# Qt Ecosystem

The Qt framework encompasses a wide range of modules that provide various functionalities for application development. Below is a list of the main modules in the Qt ecosystem.

Before continue!

Treat the list below as a reference only. It does not include all modules in the Qt framework, and it should not, actually. When you read it, you should not focus on any specific module or try to differentiate between modules, but you should **focus only on the key point: Qt framework is huge**. It includes all major features that you're expecting on a UI framework, including *open-source*, *cross-platforms*, *cross-languages*, *modern dynamic UI-UX*, *in-depth platform libraries*, *dev tools*, *test tools*, etc. All of these advantages make Qt a worthy-to-use framework.

## Modules

### Core

* **QtCore**: Basic non-GUI functionality used by other moduels, including event handling, data types, object model, etc.
* **QtGui**: GUI components, including 2D graphics, fonts, and basic windowing.
* **QtWidgets**: Traditional desktop-style widgets for building user interfaces.
* **QtQuick**: Framework for building fluid and dynamic user interfaces using **QML** (Qt Modeling Language) – a declarative language.
* **QtQml**: QML and JavaScript.

### Networking and Data

* **QtNetwork**: Network programming, including TCP/IP, UDP, HTTP, FTP, Bluetooth
* **QtSql**: Database access and SQL support.
* **QtXml**: XML handling and parsing.
* **QtWebSockets**: WebSocket communication.
* **QtWebEngine**: Web rendering and integration for displaying web content.
* …

### Multimedia

* **QtMultimedia**: Audio and video playback and recording.
* **QtMultimediaWidgets**: Widgets for multimedia content.
* …

### Graphics and Animation

* **Qt3D**: 3D graphics rendering and interaction.
* **QtCharts**: Charting and data visualization.
* **QtDataVisualization**: 3D data visualization.
* **QtOpenGL**:
* …

### Scripting and Language Integration

* **QtScript**: (Deprecated) Scripting support using JavaScript.
* **QtConcurrent**: Support for concurrent programming.
* **QtTest**: Unit test framework for Qt applications and libraries.
* **QtQuickTest**: Unit test framework for QML applications, where the test cases are written as JavaScript functions.
* …

### Utilities

* **QtLottie**: Support for rendering Lottie animations.
* **QtHelp**: Help documentation system.
* **QtLocation**: Location-based services and mapping.
* **QtPositioning**: Positioning and location services.
* …

### Advanced Features

* **QtRemoteObjects**: Communication between different processes, devices, or platforms.
* **QtWebEngineCore**: Core functionalities of the Qt WebEngine module.
* …

## IDEs

* **Qt Creator**: IDE is for developing applications with a comprehensive set of tools for writing (syntax highlighting, code completion, etc.), debugging, version control, build system integration, and deploying Qt applications. If you already worked with MS Visual Studio, then Qt Creator is similar.
* **Qt Designer**: Design tool specifically focused on designing UI components using *QtWidgets*. It allows developers to create and arrange widgets visually without needing to write code for layout.

Note: Qt Creator includes Qt Designer.

* **Qt Design Studio**: A more advanced and modern design tool focused on UI/UX, allowing designers to create and prototype user interfaces using *QML*.

## Platform-Specific Modules

* **QtAndroid**: Android-specific features and functionalities.
* **QtIOS**: iOS-specific features and functionalities.
* **QtMacExtras**: macOS-specific features.
* **QtWinExtras**: Windows-specific features.

