mariadbd`. Consequently, a dirty plugin must be compiled for the ABI of the installed server, not merely for the platform and MariaDB version.

The generated build profile must capture at least these settings:

- `DBUG_OFF`. Many internal declarations and inline implementations depend on it. MariaDB normally defines it for `Release`, `RelWithDebInfo`, and `MinSizeRel`, but not for `Debug`. A Debug builder must therefore export `MARIADB_PLUGIN_SERVER_DBUG_OFF=FALSE`.
- `WITH_WSREP`, when the installed server was built with WSREP. It adds members to internal structures including `system_variables` and `THD`; a mismatch changes member offsets. Do not define it as zero because the headers use `#ifdef WITH_WSREP`.

- `_FILE_OFFSET_BITS`, when MariaDB defines it. A mismatch can change `off_t` and any structure containing it.
- On Windows, the MSVC runtime and STL ABI. MariaDB currently defaults `CMAKE_MSVC_RUNTIME_LIBRARY` to `MultiThreadedDLL`, which means `/MD` even for a Debug server configuration unless the builder overrides it.

`NDEBUG` should normally remain the third-party project's choice. It controls the standard C and C++ `assert()` facility; it should not be part of MariaDB's public compile profile.

There is a likely bug in the current `sql/my_json_writer.h` and `sql/my_json_writer.cc`: several conditionals use `NDEBUG` to add or remove debug-only state, including STL members of `Json_writer`. MariaDB's own switch for this purpose is `DBUG_OFF`, as the same header already demonstrates in `Json_writer_nesting_guard`. These conditionals should use `DBUG_OFF` instead, in both the header and implementation. Otherwise a release server and a dirty plugin built without `NDEBUG` disagree about the layout of `Json_writer`.

Fixing the header is preferable to exporting `NDEBUG`. If a package is made for an existing server version in which the header has not been fixed, adding `NDEBUG` to that version's `MariaDB::server_headers` is a compatibility workaround required to reproduce the release server's existing ABI. It should be clearly marked as such, rather than presented as a general plugin API requirement.

The settings belong to `MariaDB::server` through `MariaDB::server_headers`. They must not be attached to `MariaDB::mysqlservices`: a pure plugin does not use internal server classes, and its private implementation need not inherit the server's CRT choice.

Do not derive these settings from the third-party project's configuration. In particular, this is wrong for a release MariaDB installation:

```
# Wrong: this describes the consumer configuration, not the installed server.
$<$<NOT:$<CONFIG:Debug>>:DBUG_OFF>
```

A developer may compile a dirty plugin with `/Od` and `/Zi`, but it must still use the installed server's `DBUG_OFF`, CRT, and STL ABI. Conversely, a plugin for a Debug server must not acquire `DBUG_OFF` merely because the third-party project selected a non-Debug configuration. `NDEBUG` is needed only as the compatibility workaround described above when using an affected installed header.

Compile definitions are transitive CMake usage requirements, so the package config can attach them to `MariaDB::server_headers`. The `MSVC_RUNTIME_LIBRARY` target property is not transitive. The package therefore exports its value as `MARIADB_PLUGIN_SERVER_MSVC_RUNTIME_LIBRARY`, and `mariadb_add_plugin()` applies it directly to each dirty plugin target. A developer who creates a dirty target manually must set the same property.

Existing MariaDB 11.8 builds record `WITH_WSREP` in generated `my_config.h`. However, the current MDEV-40608 pull request removes it from `my_config.h` and instead adds `-DWITH_WSREP` only while the server source tree is built. Its installed plugin config does not currently reproduce that definition. That is safe for pure plugins, but not for dirty plugins built against a WSREP-enabled server. The exported `MariaDB::server_headers` target must therefore add `WITH_WSREP` when the installed server used it, as shown earlier. It must not add the definition to `MariaDB::mysqlservices`.

