创建项目

通过创建项目，您可以：

- 将文件分组在一起
- 添加自定义构建步骤
- 包括表单和资源文件
- 指定正在运行的应用程序的设置

在Qt Creator中设置新项目由向导辅助，该向导将指导您逐步完成项目创建过程。向导会提示您输入该特定类型的项目所需的设置，并为您创建必要的文件。您可以添加自己的自定义向导，以标准化将子项目和类添加到项目中的方式。

大多数 Qt 创建器项目向导都允许您选择要用于构建项目的构建系统：qmake、CMake 或 Qbs。如果未显示任何选项，则项目将设置为使用 qmake。

您还可以使用向导创建纯 C 或C++使用 qmake、Qbs 或 CMake 的项目，但不使用 Qt 库。

此外，还可以将项目作为不使用 qmake、Qbs 或 CMake 的*通用项目*导入。这使您能够使用Qt Creator作为代码编辑器，并完全控制用于构建项目的步骤和命令。

您可以将设备工具作为Qt发行版的一部分进行安装。已安装设备类型的工具包以及构建和运行设置是自动设置的。但是，您可能需要在设备上安装和配置一些其他软件，以便能够从开发PC连接到它们。

选择生成系统

大多数 Qt 创建器项目向导都允许您选择要用于构建项目的构建系统：qmake、CMake 或 Qbs。如果未显示任何选项，则项目将设置为使用 qmake。

qmake是一个用于构建自动化的跨平台系统，有助于简化跨不同平台的开发项目的构建过程。qmake 自动生成构建配置，因此创建每个配置只需要几行信息。qmake 是在安装 Qt 时安装和配置的。要使用其他受支持的生成系统之一，需要对其进行设置。

CMake 是 qmake 的替代方案，用于自动生成构建配置。有关详细信息，请参阅设置 CMake。

介子Meson是一个开源构建系统，意味着既要非常快，更重要的是，要尽可能地用户友好。Meson的主要设计点是，开发人员花在编写或调试构建定义上的每一秒钟都是浪费的。等待构建系统实际开始编译代码所花费的每一秒钟也是如此。有关更多信息，请参阅设置介子。

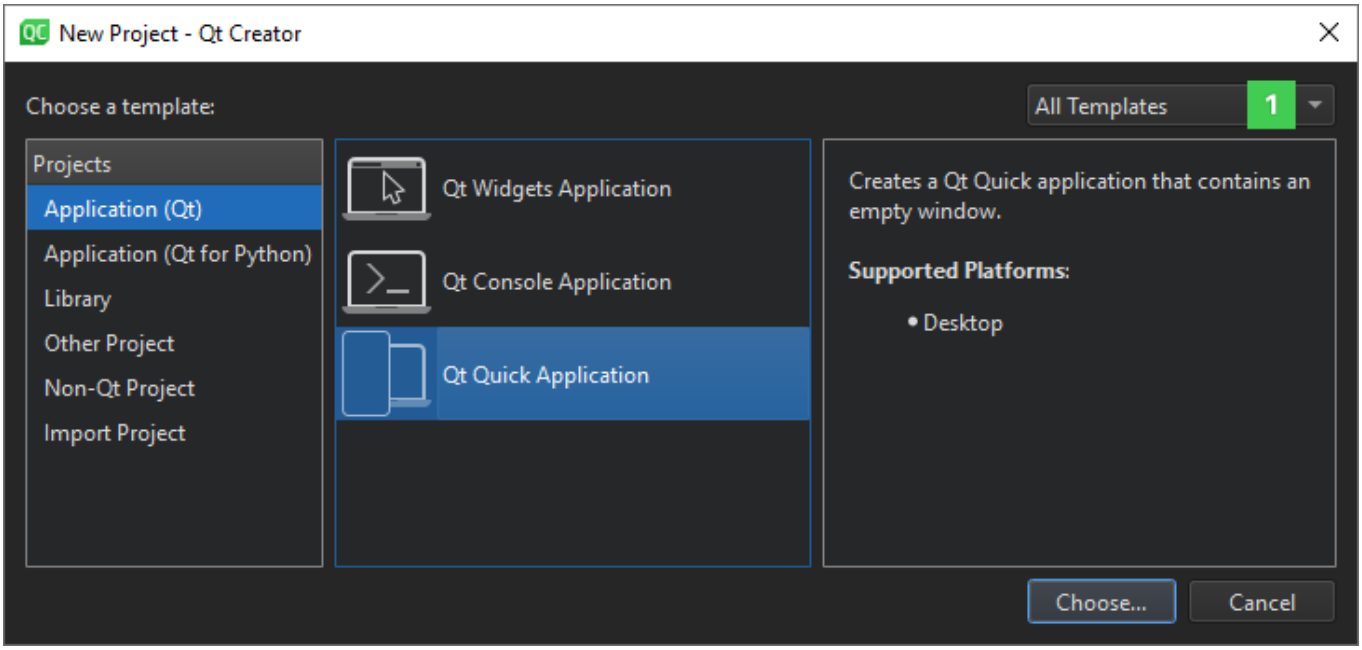
Qbs是一个多合一的构建工具，它从高级项目描述（如qmake或CMake do）生成构建图，并执行低级构建图中的命令（如make do）。有关更多信息，请参阅 设置 Qbs。