## Programing Languages

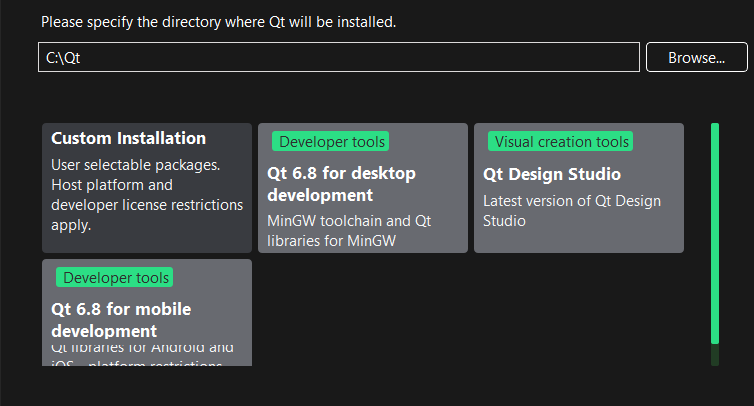
* **C++:** Main language for Qt. All modules in the Qt ecosystem are compatible with C++.
* **Python:** Second most popularlanguage for Qt with two options of Python binding:
  + **PyQt**: The unofficial yet famous set of Python bindings for Qt, developed by the Riverband Computing.
  + **PySide**: The official set of Python bindings for Qt, developed by the Qt Company.
* **JavaScript**: Qt provides QML which is often combined with JavaScript for scripting.
* **Ruby**:
  + **QtRuby**: Ruby bindings for the Qt framework, allowing you to develop Qt applications using Ruby.
* **Java**:
  + **Qt Jambi**: A set of Java bindings for the Qt framework. It allows Java developers to create applications using Qt.
* **Go**:
  + **therecipe/qt**: Go bindings for Qt, enabling you to write Qt applications in Go.
* **C#:**
  + **Qml.Net**: A library that allows .NET applications to use QML for UI design.
  + **QtSharp**: A set of .NET bindings for Qt, making it possible to build Qt applications using C#.

# Qt Installer

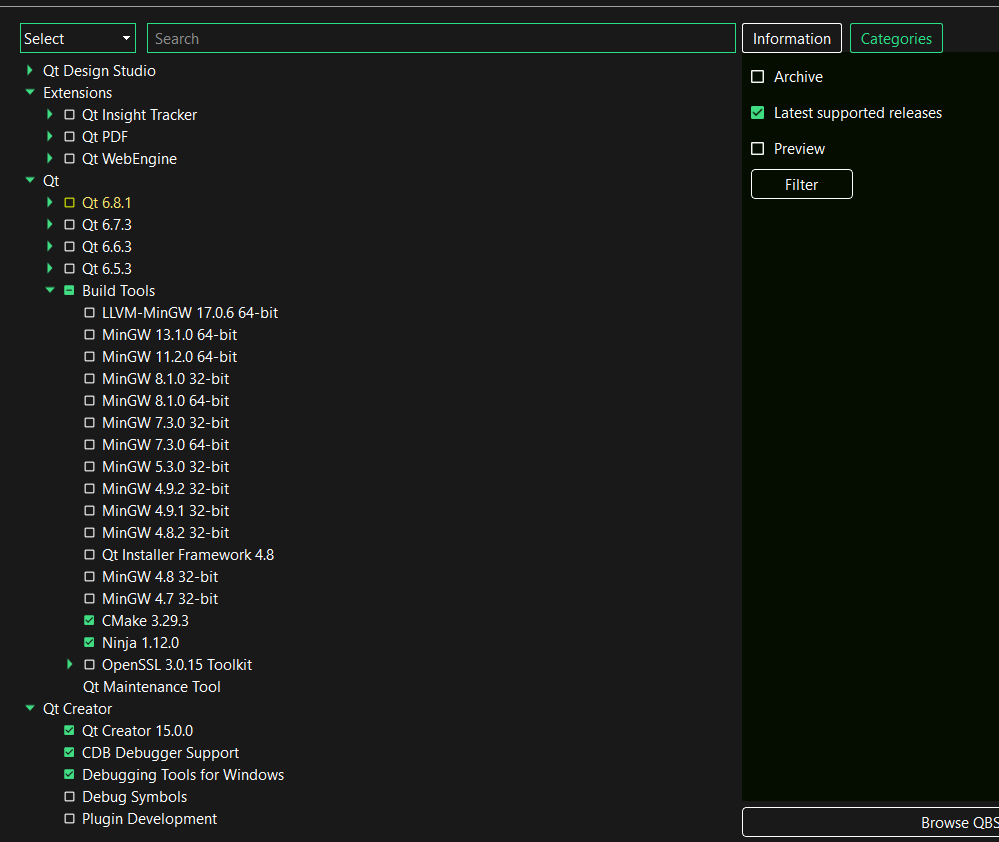
The first step to use Qt Framework is downloading necessary libraries and tools. Qt Installer is the best way to go.