Most other feature and platform definitions do not need to be copied into the CMake package individually. MariaDB's installed generated `my_config.h` records storage-engine feature macros, `HAVE_*`, and type sizes, and `my_global.h` includes it. The exported target must ensure that the matching installed `my_config.h` is found.

Debug-server development packages are a separate problem. MariaDB Debug builds can additionally use definitions such as `ENABLED_DEBUG_SYNC`, `PROTECT_STATEMENT_MEMROOT`, `SAFE_MUTEX`, `_GLIBCXX_DEBUG`, and `_GLIBCXX_ASSERTIONS`. Several of these change `THD`, lock structures, internal classes, or the libstdc++ ABI. If MariaDB chooses to distribute development files for a Debug or sanitizer build, generate a separate package containing that build's exact compile profile. Do not represent it by merely removing `DBUG_OFF` from the ordinary release package.

`MYSQL_SERVER` should not be added automatically. It exposes large additional parts of internal headers, and in-tree storage engines do not uniformly define it. A specialized plugin may request it explicitly, but `DIRTY` alone should not imply it.

8. Third-party plugin example

A third-party project can build all three plugin kinds with one small `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.15)
project(my_mariadb_plugins LANGUAGES C CXX)

find_package(mariadb-plugin CONFIG REQUIRED)

# A. Pure server plugin
# Uses only the public plugin API and mysqlservices.
# On Linux, --no-undefined is inherited from MariaDB::mysqlservices.
add_library(my_pure_plugin MODULE pure_plugin.c)
set_target_properties(my_pure_plugin PROPERTIES PREFIX "")
target_link_libraries(my_pure_plugin PRIVATE
  MariaDB::mysqlservices
)

# B. Dirty server plugin or storage engine
# May include internal headers and refer directly to server symbols.
# Windows links server.lib. Linux adds no library and permits unresolved
# server symbols.
add_library(my_storage_engine MODULE storage_engine.cc)
set_target_properties(my_storage_engine PROPERTIES PREFIX "")
target_link_libraries(my_storage_engine PRIVATE
  MariaDB::server
)
if(MSVC)
  # MSVC_RUNTIME_LIBRARY is not inherited through target_link_libraries().
  set_property(TARGET my_storage_engine PROPERTY
    MSVC_RUNTIME_LIBRARY
      "${MARIADB_PLUGIN_SERVER_MSVC_RUNTIME_LIBRARY}"
  )
endif()

# C. Client authentication plugin
# MariaDB::client_plugin is an INTERFACE target. It supplies Connector/C
# headers but adds no binary library to the linker command.
add_library(my_client_auth MODULE client_auth.c)
set_target_properties(my_client_auth PROPERTIES PREFIX "")
target_link_libraries(my_client_auth PRIVATE
  MariaDB::client_plugin
)
```

Although `target_link_libraries()` is used for the client plugin, `MariaDB::client_plugin` is header-only. No MariaDB library appears on the actual linker command line. In modern CMake, `target_link_libraries()` is also the normal way to inherit usage requirements from an interface target.

Typical source-file includes are:

```
/* pure_plugin.c: public server plugin API */
#include <mysql/plugin.h>
#include <mysql/service_my_snprintf.h>
```

```
/* storage_engine.cc: internal server API */
#include <mysql/plugin.h>
#include <handler.h>
#include <sql_class.h>
```

```
/* client_auth.c: Connector/C client plugin API */
#include <mysql.h>
#include <mysql/client_plugin.h>
```

The plugin source still needs the appropriate `maria_declare_plugin(...)` or `mysql_declare_client_plugin(...)` declaration. The package changes only how it is compiled and linked; it does not change the plugin ABI.

Build the third-party project by pointing CMake at the MariaDB installation:

```
cmake -S . -B build -DCMAKE_PREFIX_PATH=/path/to/mariadb
cmake --build build --config Release
```

Alternatively, point directly at the config directory:

```
cmake -S . -B build \
  -Dmariadb-plugin_DIR=/path/to/mariadb/lib/cmake/mariadb-plugin
```

`CMAKE_PREFIX_PATH` is usually preferable because it names the installation as a whole and does not depend on whether a platform uses `lib` or `lib64`.

The example uses CMake 3.15 because it builds a dirty plugin with MSVC. A project containing only pure or client plugins may retain `cmake_minimum_required(VERSION 3.12)`.

9. Relocatability test

Test the installed package, not only the MariaDB build tree:

1. Install only the `Development` component into a temporary prefix.
2. Move or copy that complete prefix to a different absolute directory.
3. Configure the third-party example with only the new directory in `CMAKE_PREFIX_PATH`.
4. Build all three modules.
5. Inspect the link commands: • the pure plugin contains `mysqlservices` and rejects unresolved symbols; • the dirty plugin contains `server.lib` on Windows and no server library on Linux; • the client plugin contains no MariaDB library.
6. Search the installed `mariadb-plugin-*.cmake` files for the original source, build, and installation paths. None should occur.
7. Call `mariadb_install_plugin()` from the example and stage the result. Verify that the module goes to the plugin directory exported by the moved package, not to the original prefix.
8. On Windows, repeat with `RUNTIME_DEPENDENCIES` and only the Development component present. The scan must not require or copy `server.dll`.

This catches the common mistake of producing a config file that works on the build machine only.

10. Add the `mariadb_add_plugin()` convenience function

Put this function in `cmake/mariadb-plugin-functions.cmake`. It intentionally resembles the existing in-tree `MYSQL_ADD_PLUGIN` call style: the plugin name and sources are positional, while special behaviour is selected by keywords. It only creates and configures the module target; installation and packaging are deliberately outside its scope.

```
function(mariadb_add_plugin plugin_name)
  cmake_parse_arguments(ARG
    "CLIENT;DIRTY;STORAGE_ENGINE;MODULE_ONLY;RECOMPILE_FOR_EMBEDDED"
    "MODULE_OUTPUT_NAME;COMPONENT"
    "LINK_LIBRARIES;DEPENDS"
    ${ARGN}
  )

  # RECOMPILE_FOR_EMBEDDED is accepted only as a compatibility spelling for
  # existing in-tree declarations. Externally it means DIRTY; no embedded
  # server is built.
  if(ARG_RECOMPILE_FOR_EMBEDDED)
    set(ARG_DIRTY TRUE)
  endif()

  if(ARG_CLIENT AND (ARG_DIRTY OR ARG_STORAGE_ENGINE))
    message(FATAL_ERROR
      "mariadb_add_plugin(${plugin_name}): CLIENT cannot be DIRTY or a STORAGE_ENGINE"
    )
  endif()

  if(NOT ARG_UNPARSED_ARGUMENTS)
    message(FATAL_ERROR
      "mariadb_add_plugin(${plugin_name}): no source files were specified"
    )
  endif()

  string(TOLOWER "${plugin_name}" target)
  if(TARGET "${target}")
    message(FATAL_ERROR
      "mariadb_add_plugin(${plugin_name}): target ${target} already exists"
    )
  endif()

  add_library("${target}" MODULE ${ARG_UNPARSED_ARGUMENTS})

  if(ARG_MODULE_OUTPUT_NAME)
    set(output_name "${ARG_MODULE_OUTPUT_NAME}")
  elseif(ARG_STORAGE_ENGINE)
    set(output_name "ha_${target}")
  else()
    set(output_name "${target}")
  endif()

  set_target_properties("${target}" PROPERTIES
    PREFIX ""
    OUTPUT_NAME "${output_name}"
    MARIADB_PLUGIN_MODULE_NAME "${output_name}"
  )

  if(ARG_CLIENT)
    set(plugin_type CLIENT)
  elseif(ARG_STORAGE_ENGINE)
    set(plugin_type STORAGE_ENGINE)
  elseif(ARG_DIRTY)
    set(plugin_type DIRTY_SERVER)
  else()
    set(plugin_type PURE_SERVER)
  endif()
  set_property(TARGET "${target}" PROPERTY
    MARIADB_PLUGIN_TYPE "${plugin_type}"
  )
  if(ARG_COMPONENT)
    set_property(TARGET "${target}" PROPERTY
      MARIADB_PLUGIN_COMPONENT "${ARG_COMPONENT}"
    )
  endif()

