

Qt 6.4 > Build with CMake > [Building a reusable QML module](#)

## Building a reusable QML module

The example below demonstrates how to create a library which exposes C++ to QML. The directory structure for the example looks like this:

```
├── CMakeLists.txt
├── example
│   └── mylib
│       ├── CMakeLists.txt
│       ├── mytype.cpp
│       └── mytype.h
```

The toplevel `CMakeLists.txt` file does some basic setup, and then uses `add_subdirectory` to include the one in `mylib`. The subdirectory structure corresponds to the QML module's URI, but with the dots replaced by slashes. That's the same logic the engine uses when it searches for a module in the [import paths](#). `mytype.h` declares a class and uses the declarative registration macros to expose it to the engine.

In the subdirectory's `CMakeLists.txt` we again call `qt6_add_qml_module`. However, the invocation is slightly different:

```
qt6_add_qml_module(mylib
    URI example.mylib
    VERSION 1.0
    SOURCES
        mytype.h mytype.cpp
)
```

To add C++ types, the `SOURCES` parameter needs to be specified. The target for `mylib` is not created. Therefore, if the target passed to `qt6_add_qml_module` does not exist, a library target is automatically created, which is needed in this case.

When the project is built, in addition to the library, a QML plugin is also built. The plugin's auto-generated class extends from `QQmlEngineExtensionPlugin`. The `mylib` library itself already contains the code to register the types with the engine. However, that is only useful in cases where we can link against the library. To make the module usable in a QML file loaded by qml, the [QML Runtime Tool](#), a plugin is needed that can be loaded. The plugin is then responsible for actually linking against the library, and ensuring that the types get registered.

Note that the automatic plugin generation is only possible if the module does not do anything besides registering the types. If it needs to do something more advanced like registering an image provider in `initializeEngine`,

Also, following the directory layout convention helps tooling. That layout is mirrored in the build directory. Which means that you can pass the path to your build directory to the QML tool (via the `-I` flag), and it will find the plugin.

Before concluding add a QML file to the module. In the `lib` subfolder, add a `Mistake.qml` file

```
import example.mylib

MyType{
    answer: 43
}
```

and adjust the `qt6_add_qml_module` call:

```
qt6_add_qml_module(mylib
    URI example.mylib
    VERSION 1.0
    SOURCES
        mytype.h mytype.cpp
    QML_FILES
        Mistake.qml
)
```

As mentioned, we made a mistake because `answer` is actually a read-only property. This illustrates `qmlint` integration: CMake creates a `qmlint` target, and once we run it, `qmlint` warns about the issue:

```
$> cmake --build . --target mylib_qmlint
...
Warning: Mistake.qml:4:13: Cannot assign to read-only property answer
    answer: 43
        ^^
```

[< Building a QML application](#)

[Building projects on the command line >](#)

© 2022 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are [trademarks](#) of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.



Contact Us

Company

- About Us
- Investors
- Newsroom
- Careers
- Office Locations

Support

- Support Services
- Professional Services
- Partners
- Training

Community

- Contribute to Qt
- Forum
- Wiki
- Downloads
- Marketplace

Licensing

- Terms & Conditions
- Open Source
- FAQ

For Customers

- Support Center
- Downloads
- Qt Login
- Contact Us
- Customer Success