Download Qt Installer at <https://www.qt.io/download-qt-installer> or <https://web.stanford.edu/dept/cs_edu/resources/qt/install-windows>

Following is what you can find from Qt Installer:



Or more details:



Note:

* At the "Welcome" step, Qt Installer will download data from its server. If the download process fails, you can stop the current Qt Installer instance, then configure Qt Installer to use a different mirror by running command line "<your-qt-installer-bin> --mirror <mirror-url>". This will open a new Qt Installer instance using your new configurations.

An example mirror URL is <https://download.qt.io/online/qtsdkrepository/windows_arm64/desktop/tools_vcredist/Updates.xml.mirrorlist>.

# Qt Widgets

In Qt (and most User Interfaces), widget is the name given to a component of the UI that the user can interact with. User interfaces are made up of multiple widgets, arranged within the window.

Qt comes with a large selection of **built-in widgets** and even allows you to **create your own custom widgets**.

## Built-In Widgets

|  |  |
| --- | --- |
| **Widget** | **What it does** |
| QCheckBox | Checkbox to tick or untick |
| QComboBox | Dropdown list box to select one item |
| QFontComboBox | Dropdown list box to select a font |
| QLineEdit | Enter a line of text |
| QDateEdit | Edit box to edit dates |
| QDateTimeEdit | Edit box to edit dates and datetimes |
| QTimeEdit | Edit box to edit time |
| QDial | Rotateable dial |
| QSpinBox | Spinner for integers |
| QDoubleSpinBox | Spinner for floats |
| QLCDNumber | Quite ugly LCD display |
| QLabel | Just a label, not interactive |
| QProgressBar | Progress bar |
| QPushButton | Push button |
| QRadioButton | Toggle set, with only one active item |
| QPaletteButton |  |
| QSlider | A slider |
| QListWidget |  |
| QTabWidget |  |
| QToolBar | Bar of icons and/or text used to perform common tasks within an app |
| QDialog | Small modal (or blocking) window that sits in front of the main app until dismissed.  Mostly used to communicate with the user. |
| QMessageBox | Simple form of diaglog where there is a message with buttons for the user to accept or cancel the dialog |
| QSystemTrayIcon |  |
| QcolorDialog |  |
| ... |  |

## Custom Widgets

### Bitmap Graphics

The first step for creating custom widgets is understanding bitmap (pixel-based) graphic operations. All **standard widgets draw themselves as bitmaps** on a rectangular "canvas" that forms the shape of the widget. Once you understand how this works, you can draw any widget you like.

A *bitmap* is a rectangular grids of pixels, where each *pixel* is stored individually as a number of bits. Contrast with vector graphics, where the image is stored as a series of drawing instructions which are repeated to form the image.

QPainter is the Qt's API for performing bitmap graphic operations and the basis for drawing your own widgets.

### Examples

* Animated toggle button: <https://www.pythonguis.com/tutorials/pyside6-animated-widgets/>
* Equalizer Bar, Power Bar, Password Edit, etc.: <https://pypi.org/project/qtwidgets/>. Note: This library does not support PySide6. So, we need to modify it a little bit by replacing all "PySide2" and "qtpy" by "PySide6" . There might be side effect, but not much, I think.