  if(MSVC AND (ARG_DIRTY OR ARG_STORAGE_ENGINE))
    if(CMAKE_VERSION VERSION_LESS 3.15)
      message(FATAL_ERROR
```

```
        "Dirty MariaDB plugins built with MSVC require CMake 3.15 or newer"
      )
    endif()
    cmake_policy(GET CMP0091 cmp0091_state)
    if(NOT cmp0091_state STREQUAL "NEW")
      message(FATAL_ERROR
        "Set cmake_minimum_required(VERSION 3.15) before project() so CMP0091 is NEW"
      )
    endif()

    if("${MARIADB_PLUGIN_SERVER_MSVC_RUNTIME_LIBRARY}" STREQUAL "")
      message(FATAL_ERROR
        "The mariadb-plugin package does not describe the server's MSVC runtime"
      )
    endif()

    # MSVC_RUNTIME_LIBRARY is not a transitive usage requirement. Apply the
    # exact value captured from the packaged server build.
    set_property(TARGET "${target}" PROPERTY
      MSVC_RUNTIME_LIBRARY
        "${MARIADB_PLUGIN_SERVER_MSVC_RUNTIME_LIBRARY}"
    )
  endif()

  if(ARG_CLIENT)
    # Header-only target: no MariaDB binary library is linked.
    target_link_libraries("${target}" PRIVATE MariaDB::client_plugin)
  elseif(ARG_DIRTY OR ARG_STORAGE_ENGINE)
    # server.lib on Windows; headers only on Linux.
    target_link_libraries("${target}" PRIVATE MariaDB::server)
  else()
    # Pure server plugin; also enforces --no-undefined on Linux.
    target_link_libraries("${target}" PRIVATE MariaDB::mysqlservices)
  endif()

  if(ARG_LINK_LIBRARIES)
    target_link_libraries("${target}" PRIVATE ${ARG_LINK_LIBRARIES})
  endif()

  if(ARG_DEPENDS)
    add_dependencies("${target}" ${ARG_DEPENDS})
  endif()
endfunction()
```

`MODULE_ONLY` is accepted as a no-op because an external plugin is always a loadable module. `RECOMPILE_FOR_EMBEDDED` is accepted so that a reasonable subset of existing plugin declarations can be reused outside the server tree. New external projects should use `DIRTY`, which describes the real dependency.

The compatibility is deliberately limited. In-tree options such as `MANDATORY`, `DEFAULT`, `DISABLED`, `STATIC_ONLY`, and `NOT_EMBEDDED` control how MariaDB itself is assembled. They have no useful meaning in an external module project. The call style can be shared without pretending that the two build environments have identical policy or side effects.

Typical third-party calls are:

```
find_package(mariadb-plugin CONFIG REQUIRED)

# Pure server plugin: links mysqlservices and forbids unresolved symbols.
mariadb_add_plugin(example example.c)

# Dirty server plugin: server.lib on Windows, unresolved server symbols on Linux.
mariadb_add_plugin(type_example type_example.cc DIRTY)

# Storage engines are dirty and use the conventional ha_ filename by default.
mariadb_add_plugin(example_engine ha_example.cc STORAGE_ENGINE)

# Client authentication plugin: Connector/C headers, no MariaDB library.
mariadb_add_plugin(example_auth example_auth.c CLIENT)