若要更改项目目录的位置，并指定用于生成和运行项目的设置，请选择"**编辑**>**首选项**">"**生成**">"**运行**>**常规**"。"**断续器**"选项卡包含"断续器"的其他设置。您可以在 **CMake** >**编辑**>**首选项**>**工具包**中找到 CMake 的更多设

设置。

使用项目向导

在第一步中，为项目选择一个模板。您可以筛选模板 （1） 以仅查看适用于特定目标平台的模板。



接下来，选择项目的位置并为其指定设置。

完成这些步骤后，Qt Creator会自动生成项目，其中包含向导定义的所需标头、源文件、用户界面描述和项目文件。

例如，如果您选择创建Qt快速应用程序，Qt Creator会生成一个QML文件，您可以在**编辑**模式下修改该文件。

选择项目类型

The following table lists the wizard templates for creating projects.

Category	Wizard Template	Purpose
Application (Qt for MCU)	MCU Support Application	Creates an application that uses a subset of Qt QML and Qt Quick Controls types (as supported by Qt for MCUs) that you can deploy, run, and debug on MCU boards. For more information, see Connecting MCUs.
Application (Qt)	Qt Widgets Application	Uses Qt Designer forms to design a Qt widget based user interface for the desktop and C++ to implement the application logic.
	Qt Console Application	Uses a single main.cpp file.
	Qt Quick Application	Creates a Qt Quick 2 application project that can contain both QML and C++ code. You can build the application and deploy it to desktop, embedded, and mobile target platforms.
Application (Qt for	Empty Application	Creates a Qt for Python application that contains only the main code for a QApplication.