## Notes

* Qt Widgets is an **old technology** and does not have a good support for high DPI settings, making these images look distorted when your system has DPI applied above 100%.
* You should design your UI graphically using the *Qt Designer*. The same code is just to help you understand the underlying system.

# Qt QML

Check *Qt\QML.docx* tutorial.

QML vs Qt Widgets:

You have the option of using QML for user interface design rather than widgets. It is a much easier way to write many kinds of applications. It enables a completely customizable appearance, touch-reactive elements, and smooth animated transitions, taking advantage of hardware acceleration.

# Events, Signals and Slots

## Event

*Events* are **every interaction the user has** with the application. There are many types of events, each representing a different type of interaction. Qt represents these events using event objects which package up information about what happened. These events are passed to specific *event handlers* on the widget where the interaction occurred.

By defining custom, or *extended event handlers*, you can alter the way your widgets respond to these events. Event handlers are defined just like any other method, but the name is specific for the type of event they handle.

### Example

One of the main events which widgets receive is the QMouseEvent. It's created for each and every mouse movement and button click on a widget. The following event handlers are available for handling mouse events:

|  |  |
| --- | --- |
| **Event handler** | **Event description** |
| mouseMoveEvent() | Mouse moved |
| mousePressEvent() | Mouse button pressed |
| mouseReleaseEvent() | Mouse button released |
| mouseDoubleClickEvent() | Double click detected |

For example, clicking on a widget will cause a QMouseEvent to be sent to the mousePressEvent() event handler on that widget. This handler can use the event object to find out information about what happened, such as what triggered the event and where specifically it occurred.

### Event Hierarchy

In Qt, every widget is part of two distinct hierarchies:

* The object (in code) hierarchy
* The Qt layout hierarchy
  + When you add a widget to your app, it also gets a parent from the layout. The widget parent can be found by calling .parent().
  + When UI events are triggered, these are passed to the uppermost widget. So, if you have a QPushButton on a QVBoxLayout and click the button, then the QPushButton will receive the event first. If the button cannot handle the event or chooses not to, then the event will bubble up to the parent widget which is the QVBoxLayout. This bubbling continues all the way up nested widgets until the event is handled or it reaches the main window.

How you respond or ignore events can affect how your UI behaves.

In your event handlers, you can choose to mark an event as *handled* by calling the accept() :

# Python

class CustomButton(QPushButton)

def mousePressEvent(self, e):

e.accept()

Or you can mark it as *unhandled* by calling ignore() . In this case, the event will continue to bubble up the hierarchy:

# Python

class CustomButton(QPushButton)

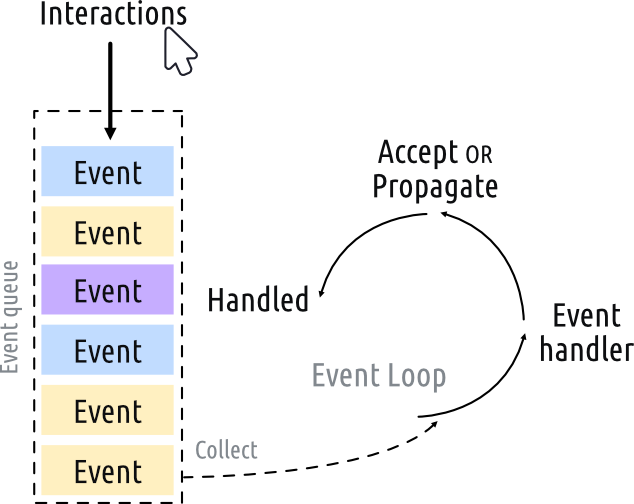
def event(self, e):

e.ignore()

If you want your widget to appear transparent to events, you can safely ignore events that you've actually responded to in some way. Similarly, you can choose to accept events you are not responding to in order to silence them.