# Optional output-name override and non-MariaDB dependencies are also supported.
mariadb_add_plugin(foo foo.cc DIRTY
```

```
  MODULE_OUTPUT_NAME mariadb_foo
  LINK_LIBRARIES third_party_library
)
```

11. Installing and packaging

Building a plugin and installing it are separate decisions.

The existing in-tree `MYSQL_ADD_PLUGIN()` macro creates a target and, for a dynamic plugin, installs it into MariaDB's plugin directory. It also contains server-build policy for static plugins, embedded builds, test suites, CPack components, RPM metadata, configuration files, debug symbols, signing, and Windows runtime dependencies.

That behaviour is appropriate in the MariaDB source tree, which owns the complete product layout. It is not an appropriate default for a third-party project. A developer may want to run from the build directory, copy the module to a test server, use a private prefix, write a native RPM spec or Debian package, or embed the result in another product.

Therefore `mariadb_add_plugin()` never creates an install rule. A developer who wants MariaDB's normal layout opts in by calling:

```
mariadb_install_plugin(my_plugin)
```

This is the only purpose of the convenience installer: install into the MariaDB installation selected by `find_package(mariadb-plugin)`. Projects that want a different destination use CMake's ordinary `install()` command.

Export the selected MariaDB layout

The package config exports these resolved paths:

Variable	Intended contents
`MARIADB_SERVER_VERSION`	Version of the MariaDB server that supplied the package
`MARIADB_INSTALL_PREFIX`	Root of the selected MariaDB installation
`MARIADB_INSTALL_PLUGINDIR`	Server and client plugin modules
`MARIADB_INSTALL_BINDIR`	Executables and Windows runtime DLLs
`MARIADB_INSTALL_SCRIPTDIR`	MariaDB-related scripts
`MARIADB_INSTALL_LIBDIR`	Additional libraries
`MARIADB_INSTALL_SYSCONF2DIR`	Additional server configuration files, or empty if the layout has none
`MARIADB_INSTALL_DOCDIR`	Documentation
`MARIADB_INSTALL_MANDIR`	Manual pages
`MARIADB_INSTALL_SHAREDIR`	Architecture-independent shared files

The paths belong to the package that was found, not to the third-party project's `CMAKE_INSTALL_PREFIX`. For a relocatable ZIP installation they are computed relative to the package config's location. System configuration directories such as `/etc/my.cnf.d` remain absolute because they are outside the relocatable installation prefix.

`mariadb_install_plugin()` uses `MARIADB_INSTALL_PLUGINDIR`, `MARIADB_INSTALL_BINDIR`, and, when requested, `MARIADB_INSTALL_SYSCONF2DIR`. There are no special convenience functions for scripts, documentation, manual pages, helper libraries, or other uncommon files. A project can use the exported variables directly:

```
install(PROGRAMS inspect_my_plugin.py
  DESTINATION "${MARIADB_INSTALL_SCRIPTDIR}"
)

install(FILES README.md
  DESTINATION "${MARIADB_INSTALL_DOCDIR}/my_plugin"
)
```

Using these variables is optional. Prefix-relative destinations remain under the third-party project's control:

```
install(PROGRAMS inspect_my_plugin.py DESTINATION bin)
```

The `mariadb_install_plugin()` function

Add the following function to `mariadb-plugin-functions.cmake`, after `mariadb_add_plugin()`:

```
function(mariadb_install_plugin target)
  cmake_parse_arguments(ARG
    "WITH_PLUGIN_CONFIG;RUNTIME_DEPENDENCIES"
    "COMPONENT"
    ""
    ${ARGN}
  )

  if(NOT TARGET "${target}")
    message(FATAL_ERROR
      "mariadb_install_plugin(${target}): target does not exist"
    )
  endif()

  get_target_property(plugin_type "${target}" MARIADB_PLUGIN_TYPE)
  get_target_property(module_name "${target}" MARIADB_PLUGIN_MODULE_NAME)
  if(NOT plugin_type OR NOT module_name)
    message(FATAL_ERROR
      "mariadb_install_plugin(${target}): target was not created by mariadb_add_plugin()"
    )
  endif()

  if(NOT ARG_COMPONENT)
    get_target_property(ARG_COMPONENT "${target}" MARIADB_PLUGIN_COMPONENT)
    if(NOT ARG_COMPONENT)
      set(ARG_COMPONENT "")
    endif()
  endif()
  if(ARG_COMPONENT)
    set(component_args COMPONENT "${ARG_COMPONENT}")
  else()
    set(component_args)
  endif()