Category Wizard Template Purpose

	Window UI	Creates a Qt for Python application that contains an empty window with a widget-based UI. Preferred approach that requires you to generate a Python file from the .ui file, to import it directly into your application.
	Window UI - Dynamic load	Creates a Qt for Python application that contains an empty window with a widget-based UI.
Qt Quick Application - Empty	Creates a Python project that contains an empty Qt Quick Application.	
Library	C++ Library	A shared or static C++ library based on qmake.
	Qt Quick 2 Extension Plugin	Creates a C++ plugin that makes it possible to offer extensions that can be loaded dynamically into Qt Quick 2 applications by using the QQmlEngine class.
	Qt Creator Plugin	Creates a Qt Creator plugin.
Other Project	Qt Custom Designer Widget	Creates a custom Qt Designer widget or widget collection.
	Qt Quick UI Prototype	Creates a Qt Quick UI project with a single QML file that contains the main view. You can preview Qt Quick 2 UI projects in the QML Scene preview tool. You do not need to build them because they do not contain any C++ code. Use this template only if you are prototyping. You cannot create a full application by using this template. Qt Quick UI projects cannot be deployed to embedded or mobile target platforms. For those platforms, create a Qt Quick application instead.
	Auto Test Project	Creates a project with boilerplate code for a Qt or Google test. For more information, see Creating Tests.
	Subdirs Project	Creates a subproject that enables you to structure your qmake projects as a tree hierarchy.
	Empty qmake Project	Creates an empty qmake project that is based on qmake but does not use any default classes.
	Code Snippet	Creates a qmake project from a code snippet. When fixing bug reports that contain a code snippet, you can place the code snippet into a project to compile and check it.
Non-Qt Project	Plain C Application	Creates a plain C application that uses qmake, Qbs, or CMake but does not use the Qt library.
	Plain C++ Application	Creates a plain C++ application that uses qmake, Qbs, or CMake but does not use the Qt library.
	Nim Application (experimental)	Creates a Nim application that uses Nimble, but does not use the Qt library. For more information, see Setting Up Nimble.
	Nimble Application (experimental)	Creates a Nimble application that uses Nimble, but does not use the Qt library. For more information, see Setting Up Nimble.
Category	**Wizard Template**	**Purpose**

		are integrated in Qt Creator, see Using Version Control Systems.
	Import as qmake or CMake Project (Limited Functionality)	Imports an existing project that does not use any of the supported build systems: qmake, Qbs, CMake, or Autotools. The template creates a project file, which enables you to use Qt Creator as a code editor and as a launcher for debugging and analysis tools. However, if you want to build the project, you might need to edit the generated project file.
	Import Existing Project	Imports an existing project that does not use any of the supported build systems: qmake, Qbs, CMake, or Autotools. This enables you to use Qt Creator as a code editor.

To create a new project, select **File** > **New Project** and select the type of your project. The contents of the wizard dialogs depend on the project type and the kits that you select in the **Kit Selection** dialog. Follow the instructions of the wizard.

For examples of creating different types of projects, see Tutorials.

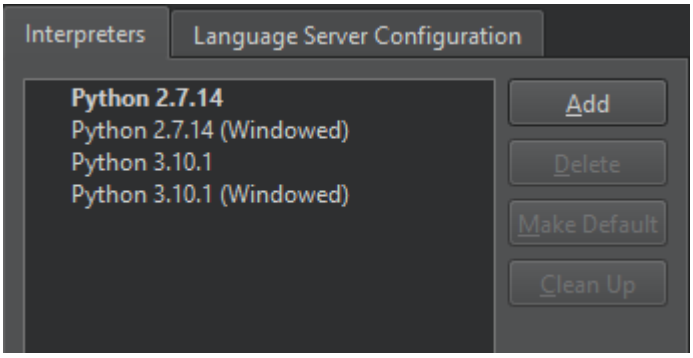
For more information about creating Qt Quick projects, see Creating Qt Quick Projects.

Creating Widget-Based Qt for Python Applications

Qt for Python enables you to use Qt 6 API in Python applications. You can use the PySide6 modules to gain access to individual Qt modules, such as Qt Core, Qt GUI, and Qt Widgets.

If you have not installed PySide6, Qt Creator prompts you to install it after the project is created. Further, it prompts you to install the Python language server that provides services such as code completion and annotations. Select **Install** to install PySide6 and the language server.

To view and manage the available Python interpreters, select **Edit** > **Preferences** > **Python** > **Interpreters**.

