Building a GUI

A graphical interface earns its keep when an experiment needs live feedback — watching data arrive in real time, starting and stopping a run with a button, adjusting a setpoint without retyping a command. This guide builds a **live-plotting desktop app** with **PySide6** (the Qt framework for Python) and **PyQtGraph** (a fast plotting library built for real-time data).

It's a big step up in complexity from a script, so the guide focuses on the few ideas that actually matter: the event loop, signals and slots, embedding a live plot, and — the one that trips everyone up — updating the GUI safely from a background thread.

> **Prerequisites**
>
> Structuring a Data-Acquisition App (threading and the acquisition loop) and comfort with Python classes. `pip install pyside6 pyqtgraph`.

1 Which GUI Toolkit?

Toolkit	Reach for it when…
Tkinter	You need a few buttons and fields, no live plotting. It ships with Python — nothing to install — and is the lightest option for a simple control panel.
PySide6 (Qt)	You need real-time plots, a polished interface, or an app that will grow. More to learn, far more capable.

For plotting *within* a Qt app:

- **PyQtGraph** for live, interactive data (zoom, pan, updates many times per second).
- **Matplotlib** for static, publication-quality figures (reports, papers, slides).

This guide uses PySide6 + PyQtGraph because live data is the case that justifies a GUI in the first place. For a static control panel, a few Tkinter widgets are plenty.

2 Event Loops, Signals, and Slots

A script runs top to bottom and exits. A GUI instead starts an **event loop** that sits and waits for things to happen — a click, a keypress, a timer — and responds to each. In Qt the mechanism for "respond to each" is **signals and slots**:

- A **signal** is emitted when something happens (a button's `clicked` signal).
- A **slot** is a method that runs in response, once you **connect** the signal to it.

```
button.clicked.connect(self.on_click)   # when clicked is emitted, on_click runs
```

That one line of wiring is the heart of every Qt program. Everything else is creating widgets and deciding what their slots do.

3 A Minimal Window

The smallest complete PySide6 app: create the application, show a window, start the event loop.

```python
import sys
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Lab App")
        button = QPushButton("Click me")
        button.clicked.connect(self.on_click)
        self.setCentralWidget(button)

    def on_click(self):
        print("clicked")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec())     # starts the event loop; blocks until the window closes
```

`app.exec()` is the event loop. Nothing after it runs until the window closes — which is why long work can't go on the main thread (more on that below).

4 Embedding a Live Plot

PyQtGraph's `PlotWidget` drops straight into a Qt layout. Make it the central widget, keep a handle to a curve, and update the curve's data whenever new points arrive.

```python
import pyqtgraph as pg
from PySide6.QtWidgets import QMainWindow

class PlotWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Live Plot")
        self.plot_widget = pg.PlotWidget()
        self.plot_widget.setLabel("bottom", "Time (s)")
        self.plot_widget.setLabel("left", "Voltage (V)")
        self.setCentralWidget(self.plot_widget)

        self.curve = self.plot_widget.plot([], [], pen="y")
        self.xs, self.ys = [], []

    def add_point(self, x, y):
        self.xs.append(x)
        self.ys.append(y)
        self.curve.setData(self.xs, self.ys)   # redraws with the new point
```

`curve.setData(...)` is the entire live-update mechanism: append to your data arrays, hand them to the curve, and PyQtGraph repaints.

5 The Rule That Matters: Don't Touch the GUI From a Thread

Acquisition has to run off the main thread, or the event loop freezes and the window goes unresponsive. But Qt requires that **all GUI updates happen on the main thread** — calling `curve.setData(...)` directly from a worker thread will crash or corrupt the display.

The solution is to use signals as a thread-safe bridge. A worker emits a signal; Qt delivers it to a connected slot **on the main thread**, where touching the GUI is safe.

```
import time
from PySide6.QtCore import QObject, QThread, Signal

class Acquisition(QObject):
    """Runs in a worker thread; reports points via a signal."""
    new_point = Signal(float, float)     # carries (x, y)
    finished = Signal()

    def __init__(self):
        super().__init__()
        self.running = True

    def run(self):
        start = time.time()
        while self.running:
            t = time.time() - start
            value = self.read_instrument()      # your Device/DeviceClient call
            self.new_point.emit(t, value)        # safe: just emits a signal
            time.sleep(0.1)
        self.finished.emit()

    def read_instrument(self):
        import random
        return random.gauss(0, 1)                # stand-in for a real reading
```

Wire it together in the window: move the worker to a `QThread`, connect its `new_point` signal to the plot's `add_point` slot, and start it.

```
from PySide6.QtCore import QThread

class PlotWindow(QMainWindow):
    def start_acquisition(self):
        self.thread = QThread()
        self.worker = Acquisition()
        self.worker.moveToThread(self.thread)

        self.thread.started.connect(self.worker.run)
        self.worker.new_point.connect(self.add_point)   # delivered on main thread
        self.worker.finished.connect(self.thread.quit)

        self.thread.start()
```

```
    def stop_acquisition(self):
        self.worker.running = False     # loop exits, emits finished, thread quits
```

The pattern in one sentence: **worker threads emit signals, the main thread updates the GUI in the connected slots.** Internalize that and the rest of Qt threading follows. (If you also need *separate processes* to share the instrument bus, combine this with the instrument server.)

6 Where This Leads

This window mixes three concerns — talking to the instrument, deciding what to acquire, and drawing the interface — in one class. That's fine here; it becomes a problem as an app grows. **Architecting a Lab App (MVC)** shows how to separate them so each can change independently.

Back to Python for Lab Work