  set(runtime_set_args)
  if(ARG_RUNTIME_DEPENDENCIES)
    if(NOT WIN32)
      message(FATAL_ERROR
        "RUNTIME_DEPENDENCIES is supported only on Windows"
      )
    endif()
    if(CMAKE_VERSION VERSION_LESS 3.21)
      message(FATAL_ERROR
        "Windows runtime-dependency installation requires CMake 3.21 or newer"
      )
    endif()
    set(runtime_set "mariadb_plugin_${target}_runtime_dependencies")
    set(runtime_set_args RUNTIME_DEPENDENCY_SET "${runtime_set}")
```

```
  endif()

  install(TARGETS "${target}"
    ${runtime_set_args}
    RUNTIME DESTINATION "${MARIADB_INSTALL_PLUGINDIR}" ${component_args}
    LIBRARY DESTINATION "${MARIADB_INSTALL_PLUGINDIR}" ${component_args}
  )

  if(ARG_WITH_PLUGIN_CONFIG)
    if(WIN32)
      message(FATAL_ERROR
        "WITH_PLUGIN_CONFIG is not used on Windows"
      )
    endif()
    if(plugin_type STREQUAL "CLIENT")
      message(FATAL_ERROR
        "WITH_PLUGIN_CONFIG is valid only for server plugins"
      )
    endif()
    if("${MARIADB_INSTALL_SYSCONF2DIR}" STREQUAL "")
      message(FATAL_ERROR
        "The selected MariaDB layout has no configuration include directory"
      )
    endif()

    file(MAKE_DIRECTORY "${CMAKE_CURRENT_BINARY_DIR}/CMakeFiles")
    set(config_file
      "${CMAKE_CURRENT_BINARY_DIR}/CMakeFiles/${target}-plugin.cnf"
    )
    file(WRITE "${config_file}"
      "[mariadb]\nplugin-load-add=${module_name}\n"
    )
    install(FILES "${config_file}"
      DESTINATION "${MARIADB_INSTALL_SYSCONF2DIR}"
      RENAME "${module_name}.cnf"
      ${component_args}
    )
  endif()

  if(ARG_RUNTIME_DEPENDENCIES)
    file(TO_CMAKE_PATH "$ENV{PATH}" runtime_search_dirs)
    list(PREPEND runtime_search_dirs "${MARIADB_INSTALL_BINDIR}")
    if(VCPKG_INSTALLED_DIR AND VCPKG_TARGET_TRIPLET)
      list(APPEND runtime_search_dirs
        "${VCPKG_INSTALLED_DIR}/${VCPKG_TARGET_TRIPLET}/bin"
        "$<$<CONFIG:Debug>:${VCPKG_INSTALLED_DIR}/${VCPKG_TARGET_TRIPLET}/debug/bin>"
      )
    endif()

    install(RUNTIME_DEPENDENCY_SET "${runtime_set}"
      DESTINATION "${MARIADB_INSTALL_BINDIR}"
      ${component_args}
      PRE_EXCLUDE_REGEXES
        "^api-ms-"
        "^ext-ms-"
        "^server\\.dll$"
        "^vcruntime"
        "^clang_rt"
      POST_EXCLUDE_REGEXES
        ".*[Ss]ystem32/.*\\.dll"
      POST_INCLUDE_REGEXES
        "libssl"
        "libcrypto"
      DIRECTORIES ${runtime_search_dirs}
    )
  endif()
endfunction()
```

The function installs both Unix `MODULE` libraries and Windows plugin DLLs. The optional arguments are independent:

```
mariadb_install_plugin(my_plugin)

mariadb_install_plugin(my_plugin
  WITH_PLUGIN_CONFIG
)
```

```
mariadb_install_plugin(my_plugin
  RUNTIME_DEPENDENCIES
)
```

`COMPONENT` is also optional:

```
mariadb_install_plugin(my_plugin COMPONENT MyPlugin)
```

It merely labels the generated CMake install rules. It permits commands such as `cmake --install build --component MyPlugin`; it does not enable component packaging or create a separate RPM or DEB. Most external projects containing one plugin need no component at all.

If a component is specified, the module, generated activation file, and collected Windows DLLs use the same component. Without one, all use the project's normal default installation.

Optional server activation file

On Unix, a server plugin can request a small activation file:

```
mariadb_install_plugin(my_plugin WITH_PLUGIN_CONFIG)
```

For a module named `my_plugin`, the generated file contains:

```
[mariadb]
plugin-load-add=my_plugin
```

MariaDB adds the platform module suffix if it is omitted. A storage engine target named `example` therefore produces:

```
[mariadb]
plugin-load-add=ha_example
```

`WITH_PLUGIN_CONFIG` is valid for pure server plugins, dirty server plugins, and storage engines. It is invalid for client plugins and is not used on Windows. If the selected layout has no additional configuration directory, the function reports an error rather than guessing one.

This generated file does nothing except load the module. A plugin that needs real configuration should supply its own file and install it with `install(FILES ...)`. RPM `%config(noreplace)` and Debian conffile policy also belong to the third-party package, not to this convenience function.

Windows runtime dependencies

`RUNTIME_DEPENDENCIES` asks CMake to inspect the plugin DLL recursively and install unusual non-system dependencies, such as a private OpenSSL build, into the selected MariaDB `bin` directory:

```
mariadb_install_plugin(my_plugin RUNTIME_DEPENDENCIES)
```

The design follows MariaDB's existing Windows installer logic. Windows API-set DLLs, System32 DLLs, the MSVC runtime, sanitizer runtimes, and MariaDB's own `server.dll` are excluded. `libssl` and `libcrypto` retain the existing special handling.

Excluding `server.dll` before dependency resolution is essential. A Development-only installation contains `server.lib` but is not required to contain `server.dll`; the DLL belongs exclusively to the Server component. Runtime scanning must therefore neither copy it nor fail when it is absent.

CMake 3.21 is required only when this Windows option is used. Windows builders can use a current CMake without increasing the minimum required by older Linux distributions.

Custom destinations and package staging

A project that does not want the selected MariaDB layout should not call `mariadb_install_plugin()`. It can write ordinary prefix-relative rules:

```
install(TARGETS my_plugin
  LIBRARY DESTINATION lib/my-product/plugins
  RUNTIME DESTINATION lib/my-product/plugins
)
```

On Unix, native RPM and Debian builds can stage MariaDB-aware absolute destinations with `DESTDIR`:

```
DESTDIR=/tmp/my-plugin-package cmake --install build
```

This produces the final MariaDB directory structure below the staging root. CPack's RPM and DEB generators normally handle this staging themselves. `CPACK_SET_DESTDIR` is available when a particular CPack setup needs explicit `DESTDIR` handling for absolute destinations.

RPM and DEB dependencies

RPM and Debian packaging tools discover ordinary shared-library dependencies from the finished plugin. For example, they can discover dependencies on `libcurl`, `libssl`, `libstdc++`, and the C runtime. CPack RPM enables its automatic shared-library dependency detection by default. CPack DEB projects can request `dpkg-shlibdeps` with:

```
set(CPACK_DEBIAN_PACKAGE_SHLIBDEPS ON)
```

This scanning cannot discover the logical dependency on MariaDB Server. `mysqlservices` is static, and a dirty Linux plugin intentionally leaves its server references unresolved instead of recording a dynamic dependency on `mariadbd`. Installing a file into a MariaDB plugin directory also creates no package dependency.

The third-party package must therefore declare its server dependency itself. For a normal, non-component CPack package the relevant variables are:

```
set(CPACK_RPM_PACKAGE_REQUIRES
  "the appropriate MariaDB server RPM requirement"
)

set(CPACK_DEBIAN_PACKAGE_DEPENDS
  "the appropriate MariaDB server DEB requirement"
)
```

The package names differ between official MariaDB packages and distribution packages, so `mariadb_install_plugin()` must not guess them. A pure plugin can normally require a compatible server version. A dirty plugin or storage engine uses MariaDB's private ABI and should require the exact server build, or a deliberately restricted set of compatible builds.

Third-party projects are not required to use CPack component packaging. If a project deliberately enables `CPACK_RPM_COMPONENT_INSTALL` or `CPACK_DEB_COMPONENT_INSTALL`, it uses the corresponding component-specific dependency variables. The existing in-tree macro uses these variables because one MariaDB build produces many related packages. That is not the normal external-plugin case.

The in-tree RPM expression involving `%{release}` must not be copied blindly. Inside the server build it denotes the common MariaDB package release. In an external RPM spec it would normally denote the third-party plugin package's release, which need not match the installed server package.

Difference from the in-tree macro

Operation	In-tree	Public package functions
Create a plugin target	Yes	`mariadb_add_plugin()`
Select static, embedded, mandatory, or default plugins	Yes	Not applicable externally
Install a dynamic plugin	Automatically	Only after `mariadb_install_plugin()`
Choose a custom destination	Server layout controls it	Third party may use ordinary `install()`
Generate an activation file	Sometimes, through packaging logic	Explicit `WITH_PLUGIN_CONFIG`
Collect Windows DLL dependencies	Global server install logic	Explicit `RUNTIME_DEPENDENCIES`
Install PDBs or sign binaries	MariaDB's global install policy	Third party decides
Install `mysql-test` suites	Detected automatically in the source tree	Third party decides
Assign an install component	Common	Optional
Enable component packaging	MariaDB packaging decides	Never enabled by the functions
Generate RPM/DEB packages	MariaDB packaging decides	Third party decides
Install scripts or documentation	MariaDB-specific helpers	Ordinary `install()` plus exported layout variables

The positional syntax of `mariadb_add_plugin()` can remain compatible with the useful subset of an in-tree declaration. That does not require the external function to reproduce all the in-tree macro's policy and installation side effects.

Short checklist

- Present libraries to consumers through CMake targets, not path variables.
- Put header include directories and compile definitions on those targets.
- Use `BUILD_INTERFACE` for source/build-tree paths.
- Use relative `INSTALL_INTERFACE` paths for installed headers.
- Export every interface target referenced by an exported library.
- Install `server.lib` and `mysqlservices` in `Development`.
- Keep `server.dll` exclusively in `Server`; `Development` must not depend on it.
- Represent `server.lib` with a relocatable imported target, without exporting the real server DLL target.
- Make `MariaDB::server` interface-only on Linux.
- Keep `mysqlservices.lib` CRT-neutral on MSVC with `/Zl`.

- Generate an ABI profile for the server configuration that is actually installed; a Debug builder must record that `DBUG_OFF` is absent.
- Put server ABI definitions such as `DBUG_OFF`, `WITH_WSREP`, and `_FILE_OFFSET_BITS` on `MariaDB::server`, not on `MariaDB::mysqlservices`.
- Export `NDEBUG` only as a compatibility workaround for affected headers.
- Make dirty MSVC plugins use the exact exported `MSVC_RUNTIME_LIBRARY` value; do not assume it from the plugin's build type.
- Treat Debug and sanitizer server development packages as distinct ABI profiles.
- Do not apply `--no-undefined` to dirty server plugins.
- Keep `mariadb_add_plugin()` free of installation side effects.
- Make `mariadb_install_plugin()` optional; never force MariaDB's layout on a third-party project.
- Do not enable CPack component packaging on behalf of a third party.
- Let RPM and Debian tools discover ordinary binary dependencies, but document the explicit MariaDB Server package dependency.
- Exclude `server.dll` from Windows runtime-dependency collection.
- Test after moving the complete installation prefix.