### Event Loop

The core of every Qt Applications is the QApplication class. Every application needs one and only one QApplication . Also, **QApplication  holds the *event loop*** **of your application** — the core loop which governs all user interaction with the GUI.



Each interaction with your application — whether a press of a key, click of a mouse, or mouse movement — generates an *event* which is placed on the *event queue*. In the event loop, the queue is checked on each iteration and if a waiting event is found, the event and control is passed to the specific *event handler* for the event. The event handler deals with the event, then passes control back to the event loop to wait for more events. There is **only *one* running event loop per application and at any time**. Your application sits waiting in the event loop until an action is taken.

## Signal

When an **event occurs, the widget emits a signal** to let you know that the event has occurred.

In addition to notifying about event happening, signals can **also send data** to provide additional context about what happened. Although there is a limitation that the signal can only emit the data it was designed to, this is a powerful way to [extend or modify the built-in signals provided by Qt](#_Transmitting_Extra_Data).

Most widgets implement **built-in** signals that are emitted when a given event (like a mouse click) occurs on the widget

## Slot

Slots are **methods that are executed** as a response to events. And you need to **connect signals to specific slots**.

Most widgets provide **built-in** slots that allow you to perform certain actions on them. For example, a QTextEdit object provides a .clear() slot that you can connect to a button or to a menu option to clear the content of the widget.

# Layouts

There are 4 basic layouts available in Qt:

|  |  |  |
| --- | --- | --- |
| **Layout** | **Behaviour** |  |
| QHBoxLayout | Linear horizontal layout  (widgets one above the other linearly) | A QVBoxLayout, filled from top to bottom. |
| QVBoxLayout | Linear vertical layout  (widgets one above the other horizontally) | A QHBoxLayout, filled from left to right. |
| QGridLayout | In indexable grid XxY  (widgets in a grid) | A QGridLayout with unfilled slots. |
| QStackedLayout | Stacked (z) in front of one another  (multiple widgets in the same space) | QStackedLayout, with the 2nd (1) widget selected and brought to the front. |

# Resources

In the context of Qt, a resource refers to **any data file or asset packaged within the application**, such as images, audios, videos, fonts, CSS, configuration files (XML, JSON, etc.), translation files, etc. Your application can access and use them at runtime without needing to be located externally.

Resources are defined in .qrc files. They're simply XML files which store all resource files of a Qt program.

**Syntax**:

|  |  |
| --- | --- |
| **Element** | **Description** |
| Root Element | <RCC> </RCC>  The root element for every QRC file. |
| version=  Optional attribute to specify the **version of the resource** collection. |
| Resource Section | <qresource> </qresource>  Resource section where resources are defined.  You can have multiple resource sections with different prefixes in a single QRC file. |
| prefix=  Prefix for referencing resources in your application.  It's like a namespace to **group resources** together. |
| <file>  **Path** to the file, **relative** to the resource file |
| alias=  Optional **short name** for the file. You it to simplify your app, while keeping the original name externally. |

**Example**:

File resource.qrc

<RCC version="1.0">

    <qresource prefix="/icons">

        <file alias="logo">images/logo.png</file>

        <file alias="backgroud">images/background.jpg</file>

    </qresource>

    <qresource prefix="/">

        <file>pic1.png</file>

        <file>pic2.png</file>

    </qresource>

</RCC>

To use a .qrc file, you need to:

* Compile it to code:
  + In Python and PySide6, use command: pyside6-rcc resources.qrc -o resources.py
  + In C++, use command: xxx
* Import resources to code:
  + In Python:

import resource # Suppose resource.qrc is compiled to resource.py

icon = QtGui.QIcon(":/icons/logo")

button = QtWidgets.QPushButton("My Button")

button.setIcon(icon)

The prefix :/ indicates that this is a resource path.

**When to use resource?**

The main advantage comes when **packaging and distributing your apps**. Because your data is bundled with the Python source code, you eliminate all the potential path problems and guarantee your data files will be accessible to your app.