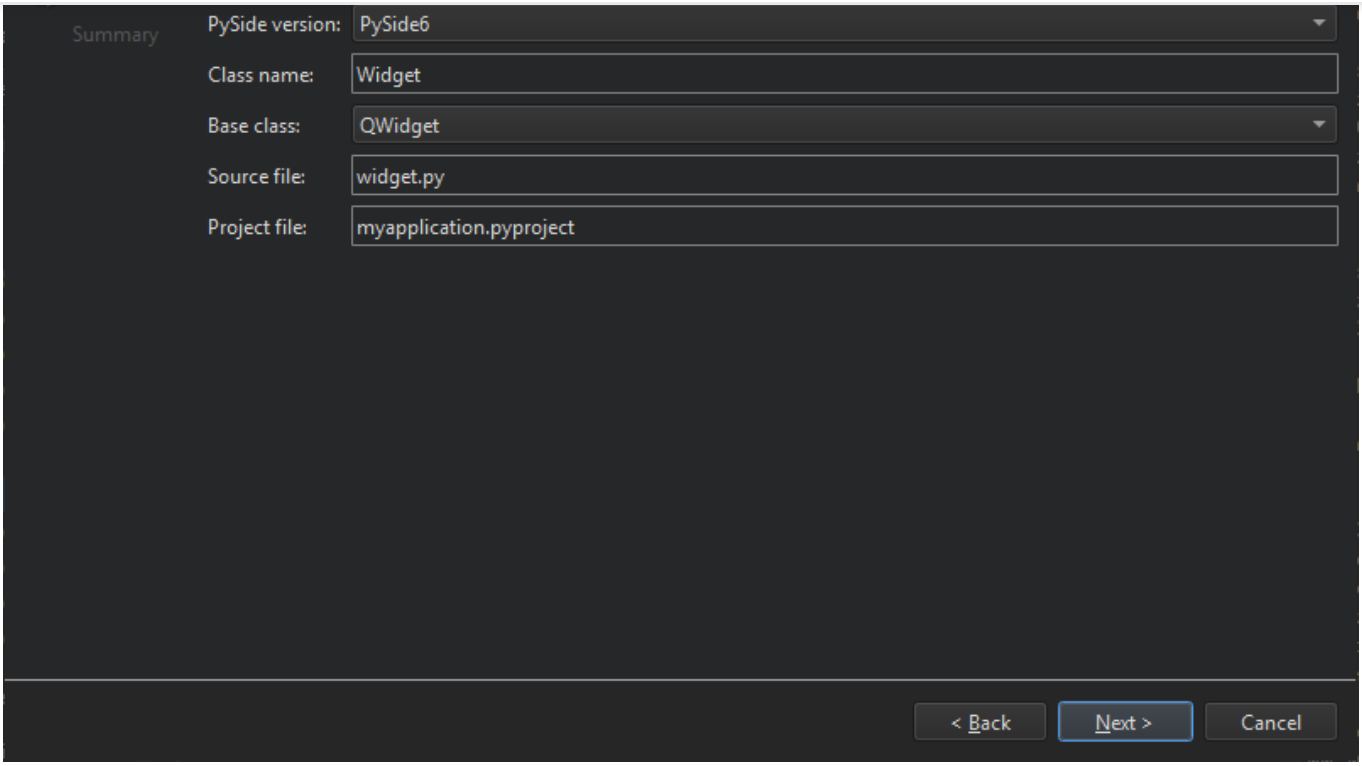


You can add and remove interpreters and clean up references to interpreters that have been uninstalled, but still appear in the list. In addition, you can set the interpreter to use by default.

The Qt for Python Application wizards generate a file that lists the files in the Python project and a file that contains some boilerplate code. In addition, the widget based UI wizard creates a file that contains a Qt Designer form, and the Qt Quick Application wizard creates a file that contains Qt Quick controls. `.pyproject.py.ui.qml`

The files are JSON-based configuration files that replace the previously used configuration files. You can still open and use files, but we recommend that you choose files for new projects. `.pyproject.pyqtc.pyqtc.pyproject`

The **Window UI** wizard enables you to create a Python project that contains the source file for a class. Specify the PySide version, class name, base class, and and source file for the class.



The wizard adds the imports to the source file to provide access to the QApplication, the base class you selected in the Qt Widgets module, and Qt UI tools:

```
import sys

from PySide6.QtWidgets import QApplication, QWidget
```

Note: It is important that you first create the Python code from your UI form. In PySide6, you can do this by executing on a terminal. This enables you to import the class that represents your UI from that Python file.`pyside6-uic form.ui -o ui_form.py`

Once you generate the Python code from the UI file, you can import the class:

```
from ui_form import Ui_Widget
```

The wizard also adds a main class with the specified name that inherits from the specified base class:

```
class Widget(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
```

The following lines in the main class instantiate the generated Python class from your UI file, and set up the interface for the current class.

```
self.ui.setupUi(self)
```

> **Note:** UI elements of the new class can be accessed as member variables. For example, if you have a button called *button1*, you can interact with it using `self.ui.button1`

Next, the wizard adds a main function, where it creates a QApplication instance. As Qt can receive arguments from the command line, you can pass any arguments to the QApplication object. Usually, you do not need to pass any arguments, and you can use the following approach:

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
```

Next, the wizard instantiates the class and shows it:`MainWindow`

```
widget = Widget()
widget.show()
...
```

Finally, the wizard calls the method to enter the Qt main loop and start executing the Qt code:`app.exec()`

```
sys.exit(app.exec())
```

You can now modify the boilerplate code in the Edit mode to develop your Python application. Select **REPL** on the toolbar to start the Python interactive shell. To start the shell and import the current file as a module, select select **REPL Import File**. To also import all functions from the file, select **REPL Import** *.

Always regenerate the Python code after modifying a UI file.

Open the .ui file in the **Design** mode to create a widget-based UI in Qt Designer.

The **Window** wizard adds similar code to the source file, without the UI bits.

The **Empty** wizard adds similar code to the source file, but it does not add any classes, so you need to add and instantiate them yourself.

For more information about the **Qt for Python - Qt Quick Application - Empty** wizard, see Creating Qt Quick Based Python Applications.

For examples of creating Qt for Python applications, see Qt for Python Examples and Tutorials.

Specifying Project Contents

A project can contain files that should be:

- Not installed, but included in a source package created with `make dist`
- Not installed, nor be part of a source package, but still be known to Qt Creator

Qt Creator displays all files that are declared to be part of the project by the project files in the Projects view. The files are sorted into categories by file type (.cpp, .h, .qrc, and so on). To display additional files, edit the project file. Alternatively, you can see all the files in a project directory in the File System view.

Declaring files as a part of the project also makes them visible to the locator and project-wide search.

CMake Projects

When using CMake, you can specify additional files for a project by either adding them as sources or installing them.

In the CMakeLists.txt file, define the files as values of the target_sources command using the property, for example.`PRIVATE`

You can prevent CMake from handling some files, such as a .cpp file that should not be compiled. Use the set_property command and the HEADER_FILE_ONLY property to specify such files. For example:

```
set_property(SOURCE "${files}" PROPERTY HEADER_FILE_ONLY ON)
```

Alternatively, to install the files, use the install command with the or property.`FILESDIRECTORY`

qmake Projects

Use the following variables in the .pro file:

- `SOURCES` and for files to compile`HEADERS`
- `INSTALLS` for files to install
- `DISTFILES` for files to include in a source package
- `OTHER_FILES` for files to manage with Qt Creator without installing them or including them in source packages

For example, the following value includes text files in the source package:

```
DISTFILES += *.txt
```

Adding Subprojects to Projects

In addition to Qt libraries, you can link your application to other libraries, such as system libraries or your own libraries. Further, your own libraries might link to other libraries. To be able to compile your project, you must add the libraries to your project. This also enables code completion and syntax highlighting for the libraries. The procedure of adding a library to a project depends on the build system that you use.

CMake Projects

qmake Projects

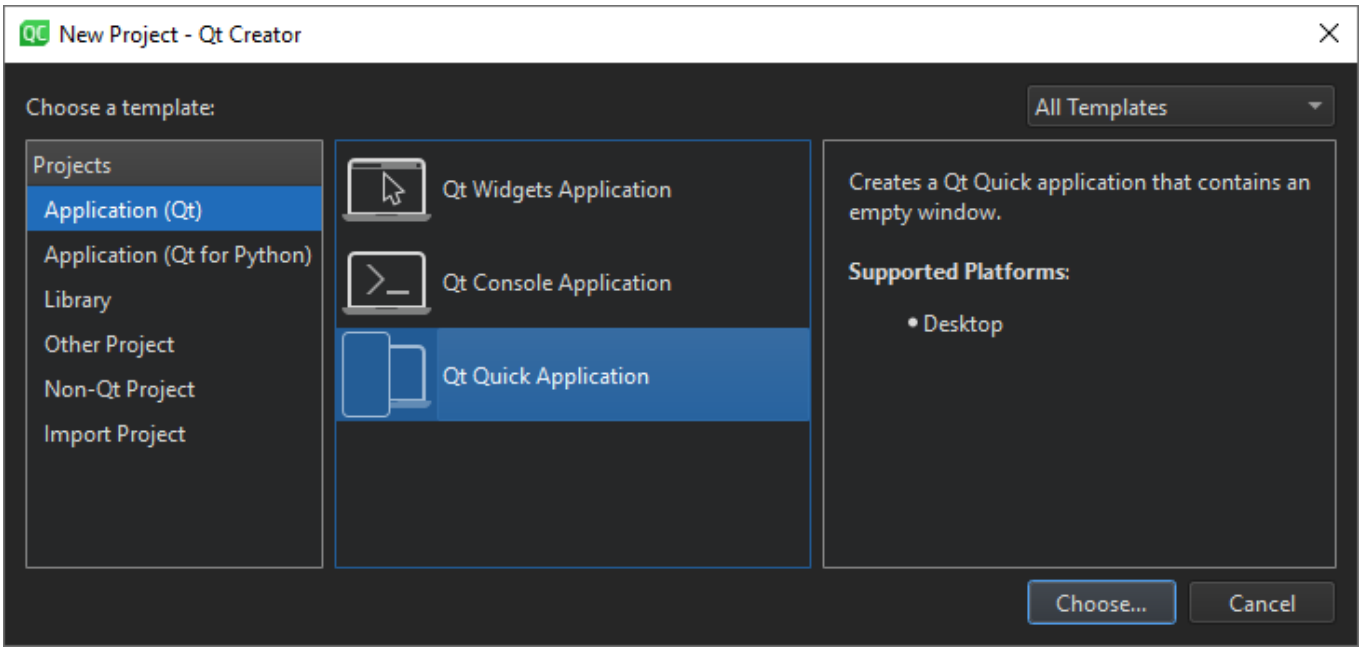
When you create a new project and select qmake as the build system, you can add it to another project as a subproject in the **Project Management** dialog. However, the root project must specify that qmake uses the template to build the project.`subdirs`

To create a root project, select **File** > **New Project** > **Other Project** > **Subdirs Project** > **Choose**.

On the **Summary** page, select **Finish & Add Subproject** to create the root project and to add another project, such as a C++ library.

The wizard creates a project file (.pro) that defines a template and the subproject that you add as a value of the SUBDIRS variable. It also adds all the necessary files for the subproject.`subdirs`

To create more subprojects, right-click the project name in the **Projects** view to open the context menu, and select **New Subproject**. Follow the steps in the **New Subproject** wizard to create a subproject.



To add an existing project as a subproject, select **Add Existing Projects** in the context menu. In the file browser dialog, locate your subproject.

To remove subprojects, right-click the project name in the **Projects** view, and select **Remove Subproject** in the context menu.

To specify dependencies, use the **Add Library** wizard. For more information, see Adding Libraries to Projects.

Binding Keyboard Shortcuts to Wizards

If you use a wizard regularly, you can bind a custom keyboard shortcut to it. Triggering this keyboard shortcut will directly open the wizard without the need to navigate to **File** > **New File** or **New Project**.

Keyboard shortcuts for wizards can be set in **Edit** > **Preferences** > **Environment** > **Keyboard** > **Wizard**. All wizard actions start with **Impl** there.

Related Topics

‹ Managing Projects

Creating Files ›