You can also use Qt Designer to manage and group icons for your application.

The downside of course is you need to **re-compile your resources any time** you add/remove new resources.

Details: <https://doc.qt.io/qt-6/resources.html>

# Multi-Threads

There are two main approaches to running independent tasks within a Qt app: **threads** and **processes**.

* **Threads** share the same memory space, so consume minimal resources and convenient to pass data between threads, however reading/writing memory from different threads can lead to race conditions or segfaults. In C++, multiple threads are fast. However, in Python, multiple threads are bound by the same Global Interpreter Lock (**GIL**) — meaning non-GIL-releasing Python code can only execute in one thread at a time.
* **Processes** use separate memory space and an entirely separate Python interpreter. This side-steps any potential problems with the GIL, but at the cost of slower start-up times, larger memory overhead and complexity in sending/receiving data. For simplicity's sake it usually makes sense to use threads, unless you have a good reason to use processes (see caveats later). Subprocesses in Qt are better suited to running and communicating with external programs.

Here we focus on the threads. In the next section, we will talk about processes.

## QRunnable and QThreadPool

Qt provides a very simple interface for running jobs in multiple threads. This is built around two classes: QRunnable and QThreadPool. The former is the container for the work you want to perform, while the latter is the method by which you pass that work to alternate threads.

The neat thing about using QThreadPool is that it handles queuing and execution of workers for you. Other than queuing up jobs and retreiving the results there is not very much to do at all.

# Multi-Processes

## QProcess

Qt system for running external programs from within your own app.

# Transmitting Extra Data with Qt Signals

<https://www.pythonguis.com/tutorials/pyside6-transmitting-extra-data-qt-signals/>

# Model-View-Controller

## How Qt Defines MVC

The MVC design pattern decouples three major components:

* **Model** holds the data and internal logics which the app is working with.
* **View** is any representation of information as shown to the user, whether graphical or tables. Multiple views of the same data model are allowed.
* **Controller** accepts input from the user, transforming it into commands to for the model or view.

It Qt, the distinction between the View & Controller gets a little murky. Qt accepts input events from the user and delegates these to the widgets (Controller) to handle. These widgets also handle presentation of the current state to the user, putting them squarely in the GUI (View). Rather than agonize over where to draw the line, in Qt-speak the **View and Controller are merged** together creating a Model/ViewController architecture — called "**Model View**" for simplicity sake.

There is an in-depth discussion of the Qt architecture [in the documentation](http://doc.qt.io/qt-5/model-view-programming.html).

# Supported Tools

## VSCode Extensions

### [Qt Extension Pack](https://marketplace.visualstudio.com/items?itemName=TheQtCompany.qt)

Includes all the extension packs for developing Qt and Qt Quick applications with Visual Studio Code.



### [Qt for Python](https://marketplace.visualstudio.com/items?itemName=seanwu.vscode-qt-for-python)

**QML**

* Syntax highlighting (qml and qmldir files)
* **Linting** (requires PySide6 >= 6.3)
  + Support .qmllint.ini configuration file
* **Code completion** (requires PySide6 >= 6.4)
* **Preview** QML file in a separate window (requires PySide6)
* **Format** QML file (requires PySide6 >= 6.5.2)

**Qt UI Files**

* Syntax highlighting (ui files)
* **Compile to Python** code when running command or **on Save**
* **Open Qt Designer from UI file** (requires PySide6 or PySide2)

**Qt Resource Files**

* Syntax highlighting (qrc files)
* **Compile to Python** code when running command or **on Save**

**Qt Style Sheets**

* Syntax highlighting (qss files)
* Provide **color picker** for HEX, RGBA, HSVA, and HSLA code

**Qt Translation Files**

* Syntax highlighting (ts files)
* Extract translation strings from Python, QML and UI files
* Edit translations with Qt Linguist (requires PySide6)
* Compile to binary translation files (requires PySide6)